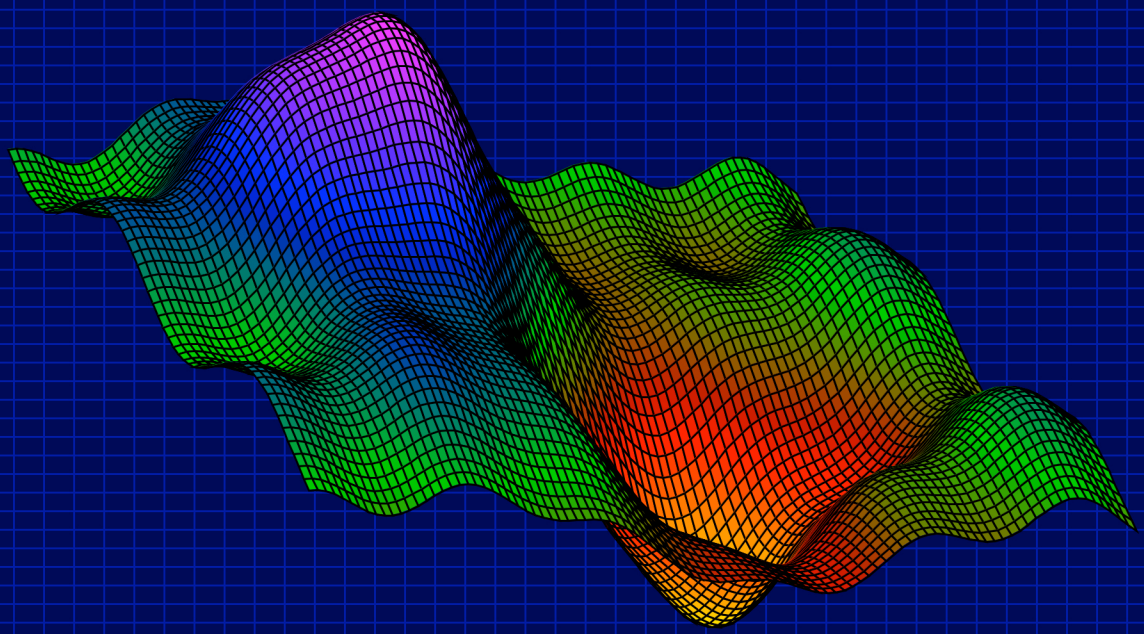
An Introduction To Object-Oriented Scientific Programming



Robert R. Birge
University of Connecticut

Based on MathScriptor 3.6.2

An Introduction To Object-Oriented Scientific Programming

Departments of Chemistry and of Molecular and Cell Biology
University of Connecticut

Robert R. Birge
Harold S. Schwenk Sr.
Distinguished Chair of Chemistry
University of Connecticut
Storrs CT 06269-3060 USA

based on
Scriptor and MathScriptor Versions 3.6.2

January 2015

TABLE OF CONTENTS

Chapter 1
Programming using Scriptor and MathScriptor

MathScriptor, the programming environment used in this course, is designed to serve two functions. First and foremost, Scriptor is designed as a teaching environment to help students learn object oriented programming efficiently. The user interface has been created to be as friendly as possible, and it can provide the student with syntax feedback on a real-time basis. It is simple to learn syntax and fix errors when the correct syntax is visible as one types. Second, MathScriptor provides the student with access to over 400 internal scientific, engineering, graphics and business functions that help carry out tasks relevant to both course work and research. The purpose of this chapter is to introduce the Scriptor and MathScriptor environments. A modest understanding of programming is needed to fully appreciate some of the material, and those readers with no background in programming may prefer to skim this chapter and return to it later after reading chapters 2 and 3.

MathScriptor is a cross-platform Tabbed Integrated Development Environment (TIDE) created using Xojo, an extended basic language developed by Xojo Inc. (www.xojo.com). Xojo (previously called RealBasic) is a high-level object oriented language that shares more similarities to Fortran 90, Pascal and C++ than to the original versions of Basic. MathScriptor is an extended version of Scriptor which resides in the same program, but which can only be activated by registered users. MathScriptor is designed for scientific programming. Scriptor can also generate stand-alone applications in the form of packages (see Appendix 1), but all applications developed using Scriptor must be run within the Scriptor TIDE. This is not a limitation but an advantage during the learning process because the student is freed from having to learn the intricacies of writing code associated with a graphics user interface. And because Scriptor is cross-platform, code written on a PC can be run on an Apple Macintosh computer under OSX and vice versa. All Scriptor programs can be distributed to and run by other Scriptor users. Furthermore, the Scriptor source code can be copied directly into Microsoft Word while retaining the colored styled-text formatting which provides the reader of the program with a better understanding of the program objects and structure.

This chapter describes the options and capabilities of the Scriptor TIDE. A full appreciation for these capabilities will necessarily require exploration of the programming language elements that are presented in subsequent chapters. The primary goal of this chapter is to outline the capabilities of each of the eight panels. It should be emphasized that until you are comfortable and experienced in using Scriptor, you will likely spend, and probably should spend, a majority of your time working in the Main Panel.

1.1. The Tabbed Integrated Development Environment

The Scriptor TIDE opens up in the Main Panel as shown in Fig. 1.1. The window on the left side is the text field into which the program is typed. The two windows on the right-hand side provide a small graphics canvas at top and a text output window directly below. The relative size of these two windows is adjustable using the slider at bottom right of the panel or from within the program using the statement set_graphics_slider. The small rectangular box directly above the set of buttons is an input text area, which allows the user to enter data to be read by the program. Most programming work is normally done in the Main panel and the program defaults to this panel when launched (see below).

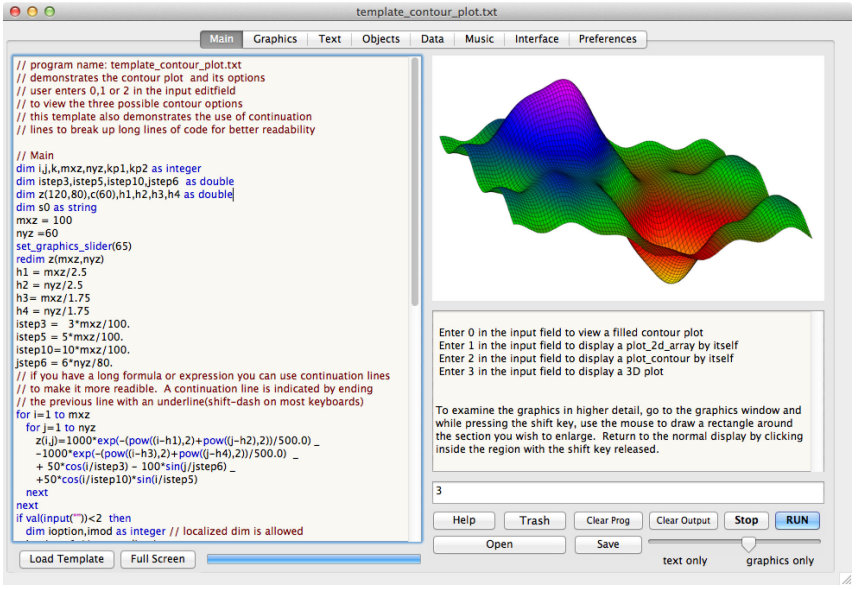


Figure 1.1. The Main Panel of the Scriptor TIDE. This panel provides the principal work area for writing and testing your program. A small graphics canvas, a text output window and a user input window are provided. The slider at lower right allows the user to adjust the relative size of the graphics versus text output windows. The user input window is the small rectangular window just above the two rows of buttons. Templates can be loaded by pressing the button at lower left.

The Scriptor program should be typed into the window on the left of the Main Panel. Any functions or subroutines that are to be included in this window should be placed at the top above the main program. You can also place subroutines, functions and/or modules in a separate panel ("Objects") as will be discussed later. As programs get larger and more complex, it will more efficient to work on only a small portion of the program at a time, and when your methods or classes have been debugged, make them available to the larger program using the Objects Panel. Because the Main panel is such

a critical work area, a more detailed discussion of this work area is presented in Section 1.2.

1.1.1. Graphics and Text Panels

The main panel provides relatively small graphics and text output windows, but Scriptor provides larger (nearly full-screen) versions of both types of output in the second and third tab panels. The Graphics tab panel is shown in Fig. 1.2 and the Text tab panel is shown in Fig. 1.3. You can write graphics to an invisible buffer (target=0), the graphics window in Main (target=1) or the graphics window in Graphics (target=2). Graphics can also be written directly to these windows. The Scriptor language provides a complete set of graphics statements that will be introduced in Chapter 3. Writing text output is simpler and will be discussed in Chapter 2.

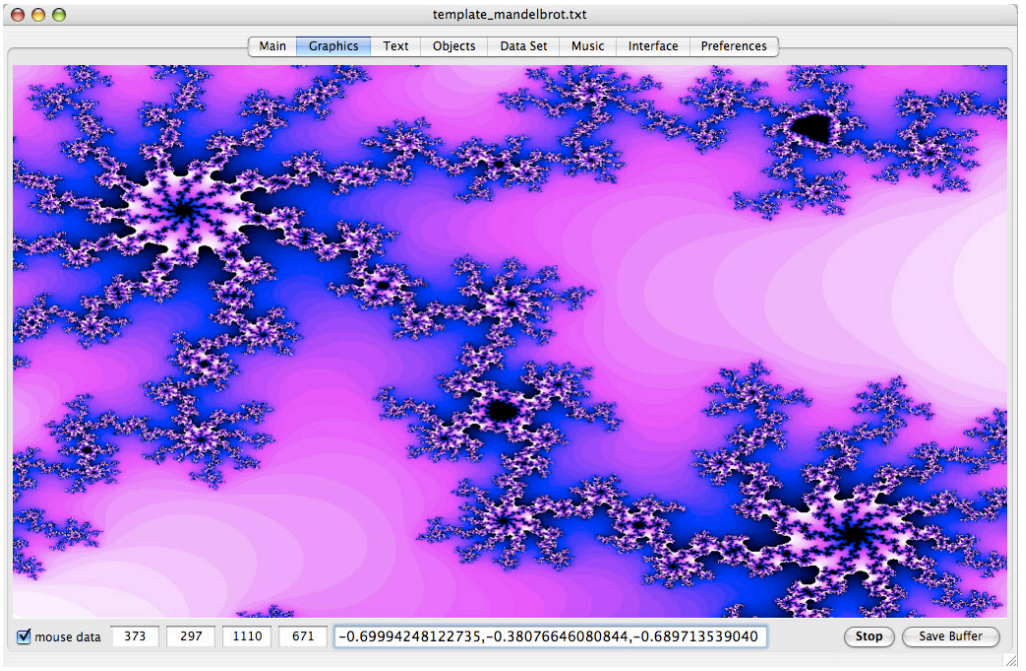


Figure 1.2. The Graphics panel of the Scriptor TIDE. This panel provides the large graphics output area which is addressed within a program by using target=2. The Save Buffer button allows the user to save the graphics to a user assigned file on disk. This panel also allows the user to run a program and provide input to the program via either the mouse or the user input field (the input field to the left of the stop button).

The graphics panel is the only panel which allows the user to enter data into a program via movement and location of the mouse with reference to the graphics canvas, or a picture presented inside the canvas. When the left or center mouse button is clicked inside the canvas, the location of the mouse is loaded into the Xdown and Ydown edit fields. When the button is released the location of the mouse is loaded into Xup and

Yup. Thus a mouse drag can be recorded and is available to the program if desired. If you press the shift key while dragging the mouse over a section of the canvas, that section is magnified into the entire canvas for inspection. The red rectangle that indicates the mouse drag region can be turned on and off under program control.

The text panel is for the presentation of character output. All text output generated by the Print statement is automatically sent to both of the text output windows (in Main and in Text).

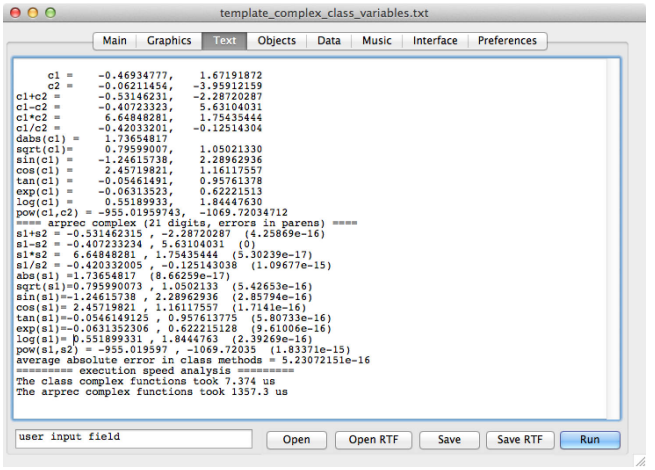


Figure 1.3. The Text panel of the Scriptor TIDE. This panel provides a large ASCII text output area and whenever the Print("...") statement is executed, the text in quotes is added to the bottom of this window. You can format output generated via the print() statement by using the set_text_style statement prior to the print() statement. You can also format text input by hand via access to the Style menu.

1.1.2. The Objects Panel

This panel is only relevant to MathScriptor programs. Objects are components of a program that add a new behavior or capability to the program. An object oriented programming environment provides programming objects that are encapsulated so that the interaction of these objects with the other program elements is fully controlled. The scientific programming environment provided by MathScriptor provides four types of objects: Functions, Subroutines, Modules and Classes. These can be included in the main program. But the Object tab panel provides four windows into which additional objects can be loaded. This panel is shown in Fig. 1.4. Four object windows is not a serious limitation, because one can have multiple functions and subroutines in the same window. While only one class can be loaded into each window, a class can be as large and complicated as desired. One can, for example, define one class that contains all of your functions and subroutines, although that would not be recommended as it would ultimately represent a poor choice with respect to organization and memory usage. In practice one rarely needs more than two objects available ("instantiated") in addition to the Main program. The example shown in Fig. 1.4. shows an example in which four classes are active at one time.

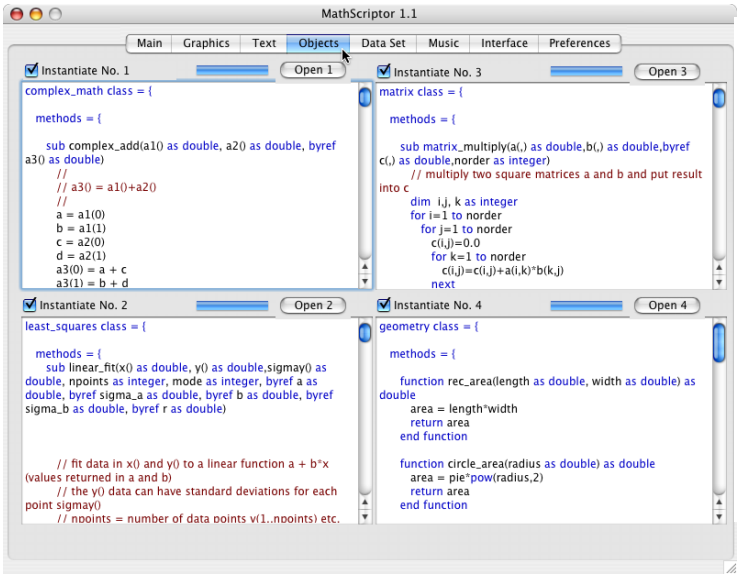


Figure 1.4. The Objects panel of the *Scriptor* TIDE. This panel provides four text windows into which *Scriptor* classes, functions or subroutines can be loaded using the Open button. The instantiation button at upper left of each window must be checked before the compiler will link the object to the program in Main.

At this point we remind the reader that if you do not know what a class is, do not be concerned. You will learn soon about functions, subroutines, modules and classes and how these objects communicate with your program to enhance functionality. The simplest of these objects are called functions. There are numerous internal functions such as Sqrt(x), which returns the square root of the argument, x. Indeed, Scriptor provides roughly 350 math, string and graphics functions which provide a rich programming environment. But the principal power of programming derives from the ability to create new functions as needed. This topic will be covered briefly in Chapter 2 and in more detail in subsequent chapters.

Note that in the previous paragraph, the first letter of Sqrt(x), was capitalized. Object oriented programming languages are case insensitive, so the decision to write "Sqrt()" with a capital S is one of personal preference. One can use any of the following strings to represent the same function: SQRT(x), Sqrt(x), sQrT(x) or sqrt(x). The compiler recognizes all of these functions as identical, and would treat the arguments X and x as identical.

Comments. Comments are critical parts of any program. The programmer should provide enough commentary to make it clear what the code sections are doing. Comments can be added to the program by using any of three identifiers: ‘, // or REM. Note that once a comment identifier is found, all text that follows is ignored until an end-of-line character is found. Thus you cannot use:

```
// Now we assign y to the square root of x, y=sqrt(x)
```

and expect the compiler to assign the sqrt(x) to the variable y. The compiler has no way to know where the comment stops and where you want the coding to begin. The above code is accepted by the compiler but the entire line is treated as a comment. One must be careful to provide the compiler with clear directions. Object oriented programming is powerful because it provides a framework that forces the code to be organized into compartments that communicate with each other using well defined rules. Object oriented code is easier to maintain by the programmer and those programmers that seek to maintain or update the code of others. The Objects panel is nothing more than a way to view objects and inform the compiler which of these are to be used and which are to be ignored for a given run.

1.1.3. The Data Panel

The Data Panel is shown in Fig. 1.5 and provides a scientific spreadsheet. A scientific spreadsheet differs from a free-form spreadsheet (for example, Excel) by treating each row as a coupled entity. Carrying out a sort on any column will automatically keep the rows intact. In a scientific spreadsheet, a row is viewed as an experiment where each column represents a different measurement.

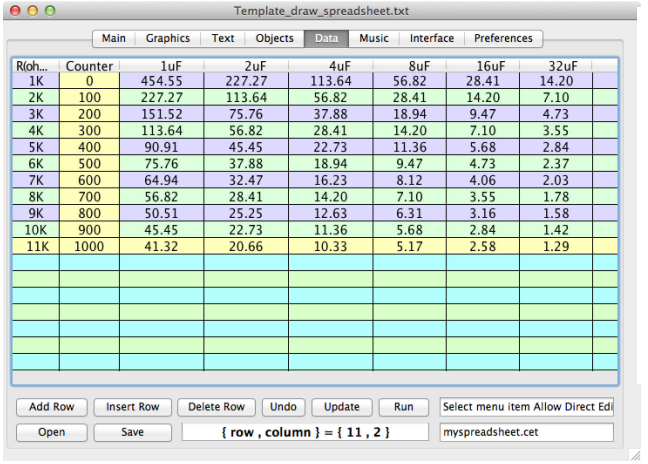


Figure 1.5. The Data panel is shown at left. This panel provides a single spreadsheet window into which one can import standard tab or comma delimited files. Scriptor spreadsheet intrinsics are available to read and write to individual cells or create a new spreadsheet with assigned rows, columns and headers. One can output the results of a calculation directly into the spreadsheet, and save the spreadsheet to a file to be read by users or other programs.

The spreadsheet can be created and modified under program control to have any number of rows limited only by computer memory. However, a maximum of 64 columns is allowed. Those familiar with programming will quickly appreciate how much easier it

is to organize tabular or matrix output using the Data Panel of Scriptor than using the text formatting statements available in Fortran, C or C++. The use of a spreadsheet simplifies this process significantly by automatically organizing the fields in well defined and visible rows and columns. Furthermore, the spreadsheet data can be exported into Microsoft Excel or any spreadsheet program that handles comma or tab delimited format. The data can also be exported as a figure for insertion into presentations or papers.

Inserting or Deleting Rows and Columns. The Data panel includes a number of buttons which are useful for setting up or manipulating the data. The first row of buttons provide control over the insertion or deletion of rows or columns. A row is defined as the data enclosed between two horizontal lines and a column is defined as the data enclosed between two vertical lines. When you click on any cell within the spreadsheet, the first click selects an entire row and the second click selects the cell directly underneath the cursor. Thus you will normally see row-based buttons in the first line. But if you select a header, you not only select the header, but the entire column and the buttons in the first row now change their names to indicate column-based activity. You can now delete the selected column, insert a new column to the left of the selected column, or add a new column at the far right of the spreadsheet. You have only one level of undo for any row or column deletions, so be careful. Any fancy manipulation of rows and columns is best done under program control.

Manipulating Data. When you click on a cell, the data in that cell are written into the small rectangular editfield to the right of the first row of buttons. You can modify the data within this editfield by selecting the data and typing in a replacement. The data in the cell within the spreadsheet are not modified until you press the **update** button. This sequence is the only way you can modify a header. However, you have the option to modify data directly within the spreadsheet by clicking a cell twice and typing directly into the cell. Note that you must click within the desired cell twice to select it for direct editing. The first click selects the entire row. The second click selects the cell for direct editing. Direct editing does not work on a header which can only be modified by using the editfield and the **update** button.

Resizing Columns. You can alter the width of any column by placing your mouse cursor over the separators within the header region. The cursor changes from a single arrow to a double arrow with a bar in between. If you press the mouse button at this point, you can drag the bar to alter the column size.

Adjusting Column Alignment. Each column in the spreadsheet has a defined alignment that can be set by selecting the header and using the Spreadsheet menu to select left, right, center or decimal alignment. The latter option also allows you to shift the decimal point to the left or the right within the column by using the shift menu

items. Pressing command+shift+< or command+shift+> allows for rapid adjustment of the horizontal position of the decimal point.

Sorting Data in the Spreadsheet. If you select a column by clicking on the header, you have the option to sort the rows by reference to the data in the selected column. Scriptor always requires that the rows be kept intact so it is not possible to sort only a few selected columns. The data can be sorted either numerically or alphanumerically in either ascending or descending order. When the data in the column are a mixture of numbers and alphanumerics you can still sort the data numerically, but all of those variables that begin with a non-numerical value are interpreted to equal zero which means they will be collected together between the negative and the positive numbers. If an alphanumeric begins with a number, the number is extracted from the beginning and used for sorting. Thus a numerical sort will yield the following list order: −4, −3.5, −3abc, −2, abc, 3.1, 3.2abc, 3.3. To understand the details, it is necessary to understand the nature of the data stored within a spreadsheet, the topic of the next section.

All of the Data in the Spreadsheet are Strings. The discussion of sorting raises an important point that will become clearer when we discuss the programming statements associated with manipulation of data sets in chapter 4. All of the spreadsheet cells are populated by string variables, regardless of the nature of the data. Thus, if the data cell contains the number 3.14159, it is stored in the form of a string [i.e. spreadsheet_cell(row, column)="3.14159"]. The exclusive use of strings provides the programmer with maximum flexibility in how the cell is populated and manipulated. But the exclusive use of strings can cause unanticipated problems when there is a mixture of numeric and alphanumeric in a column that is sorted. A numerical sort is carried out by converting all of the cells to numerical values using the internal function value(). This function is clever enough to convert "$dd.cc" to dd.cc, "12%" to 0.12, "1.234E4" to 12340, "-12.456" to –12.456, "(12.456) to –12.456 as well as handle comma delineators to convert "12, 345.67" to 12345.67. However, be warned that this function will convert a mixed string such as 12.34abc to 12.34 and convert abc12.34 to 0. An alphanumeric sort is easier to understand in that the sort orders the list based on standard alphanumeric (dictionary) ordering.

Importing Data. If you have a large data set in another spreadsheet program, you can import the data (via important menu options) into Scriptor by saving the spreadsheet as a comma or tab delimited file. Both options are provided by Microsoft Excel. You then have access to all of the data within your program using the **spreadsheet_cell**(i, j) statement, which returns a string.

Saving and Opening Spreadsheets. The two buttons at lower left of the data panel allow the user to manually open or save Scriptor spreadsheets files. Scriptor uses a tab delimited text format that can be read by other programs. However, the first few lines of a spreadsheet file contain control information unique to Scriptor. These data are clearly delineated from the spreadsheet contents and can be ignored or deleted.

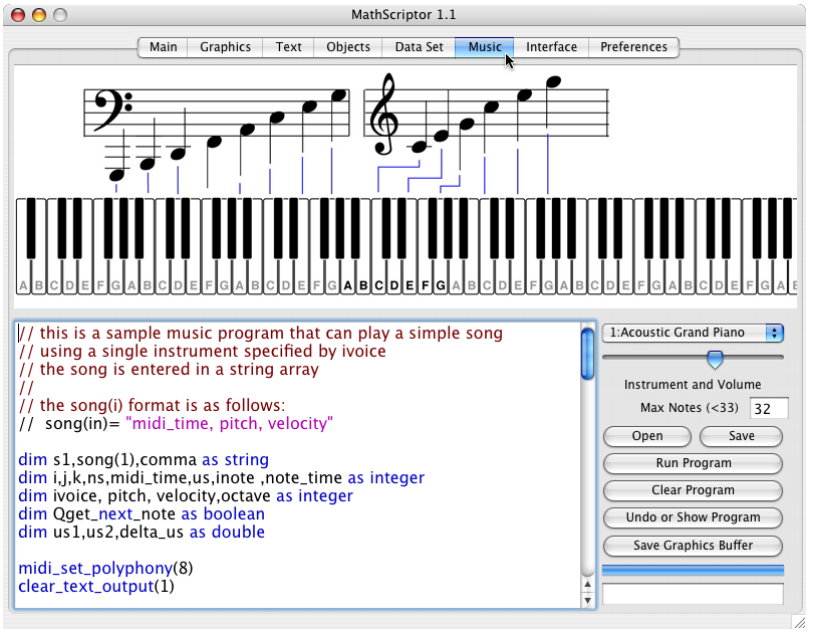


Figure 1.6. The Music panel provides program access to 128 instruments as well as a drum kit. The keyboard at the top can be used to test the instruments and provides a visual identification of notes when they are played by hand or under program control. This panel provides a complete programming environment which can be used for programs other than music (see text).

1.1.4. The Music Panel

Programming should be fun. The Music Panel (Fig. 1.6) provides access to the 127 high quality instruments that are available for free by downloading Quicktime from the Apple Computer web site:

http://www.apple.com/quicktime/products/qt/

There are Quicktime versions for both Windows and Macintosh computers and you do not need to purchase the professional version to have access to the instruments. When you switch to the Music Panel, Scriptor checks to make sure that Quicktime has been correctly installed and upon verification it displays the message:

Quicktime has been installed
and Midi statements are available

This statement only appears when the Music Panel is first selected, so if it is not visible, try selecting a different panel and then returning to the Music Panel. Another way to verify that the instruments are present is to play the keyboard making sure to adjust the main computer volume so that you can hear the sounds. Note that the volume control in the Music Panel (Fig. 1.6) is the local volume and that the master computer volume control must be turned up for you to hear anything. On Windows XP, this volume may be ignored and you will have to adjust the volume externally.

NOTE: Users of Scriptor versions higher than 3.5.0 that end in LLVM or XS can ignore the Quicktime discussion and the installation test. In these versions, Quicktime has been replaced by Core Audio. Core Audio is a cross-platform set of internal methods included in modern Windows and Mac operating systems. If your operating system does not include it, Core Audio can be downloaded and its functionality added. There is no harm in having Quicktime installed on your computer, however, as versions of Scriptor that use Core Audio will ignore Quicktime if it is present.

Quicktime and Core Audio Instruments. The popup menu just above the slider lists the instruments which are available along with the instrument identification number. You will need to know this number if you are going to access the desired instrument using software control. If you want to preview the sound, select this instrument using the popup menu and play it on the keyboard by clicking on the desired note. The number of notes that will play simultaneously is determined by the entry in the Max Notes (<33) window directly below the volume. You can set this to any number from 1 to 32. If you are playing the keyboard and want to manually turn the note(s) off, just click inside the keyboard window but above the keyboard. As you play, the notes are marked on the keyboard. This option is also available from the program so you can

monitor the notes that are active during program execution. Details can be found in Chapter 10.

Programming Features of the Music Panel. The Music Panel can be used for programming and graphics applications without reference to any of the musical instruments. This capability may seem superfluous but for some applications, the music panel may be preferred. First, the programming window is larger. Second, the graphics window is much wider than it is tall and for some graphics output, this canvas aspect ratio is an advantage. You can open any of the program files into this window by using the "Open" button, and when this button is used to open a program, it is automatically placed in the Music Panel Programming Window. When you press "Run" the text window is used to display the Text output. The graphics output can be directed to the Music Panel Graphics Canvas, and you can remove the keyboard under program control and replace it with graphics that you have created. The small rectangular window in the lower right serves as the user input window. After the program has run and you have examined the output, you can return to the program by pressing the "Undo or Show Program" button and the output text will be replaced with the program. Those users who prefer to do their programming within the Music Panel are often those working with a small computer screen (800 x 600). The only disadvantage of working in the Music Panel is that you do not have access to the help screen and keyword monitoring, which are only available when you are programming in the Main Panel.

1.1.5. The Interface Panel

The interface panel serves two purposes. This panel will help the user test or interrogate phidget interface boards that are connected to the computer via usb. These boards allow manipulation of servo motors, stepper motors, and the reading of data from temperature and light sensors, and many other instruments. There are a number of program statements dedicated to controlling phidgets (see Appendix 1 for details).

When phidgets are not connected, this panel is used to provide a list of ASCII codes which will be helpful to those programming. You have the option of replacing the ASCII codes with the help screen by selecting the appropriate option in the preferences panel. This option is particularly useful if you prefer to do your programming in the Music Panel where no help screen is available. You can then flip back and forth between the Music and Interface panels to provide access to a large programming window and a full window help file.

1.1.6. Preferences

The final panel in the TIDE is the Preference Panel which is shown in Fig. 1.7. This Panel will only be used occasionally. This panel is used to register your program, and subsequently to set preferences. We will discuss registration first, and then go over what some of the preferences represent. There are other preferences that are self-explanatory, and the user has access to a description by placing the mouse pointer over the preference and reading the text shown in the lower-right section of the preferences panel.

Registering Scriptor. Development of Scriptor was funded by the National Science Foundation, the National Institutes of Health and the Harold S. Schwenk Sr. Chair in Chemistry at the University of Connecticut. Continued funding of program development requires that we maintain metrics, and these metrics require that we stay in touch with the users. To accomplish this goal, we require that all users of Scriptor be registered so that we have contact information. Registration is free for students and other academic users. Please fill out the appropriate registration form inside the Registration_Emails folder and email this to rbirge@uconn.edu with Scriptor Registration in the Subject line. You will receive an email in return with all of the registration information. After receipt of the email, the easiest way to register the program is to save the registration email, or the attached text file, in the Registration_Emails folder. Next, open the text file using the "Open Registration Email" button on the right-hand-side of the panel. If you prefer, you can enter your first name, last name and Email address in the appropriate boxes and type in your license key, which will be of the form xxxx-yyyy-xxxx-yyyy-xxxx in the five license key edit fields. Then press "Register Program". If the process fails, check to make your name and email address are identical to those provided in the registration email form. Each user of Scriptor requires a separate license, but that license can be used on as many computers as desired. Please do not share your license key with other users or use the license numbers that can be found on the internet.

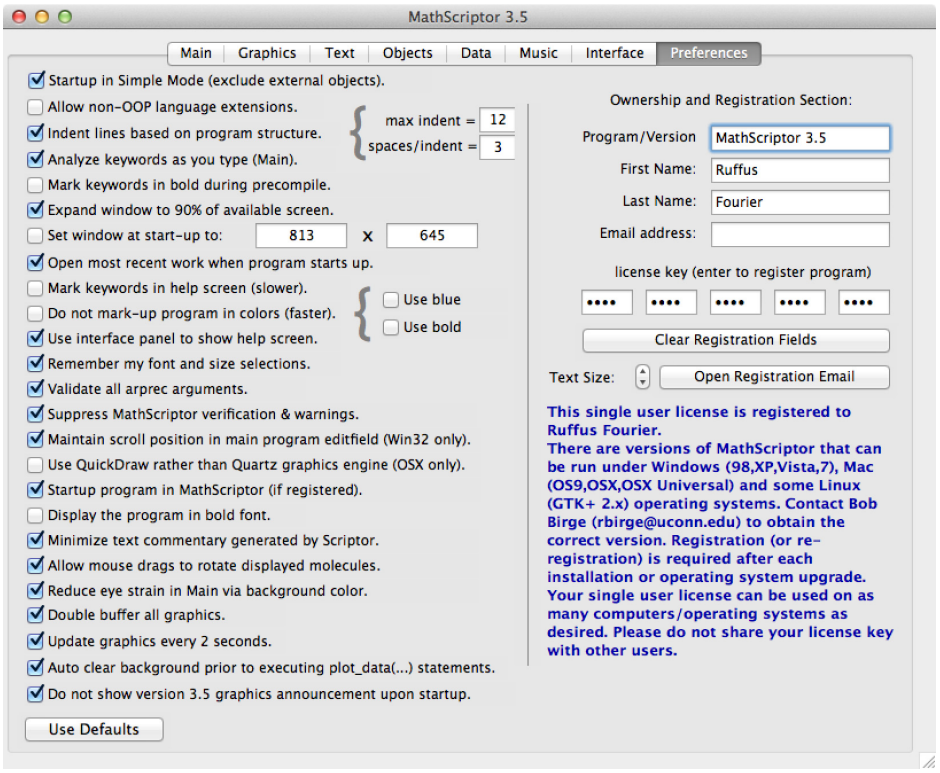


Figure 1.7. The Preferences panel of the Scriptor TIDE. This panel provides preference settings on the left-hand side and a registration section on the right-hand side. The program will need to be registered to gain access to the functions provided by MathScriptor as well as the option of saving compiled versions of your programs. The Use Defaults button provides a convenient way to initialize the preferences after you first register your program. Moving the mouse pointer over a preference displays a brief discussion of what the preference does on the right-hand side of the window.

Preference settings. Most of the preference settings are self explanatory. A brief description of the preference is presented when the user positions the mouse pointer over the preference description. Brief overviews of the more important preference settings are presented below. Notice that the Use Defaults button should be pressed by new users to select a default set of preferences considered appropriate for new users.

Startup in Simple Mode. Scriptor is designed to serve both as a learning tool and as a high-level programming and numerical methods environment. Simple mode allows a student to exclude external objects, which makes error checking faster and line error analysis more accurate. Students should use this option during their early learning experience until they need to make use of external objects. Experienced users should also select this option when no external objects are required simply to speed up execution.

Indent lines based on program structure. This preference sets the option of using line indentations to reflect program structure. Loops and control statements are easier to follow if each layer of the structure is similarly indented as shown in Fig. 1.8. However, some find program indentation annoying, particularly if they are working on a small screen computer. This preference panel allows each user to set this to suit their visual preferences. The program structure is only analyzed during a precompile which is initiated when you click on the progress bar at the bottom of the page or select clean code using the menu item or ctrl-K (command-K on a Mac).

```
for i=1 to n1
   i1 = max(1, i-dspan_half)
   i2 = i1+deltaspan-1
   if i2>nmax then
      i2=n1
      i1=n1-deltaspan+1
   end if
   k=0
   hw=0.0
   for j=i1 to i2
      k=k+1
      hw = hw + hscore(j)
   next
   if k<>deltaspan then
      print(" error, k="+str(k))
   end if
   yh1(i)=hw/k
   ymin = min(ymin, yh1(i))
   ymax = max(ymax, yh1(i))
next
```

Figure 1.8. Example of marking up a program using color to identify keywords and line indentations to reflect program structure.

Analyze keywords as you type in Main Panel. Scriptor has the useful ability to monitor the program during entry and recognize any keywords that you type. This capability is of significant help during the early learning stages, and is turned on when this preference is selected. For example, if we move to a new line and type the word matrix we get the following display in the text output editfield:

```
matrix_diagonalize(h2(), v2(), e1(), nsize, nroots)
matrix_gauss_jordan(a2(), b2(), nsize, ms)
matrix_invert(a2(), det, nsize)
matrix_print(a2(), nrows, ncols, ncols_per_set) as string
matrix_svd(a2(), v2(), w1(), m, n)
matrix_svd_backsubstitute(u2(), v2(), w1(), m, n, b1(), x1())
```

This listing presents all the functions that begin with the word matrix. If you then continue typing "matrix_i", there is only one routine that matches and the output editfield is updated to provide a brief discussion of the matrix_invert capability.

```
matrix_invert(a2(), det, nsize)
use LU decomposition to replace the square matrix
a2(1..nsize, 1..nsize) with its inverse and return the
determinant in det. Provides the determinant but no solution
vectors. Most efficient method.
```

If matrix_invert is not the desired function, a simple backspace returns the full list for inspection. The display provides rapid, real-time feedback and increases the efficiency of programming significantly. **It is strongly recommended that this preference be selected during the early learning process**. Real-time analysis does have one disadvantage. Because the text output editfield is used to display the keyword options, any output from the previous calculation is lost. Thus, there are situations when you might want to turn this preference off. You can turn preferences on and off at any time (but note that some preferences are only relevant during Scriptor startup).

Expand TIDE to 90% of available screen. &
Set TIDE at startup to width x height. These two preferences provide different methods of controlling the size of the TIDE. The default startup size for Scriptor is a window of size 800 by 600 pixels, which is the minimum size recommended. The first option automatically sets the application size to 90% of the available screen. The second option sets a specific size that you can enter by hand. It is easier, however, to turn this option off, adjust the application size with the grow icon at the bottom right, and let the program automatically set the width and the height values. Then when you have the program at the desired size, select the option again and it will always startup with this size. The fifth button has one additional attribute worth noting. If you ever want to return to the original default size, unchecking this button automatically returns the program to a window of size 800 by 600 pixels.

Open most recent work when program starts up. This preference provides the option of starting up each Scriptor session with the identical environment that was present when the program was shut down. This option is very useful if you are working on a complicated program and want the next session to pick up where you left off. However, this option should not be used in place of backing up your work in stages to make sure it is not lost due to a crash, power outage or user incompetence. In that regard, it is worth noting that each time you run your program, it is automatically saved inside your Programs folder in a folder called backups. In the event that your program locks-up or crashes Scriptor, you can retrieve the most recent version by directing the open dialogue to this folder and selecting the most recent backup file. The state of the Scriptor environment is only saved during a controlled Quit from the File menu.

Do not markup programs in color. When you do a precompile by pressing the progress bar or selecting the precompile option from the Debug menu, the keywords and internal routines are marked in blue, comments in red, compiler directives in green, and quotes

in purple. For some programmers, this is very helpful. For others, it is an annoyance. You can turn colorization on or off with this preference. Note that turning off colorization does not turn off structural indentation, which is controlled by a separate preference. An additional reason to consider turning off colorization is that it can be a slow process if the computer is slow or if the program is quite large. The speed issue derives from the fact that Scriptor must carry out colorization within a rich-text (stylized) editfield, and populating this field carries significant overhead. The loss of speed is compensated by the ability to copy and paste programs into Microsoft Word and most email programs without loosing the style information (font, text size and color are preserved). This capability is particularly useful for students preparing a discussion of their programs.

Use interface panel to show help screen. This preference allows the user to replace the editfield in the Interface Panel with a full-width help screen. This option has the advantage of providing the programmer with a help screen that is readily accessible by simply selecting the Interface Panel rather than using the output text field in the Main Panel. There are, however, two disadvantages. First, you are replacing the ASCII table which is normally present on this panel, and for some applications the ASCII table may be more useful. Second, it takes time to load the help screen and mark it in color, and this latency can be annoying on slow computers.

Reduce eye strain in Main via background color. When checked, the user can select an off-white color for the background of the editfields in Main. For example, selecting RGB(250, 248, 243), which is automatically shown the first time this option is selected, will significantly reduce eye strain by reducing the contrast between the program and the background. If programming causes a headache, this preference might help. Experienced programmers invariably use this option.

Double buffer all graphics. When checked, all graphics are double buffered prior to display on any of the canvases. However, on Windows computers, all graphics are automatically double buffered, so checking this preference will add one additional level of buffering which usually results in no graphics display at all. In general, this preference should only be used for the new cocoa Mac versions which are designed to make use of this feature to provide optimal high-resolution, flicker-free graphics.

Update graphics every 2 seconds. When checked, a filled buffer will be transferred automatically to the visible canvas every 2 seconds. This option is useful, particularly for students who often forget to add the buffer_copy_to_canvas() statement. More experienced users should turn this preference off as it can generate flicker during the update process.

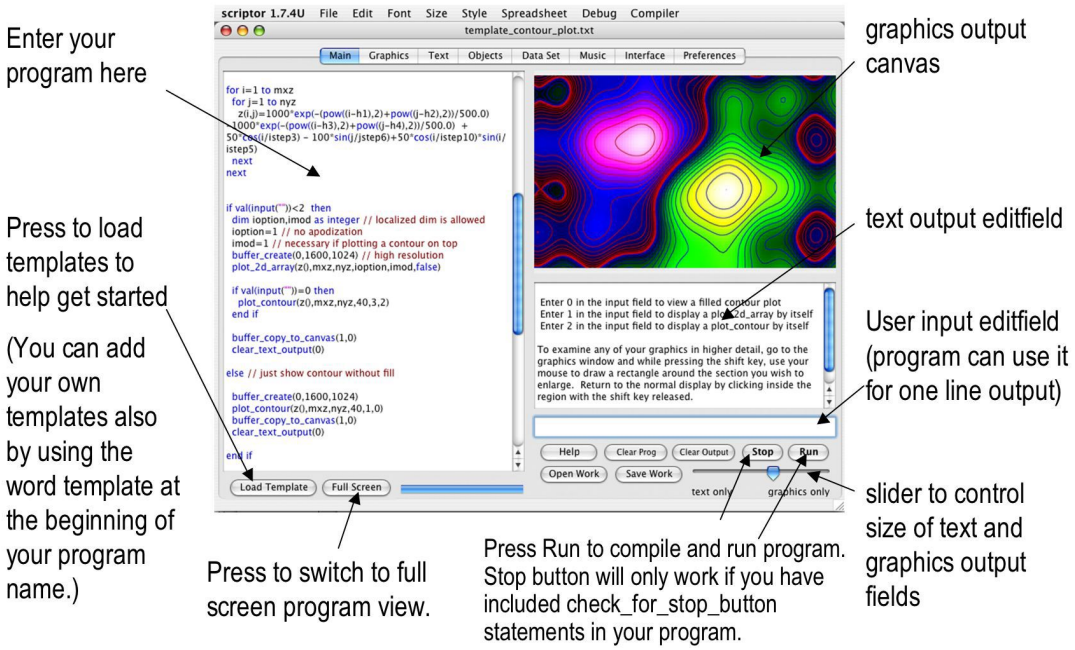


Figure 1.9. The Main Tab Panel provides the most flexible work space for writing and debugging programs. The output editfield can display the help screen when desired. The relative sizes of the graphics canvas and the text output editfield can be controlled by the slider at bottom right, or via program control.

1.2. The Main Panel

A majority of programming should be carried out in the Main Panel, because it is designed to simplify and enhance the process. These capabilities come with some degrees of freedom that require more detailed discussion. This section will provide the necessary details and some suggestions to make the programming experience more efficient.

When starting to write a new program, one needs to clear the programming window and this is done by pressing the Clear Program button. When **Clear Prog** is pressed, a blank header with sample declarations is created. If the shift key is pressed simultaneously with **Clear Prog**, a completely blank program is created. If the option key is pressed simultaneously, a scientific template including declared fundamental constants is loaded. These options are explored more fully in section 1.2.5. Alternatively, you can start out by opening a template program that does something similar to what you plan to do (see next section).

Run Button. The Run button does two things. First, it checks the program for errors and reports the errors in the text window on the right-hand side of Main. Second, if no errors are found, the program is compiled and execution is started. For a typical program, this process takes only a few seconds. If errors are found, however, each error is reported with reference to a line number and the program is modified to display the line numbers to help direct the programmer to the error (see Section on understanding error messages below). After the errors are corrected, press the **RESET** button and the line numbers are removed. (Line numbers can also be toggled on and off using the "Add or Remove Line Numbers" menu item under the Debug menu. It is easier to do this using the keyboard combinations: ctrl-L on windows or command-L on the Mac The **RUN** button will then reappear and you can try to run the program again. The cycle continues until the program compiles successfully. Sometimes the process fails to return the **RUN** button due to background tasks that interfere with the update process. You can force the **RUN** button to appear by pressing the **Full Screen** button twice.

Templates. The lowest left button in Main accesses the programs in the Templates folder. If Scriptor is being used in a programming or numerical methods course, the instructor will provide the class with a templates folder that includes examples relevant to the class material. To access each template in alphabetical order, simply press the button and monitor the name of the template as it is loaded. If you accidentally go past the template you want, simply hold down the alt (or on the Mac, option) key and the next button press will select the previous Template. More precisely, the alt (or option) button makes each subsequent selection in reverse alphabetical order. To access an individual template by name, hold down the shift key when you press the template button. This operation will open up a dialogue box which will allow the direct selection of a template by name. Templates serve three purposes. First, they provide short examples of how to do certain tasks designed to be easily updated or modified for a similar task. Second, they demonstrate programming techniques that are best illustrated by example rather than textbook discussion. If you are new to programming in Scriptor, stepping through the templates is an excellent way to learn. Third, the user can use the Templates folder to store programs that will be reused later as templates for other programs. To write and store a template is simple. Starting with a working program, make sure it is well commented, remove any programming elements that you consider are redundant or irrelevant, and rename the program so that the name begins with "Template_". When this program is saved, it will automatically be placed in the Templates folder.

In the event that the Templates button fails to open a template, the problem is invariably due to running Scriptor outside of its environment. The Scriptor program must be in a folder that also contains the Programs, Templates, data_sets, etc. folders that are used by the program as critical resources. Although there are times when one may wish to make a copy of the program and run it in a new location, if one wants access to templates, the

Templates folder must also be copied and moved into the parent folder. And if MathScriptor is to be used, many of the math functions require access to files within the data_sets folder.

The Help Screen. The Output editfield serves a dual purpose. When a program is running, this editfield is used by the Print statement. But while you are writing a program, you can select the preference "Analyze keywords as you type" , this output field will be used to provide real-time analysis of the code that you are typing in. A description of this capability was provided in the above discussion on preferences, and is turned on be checking the "Analyze keywords as you type in Main Panel" preference. An alternative is to press the help key and load the help screen. The help screen comes in two versions. Normally, a relatively short Help Screen is loaded which provides a discussion of each keyword and can be scanned quickly using the scroll bar. If you press the shift key, a longer version of the Help Screen is loaded which provides a more detailed discussion of the elements of the Scriptor language. Appendix 1 of this book provides a more detailed discussion of the language.

You are welcome to modify the help screen (either the short or long version) with your own comments and clarifications. If you do, make sure you save the revised file by pressing the Save Help button. You would be well advised to save backup copies of the original short (bschlp_short.txt) and long (bschlp.txt) help files prior to making any significant changes. The backup can be created by using the Save Help button or simply selecting and duplicating the files, which are found in the help_files folder.

Open and Save Buttons. These buttons are used to open programs and to save the current program. Programs can be stored anywhere the user wishes, but the optimal location is within the Programs folder. You can add folders inside the Program folder to help organization.

The Trash Button. If you have a file in any of the folders within the Scriptor Environment that you wish to delete, it is often convenient to carry out the process from the Main panel by pressing the Trash button and directing the window dialogue to the desired file. The selected file is moved to the Trash folder inside the Scriptor environment. If you make a mistake, you can always move the file by hand to the original or alternate location. The theory behind using a Trash folder is safety. An accidental "trashing" is easily fixed and the trash process is cross-platform in scope.

Understanding error messages. While you are in the Main Tab Panel, you have access to a number of menu items that help debugging. When an error occurs, the compiler does its best to identify the line number in which the error occurred. Errors are then printed out as shown in the following example:

```
 line | error number and error message
0025 | Error No. 11, Undefined identifier.
0030 | Error No. 1, Syntax does not make sense
```

Versions of Scriptor ending in XS or XSC use a slightly different error presentation in which the Error No. is replaced with a description and identification of the actual error in the line within brackets {}:

```
Program errors are listed below:
E/W | line number, error, {bad code}
E | 0012 | Undefined identifier. { hu1 }
E | 0013 | Undefined identifier. { vall2 }
W | 0014 | Converting between types causes a possible loss of precision,
which can lead to unexpected results. { 1\3 }
```

Note that the XS and XSC versions also provide the option to print out warnings, situations where the compiler has identified either unsafe or poor programming practices which can lead to errors.

In versions 3.5.0 and beyond, the line numbers are automatically added to the program, and those with errors (not warnings) are marked in red. Errors must be corrected before the program will run. Warnings should be examined but most can be ignored. If you are instantiating objects from the Objects Panel, these objects are added at the top of the script to be compiled. Now you must select Display Script on Run from the Debug Menu if you are to track down which line is involved in generating the error. Your program will be replaced with a line-numbered version that includes all of the expanded source code. The resulting code can be quite large but you should have no problems tracking down the error. But do not make changes in this expanded version of your code. Identify the location of the error and then select Restore Program after Script Display to return the program for modification. If you make changes in the displayed script, the changes will be lost upon restoration. If you have numerous errors, you are advised to print out the program and mark down the error locations by reference to the displayed script. Then restore the program and make the changes by reference to your printout.

Handling Properties. Many of the common programming errors involve misspelling Scriptor methods or using an incorrect number or ordering of the calling parameters. These errors are best handled by making reference to the online help or using Appendix 1 to examine the method calls. These errors are usually easy to fix. One of the more

frustrating errors to encounter is Err(11), an undefined identifier. An identifier is a generic term that is used to represent a named property or object. The term is purposely vague because when the compiler encounters something like blimdiddy(i), and blimdiddy() is not declared or defined as a subroutine or function, the compiler is incapable of figuring out whether the programmer intended blimdiddy(i) to be a function, subroutine or array variable. If the programmer meant to define a function, for example, this error suggests the reference was not spelled correctly. But if blimdiddy() was intended to be an array, then this error suggests it has not been declared, or declared with a different spelling. At this point it is often very helpful to get a list of all the variables that have been declared in the program, or have been defined by other objects as public properties. Under the debug menu you will find two menu options:

List Public Properties and
List Main Program Properties

By selecting one of these items you generate a list of the requested properties in the output editfield. Note that when the second option is chosen, all of the properties available to the Main Program (either defined within Main and or provided by external objects) are listed. Below is a typical example.

```
a2(1, 1)        [double from line 8, redimmed in line 15]
i               [integer from line 9]
imod            [integer from line 9]
masterlength    [double = 3.0 from public(geometry)]
nbasis          [integer from line 9]
nc              [integer from line 9]
nradius         [integer from line 9]
nside           [integer from line 9]

========== global constants follow ===========

const_degree    [global double = 0.0174533...]
const_e         [global double = 2.7182818...]
const_pi        [global double = 3.1415927...]
```

The variables are listed in alphanumeric order regardless of the order in which they were declared. This helps one track down the variable in the list. Furthermore, if an array variable is redimensioned anywhere in the program, all such events are listed and the line number is shown. If there are any classes instantiated, the public variables declared are also shown. In the above example, one such variable is found, masterlength. This is a double which was defined as a public variable in an instantiated class called "geometry". It matters not that this class is relevant to the program, only that it has been instantiated. Finally, after all the user defined variables are listed, the Scriptor global constants are listed whether or not they are used. Indeed, the property listing does not check to see if any of the variables are used in the program, only if the variables are available to the program. This listing is invaluable when tracking down variables in

the program, either those that are undefined or when the compiler identifies a variable that has been declared more than once. As a reminder, although dim statements can appear anywhere in a program, one can declare a variable only once. A redim statement does not count as it only redefines the upper bounds of, and the amount of memory allocated to, a previously declared array. One can have as many Redim statements as your programming requires. Keep in mind that when an array is redimmed to smaller dimensions, the memory immediately becomes available for use by the program. If one is working on a program that makes heavy use of RAM, one should Redim arrays to minimal size whenever possible to maintain maximal available memory. However, one cannot Redim an array to a smaller size and then Redim the array back to its original size and expect previous data in the excluded region to be valid. When an array is Redimmed to larger size, Scriptor sets the new elements created to zero or to nil, depending upon data type. However, elements that are contained within a subscript region not affected by the Redim statements remain valid and do not lose values previously assigned.

1.2.1. Listing Internal Methods via Category

There is a good chance that a particular function or set of functions needed to carry out a given programming task is available in Scriptor or MathScriptor. However, there are more than 500 internal methods that are part of the extended language and there is no point in trying to memorize them. Thus, the programmer will often need help tracking down the desired function, particularly when starting a project on a new topic. The help menu includes a List of Methods and Properties menu item under which are eleven categories. Selecting the menu item generates a list of all functions relevant to that topic. The methods are collected and listed in the Main panel text output field with enough discussion to not only indicate what the functions do, but how to use them. These menu selections are often the fastest way to track down a desired method.

1.2.2. Running in Simple Mode

The ability to use classes, modules, or methods from the objects panel provides considerable flexibility and allows the user to concentrate their programming work on that portion of the program that is being created or modified. However, when one is first learning how to program, using external objects adds additional degrees of freedom that often cause confusion. The Simple Mode option under the Debug Menu constrains the compiler to only collect code from the Main Program Editfield (when in the Main Panel) or the Music Program Editfield (when in the Music Panel). This makes debugging easier, makes error identification by line number more accurate, and speeds compilation even when no external objects are instantiated. It is recommended that

students use Simple Mode during the first semester course. This mode can easily be turned off when external objects are required.

1.2.3. Organizing Declarations

A complicated program can have hundreds of user variables of different types. Each variable represents an object, and the rules of object oriented programming require that all variables be formally assigned. This rule makes it much easier for others to read and understand your program, and much easier for you, the author, to find errors or enhance the program. Variable declarations can be placed anywhere in the program the user desires, a flexibility made possible by a multipass compiler. However, this is bad form, and readability is enhanced if all the variable declarations are collected at the top of the program. Readability is further enhanced if the variables are listed in alphabetical order. To make this process simple, the Scriptor metacompiler can collect all the variables declared in the Main program, organize the variables by type, alphabetize the variables and place the declarations at the top of the program. This is accomplished by selecting the Organize Declaration in Main under the Edit Menu. The key combinations ctrl-D (or command-D on Mac OSX) also activate this menu item.

```
do
  dx =(x2-x1)/mxz
  dy =(y2-y1)/nyz
  if Qrestore then
    picture_copy_to_buffer(2)
    Qrestore=false
  else
    for i=1 to mxz
      x0=x1+i*dx
      for j=1 to nyz
        y0 = y1 + j*dy
        z(i,j)=100-mandelfast(x0,y0)
      next
      if i mod 17 = 0 then
        plot_progress(100*i/mxz)
        if check_for_user_action("user") then return
      end if
    next
    if Qoption then
      zmax=-100
      zmin=100
      for i=1 to mxz
        for j=1 to nyz
          z0=100-z(i,j)
          z(i,j)=z0
        next
      next
      for i=1 to mxz
        for j=1 to nyz
          z(i,j)=100*(z(i,j)-zmin)/(zmax-zmin)
        next
      next
      plot_2d_array(z(),mxz,nyz,0,1,false)
    else
      for i=1 to mxz
        for j=1 to nyz
          z(i,j)=100*(z(i,j)-zmin)/(zmax-zmin)
        next
      next
      plot_2d_array(z(),mxz,nyz,0,2,false)
    end if
  end if
loop  until check_for_user_action("user")
```

1.2.4. Graphing the Program Structure

The combination of loops and conditionals can often lead to structural complexities that are hard to follow, even when the programmer is the creator of the structural complexities. Although program indentation is helpful to an understanding of the structure, for high levels of structure, this tool is of limited value. A more sophisticated tool is available. A graph of the program structure like that shown at right can be generated by marking the region to be graphed using triple slashes (///) above and below the

target section. Not only does this graphics image help display the structure, it is invaluable in tracking down structural errors. If a grey line connects two statements that are not structurally related, such errors are easily spotted visually and the error fixed. Program graphing is also useful if one seeks to insert a structured overview of the program in a presentation or manuscript. Although the graphing method does not formally limit the length of the program, in practice, the program should be divided up into sections of less than 150 lines each to generate readable graphs.

1.2.5. Loading Simple and Complex Templates

While every program is different, there are a few elements that are common to the majority of programs. Very simple templates can be loaded by pressing the **Clear Prog** button at lower right. If the button is pressed by itself, the following collection of declarations are loaded. These provide a useful start for general scientific programming.

```
// Program Name: progname.txt
// by Ruffus Fourier (2008-11-28 15:33:45)
// This program ...
dim i, j, k, m, n, ns, np, nfile, icw(1) as integer
dim x1, x2, y1, y2, x(1), y(1) as double
dim s0, s1, s2, fname, fcontents, lines(1), hdrs(1) as string
dim Q, Q1, Q2 as boolean
```

If the shift key is held down while pressing the **Clear Prog** button, no declarations are provided and a minimal header is presented.

```
// Program Name: progname.txt
// by Ruffus Fourier (2008-11-28 15:33:45)
// This program ...
```

A third option is to hold down the alt (or option) key while pressing **Clear Prog**, to generate the following scientific template.

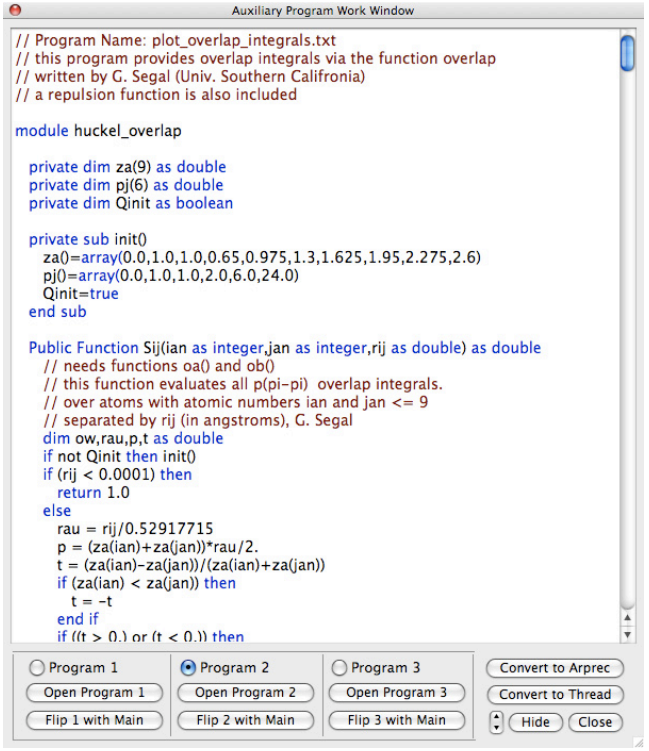
```
// Program Name: progname.txt
// by Ruffus Fourier (2008-11-28 15:39:28)
// This program ...
dim a0, a1, alpha, c0, Eh, epsilon0, G, h, hbar, k, kappa as double
dim masse, massn, massp, massu, Na, Qe, Qohm, Ryd, u0 as double
dim s0, s1 as string
// 2006 CODATA values (Rev. Mod. Phys. 80, 633-730 (2008))
masse=9.10938215E-31 // kg (mass of electron)
massp=1.672621637E-27 // kg (mass of proton)
massu = 1.88353130E-28 // kg  (mass of muon)
massn=1.674927211E-27 // kg (mass of neutron)
a0=5.291772108E-11 // m (au length = bohr radius)
Qe=1.602176487E-19 // C (charge on electron) 2008
hbar = 1.054571628E-34 // J•s (h/2pi = Dirac `s constant)
h = 6.62606896E-34 // J•s (Planck `s constant)
c0 = 299792458 // m/s (defined speed of light)
Eh = 4.35974417E-18 // J (Hartree- au of energy)
Ryd = 10973731.568527 //m^-1 (Rydberg constant)
```

```
alpha = 7.2973525376E-3 // fine structure constant
kappa = 8.9875517874E9 // 1/(4*pi*e0) N m^2/C^2 (coulombs constant)
Na = 6.02214179E23 // mol^-1 (Avogadros number)
G = 6.67428E-11 // m^3/(kg s^2) (gravitational constant)
k = 1.3806504E-23 //  J/K (Boltzmann constant)
Qohm = 12906.4037787 // ohm (h/(2*e^2))
u0 = 4*const_pi*1.0E-7 // N/A^2  [magnetic constant (exact)]
epsilon0 = 1/(c0^2*u0) // F/m (electric constant)
```

The physical constants selected are relevant to many of the calculations that students encounter in undergraduate science courses or research. The programmer can easily add additional physical constants or variables, and remove those that are unneeded. The user also has the option of selecting one of the more substantive templates which are stored in the Templates Folder. And it is important to remember that new templates can be added at any time by simply naming the program with the header "template_". The save process will then preferentially place this program in the Templates folder and the program will be available by pressing the templates button.

1.2.6. Auxiliary Program Work Window

A new program may require the incorporation of sections from one or more other programs that were saved previously in the program folder. Under these circumstances, it is convenient to have the source programs open simultaneously to facilitate copying and pasting. Method and code transfers like this are easily handled by using the Auxiliary Program Work Window which is opened by selecting "Open Program Work Window" under the File Menu. An example instance of this window is shown at right. As many as three other programs can be opened and placed in the three slots provided by this window. The individual programs are loaded into the window buffers by pressing the **Open Program 1**, **Open Program 2** or **Open Program 3** buttons. Only one program is presented in the work window at a time, and the active program is selected using the radio buttons directly above the Open Program buttons. One can copy and paste between this window



and the Program Editfield in either direction. One can also flip any of the three programs with the Main Program by pressing the appropriate button. As of this writing, the "Convert to Arprec" and "Convert to Thread" buttons have not yet been implemented fully. Feel free to experiment if interested, but do not expect competence. This window is designed to always float in front of the Main Window, but can be hidden by pressing the **Hide** button. The window can then be returned by pressing the text "click here to restore work window" underneath the slider at lower right on the Main panel. The window is closed permanently by pressing the **Close** button.

1.2.7. The Debug and Optimization Window

After a program has been debugged and is working as desired, it is important to do a final check and optimization process. This process is facilitated by transferring the program to the Debug and Optimization Window which is accessed under the Debug menu and selecting Open Debug Window (ctrl-W) or (command-W on Mac). This window provides a systematic analysis of the variables used in your program and allows one to remove any declarations to variables that are not used in the program. The process must be done in a particular order as follows:

1. **List Errors:** This button generates a list of errors. Any errors must be fixed before proceeding to the variable optimizations listed below.

2. **Expand Declarations:** This button expands the declarations to one line per variable. This operation is necessary before pressing "Remove All Unused Variables".

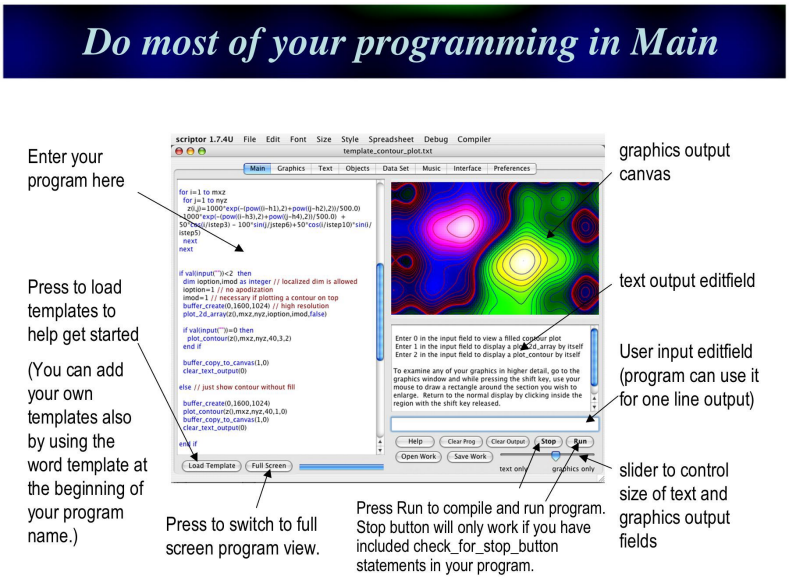
3. **Remove All Unused Variables:** After you Expand Declarations this button will remove all variables in the program that are unused. The variables removed will be listed in this window.

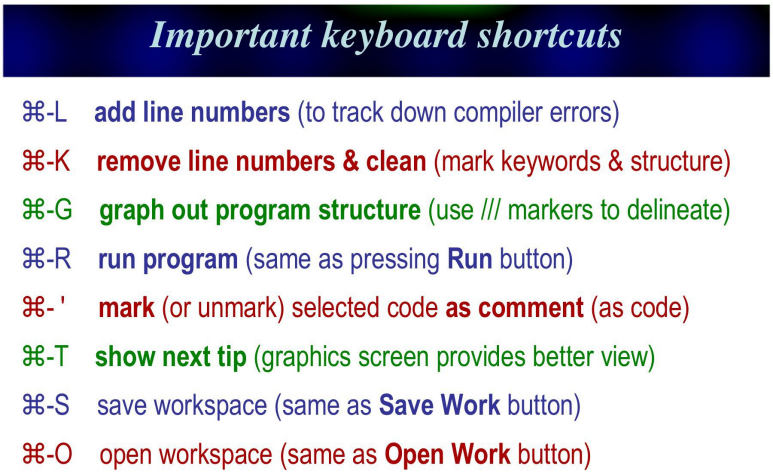
4. **Organize Declarations:** This button collects and alphabetizes all of the variable declarations into a minimum number of statements.

The XS and XSC versions of Scriptor will also provide a button for listing compiler warnings. Most of these warnings can be ignored, but if one plans to distribute a program to other users, each warning should be considered seriously.

1.2.8. Tips

Sometimes guidance in programming or using the programming environment is most efficiently presented in graphical form. In Scriptor, graphical help files are called tips and can be accessed under the Help menu by selecting either "Show Next Tip" or "Show Previous Tip". The tips are presented in the graphics canvases in both the Main and Graphics Panels. Some selected tips are shown below.





Exit statements exit the current loop

```
n1=0
for i=1 to 10
   for j=1 to 10
      n1=n1+1
      if j>2 then
         exit
      end if
   next
next
print("n1 = "+str(n1))
```

```
n1=0
for i=1 to 10
   for j=1 to 10
      n1=n1+1
      if j>2 then exit
   next
next
print("n1 = "+str(n1))
```

The exit statement transfers execution to the first statement after the current loop's next statement. And the exit statement is only executed once, and next time the loop is entered, will need to be executed again if the loop is to be exited again.

Exit Statements Work in all Loop Types

```
n1=0
i=1
do
   j=0
   while j<10
      j=j+1
      n1=n1+1
      if j>2 then exit
   wend
   i=i+1
loop until i=10
print("n1 = "+str(n1))
```

The exit statement operates the same way regardless of loop type.

When the exit statement is encountered,

execution is transferred to the first statement after

the closing statement (next, wend, loop) of the current loop.

Functions & Subroutines without parameters still need ()

```
sub tad1()
   // prints out date and time
   update_time
   print(string_time_and_date)
end sub

function tad2() as string
   // returns date and time
   update_time
   return string_time_and_date
end function

// main program
tad1()
print(tad2())
```

You still need to include parentheses in both the

*definitions **and** calling statements*

for functions and subroutines without parameters.

Functions can have multiple returns

```
function autofix(s0 as string) as string
  // this function checks for common
  // misspelling and returns correction
  select case s0
  case "modelling"
    return "modeling"
  case "catagory"
    return "category"
  case "seperate"
    return "separate"
  else
    return s0
  end select
end function
```

A function can have as many return statements as desired. Each one, however, must return a value of the correct type as defined in the function header.

Subroutines can have multiple returns

```
sub autofix(byref s0 as string)
  // this function checks for
  // common misspellings and
  // returns the corrected
  // value ByRef in s0
  select case s0
  case "modelling"
    s0= "modeling"
    return
  case "catagory"
    s0= "category"
    return
  case "seperate"
    s0= "separate"
    return
  end select
end sub
```

A subroutine can have as many return statements as desired. However, know that in most cases, you need no returns at all.

Return (in Main) Exits the Program

```
n1=0
for i=1 to 10
  for j=1 to 10
    n1=n1+1
    if j>2 then return ──────► exits program!
  next
next
print("n1 = "+str(n1))
```

The return statement is special. If you are in a subroutine or function, you return to the program that called the method.

But if you are in the main program, return means return to Scriptor. The program stops and control is returned to you.

ByVal versus ByRef

If passed by value (ByVal), the value of the variable is copied into the local variable. If the value of the variable is modified inside the subroutine, the calling variable remains unaltered. In contrast, if a variable is passed by reference (ByRef), the memory location is passed and upon exit, if a change in the value has occurred within the function or subroutine, the change is retained by the variable upon exit. All parameters default to ByVal except for arrays, which are always passed ByRef. Accordingly, you normally only need to indicate ByRef as in the following example. Below, a2(,) and bb are passed ByRef.

```
Sub sub_name(aa as double, ByRef bb as double,
             ic as integer, a2(,) as double)
```

Method Overloading and Polymorphism

Functions and subroutines can be overloaded, which allows two or more methods to be defined with the same name but different numbers or types of parameters. This is an example of polymorphism, an important OOP capability.

```
function max1(a as double, b as double) as double
  return max(a,b)
end function

function max1(s1 as string, s2 as string) as double
  return max(value(s1),value(s2))
end function

function max1(a as double, b as double, c as double) as double
  return max(a, max(b,c))
end function
```

The New SQL Date Statements

Date_seconds_to_SQL(total_seconds) **as string** *returns the SQL date and time as a function of the number of seconds that have elapsed since January 1, 1904. SQL format is YYYY-MM-DD HH:MM:SS*

Date_SQL_now as string *returns the SQL date in the format YYYY-MM-DD HH:MM:SS for the date and time at which the function was called.*

Date_SQL_to_seconds(SQL_string) **as double** *returns the number of seconds that have elapsed since 1904-01-01 00:00:00. A date and time prior to 1904-01-01 00:00:00 will yield a negative value. Dates prior to Jan 1, 1601 are assigned based on the modern calendar and may be different from historical record. The SQL date is a string in the format YYYY-MM-DD HH:MM:SS*

Date_to_seconds(string_date) **as double** *returns the number of seconds that have elapsed since January 1, 1904. A date prior to Jan 1, 1904 will yield a negative value. Dates prior to Jan 1, 1601 are assigned based on the modern calendar.*

Chapter 2
Introduction to Scriptor Extended Basic

The extended basic language available in Scriptor is similar to the languages of Visual Basic and Xojo. Scriptor is based on XojoScript, a compiler provided by the Xojo Extended Basic language, the parent language that was used to create Scriptor and MathScriptor. The capabilities have been enhanced by including roughly 500 additional math and graphics functions, and some of these functions are only available in MathScriptor. A registered user can switch between Scriptor and MathScriptor under the Compile menu. This chapter provides an overview of the language in enough detail to allow the student to understand the programs that are introduced and just enough to allow them to modify the programs to suit their requirements. Additional resources are provided in appendices. The most important is Appendix 1, which provides a list of all the functions that are available as well as the calling procedures. Appendix 1 also provides a more detailed discussion of the language.

Introduction. The Anatomy of a Program

We start by examining a simple program that calculates the Fibonacci series, made famous by the book "The Da Vinci Code". This series could easily be calculated by hand, but serves as a useful target for a programming example. The program is shown below and is annotated with comments to provide a brief introduction into the key statements. It is important to include numerous comments throughout your program, not only for others, but for yourself. Although Extended Basic is a high level language that has a syntax designed to enhance clarity of function, comments are still important to provide a perspective on what the code is doing. We will use this program to introduce the key components. You can find a copy of this program in the book_examples folder inside the Programs folder. Here we examine the elements of this program in detail.

```
// Program Name: sample_program_2_1.txt
' This is a sample program that illustrates the main components of a program.  The first line of each
program should include the program name using the format shown above.  Use underlines to separate
words and finish with the extension .txt  This name will be used by the program as the filename.  The next
section is the declaration section which assigns the names of the variables that will be used and their data
types. All variables must be declared prior to usage.  Comments begin with // or ' and should be added
throughout your program to explain what you are doing.  Comments can be added at the end of a line as
well.
dim s0,s1,s2 as string
dim i,j,k,nf0,nf1,nf2 as integer
dim a0,a1,phi as double

' In the 12th century, Leonardo Fibonacci discovered a simple numerical series that bears his name.
Starting with 0 and 1, each new number in the series is simply the sum of the two before it.  The ratio of
each successive pair of numbers in the series approximates the number known as phi(1.618. . .).  This
program calculates the first 40 numbers in the series and then calculates the value of phi at the end.  The
first 20 terms are printed out.
set_graphics_slider(0) // set the output to text only
clear_text_output(0) // clear the text output
'The following statements are called assignment statements.  The equals sign is used in a special way in
an assignment statement.  It means place the number on the right-hand-side of the equal sign into the
variable on the left-hand-side.  After the line is executed, the two values are equal.
nf0=0
nf1=1
'Initialize the string variable s0 to hold the first part of the series.  Note that the + symbol is used to add
one string to another, but that this does not involve math but simply appending the value to the right of the
+ symbol to the string on the left of the + symbol.
s0=" The first 20 numbers in the Fibonacci series are: "
s0=s0+str(nf0)+", "+str(nf1)
'The following statement starts a for..next loop which allows for repetitive looping.  In the present case,
this statement requests a looping using the counter variable i with values of 1 to 40 in steps of 1.  The end
of the loop is indicated by using the "next" statement.  This statement increments the counter by the step
size and if greater than the end value(40 here), the loop is exited.
for i=1 to 40 step 1
   nf2=nf1+nf0
   'The following is a conditional statement that checks to see if the counter variable, i, is less than 21.  If
   so, the next value in the series is added to the string variable which will ultimately be printed out.
   if i<21 then s0 = s0 + ", "+str(nf2)
   nf0=nf1
   nf1=nf2
next
print(s0)
print("-----------------")
phi=nf1/nf0
print(" after "+str(i-1)+" terms, phi="+format(phi,19,16))
'Each program should end with the following statement.  This informs the compiler that anything below
this statement should be ignored.
// end program
```

2.1. Initialization

The first part of the program assigns memory to variables, initializes the variables and initializes the environment.

2.1.1. Program Identification Line

Although it is optional, using the following syntax for the first line of the program will automatically assign the file name when you save the program.

```
// Program Name: my_program_name.txt
```

The next few lines should also be comments lines. These lines should provide the name of the programmer and a brief discussion of what the program does and how to use it.

2.1.2. Declare Variables by Type

Computer programs use named variables for temporary storage of numbers or characters. The names are for the convenience of the programmers and can be as many as 64 characters in length and include upper and lowercase letters and the underline character. For example, the program

```
dim this_is_a_ridiculously_long_variable_name_but_it_is_legal as double
this_is_a_ridiculously_long_variable_name_but_it_is_legal = sqrt(3)
print(str(this_is_a_ridiculously_long_variable_name_but_it_is_legal))
```

will print out `1.732051`. Although some languages such as Fortran 77 automatically declare variables as they are encountered in the program, this automatic process is one of the main sources of programming error in such languages and is no longer considered good practice. Scriptor requires that all variables be declared prior to usage, and this is done using the `dim` statement. The `dim` statement can declare more than one variable of the same type. For example, `dim s0, s1, s2 as string`, declares three variables s0, s1 and s2 as string variables. These are variables that hold text, and the text can be of any length. There are many other data types. The most common types are listed below:

Integer (32-bit whole numbers in the range ± 2, 147, 483, 648)
Int64 (64-bit whole numbers in the range ± 9, 223, 372, 036, 854, 775, 807)
Single (32-bit positive or negative 7 digit real values between
1.175494×10^{-38} and $3.402823 \times 10^{+38}$)
Double (64-bit positive or negative 15-16 digit real values between
$2.2250738585072013 \times 10^{-308}$ and $1.7976931348623157 \times 10^{+308}$)
Currency (64-bit fixed point variable with 15 digits to the left of the decimal point and 3 digits to the right of the decimal- used for business and accounting accuracy)
Boolean (1 bit: true, false)
String (ASCII characters of arbitrary length – see Appendix 3)
Color (32-bit color specification, e.g. RGB(255, 255, 255))
Const (any variable type, but once defined, cannot be changed via assignment in the program)

Upper and lowercase letters are treated the same which is a very important difference between Scriptor and Mathematica, which uses a case sensitive syntax. Most object-oriented languages are case insensitive for enhanced readability and ease of debugging. For example, the variable apple and Apple are different variables in Mathematica, and that is a perfect recipe for introducing programming errors.

2.1.3. Declare and assign variables as constants

It is sometimes useful to use constants. These are variables that once assigned, cannot be reassigned by the program. For example, the value of pi is not going to change as any point in the life of your program, so it is appropriate to assign it as type Const.

Const Pi=3.14159265358979323846264338327 95

Now when this variable is used, you can be confident that it will always have the correct value. Scriptor provides the const_pi and other selected constants as part of the language (see Appendix 1).

2.1.4. Assign values to variables via default or during declaration

Declaring a variable in a dimension statement assigns a default value to that variable. The default value for all numerical variables is zero. The default value for all string variables is the null string = “”. The default value for color is black = RGB(0, 0, 0). It is not considered good practice to rely on these defaults, but it is important to know that these assignments are made. You can assign a value to a variable during declaration, as demonstrated in the examples below:

```
dim red as color = rgb(100, 0, 0)
dim months_in_a_year as integer = 12
dim x as double = 32.5
```

2.1.5. Assign values using the "=" sign

The value of any standard variable can be changed by using an assignment statement, for example:

x = 19.7

First time programmers are often confused by the use of an equals sign to assign a variable. This confusion is understandable because the equals sign serves a dual purpose in programming, and the most common usage is not mathematical but variable assignment.

The equals sign initiates the following two step process:

(1) evaluate the expression to the right of "="

(2) transfer the result of the evaluation to the memory location associated with the variable to the left of the "="

The equals sign is an appropriate symbol to use in the sense that after the operation is finished, the left and the right hand sides do indeed have equal values or when evaluated, yield identical results.

When it is necessary to assign large values, use either the common "E" symbol to indicate exponent, or use the Mathematica-like statement, 10^ :

```
a = 1.234E128
a = 1.234*10^128
```

Both statements assign the value 1.234 x 10^{128} to the variable a.

2.1.6. Initialize the Output Environment

The next step is unique to the Tabbed Panel Environment of Scriptor. We now set up the environment so that our program will have access to the appropriate output environment. If we are only interested in printing out numbers, the graphics canvas in Main is of no use so we replace it with a pure text editfield. We can do this with the statement set_graphics_slider(0) which means set the graphics slider to 0% size. The

default when the program is run is a 50:50 space allocation with graphics and the text output field set to equal size. The next step is to clear the text output which is done using the statement clear_text_output(0). The zero in parentheses in this case means clear both the editfields in both Main and Text panels. Remember, if you forget what the parameters of any internal function mean, all you need to do is type in the function or back into the name of the function and if you have the "analyze keywords as you type" preference set, a description of the function and its parameters will appear in the editfield on the right-hand-side.

2.1.7. Arrays, Vectors and Matrices

We close this section by introducing some new data types that are particularly useful for numerical methods. Arrays are variables that allow for the storage of multiple numbers or strings by using a single variable name and subscripts to access the elements. Arrays can use any data type and can by dynamically redimensioned in the program by using the Redim statement.

Arrays are assigned by type in the dimension statement, for example:

```
dim a1(20), a2(10, 20) as double
```

creates a one-dimensional array a1 with 21 elements and a two-dimensional array a2 with 231 (11 by 21) elements. The reason there are more elements than the product of the dimensions is that all the arrays have zeroth elements. That is, if the array a2 is dimension a2(1, 1) it has four elements: a2(0, 0), a2(0, 1), a2(1, 0) and a2(1, 1). You may choose to ignore these elements, but they exist and occupy memory.

It is often useful to manipulate the size of an array based on conditions that can only be determined during the execution of the program. To handle these situations, one can declare nil arrays by using the following syntax:

```
dim a1(-1), a2(-1, -1) as double
```

and then assign the size during runtime using the redim statement:

```
redim a1(8)
redim a2(3, 4)
```

The rule is that a single redim statement is required for each array, that the variable type of the array must have been declared in a prior dim statement, and that one cannot use the redim statement to alter the variable type (i.e. redim a1(8) as integer is not allowed). In contrast, redim statements involving the same variable are allowed, so one can create

a nil array, redimension it to hold all the necessary data, and then redim the array back to nil when the calculation is done using the following statement:

```
redim a2(-1, -1) // recover all memory allocated to a2
```

The capabilities inherent in the redim statement are significant. This means that a program can allocate memory to an array variable and then recover the memory under program control. There was a time in the history of computing when such capabilities were unavailable to the programmer and thus very fancy and complex methods were used to have variables share memory (e.g. use of common within Fortran IV). The fact that Scriptor provides this capability is a tribute to the sophistication of modern languages (and Xojo Inc. which wrote the compiler).

We note that the term array is used to describe both one and multidimensional arrays. Indeed, one can create an array with as many as 16 dimensions if desired. And the current size of an array can be determined by using the function ubound(array_name, dimension) to discover the current size:

```
dim a12(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12) as double
dim i as integer
i=ubound(a12, 8) // find the current size of the 8th element
print(str(i))  --------------> 8
```

One needs to take care when working with multidimensional arrays, as they require a great deal of memory. For example, the above 12-dimensional array has 13! = 6, 227, 020, 800 elements each requiring 8 bytes of memory, which totals roughly 50GB (gigabytes). It is unlikely that you computer has 50GB of RAM. Modern operating systems, however, may allow one to run this program by paging memory to the disc drive.

Quite apart from the large memories required to utilize multidimensional arrays, access to any arrays larger than two dimensions is slow. The compiler optimizes access to one and two dimensional arrays, but arrays using more than two dimensions are accessed relatively slowly compared to other variables. Limit your programs to one or two-dimensional arrays whenever possible if speed is an issue.

MathScriptor provides a large number of functions and subroutines that work on one and two-dimensional arrays. These functions will play an important role in many of the numerical methods to be discussed in chapter 4.

2.2. Math

We now have our variables declared and initialized and our output environment set up for the task at hand. The next step is to do the math and display or print the results. First we examine how we can do math with Scriptor.

There are four discrete types of math capabilities in Scriptor and MathScriptor. The first type are the standard math functions available in almost all computer languages. These are listed below:

Operation Performed	Operator	Example
Addition	+	2 + 3 = 5
Subtraction	-	3 - 2 = 1
Multiplication	*	3 * 2 = 6
Floating Point Division	/	6 / 4 = 1.5
Integer Division	\	6 \ 4 = 1
Modulo	Mod	6 Mod 3 = 0 6 Mod 4 = 2
Exponentiation	^	2^3 = 8

Exponentiation can be also be done using the C-type operator pow(a, b), which is equivalent to a^b. There are hundreds of additional math functions built in. For example, the log and exponential functions are as follows:

exp(x) = e^x where e is the irrational constant 2.718281828459
log(x) = natural logarithm of x = ln(x)
log10(x) = logarithm of x to base ten.

All of the standard trigonometric functions are available

sin(x), asin(x), cos(x), acos(x), tan(x), atan(x)

and these functions operate by default on radians. Thus, if you want to use arguments in degrees, you must multiply the argument by the number of radians per degree which equals 2 π divided by 360 = 0.0174533. A full precision representation of this number is included as one of the intrinsic constants and is called const_degree. Hence, if you seek to assign the variable b to equal cos(45°) then use

```
b = cos(45*const_degree)
```

The following hyperbolic trigonometric functions are also available:

```
sinh(x), asinh(x), cosh(x), acosh(x), tanh(x), atanh(x)
```

These functions are invaluable for working with differential equations, and using Laplace's equation in Cartesian coordinates in electromagnetic theory, fluid dynamics, heat transfer and relativistic quantum mechanics. These functions have interesting and useful relationships with the other trigonometric functions and complex numbers (see Appendix 4, Equations A4.1.13 – A4.1.24).

All of the above functions are not only available for use with singles and doubles, the standard data types for math, but also with string representations of numbers. For example,

```
s0 = "3.14159"
s1 = cos(s0)
print(s1) --------> -0.9999...
```

is acceptable syntax. (The long arrow indicates the printed output and is not part of the expression.) The reason that strings are also accepted is to provide for complex arithmetic as well as arbitrary precision arithmetic, the latter requiring Scriptor. In Scriptor, a complex number is represented by using a string and separating the real and complex components by a comma:

```
s0 = "2, 2"  // 2 + 2*I
s1 = cosh(s0)
print(s1) --------> -1.56563 , 3.29789
```

2.2.1. Ordering of Math Operations

Scriptor obeys the standard rules of math operator precedence. Calculations are carried out in the following order (the lower numbered items are done first):

1. **Inside Parentheses**
2. **Exponentiation**
3. **Multiplication and Division**
4. **Addition and Subtraction**
5. **Left to Right**
6. **Comparison Operators (Left to Right)**
7. **NOT (right to left)**
8. **AND (left to right)**
9. **OR (left to right)**

Note that parentheses take highest priority, and thus the user is in complete control of how the math and logical operations will be carried out. Put another way, parentheses can override operator precedence and force addition to be carried out before exponentiation as shown in the following example:

```
 2 + 3^2 + 4      --------> 2 + 9 + 4 = 15
(2 + 3)^(2 + 4)   --------> 5^6 = 15625
```

Programmers quickly pick up the concepts of math precedence, but often get confused when writing logical operations which often seem to give results that are contrary to intuition. We examine logical operations in the next section.

2.3. Logical Expressions

A Boolean (or logical) statement is one which evaluates to either true or false, and involves the use of both comparison and logical operators, and can include the Boolean variables True or False. The comparison operators are listed below:

Operator	Meaning	Evaluates to true if… (otherwise evaluates false)
<	less than	LHS is less than RHS
<=	less than or equal to	LHS is less than or equal to the RHS
>	greater than	LHS is greater than the RHS
>=	greater than or equal to	LHS is greater than or equal to the RHS
=	equal	LHS and RHS are equal
<>	not equal	LHS and RHS are not equal

where LHS means a number or expression on the Left-Hand-Side of the operator and RHS means a number or expression on the Right-Hand-Side. For now we will limit our discussion to math expressions, but these operators also work on strings (see below).

There are three additional operators that are available which when combined with the comparison operators allow for complete flexibility in conditional analysis. The additional operators are: **AND, OR** and **NOT.**

These operators mean what they say. But their order is important. Consider the following:

```
2<3 or 3<2 and 4<3 or 3<2
```

Try and evaluate this and see what you get. Now lets do it systematically. The comparison operators are evaluated first and so the above expression can be rewritten as:

```
true or false and false or false
```

The AND operator takes precedence over OR so it is evaluated next. false and false

```
true or (false and false) or false  ----->  true or false or false
```

The rest is evaluated from left to right so we get

```
(true or false) or false  ----->  true or false  ------> true
```

If your initial evaluation of this expression came out false, join the crowd. Our language sense drives us to see the AND operator as one which should be evaluated at the end, not the beginning. Thus, most of us would parse the above problem to have the following operational ordering:

```
(2<3 or 3<2) and (4<3 or 3<2)  ------> false
```

which is incorrect! The above example demonstrates why it is strongly recommended that parentheses be used in logical operations even when they are not needed. Parentheses are also recommend when writing complicated math expressions. In both cases, parentheses help the programmer observe the expression with greater clarity and avoid making mistakes.

A simple way to check for the value of an equality is the following three line program:

```
dim Q as boolean
Q=2<3 or 3<2 and 4<3 or 3<2
print(convert_to_string(Q))
```

The function, convert_to_string, works on any variable type and converts it to a string that can be printed. In the present case, when the program is run, the output is “true”.

There is an additional operator available, **XOR**, which stands for exclusive or. This operator returns true if the LHS and RHS expressions evaluate to different states. A summary of all of the operators is shown below:

Q1	Q2	Q1 XOR Q2	Q1 OR Q2	Q1 AND Q2
true	true	false	true	true
true	false	true	true	false
false	true	true	true	false
false	false	false	false	false

2.4. Conditionals

One of the most important capabilities available to a programmer is the use of conditionals to mediate program flow. Conditionals are statements that test for the state of a logical expression and then select a section of code based on the state of the logical expression.

2.4.1. The If Statement

The If statement coupled with the Else or Elseif statements is the most commonly used conditional and exists in one form, or another, in all high level languages. The one-liner form of this statement is,

If testcondition1 **Then** statement1 **Else** statement2

where statement1 is executed only if testcondition1 evaluates to true and statement2 is executed only if testcondition1 evaluates to false. The multiline approach is often easier to read and provides for additional options.

```
if testcondition1 then
  statements in here are executed if testcondition1 is true
Elseif testcondition2 then
  statements in here are executed if testcondition2 is true
  and testcondition1 was false when evaluated
Else
  these statements are only executed if all previous test conditions were false
End if
```

The testconditions listed in the above examples are expressions that evaluate to yield a Boolean (true or false) as discuss in the previous section. They can be simple or very complicated as the case requires. **If** statements can be nested to whatever level your creativity or computer memory allow. Although only one ElseIf statement is shown in the above example, you can have as many ElseIf sections as desired. It is important to keep in mind that once any of the test condition has evaluated to be true, the corresponding code is executed and the If statement exits to the line following the End If.

2.4.2. The Select Case Statement

A high level conditional is also provided by the Select Case statement, which is valuable in providing a highly readable but somewhat slower version of the multiline If statement examined above.

Select Case testexpression
Case testvalue1
statements in here are evaluated if testvalue1=testexpression is true
Case testvalue2
statements in here are evaluated if testvalue2=testexpression is true
Case testvalue3, testvalue4, testvalue5
statements in here are evaluated if any equal the testexpression
Case first_value **To** second_value
statements in here are evaluated if testexpression is in the range specified
Case is testvalue2
statements in here are evaluated if testvalue2 has an inequality
relative to testexpression that is true (i.e. <, <=, >, >= as specified)
Else
statements in here are evaluated if none of the above tests were true
End Select

The advantage of the Select Case statement is readability and convenience. Some programmers avoid it because it is a high level construct that slows down program execution. But the compiler does as much optimization of the conditionals as is possible, and one should not avoid this statement if it makes the code easier to read. There are instances when the Select Case statement makes code significantly easier to follow and requires fewer statements.

2.5. Looping

Virtually all programs have sections of code that are repeated multiple times. Scriptor provides three types of loops, using the combinations: For…Next, Do…Loop and While…Wend. The most useful of these options is the For…Next loop.

```
For i = istart to iend step idelta
  if testcondition then exit
Next
```

The above loop uses a loop variable i which starts out with a value of istart and at each next statement is increased by a value idelta. The loop is repeated if the loop variable i, after being incremented, is less than or equal to iend. Thus the Next statement does two jobs. First it increments the loop variable by idelta. Then it checks to see if the variable is less than or equal to iend. If so, it loops back to the top and the loop contents are executed again.

The loop can contain a conditional statement that tests for some condition and if the condition is true, exits the loop. Loops and conditionals are the heart of programming and combine to provide enormous flexibility and power. An exit statement inside a loop exits the current loop only. So if an inner loop encounters an exit statement, the statement directly following the last statement in the loop is executed. The code segment at right illustrates the process. The value of k when printed out is 100.

```
k=0
for i = 1 to 40
  for j= 1 to 20
    k=k+1
    if k>100 then exit
  next
  k=k-1
next
print(str(k))
```

You may have noted that the example program that we are studying and the snippet of code at right includes lines that delineate the program structure. These program graphs can be generated by the user by selecting one of the three options at the bottom of the edit menu. These graphs are particularly useful when analyzing a program with complex structure. It is a resource that should be used regularly during the learning process.

The loops shown above count up. But you can also count down as shown in the following example.

```
For i = istart downto iend step idelta
  if testcondition then exit
Next
```

If you want to decrement, you must use the downto statement. The step idelta is optional and if the step parameter is not present, an increment of +1 (or –1 with downto)

is assumed. The For..Next loop can also operate on floating point numbers where rdelta can be any positive real number. The if testcondition then exit statement is available in all of the Scriptor loops, and can be used to exit the loop whenever the testcondition evaluates as true.

```
For r = rstart to rend step rdelta
Next

For r = rstart downto rend step rdelta
Next
```

The following loop types are more useful in situations where a test is to be carried out during the looping process and when the test condition has been satisfied, the loop is excited. When using the do loop, you have the option of testing before, during or after the loop has been executed:

```
Do Until testcondition
Loop

Do
Loop Until testcondition

Do
  if testcondition then exit
Loop

Do Until testcondition1
  if testcondition2 then exit
Loop Until testcondition3
```

The above examples represent the most common usage, but it is valid and sometimes necessary to include conditional tests before, during and after the loop is executed as shown in the fourth example. Each testcondition can be different. The While…Wend statement provides an addition looping option that provides no additional flexibility but has the modest advantage of providing a more natural resonance with the English language.

```
While testcondition
Wend

While testcondition1
  if testcondition2 then exit
Wend
```

Here, the test condition is only available at the beginning of the loop but one does have the option of exiting at any point in the loop based on the conditional exit statement, as shown in the second example.

2.6. Input and Output

A program is of little use if you cant enter data and get the results back in a permanent form. Scriptor provides a significant number of statements that handle I/O and the following short list provides a minimal overview.

Clear_Text_Output(ioption) Clears all of the text in 0(both), 1(Main Panel), 2(Text Panel)

Input(Prompt **as string**) **as string** Retrieves input from the user with an optional prompt

Print(s0) Sends string s0 to both the Main and Text panel output windows. If running in the Music Panel, this statement also sends output to the Music Output text buffer.

Format(number, string_format) **as string** Formats number using rules discussed below

Format(number, n, m) **as string** Formats number into n digits with m to the right of the decimal point. Uses exponential format if the number if too large.

Show_progress_bar(ip) Displays the progress bar for ip=0(start) to ip=100(finished).

Show_progress_line(s0 [, fontname, fontsize]) Displays the string s0 in the input line using the default fontname and fontsize, but the user can override the defaults by explicitly specifying both the fontname and fontsize.

String_speak(Text, Qnow) Speaks the Text and if Qnow is true, immediately interrupts.

Convert_to_string(any_variable) **as string** Converts single, double, integer, boolean, or colors into their full resolution string representation for printing.

There are also a number of statements that graph data which are discussed later.

2.6.1. The Spreadsheet

One of the most useful components of the Scriptor TIDE is the spreadsheet, which is available by selecting the Data tab. Entering data into the spreadsheet can be done by hand, by importing an Excel spreadsheet in comma or tab delimited format, or under program control. All information in the spreadsheet is in the form of strings. If a cell contains a number, it is the string representation of that number, not the actual number.

The process of entering data by hand is done on a cell-by-cell basis. Select the cell you wish to modify and it will change to an editfield into which the new datum can be

entered. It is common for the spreadsheet to have cells showing that are outside the active space. If you click on those cells, no editfield will appear. You can enlarge the spreadsheet using the buttons at the bottom.

Creating a new spreadsheet is done under program control. For example, the following program creates a new spreadsheet with 20 rows and 10 columns with numbered headers.

```
dim hdrs(-1) as string
dim icw(-1), nrows, ncols as integer
nrows=20
ncols=10
redim icw(ncols)
redim hdrs(ncols)
for i=1 to 10
  icw(i)=80
  hdrs(i)=str(i)
next
spreadsheet_create(20, 10, hdrs(), icw(), 2)
```

The advantage of using a spreadsheet to handle data is the ease of viewing and manipulation of the data. The data panel provides a true scientific spreadsheet, so it is not possible to sort the data "improperly" so that rows are accidentally excluded from the sort, a problem that is often encountered during a limited sort within Excel. You can manipulate the data in any fashion that is desired using a program, however.

Program access to the spreadsheet is handled by using the spreadsheet_cell(irow, jcol) function which sets or returns the contents of the cell. For example, you can fill each cell with its row and column designation by using the following code:

```
for i=1 to nrows
  for j=1 to ncols
    spreadsheet_cell(i, j)=str(i)+", "+str(j)
  next
next
```

The spreadsheet creation and fill programs can be found in the book_samples folder as sample_program_2_6.txt. It is important to restate that each cell contains a string. The size of the string is limited only by memory. If the string does not fit, Scriptor will try to compress the string to make it visible. However, after a certain point, compression fails and the left-hand side is shown followed by an ellipsis ().

The contents of the spreadsheet can be saved to a file either from the data set panel (**save** button) or under program control using save_spreadsheet(file_name).

Alternatively, a data set can be loaded into the spreadsheet from the data set panel (open button) or under program control using open_user_data_file(ifile_number, filename). The syntax for these statements can be found in Appendix 1.

One source of confusion when using the spreadsheet is the size of a number that can be stored in an individual cell. Users often worry that a 16-digit number cannot possible fit into a cell without making the cell very wide. The visualization of the spreadsheet within the data set panel is separate from the data, which is stored in a two-dimensional string array that can handle strings of any size constrained only by the memory available. The maximum number of columns is 64. The maximum number of rows is limited only by memory, but it is recommended that this number be kept below 68, 000 to provide adequate responsivity of the data set panel vertical scroll bar. But values as large as 1, 000, 000 have been used to handle data collected from long-term experiments.

2.6.2. Formatting Output

We close this section by provide additional information on formatting numbers. The Print() statement works on strings, and hence one must convert numerical values to strings prior to printing. There are a number of formatting statements and the three most useful for individual numbers are illustrated below. The const_pi number is the value of pi to 16 significant digits. The other number, bignum= 1.2345E300, is a number with a large exponent that requires the use of exponential format.

Formatting statement	Print() Result
str(const_pi)	3.141593
convert_to_string(const_pi)	3.141592653589793
format(const_pi, "+0.0000000E")	+3.1415927E+0
format(const_pi, "-0.0000000")	3.1415927
format(const_pi, 12, 8)	3.14159265
SF1(const_pi)	3.141 592 653 589 793
str(bignum)	1.234500e+300
convert_to_string(bignum)	1.2345000000000000E+300
format(bignum, "+0.0000000E")	+1.2345000E+300
format(bignum, "-0.0000000")	?.0000000
format(bignum, 12, 8)	1.23450E+300
SF1(bignum)	1.234 500 000 000 000 e+300

The easiest number-to-string converter is str(a), but it only reports seven significant digits. In many cases, seven significant digits are sufficient, and the str() function is a good choice. If the exponent is large, str() automatically adjusts the format to maintain 6-7 significant digits. The convert_to_string function is the most comprehensive. It can take a number, color or Boolean value and convert it into a full-precision string for

printing. But by providing full precision, it often provides too many significant digits. The Format statement is the best choice in most cases. This function either takes a formatting string or uses the second and third parameters to set the total number of digits and the number of digits to the right of the decimal point. The advantage of the second formatting statement is that it automatically adjusts if it receives a number that is too large to handle. Note that the string-based formatting statement displays a ? when it encounters a number too large to display.

Finally, function **SF1**(**a1 as double**) **as string** returns a scientific number format adopted by the American Physical Society for the display of numbers with a large number of significant digits. The number is divided up into groups of three with spaces inserted to make the number more easily deciphered. A full 15 significant digits are displayed.

More details regarding the various formatting statements can be found in Appendix 1.

2.7. Methods (Functions and Subroutines)

Much of the power of MathScriptor derives from the over 500 functions and subroutines that have been added to the basic language to extend its capabilities. These functions are listed in Appendix 1, and provide functions for numerical integration, least-squares fitting, Fourier and wavelet transformations, linear algebra, singular-valued decomposition and graphing. The purpose of this section is to describe how the user can write functions and subroutines to extend capabilities.

Functions are designed to return a value, most often a single value. The structure of a function is shown below:

Function name(parameter list) **As Type**
… … …
Return value
End Function

A function must return a value (using the Return statement), and when referenced in your program, the returned value must be assigned to the appropriate variable. You can have multiple return statements, but once a return statement is encountered, the function exits and returns the value. In contrast, a subroutine returns its results by using one or more of the variables that are passed in the parameter list.

Sub name(parameter list)
… … …

End Sub

A subroutine can also have a return statement, but this statement only serves to exit the subroutine. The statement cannot have a value following it. The parameter list allows two types of variables to be passed: ByVal or ByRef. If passed by value (ByVal), the value of the variable is copied into the local variable. If the value of the variable is modified inside the subroutine, the calling variable remains unaltered. In contrast, if a variable is passed by reference (ByRef), the memory location is passed and upon exit, if a change in the value has occurred within the function or subroutine, the change is retained by the variable upon exit. All parameters default to ByVal except for arrays, which are always passed ByRef. Accordingly, you normally only need to indicate ByRef as in the following example:

Sub sub_name(aa **as double**, **ByRef** bb **as double**, ic **as integer**, a2(,) **as double**)

In this example, the variables bb and the two dimensional array a2(,) are passed ByRef while the variables aa and ic are passed ByVal. Again, it is important to remember that arrays are always passed ByRef so if you change an array element within the subroutine, that change will be preserved upon exit. If you want to pass an array Byval, you must do so programmatically by making a copy and placing the copy in the parameter list.

A function can return an entire array if desired. For example, the following is an example of a function that generates an identity matrix.

```
Function matidn2(nsize as integer) as double(, )
  dim i, j as integer
  dim a2(1, 1) as double
  redim a2(nsize, nsize)
  for i=0 to nsize
    for j=0 to nsize
      a2(i, j)=0.0
    next
    a2(i, i)=1.0
  next
  return a2() // note a2() is used, not a2(, )
end function
```

Following is a short section of code that calls this function and demonstrates two important aspects of calling functions which return arrays. First, not only are the array elements returned, including the (0, 0;0, 1;1, 0 elements even if not assigned), but the array is redimensioned to correspond to the dimension that is assigned within the function. Second, the dimension of the array is not reflected in the return statement (above) or the assignment statement in line 4 below.

```
dim a(10, 10) as double
dim n, n1 as integer
n1=4
a()=matidn2(n1) // note a() is used, not a(, )
n=ubound(a(), 1)
set_text_style("Courier", 12, rgb(0, 0, 0), false, false)
print(matrix_print(a(), n, n, n))
```

2.7.1. The Assigns Keyword

There are times when one might prefer to have one of the parameters presented to a subroutine via the equals sign. For example:

```
pixel_blend(ix, iy)=blend_color
```

where ix, iy and blend_color are the three parameters. This can be implemented using the assigns keyword as shown below.

```
sub pixel_blend(kx as integer, ky as integer, assigns c0 as color)
  // reads the color of buffer pixel
  // at kx, ky and blends the color c0
  // into that pixel by averaging the RGB values
  dim ir, ig, ib, jr, jg, jb as integer
  dim c1 as color
  c1=buffer_pixel(kx, ky)
  ir =  (c0.red+c1.red)/2
  ig =  (c0.green+c1.green)/2
  ib =  (c0.blue+c1.blue)/2
  buffer_pixel(kx, ky)=rgb(ir, ig, ib)
end sub
```

Students using this statement for the first time have a tendency to leave out the parameter which will hold the assignment, and writing something like this…

```
sub pixel_blend(kx as integer, ky as integer, assigns as color)
```

The compiler rejects this statement but does not help very much by returning the error statement "Syntax does not make sense" rather than a more relevant "You must include a variable after the keyword assigns". Unfortunately, compilers are not always good at explaining what needs to be done to fix faulty code. To more fully understand the example, think about why a variable needs to be included. The code must reflect where to put the assignment, and without a variable following the assigns keyword, there is no place to put the value.

2.7.2. Method Overloading

Scriptor allows functions and subroutines to be overloaded, which allows two or more methods to be defined with the same name but with different numbers or types of parameters. This capability provides significant flexibility in programming. The flexibility is demonstrated in the following example of a function max1, which can be called with two or three doubles or two string variable representations of numbers.

```
function max1(a as double, b as double) as double
  return max(a, b)
end function

function max1(s1 as string, s2 as string) as double
  return max(value(s1), value(s2))
end function

function max1(a as double, b as double, c as double) as double
return max(a, max(b, c))
end function
```

The above three functions do not fully explore the capabilities of method overloading. For example, one can have a version of max1 that returns a string representing the maximum value. The question that students often ask is how all of this is possible. The common assumption is that a decision is made at run-time with regard to which method is to be called, but in fact it is the compiler that determines which function is coupled to the call. And the compiler must find an unambiguous choice. If the compiler cannot make a selection, it will return Error 24:

24	Ambiguous call to overloaded method. Method overloading must be defined so that there is no ambiguity in selecting which method to call. If the number of parameters is the same, the data types must be different.

2.8. Arbitrary Precision Arithmetic

MathScriptor versions 1.8.2 and above include the capability of doing arbitrary precision (Arprec) arithmetic as well as string based complex arithmetic. The precision of Arprec arithmetic is controlled by the command **Arprec_set_precision**(idigits), where idigits is equal to the number of digits of precision in the real number, not including those digits in the exponent. An added benefit of using Arprec arithmetic is that exponents as large as ±58, 000, 000 are allowed.

Arbitrary precision functions have string parameters and return strings. These functions also work on complex numbers identified by separating the real and imaginary parts with a comma (do not include I, it is understood). The following functions are Arprec savvy: plus(s1, s2), minus(s1, s2), mult(s1, s2), div(s1, s2), real(s1), imag(s1), pow(s1, s2), log(s1), loggamma(s1), exp(s1), abs(s1), sin(s1), asin(s1), cos(s1), acos(s1), tan(s1), atan(s1), sinh(s1), asinh(s1), cosh(s1), acosh(s1), tanh(s1), atanh(s1).

Output precision can be rounded to a lower precision by using the function round_to_precision(s1, ndigits). This function returns a string which can be inserted directly into a Print statement. Alternatively, one can format Arprec strings by using the function Format(s1, nwidth, ndecimal), which also works on complex numbers where the total width of the output string equals 2*nwidth+3 for comma delimiter.

There are three comparison functions that can be used with Arprec strings. The first is the standard equals ("=") which when used in a conditional statement returns true if two Arprec strings are identical. This function, when combined with the round_to_precision function, allows identity to be established at lower precision if necessary. The two Arprec specific functions, **Q_greater_than**(s1, s2) and **Q_less_than**(s1, s2), return true if s1 is greater than, or less than, s2. These two functions will even work if s1 and s2 were calculated at different precision.

The flexibility and power provided by arbitrary precision arithmetic comes with a price. The most significant cost is in CPU time as a 32 digit Arprec multiplication takes 4550 times longer than a 16 digit precision double multiplication. The reason for this significant difference is that double precision arithmetic can take advantage of floating point hardware that is designed to manipulate double precision numbers. In contrast, all of the Arprec math must be done in software and despite use of extensive use of processor floating point arithmetic, Arprec math invariably requires thousands of processor cycles. Additional latency is associated with the use of strings to receive and return the results.

Despite the increased computation time associated with Arprec math, there are times when high precision arithmetic is needed. Salient examples include situations where

the relatively small IEEE exponent range of ±308 is inadequate for a given calculation. This limitation is often a problem in physics, chemistry and engineering calculations. Overflow or underflow problems are eliminated by switching to Arprec arithmetic. Cryptography, numerical integration, perturbation theory and Monte-Carlo methods also benefit significantly from expanded precision. It is also useful to do a sample calculation using arbitrary precision arithmetic to verify that truncation error is not a problem, and then revert to double precision after verifying that it is adequate.

2.8.1. Arprec Complex Arithmetic

Arprec complex variables are string variables that separate the real and imaginary components of a number using a comma. Thus, the complex number 3 + 2**i** will be represented as a string "3, 2". The resulting variable can be used in any Arprec function as all such functions are designed to recognize a complex number simply by the presence of a comma. Thus, sin(3 + 2**i**) = sin("3, 2") ≈ "0.531 , -3.59" or 0.531 - 3.59**i**. In section 5.3 we explore adding double precision complex arithmetic by using classes, which do not provide the same level of precision available via Arprec, but are significantly faster since all the math is done using the floating point units within the computer processor.

2.9. Modules and Classes

Modules provide a straightforward and flexible approach to providing a set of functions and subroutines that are available to other objects outside of the module, but can communicate between each other via private properties and private methods, if desired. Modules are defined using the following syntax:

```
module module_name

public and private properties are declared here with the requirement that
each variable is declared using a single line dimension statement.
Each private property is shared by all the methods within the module, but
is invisible outside of the module.  Public properties are available to all the code in your
application.  Examples include:

private dim u(10, 10) as double  // only available to code inside the module
private dim v(10, 10) as double  // only available to code inside the module
public dim w(10) as double  //  available to all code
dim pwr(10) as integer  // available to all code (default is public)
```

methods are defined next and are available outside of the module unless their name is preceded with the word private. Thus

```
private sub a1(i as integer, byref x1() as double)
// this subroutine is available to only code within the module
// any properties declared are local to the subroutine
....
end sub

public sub a2(i as integer, byref x2() as double)
// this subroutine is available to all code
// any properties declared are local to the subroutine
....
end sub

function a3(i as integer, byref x3() as double) as double
// this function is available to all code (default is public)
// any properties declared are local to the function
....
end function
```

You cannot have code outside of functions or subroutines within a module. Modules are never called by themselves but only serve as containers for properties and methods.

```
end module
```

The fact that the compiler requires that each variable be assigned in a separate (one-line) dimension statement is an inconvenience, but is a restriction that can be justified based on the significant amount of work that is required of the compiler when handling public and private variables within both modules and classes (see below). But help is available. You can collect all of your dimension statements into groups as normal and then press the clean code menu item, and the dimension statements will be expanded automatically. This saves time during the writing of your programs.

2.9.1. Classes

Classes are collections of methods that are available to objects outside of the class, but which must be "called" by using a different syntax than is used to call methods declared via modules. To use classes properly you need to learn three new programming terms, instantiation, constructors and destructors. Classes are fundamentally different than any other object, and uniquely powerful. These objects will be discussed in detail in Chapter 5, and the following overview provides a brief introduction.

Instantiation refers to the process of creating a "copy" of the class for use in your program. A variable is created to represent your class using a standard dimension statement. Lets say your class is called class1. To use this class, you would assign a variable to be of type class1 by using the statement dim variable_name as class1. Then a copy of the class is created by using the statement variable_name = **new** class1. This process is known as instantiation. Classes also need constructors.

Constructors. One or more subroutines must be added to the class with the name constructor. If two or more are present, they must have different types or numbers of parameters. Multiple methods with the same name but different parameters are called "overloaded". These methods are run when the class is instantiated. The process of instantiation uses the **new** keyword (a keyword that is sometimes called a constructor) to create a new "instance" of the class within your program. You should think of an "instance" as a copy but with properties defined by the constructor. Once created (or instantiated) you have access to the methods that have been defined by your class. The constructor is a critical part of instantiation because of the flexibility that it provides. You can use the constructors to initialize the class to behave differently, or have different properties. Because the properties that are assigned during instantiation are part of the class, each instance can define a set of properties that are remembered for the life of the class. In that way, a class can define a variable, and the methods of manipulating that variable. This is an advanced capability that will be presented and discussed in Chapter 6 in the form of an example. We use the term constructors because you can have more than one method of the same name which is selected based on the way the **new** statement is written. You can have multiple constructors and the constructors can be overloaded (see below). Some classes also need to carry out a cleanup operation when the program no longer needs them and they go out of scope. For this reason, classes have an additional but optional subroutine which is called **sub destructor()**. This subroutine is automatically called when the class is no longer available to the program. For example, if a class has been instantiated within a subroutine, after exiting the subroutine the class has gone out of scope and the class destructor subroutine is executed if present. The destructor provides the programmer with an opportunity to do any necessary cleanup operations or redimensioning of variables that were, for example, increased in size within the constructor. However, the programmer need not worry about variables that were local to the class. The memory allocated to these variables is returned to the system automatically.

Classes are powerful but complicated objects that new programmers should avoid using until they have mastered modules and the concept of method overloading. Modules are easier to use because all public methods defined within a module are immediately available to the program just as if they had been defined within the main program. The following example illustrates the definition and use of a simple class that takes a number and multiplies it by π (the default) or a user assigned number. The

example also illustrates an important aspect of classes. When they are instantiated (created by using the **new** keyword), the variable that you defined to be of type class1 is "filled" with the code associated with that class. It will not change even if another variable of type class1 is instantiated but instantiated using a different value for the internal private variables. The following example illustrates this important, but rather confusing aspect.

```
class class1  // creates a class called class1
  dim a1 as double  // all variables are private to the class
  dim k as integer // each variable must be declared separately

  public function mba1(x as double) as double
    // public not required because public is the default
    return x*a1
  end function


  sub constructor()
    // constructor uses default initialization of pi
    a1=const_pi
  end sub

  sub constructor(a1set as double)
    // allows user to select other options during new assignment
    a1 = a1set
  end sub

  function a1val() as double
    // this function returns the value of a1
    // although there are no parameters, we need ()
    return a1
  end function

  sub destructor()
    //  optional subroutine is executed when class goes out of scope
  end sub

end class

dim r1, a2 as double
dim blim, blam as class1
```

```
// create an instance of the class using default value of pi
blim = new class1
// create an instance of the class using a value of 4
blam = new class1(4)

r1 = 4.0
a2 = blim.mba1(r1)
print("a2 (based on blim.mba1)  = "+str(a2))
print("blim.a1val = "+str(blim.a1val))

r1=4.0
a2 = blam.mba1(r1)
print("a2 (based on blam.mba1)  = "+str(a2))
print("blam.a1val = "+str(blam.a1val))




// the following demonstrates that creating an instance of blam
// did not override the definition of blim.  Once an instance is
// created, it remains invariant to new instances and constructors.
r1 = 4.0
a2 = blim.mba1(r1)
print("a2 (based on blim.mba1)  = "+str(a2))
print("blim.a1val = "+str(blim.a1val))

// end program
```

The above program, when run, will generate the following output:

```
a2 (based on blim.mba1)  = 12.56637
blim.a1val = 3.141593
a2 (based on blam.mba1)  = 16
blam.a1val = 4
a2 (based on blim.mba1)  = 12.56637
blim.a1val = 3.141593
```

Note that when you want to call a class method, you need to access that method by using the syntax: class_variable.class_method where the class_variable is the variable that was declared (instantiated) to represent the class in your program and the class_method is the name of the method that resides within the class. If this seems like a lot of work without any obvious advantage, it is important to understand that a class

provides some new flexibility. The flexibility is that when a class is instantiated, it can be instantiated with various assignments made at the time of instantiation. You can thus have many different variables representing the same class, but which have functions that are altered at the time of assignment to suit your needs. The above is a trivial example, designed to illustrate the concepts, but not the power, of this flexibility. The class instantiation and construction syntax may appear to be arbitrarily complicated, but once you get used to the syntax and explore the possibilities, you will appreciate the new flexibility the class structure provides.

Chapter 3
Graphics

There are 127 graphics commands in Scriptor ranging from bit-level to complex commands that control fully the writing of both objects and text in color. There are commands which are relevant to graphics artists seeking to create something beautiful and commands relevant to scientists seeking to display information with clarity, precision and style. This chapter provides an introduction to these commands assuming the reader is a novice. If one is experienced in using graphics, the initial section of this chapter still needs to be read to explain how buffers are used to create flicker free, high resolution graphics. Experienced users can explore the many options by selecting Graphics under the Help menu and scanning through the statements and the brief descriptions.

There are individually addressable graphics canvasses available in Main (canvas 1), Graphics (canvas 2) and Music (canvas 3). This chapter introduces the most efficient approach to graphics known as buffered graphics. In this approach, the user creates a graphics buffer and does all manipulations in this buffer, which is invisible to the user. When finished, the programmer then copies the buffer into the canvas using a single statement. The latter operation has been designed to be very fast, and thus provides for flicker-free (or nearly flick-free) graphics. Furthermore, when graphics are created in a buffer, the buffer resolution can be many times greater than the canvas. By using the **buffer_copy_to_canvas** command, the user can automatically display the graphics at maximal resolution allowed by the canvas with full anti-aliasing. This makes the graphics look professional. Furthermore, the user can save the graphics buffer into a file that can be of any standard graphics type (Photoshop, jpeg, pict, tiff, bmp, etc.). Using an intermediate buffer is the only way to create high level graphics that are optimized for all platforms.

The term anti-aliasing, when applied to computer graphics, refers to the process of taking a high resolution image and transferring it to a lower resolution image while eliminating artifacts. The algorithms for this process have been optimized to such an extent that an image invariably looks better if it is produced by anti-aliasing a high-resolution image than if graphics are written directly.

3.1.1. Buffers

The first step in writing graphics is to create the graphics buffer using the statement:

buffer_create(ioption, pixel_width, pixel_height)

which creates a single buffer of size pixel_width by pixel_height with either a white background (ioption=0) or a colored background (ioption=1). If a colored background is selected, the color is assigned by using the buffer_background_color variable. For example, if you want a dark blue background, execute

buffer_background_color = rgb(0, 0, 100) // Execute this before buffer_create()

where rgb is one of three color intrinsics (see below). The user should create a buffer that is big enough to provide adequate resolution, while keeping in mind that the larger the buffer, the slower the graphics manipulations. (In general, the time necessary to write a scaled graphics object into the buffer is proportional to the square of the pixel dimensions of the buffer.) The second issue to consider is the aspect ratio, which equals the pixel_width divided by the pixel_height. Thus, a buffer of 1024x768 provides an aspect ratio of 1.3333. This value is a typical computer screen aspect ratio, but not necessarily the best choice. If one seeks to save the graphics for use as a figure to be inserted into a document, one should choose a higher resolution and an aspect ratio appropriate for the graphics that will be created. As a starting point, a good choice is 3000 by 2500.

After filling the buffer using the statements introduced below, the final step in the process is to transfer the buffer to one of more of the canvases available.

buffer_copy_to_canvas(icanvas, [ioption])

where icanvas specifies the canvas [1(main canvas), 2(graphics canvas), 3(music canvas)] and ioption defines how the buffer is written. If ioption =0 , the graphics are centered are preserve the aspect ratio, if ioption=1, the graphics are drawn starting in the upper left, but with preservation of the aspect ratio. If ioption=2, the canvas is filled with the buffer, and the aspect ratio is ignored.

3.1.2. Specifying the color

The human eye can distinguish light over ten log units of intensity, and at the focal point of the retina, distinguish color differences with a resolution of 1 part in a few million. Color plays a very important roll in how we perceive our environment, and thus we seek to control both the intensity and color of graphics objects with precision. We control the color by using what are known as color intrinsics, functions that allow the user to specify a color based on additive or subtractive space or in terms of a color wheel that helps constrain the colors to constant brightness and/or saturation.

Modern computer screens can manipulate the color of individual pixels to levels of a few parts per million although LCD displays, particularly on notebook computers, often use anti-aliasing to save display memory and in the process reduce the color space to one part in a few thousand. Fortunately, we do not need to worry about the internal workings of the computer display system, and can write graphics routines which will work on any computer which will run Scriptor. Colors are assigned by using one of following color intrinsics.

rgb(ired, igreen, iblue) **as color**, where ired, igreen, iblue are integers that range from 0 to 255. Black is rgb(0, 0, 0) and pure white is rgb(255, 255, 255). Most programmers find that RGB space is the easiest to use because it is intuitive. Although the integers are limited to a single byte range, the RGB function provides access to 255^3 = 16 million colors. Few humans can distinguish colors at this level of precision. The RGB representation has another form that is more compact, and once mastered, is preferred.

&cRRGGBB as color, where RR, GG and BB are the hexadecimal representations of ired, igreen and iblue values from the **rgb**(ired, igreen, iblue) method. Experienced users prefer to use this compact description, because it is faster to type in and they have gained enough experience working with hexadecimal numbers to be efficient. Hexadecimal numbers are base 16 and hence a byte (8 bits) of data can be represented conveniently as a two digit hexadecimal number ranging from 0 [hex(0)=0] to 255 [hex(255)=FF]. Table 3.1 is provided to help those unfamiliar with hexadecimal numbers to quickly convert for single byte integers. For example, white is &c000000, black is &cFFFFFF and the RGB color value rgb(100, 150, 200) is &c6496C8. There is an internal function, **hex**(i) **as string**, which will convert an integer to its hexadecimal equivalent.

cmy(cyan, magenta, yellow) **as color**, where cyan, magenta, yellow are floating point numbers from 0.0 to 1.0. While RGB colors are additive, CMY colors are subtractive. Those familiar with color printers will recognize that color print is created by using three color inks that are subtractive, removing white by overlaying transparent ink (or toner) that is cyan, magenta or yellow. Most printers also have a fourth ink that is black

to provide a more cost effective black and white printing and to improve contrast ratio in color prints.

hsv(hue, saturation, value) **as color** where hue, saturation and value are floating point numbers from 0 to 1.0. The values of hue from zero to one span the range from red(0.0)-orange(0.125)-yellow(0.17)-green(0.33)-aqua(0.5)-blue(0.66)-purple(0.8)-red(1.0). The saturation adjusts the amount of color versus gray scale with 1.0 providing full color. The value adjusts the brightness and goes from 0.0 (black) to 1.0 (brightest). Although this function is the hardest to use to create specific colors, it is the easiest to use to create a series of colors that have the same level of saturation and/or brightness. This capability is very helpful when generating a series of colors for plot lines that have different, easily differentiated colors.

A color, as defined by any of the above three intrinsics, can be assigned to a variable, a pixel or a graphics object. One can also read the color of an object by using the function:

color_value(color, s0) **as double**, where color is a variable containing the color and s0 is a string representing the component sought ("red", "green", "blue" or "hue", "saturation", "value" or "cyan", "magenta", "yellow"). To make programming easier, these components can be designated by using only the first letters (e.g. h, s, v, r, g, b, c, m, y). It is important to remember that the complete description of a color requires three components of a given specification: {r, g, b} or {c, m, y} or {h, s, v}.

color extensions: An alternative to using color_value is to use the extensions .red, .green, .blue to read the RGB values, .hue, .saturation, .value to read the HSV values or .cyan, .magenta, .yellow to read the CMY values. These extensions can also be used to set the values (e.g. clr.red = 200 or clr.hue = 0.7).

color_selection_window(color, text_prompt) **as Boolean**, can be called inside a program to allow the user to select a color. This statement provides access to the color selection window provided by the operating system, and as such, the window is quite different from Mac to PC, and from Windows XP to Windows 8. The text_prompt is only displayed with some operating systems, and the user should check to verify it works and provide an alternative prompt via a print() or show_progress_line() statement.

Table 3.1. Conversion of decimal values (D) to corresponding hexadecimal values (H)

D	H	D	H	D	H	D	H	D	H	D	H	D	H	D	H
0	**0**	1	**1**	2	**2**	3	**3**	4	**4**	5	**5**	6	**6**	7	**7**
8	**8**	9	**9**	10	**A**	11	**B**	12	**C**	13	**D**	14	**E**	15	**F**
16	**10**	17	**11**	18	**12**	19	**13**	20	**14**	21	**15**	22	**16**	23	**17**
24	**18**	25	**19**	26	**1A**	27	**1B**	28	**1C**	29	**1D**	30	**1E**	31	**1F**
32	**20**	33	**21**	34	**22**	35	**23**	36	**24**	37	**25**	38	**26**	39	**27**
40	**28**	41	**29**	42	**2A**	43	**2B**	44	**2C**	45	**2D**	46	**2E**	47	**2F**
48	**30**	49	**31**	50	**32**	51	**33**	52	**34**	53	**35**	54	**36**	55	**37**
56	**38**	57	**39**	58	**3A**	59	**3B**	60	**3C**	61	**3D**	62	**3E**	63	**3F**
64	**40**	65	**41**	66	**42**	67	**43**	68	**44**	69	**45**	70	**46**	71	**47**
72	**48**	73	**49**	74	**4A**	75	**4B**	76	**4C**	77	**4D**	78	**4E**	79	**4F**
80	**50**	81	**51**	82	**52**	83	**53**	84	**54**	85	**55**	86	**56**	87	**57**
88	**58**	89	**59**	90	**5A**	91	**5B**	92	**5C**	93	**5D**	94	**5E**	95	**5F**
96	**60**	97	**61**	98	**62**	99	**63**	100	**64**	101	**65**	102	**66**	103	**67**
104	**68**	105	**69**	106	**6A**	107	**6B**	108	**6C**	109	**6D**	110	**6E**	111	**6F**
112	**70**	113	**71**	114	**72**	115	**73**	116	**74**	117	**75**	118	**76**	119	**77**
120	**78**	121	**79**	122	**7A**	123	**7B**	124	**7C**	125	**7D**	126	**7E**	127	**7F**
128	**80**	129	**81**	130	**82**	131	**83**	132	**84**	133	**85**	134	**86**	135	**87**
136	**88**	137	**89**	138	**8A**	139	**8B**	140	**8C**	141	**8D**	142	**8E**	143	**8F**
144	**90**	145	**91**	146	**92**	147	**93**	148	**94**	149	**95**	150	**96**	151	**97**
152	**98**	153	**99**	154	**9A**	155	**9B**	156	**9C**	157	**9D**	158	**9E**	159	**9F**
160	**A0**	161	**A1**	162	**A2**	163	**A3**	164	**A4**	165	**A5**	166	**A6**	167	**A7**
168	**A8**	169	**A9**	170	**AA**	171	**AB**	172	**AC**	173	**AD**	174	**AE**	175	**AF**
176	**B0**	177	**B1**	178	**B2**	179	**B3**	180	**B4**	181	**B5**	182	**B6**	183	**B7**
184	**B8**	185	**B9**	186	**BA**	187	**BB**	188	**BC**	189	**BD**	190	**BE**	191	**BF**
192	**C0**	193	**C1**	194	**C2**	195	**C3**	196	**C4**	197	**C5**	198	**C6**	199	**C7**
200	**C8**	201	**C9**	202	**CA**	203	**CB**	204	**CC**	205	**CD**	206	**CE**	207	**CF**
208	**D0**	209	**D1**	210	**D2**	211	**D3**	212	**D4**	213	**D5**	214	**D6**	215	**D7**
216	**D8**	217	**D9**	218	**DA**	219	**DB**	220	**DC**	221	**DD**	222	**DE**	223	**DF**
224	**E0**	225	**E1**	226	**E2**	227	**E3**	228	**E4**	229	**E5**	230	**E6**	231	**E7**
232	**E8**	233	**E9**	234	**EA**	235	**EB**	236	**EC**	237	**ED**	238	**EE**	239	**EF**
240	**F0**	241	**F1**	242	**F2**	243	**F3**	244	**F4**	245	**F5**	246	**F6**	247	**F7**
248	**F8**	249	**F9**	250	**FA**	251	**FB**	252	**FC**	253	**FD**	254	**FE**	255	**FF**

3.1.3. variables of type color

New programmers are often surprised to discover that a variable can be declared of type color. For example, one can declare the following variables:

```
dim darkred, lightred as color
darkred = rgb(80, 0, 0)
lightred = rgb(200, 0, 0)
```

One can also define color constants, but in order to do so, one must use the hexadecimal representation described above. Thus, a constant of darkred can be defined as follows:

```
const darkred = &c500000
```

Be sure to avoid trying to redefine a constant in your program. Constants are not supposed to be changed by the user once defined. Any attempt to redefine a constant will generate error 70 and termination of the program.

3.1.4. bit level graphics

One can control and read the properties of an individual pixel if desired. However, this is done with reference to the graphics buffer, not one of the three canvases. There is a good reason for this restriction. The buffer is completely under the programmer's control, whereas a canvas is shared with Scriptor and the operating system. Although you can indeed write to the canvas, this process is mediated by Scriptor based in part on the operating system and the graphics cards that are installed in the computer. Scriptor optimizes the display in the three graphics canvases by doing anti-aliasing and buffering, which is operating system dependent. Graphics on a Macintosh computer are handled differently from graphics on a Windows PC. In contrast, the buffer is yours to handle as you see fit.

There are two statements available for setting or reading the color of a buffer pixel (assume **acolor** has been declared of type color).

```
buffer_pixel(ix, iy) = acolor
acolor = buffer_pixel(ix, iy)
```

where ix, iy are the pixel positions inside the buffer. These statements are 1-based which means the upper left pixel is buffer_pixel(1, 1) and the lower right pixel is buffer_pixel(nx, ny). Thus, pixels are numbered in a raster format which is identical to the way elements in an array are defined. This approach makes inherent sense until it is extended into writing graphics objects, at which point the user must confront the fact

that the Y axis appears to be inverted (Y values increase as you move south on the canvas). Always remember that 1, 1 is at upper left.

Lets examine a simple program that creates a very small buffer and writes into the pixels:

```
// Program Name: program_3.1
// This program writes randomly colored pixels inside a 10x10 buffer
with a dark red background.

dim i,j as integer
set_graphics_slider(100) // fullsized Main graphics
const darkred = &c500000 // hexadecimal method
buffer_background_color = darkred
buffer_create(1,10,10) // create buffer with backcolor
for i=3 to 8
  for j=3 to 8
    buffer_pixel(i,j)=rgb(255*rnd,255*rnd,255*rnd)
  next
next
buffer_copy_to_canvas(active_canvas,0)

// end program
```

A copy of the graphics output from the above program is shown if Fig. 3.1.

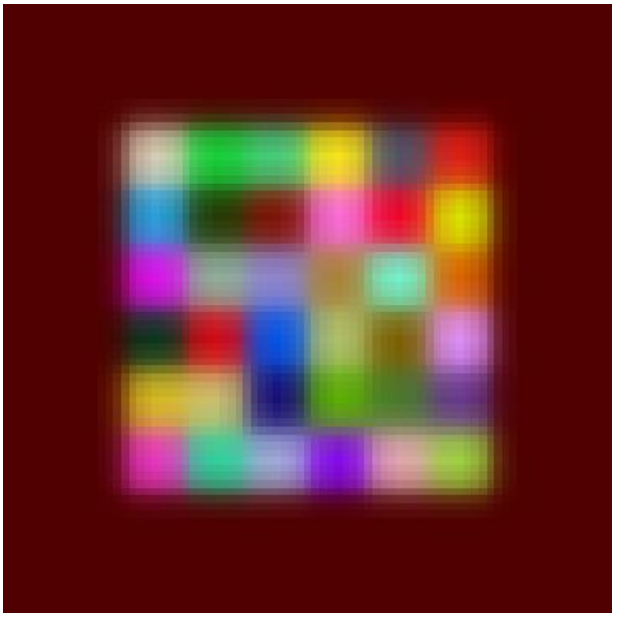


Figure 3.1. The output from the above program is shown at right. The background color is assigned to the color constant darkred. Individual randomly colored pixels are written into a smaller 6x6 pixel square inside the 10x10 pixel buffer. The small buffer is then expanded into the canvas by using the buffer_copy_to_canvas statement which means each pixel is expanded into a space of about 12x12 pixels. The expansion is blurred in the process of anti-aliasing.

The ability to read the color of a buffer pixel may appear to be pointless given the fact that the programmer was likely responsible for writing it. However, it is possible to load an external graphics object into the buffer and then read the color of each of the bits by using the **buffer_pixel**(i, j) statement. Reading a large buffer pixel-by-pixel is possible, but it is slow and inefficient. When the entire buffer is to be examined or manipulated at the pixel level, there are statements that can read the entire buffer into arrays. One example is the following:

buffer_to_arrays((ired(), igreen(), iblue())

The red channel is stored into ired(1..buffer_width, 1..buffer_height), the green into igreen(1..buffer_width, 1..buffer_height) and the blue into iblue(1..buffer_width, 1..buffer_height). The three arrays are redimensioned to the size of the buffer. One can then manipulate these arrays and load them back into the buffer using:

buffer_fill_from_arrays(ired(), igreen(), iblue()).

We will explore the use of these, and other similar statements, in more detail below.

3.2. Drawing Objects

Before we draw any objects, we need to assign a color and a linewidth to use for the drawing. These two requirements are handled by using the following two statements:

graphics_forecolor(desired_color)

where color is replaced by a variable of type color or a color intrinsic (e.g. rgb()). The default graphics_forecolor is black. Similarly, we assign the stroke width by using:

graphics_stroke_width(npixels)

where npixels is the stroke width in pixels. The default stroke_width is 1.0 At any time during and/or after the graphics manipulations, the buffer can be written by using the statement:

buffer_copy_to_canvas(icanvas, [ioption])

where icanvas specifies the canvas [1(main canvas), 2(graphics canvas), 3(music canvas)] and ioption defines how the buffer is written. If ioption =0 , the graphics are centered are preserve the aspect ratio, if ioption=1, the graphics are drawn starting in the upper left, but with preservation of the aspect ratio. If ioption=2, the canvas is filled by the buffer, and the aspect ratio is ignored.

3.2.1. Lines and Fills

Lines are the simplest of all the objects that can be drawn, but in combination, have the potential to draw any complex object. All objects are drawn to the buffer, or to buffer 1 is multiple buffers have been created. One can draw lines of any color or thickness by using the following statement:

draw_line(ix1, iy1, ix2, iy2)

which draws a line from ix1, iy1 to ix2, iy2. The thickness of the line defaults to 1 pixel, but can be set to any number of pixels by using **graphics_stroke_width.** The color of this line is assigned by using graphics_forecolor. If there is a complex object involving a series of lines connected to each other, a more powerful statement is available:

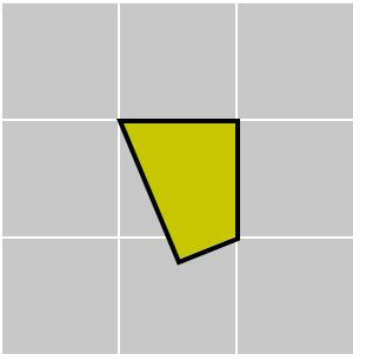
draw_arrays(ix(), iy(), n1, n2)

which draws a series of connected lines with vertices at ix(n1..n2) and iy(n1..n2). The power of this statement is more evident when one wishes to draw a complex object that is filled with a solid color. The following statement will do exactly that:

fill_arrays(ix(), iy(), n1, n2)

which is identical to the draw_arrays statement except the outline of the object is not drawn, but rather the object is filled with the **graphics_forecolor**. This is only successful if the first and last pairs of points are identical: ix(n1)=ix(n2) and iy(n1)=iy(n2). However, if the first and last points do not coincide, the fill routine closes the object by drawing a line between the first and last points and then completing the fill. For example, the following code segment draws the object shown at right (the grid lines are in increments of 100 pixels):

```
graphics_stroke_width(4)
graphics_forecolor(rgb(200, 200, 0))
ix()=array(0, 100, 200, 200, 150)
iy()=array(0, 100, 100, 200, 220)
fill_arrays(ix(), iy(), 1, 4)
graphics_forecolor(rgb(0, 0, 0)
draw_arrays(ix(), iy(), 1, 4)
```



The above code segment demonstrates a statement that is quite useful when loading a series of values into an array. This statement is shown below:

array(comma delimited list of any type variable) **as variant 1d array**

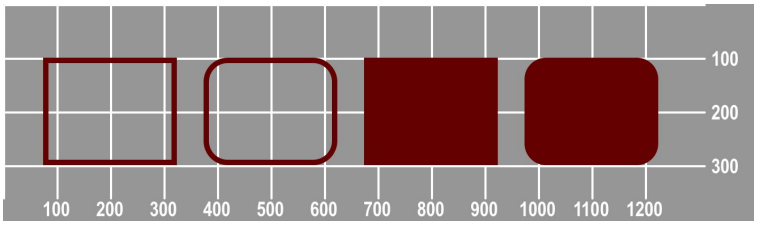
which transfers the comma delimited set of variables into a one-dimensional array of the appropriate type starting at the 0th element. If you want to start at the array(1) element, simply insert a zero or null value (0.0, 0 or "") in the first position as shown in the code example above. The recipient array is dynamically re-dimensioned to be of the exact size as required to hold the data.

3.3. Predefined Objects

Although the draw_array() and fill_array() methods provide the programmer with the flexibility to draw or fill any object, these statements are complicated to use because the programmer has to specify the vertices of the object. Thus, internal methods are predefined for squares, rectangles and circles. The commands are summarized below:

3.3.1. Squares and Rectangles

Squares and rectangles are drawn using the same statements:



draw_rect(ix_center, iy_center, iwidth, iheight, [iarcwidth, iarcheight])
fill_rect(ix_center, iy_center, iwidth, iheight, [iarcwidth, iarcheight])

where the center of the object is at ix_center, iy_center, the width of the rectangle is given by iwidth and the height by iheight. The last two parameters are optional, and if left out, the rectangle has sharp edges. If included, iarcwidth and iarcheight indicate where to start the rounded arcs relative to the edges in the x and y axes. The following four statements draw the rectangles shown in the figure above.

```
draw_rect(200, 200, 250, 200)
draw_rect(500, 200, 250, 200, 80, 80)
fill_rect(800, 200, 250, 200)
fill_rect(1100, 200, 250, 200, 80, 80)
```

To draw a square, one sets iwidth = iheight.

3.3.2. Circles and Ovals

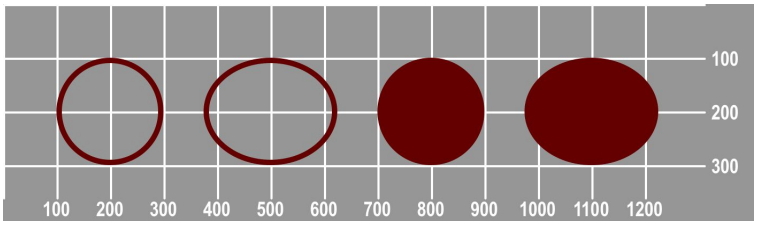
There are four statements that draw circles and ovals. Circles are defined in terms of their center and radius:

draw_circle(ix_center, iy_center, radius)
fill_circle(ix_center, iy_center, radius)

whereas ovals are defined in terms of their center and both a width and a height:

draw_oval(ix_center, iy_center, iwidth, iheight)
fill_oval(ix_center, iy_center, iwidth, iheight)

For example, the following four statements generate the circles and ovals shown in the figure at right.



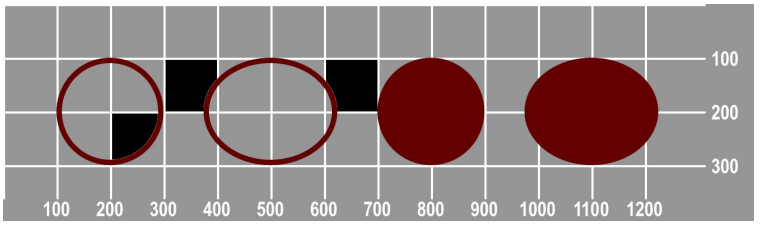

```
draw_circle(200, 200, 100)
draw_oval(500, 200, 250, 200)
fill_circle(800, 200, 100)
fill_oval(1100, 200, 250, 200)
```

3.3.3. The Paintbrush

There is a very powerful command which can carry out a local fill based on the local pixel environment:

buffer_write_paintbrush(ix, iy, ired, igreen, iblue)

This command reads the pixel color at position ix, iy and replaces that pixel and all contiguous pixels of the same color with the new color specified by ired, igreen and iblue. For example, the following statements, modify the previous picture as shown above right.




```
buffer_write_paintbrush(250, 220, 0, 0, 0)
buffer_write_paintbrush(320, 120, 0, 0, 0)
buffer_write_paintbrush(630, 120, 0, 0, 0)
```

The paintbrush is a tool that may appear to be more artistic than pragmatic, and it is true that clever use of the paintbrush can create complex and unique graphics. However, it is also a very useful tool for scientific and engineering graphics, particularly when one seeks to illuminate the overlap of two complex objects or execute a ray tracing algorithm.

3.4. String Graphics

Prior to writing text into the graphics panel, one of the local fonts must be selected using the following statement.

```
graphics_font(font_name, isize, Qitalics, Qbold) as boolean
```

which sets the buffer graphics font to the string font_name, and simultaneously assigns the font size using the integer isize. The booleans Qitalics and Qbold set italics and bold options, respectively. This function returns true if the specified font is found in the font folder. This function is not smart enough to look for the font in other locations on the computer, and thus the programmer should either make sure the font is present or provide a series of statements that test for a number of different fonts until a valid font is found.

If one is working on a program on the same computer that will be used to run the program, it is a simply matter to check the font menu. If the font is there, then Scriptor will be able to find it with the above statement. Make sure the font_name is spelled identically to the one that appears in the font menu. Alternatively, a font type can be specified by using one of the following variables to assign the fontname:

```
System_fontname_sans as string
System_fontname_serif as string
System_fontname_label as string
System_fontname_mono as string
System_fontname_narrow as string
```

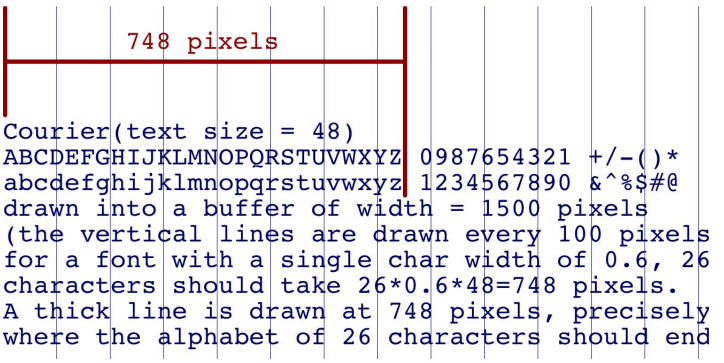
These five strings are assigned when Scriptor is starting up at which time the program scans the local system font folder, loads all of the viable fonts into the font menu, and then analyzes them to find fonts that are of the three following forms:

serif fonts. These are fonts which have non-structural detail added at the edges of the font to make them easier to read or more artistic. The main text of this book is written using Times, which is one of the more common serif fonts, and very readable when on the printed page. Other common examples of serif fonts include: Times New Roman

(traditional serif), Garamond (old-style serif), Rockwell (slab serif) and Bodoni (modern serif).

sans or sans-serif fonts. The term sans-serif means “without serif” features and is derived from the French word sans (without). It is now common to use the term sans by itself to represent sans-serif. Removing non-structural detail makes a font more readable on a computer screen, and makes titles and headers stand out. This paragraph is printed in Arial, a very common version of a sans-serif typeface. Other common examples of sans-serif fonts include: Helvetica (traditional sans), Royal Gothic (grotesque or early sans), Gill Sans (humanist sans) and Avant Garde (geometric sans).

monospace or non-proportional fonts. A majority of fonts are designed to be proportional. That means, the horizontal space for each letter is determined by the nature of the letter itself. This makes the type more compact and easier to read. However, if one is preparing a table of numbers and text, the table is more legible if all the letters and numbers line up in columns. Non-proportional or monospace fonts have identical widths for each glyph (letters, symbols and numbers). This paragraph and the spacing example at upper right is printed in Courier, the most common monospaced font. The single character width is 0.6 times the font size. Other common monospaced fonts and their character width include: Courier New (0.6, a thinner version of Courier), Andale Mono (0.6), Monaco (0.6), Letter Gothic (0.6), Prestige Elite (0.6), Osaka (0.5), OCR A Std (0.72). If a table of numbers is being prepared, it is worth knowing that many proportional fonts nevertheless have monospaced numbers. Examples and relative widths of the number characters include: Arial Black (0.667), Arial Rounded MT Bold (0.594), Baskerville (0.5), Bookman Old Style (0.62), Century Schoolbook (0.556), Garamond (0.469), Gill Sans (0.5), Lucida Grande (0.632), Marker Felt (0.57), and Rockwell (0.542). However, none of the monospaced number fonts listed in the previous sentence provide monospaced numerical symbols (e.g. {+, -.}). In most cases, proportional symbols can be used while still allowing the digits to line up properly.



3.4.1. Drawing strings

A string is drawn to the graphics target by using one of the following three statements:

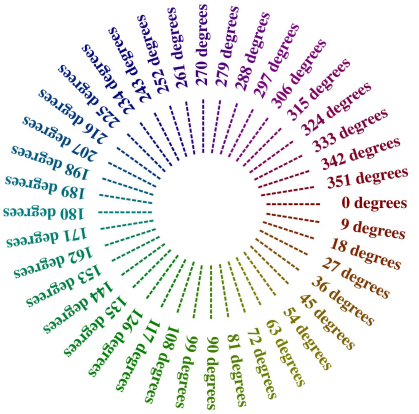
draw_string(text, ix_center, iy_center)

which draws a single line of text centered at ix_center and iy_center. The color of the text is that assigned via graphics_forecolor and the text size is that assigned previously via graphics_font (see above). If a multiline string is to be drawn, ix_center and iy_center now assign the upper left position of the string. A multiline string is assigned by the method when it finds one or more endofline characters (=const_eol). The multiline string will be printed left justified. The above two statements draw text that is horizontal, but as shown in the example below, text can be rotated to any angle desired.

draw_rotated_string(text, ix_center, iy_center, angle_degrees)

where ix_center and iy_center are the center of the text. If the goal is to rotate the text about a fixed position, as shown in the figure at right, then one must rotate the ix_center, iy_center values around a selected position. A program that does exactly that is shown below and produces the figure at right.

```
dim ixt, iyt, radius, theta as integer
dim s0 as string
set_graphics_slider(100)
buffer_create(0, 2000, 2000)
radius = 500
if graphics_font(system_fontname_serif, 64, false, true) then
  for theta = 0 to 359 step 9
    graphics_forecolor(hsv(theta/360., 1, 0.5))
    ixt = 1000 + radius*cos(theta*const_degree)
    iyt = 1000 + radius*sin(theta*const_degree)
    s0="----------"+format(theta, 6, 0)+" degrees"
    draw_rotated_string(s0, ixt, iyt, theta)
    buffer_copy_to_canvas(active_canvas, 0)
  next
end if
```



3.4.2. Selecting Cross-Platform Fonts

It is now common for both Windows and Macintosh operating systems to have a large number of fonts provided by the operating system. Unfortunately, the fonts commonly found on a Mac differ from those on a Windows machine except for a select few that seem to be present on most systems, and which are usually included in the font collection installed on printers. If one is writing cross platform programs, it is a good idea to select from the following list of common cross-platform fonts:

Arial (character width in pixels for size 12 = 5.825)
Arial Narrow (4.772)
Book Antiqua (5.895)
Bookman Old Style (6.368)
Century (6.202)
Courier (7.132)
Garamond (5.561)
Palatino (5.868)
Tahoma (5.974)
Times New Roman (5.596)
Verdana (6.851)
Symbol {αβγεφγηψμλ} (5.974)

where the font name is printed in the font (except for symbol) and the number in parentheses is the average width of the characters in pixels when plotted in a canvas using a font size of 12. Note that while all of these fonts are the same size (12), they have a surprisingly different height and width when printed (or plotted). That is why the average character width is important to know.

An alternative approach to cross-platform font selection is to use the font names assigned by the system automatically at startup to the five variables shown below:

```
System_fontname_sans as string
System_fontname_serif as string
System_fontname_label as string
System_fontname_mono as string
System_fontname_narrow as string
```

That approach more-or-less guarantees that a font will be available of the desired type, but it does not guarantee that the same fonts will be used at run-time. If there are no fonts in the system folder of a given type, the system_fontname_option string will be null.

3.5. Graphics using Multiple Buffers

The user can create multiple buffers and in the process prepare complex, encapsulated figures. This approach is particularly useful when a more complex graphic is made up of separate components that are more easily prepared separately. The key statement is the following:

buffer_create_multiple(nbuffers, ioption, iwidth, iheight)

where nbuffers is the number of individual buffers to be created, ioption indicates whether the buffers would have a white (ioption=0) or colored (ioption=1) background. The size of each buffer is iwidth by iheight in pixels, and it is critical to keep in mind that the total work area is given by abs(nbuffers)*iwidth*iheight. Memory allocation can therefore become an issue if one seeks to create six buffers with a size of 3000 by 2000 pixels, which would require 6*3000*2000*32 = 144 MB. This memory amount is, by itself, of little consequence but on a computer with 512MB total memory, a potential deal breaker. The Scriptor program will make every effort to reallocate objects to handle the additional memory requirement, but if that fails, execution will generate a run-time error and the following error message:

nil-object error- not enough RAM to create graphics buffer.

For reference, the term nil-object means the system attempted to create an object (in this case, a large graphics buffer), but when this object was checked for existence, it was not found (i.e. nil-object). Often the operating system will take such umbrage at the audacity of asking for more memory than it can provide, it will shut down the application entirely. Do not despair. If one has checked "open most recent work when starting up" in the preferences panel, the program will be reloaded when you restart Scriptor. If not, you can find it in the "backup" folder where each program run is saved with a label of the form "B"+date+time+truncated_name.txt.

There are eight options for nbuffers, where the absolute value determines the number of canvases within the buffer and the sign determines the canvas placement:

2 (1 2 --- left right)
-2 (1 above 2 -- up down)
3 (1 2 3 --- left middle right)
-3 (1 above 2 above 3 ---- top middle bottom)
4 (1 2 3 4 --- left to right in line)
-4 (1 2 on top, 3 4 underneath)
6 or -6 (1 2 3 on top 4 5 6 underneath)
9 or -9 (1 2 3 on top 4 5 6 in middle and 7 8 9 on bottom)

Explicit locations of the canvases for each of these options are shown in Figure 3.5.1.

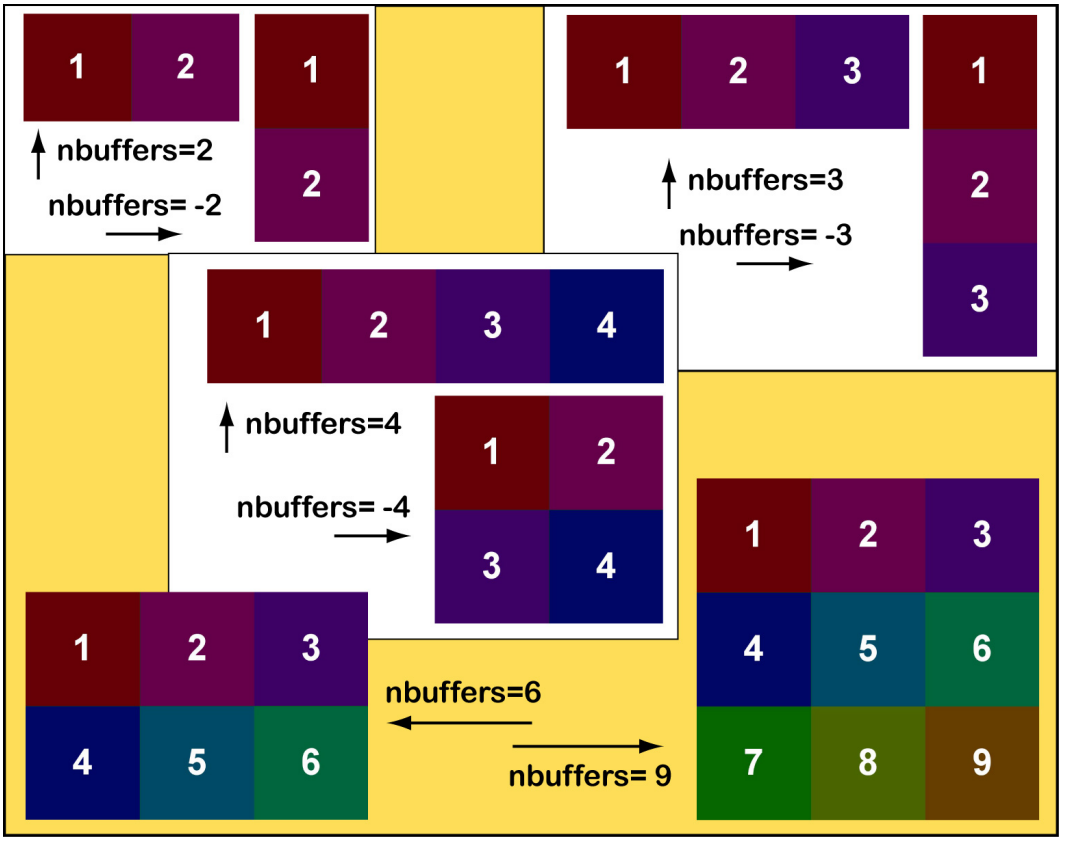


Figure 3.5.1. The location of the individual canvases within the larger buffer as a function of nbuffers and the sign of nbuffers.

When using multiple buffers, one always writes into buffer 1, and then moves or swaps buffer 1 into one of the other canvases. Each canvas is labeled buffer 1, buffer 2, etc. Manipulations are carried out by using the following statements:

buffer_copy_to_buffer([isource,]itarget)

where the optional parameter, isource, is the source buffer and itarget is the destination. Note that the operation is a copy, so that the contents of buffer itarget are overwritten by the contents of buffer 1 (of buffer isource). These operations all depend on the user

having previously created at least itarget canvases. If that is not the case, the operation is canceled and the following error is printed in

Run Time Error in buffer_copy_to_Buffer: target canvas number exceeds number of created buffers.

The alternative is to flip two buffers by using the following statement:

buffer_flip_buffers(i, j)

which flips the contents of buffers i and j. This statement does not destroy the contents of any of the canvases, simply flips those designated by the two integer parameters. If either value exceeds the number of canvases available, the operation is canceled and the following error is printed out.

Run Time Error in buffer_flip_buffers: one or more of the targets exceeds number of created buffers.

Multiple buffers are invaluable for preparing comparative figures for professional graphics applications. A number of examples are presented below. By allowing the user to work on a small section at a time, the programming is much easier and the complicated offsets required to create a large picture are obviated. However, one always has the option of putting a larger figure together by using Adobe Illustrator, Adobe Photoshop or a comparable graphics editing program.

3.6. Saving and Loading Graphics Files

After creating an artistic picture or scientific graphic, the next step is usually to save the buffer (which contains the picture or pictures) for insertion into a document or emailing to a friend or collaborator. This process is accomplished by moving to the graphics panel and using the save button. The user is then presented with the option of not only naming the graphics file and selecting a location for the file, but selecting the type of graphics file into which the buffer is saved. A brief description of the various graphics file types is therefore presented below. Remember that the save button is not saving the graphics canvas that is observed but rather the buffer that the user created to store the graphics. Normally, the buffer will always be higher resolution than the canvas.

3.6.1. Saving Graphics via the Save Command

Scriptor can save the buffer in various formats. In some cases, the dialogue includes options that determine the degree of compression. The sophistication of this dialogue is

directly dependent upon whether Quicktime has been installed on the users computer. Because Quicktime is free and is required if one is to make use of the Music panel, the following discussion assumes the user has already installed the free version. The Pro version is not required for any of these options. Quicktime is cross-platform and is available at **www.apple.com/quicktime**. The various formats are described below:

Photoshop (filename.psd). Adobe Photoshop has become the de facto standard for image editing, and is used by a majority of graphics professionals as well as students. The Photoshop format is a potentially loss-less compressed form that is both efficient and of high quality. One has the option of working with a smaller color space (best depth to 256 shades), and if the color space is reduced by user selection, one will create a smaller file with a corresponding loss in color depth.

BMP (filename.bmp). The abbreviation refers to a Bit-Mapped Picture, and is a common choice for Windows computers. Most applications on a Mac can read this as well, so this is a good choice and options are available for various color depths from black and white to millions of colors. The picture can also be saved in greyscale.

JPEG (filename.jpg). The abbreviation JPEG stands for Joint Photographic Experts Group and is the most common choice for photographs where file size is important. JPEGs are stored using a clever method of minimizing file space by a variety of methods that allow the user to select both grey scale or color as well as quality of the image. One can alternatively assign a file size, and the compression will be carried out to achieve the goal. The problem with this format is that it introduces artifacts that are permanent and blotchy. While the best choice for web based pictures where download time is an issue, jpegs should be avoided if quality is the primary issue. However, there is one exception. The user can save the picture using jpeg format and "best quality" and generate a file that is nearly lossless. Journals often request high resolution, best quality jpeg figures as they represent a good compromise of quality and size.

JPEG2000 (filename.jp2). An updated version of the JPEG format (see above) which allows for lossless compression and high quality. An excellent choice if the target application can read this format. The only downside to this choice is the time it takes to decompress the image, because the wavelet-based image compression requires heavy computer usage.

PICT (filename.pct). This format was historically used on Macintosh computers, and is both powerful and flexible. Options include a variety of compression schemes and quality options. The flexibility of this format is impressive, but if the target computer does not have Quicktime installed, the ability to read this format should not be assumed. This format has been deprecated in favor of the pdf format.

PDF (filename.pdf). The abbreviation stands for portable document format, and was developed by Adobe corporation as a general approach to encapsulation of documents and pictures. In the case of pictures, both vector and pixel graphics are supported, and LZW compression is available to adjust the size at the expense of resolution.

PNG (filename.png). The abbreviation usually stands for persona non grata, but in the present case it stands for Portable Network Graphics. Unless someone asks for this format, don't use it.

Quicktime (filename.qtif). This format is excellent, but requires that the target computer have Quicktime installed. This format is not as commonly used as PICT, and has no intrinsic advantages over PICT. Unless someone requests this format, don't use it.

TIFF (filename.tif). The abbreviation stands for Tagged Image File Format and the format is now owned by Adobe. It is known as a container format because one can store both bit-mapped and line-art images. Although this format allows for compression, it is not as flexible as other formats. Furthermore, there are Mac and PC versions of the TIFF format which often lead to incompatibilities. The advantage of this format is that it can be lossless, but other lossless image formats are available (e.g. Photoshop and jpeg2000) and the potential for image corruption due to corrupted tags or format problems must be considered. Although historically important, other formats are preferable when bit mapped images are involved.

3.6.2. Saving Graphics via the Picture Conversion Window

Under the file menu is the Open Picture Conversion Window item that opens up a separate window and automatically loads the buffer image into the window. A picture of this window is shown in Fig. 3.6.2. This window allows the user to manipulate the resolution and down-convert the picture to grey scale, if desired. While increasing the resolution is available, it is preferable to do that type of manipulation in a more advanced graphics program like Photoshop. When decreasing the resolution is desired, this program carries out full anti-aliasing and produces results comparable to Photoshop.

The save button at lower right allows the user to save the modified graphics. The dialogue provides the same selection of options as the save button in the Graphics Window (see Section 3.6.1.). All of the operations that are done in this window can also be done using programming, but this window provides immediate feedback and is more convenient to use when relatively few operations are involved.

3.6.3. Opening Graphics via the Picture Conversion Window

The Picture Conversion Window also allows the user to open a graphics file and subsequently load the picture into the buffer. This same operation can be done from within a program, and thus the only reason to use the window is if the picture is to be manipulated. This option is best if a picture needs manipulation with regard to transparency as the user usually needs to look at the picture to carry out the operation (and select the color to be made transparent). When the open button is selected, the open dialogue is directed to the user_pictures folder provided that the scripter program is run from the folder containing the user_pictures folder.

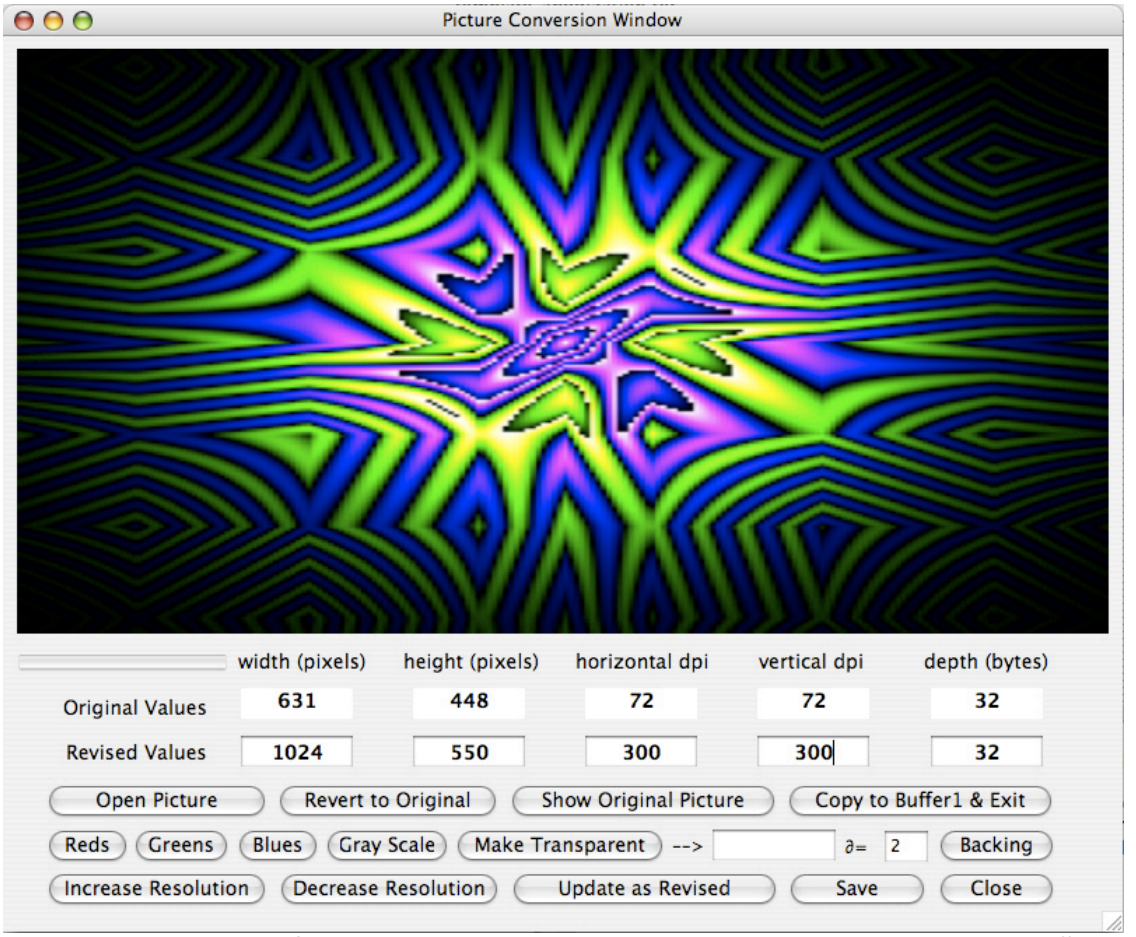


Figure 3.6.2. The Picture Conversion Window. When opened, this window displays the current buffer and allows the user to manipulate the buffer in preparation for saving. This window will also open a graphics file, adjust its color or resolution and copy the picture into buffer 1 (use the **Copy to Buffer1 & Exit** button).

3.7. Manipulating Pictures

Individual pictures can be loaded into memory and analyzed or manipulated. Although the actual analysis must take place in the buffer, the user opens the picture into a previously created picture object created by using the following statement:

picture_create(inum, nwidth, nheight, Qtransparent)

which creates a new picture file in the picture slot ipicture with size nwidth by nheight If Qtransparent is true, pure white becomes transparent. Pure white is RGB(255, 255, 255) = CMY(0, 0, 0) = HSV(0, 0, 1). (The picture conversion window, available under the Edit menu, can be used to manipulate the location and amount of transparency for any given picture.) One can create as many pictures as memory allows, but it is important that the picture slots be created in numerical order (low to high) to optimize memory utilization. The following statement can subsequently load a picture from a disk file into buffer 1 or the picture object designated by idestination.

open_user_picture_file(ifilenumber, filename [, idestination]) **as boolean**

operates in a fashion identical to the open_user_text_files routine but goes to the user_pictures file and then loads the picture into the buffer. If multiple buffers have been created, then the picture is loaded into buffer 1. If the optional parameter idestination is included, it designates the picture slot into which you want to place the opened picture. One does not need to create a picture object if the above statement is to be used to load a picture into the buffer. Although a picture can be designated to have transparency when the object is first created, the following statement allows the user to turn transparency on or off:

picture_make_transparent(ipicture)

makes the ith_picture transparent. If the value of ith_picture is negative, the abs(ith_picture) is made non-transparent, which means that purewhite is now solid. This function does nothing if the ith_picture has not yet been created. There are two ways to copy a picture (from the picture set) to the buffer. The fastest method uses a pixel-to-pixel transfer via the following statement:

picture_copy_to_buffer(ipicture)

which copies picture number ipicture into the buffer on a 1-to-1 pixel ratio starting at the upper left. The buffer must be large enough to handle this copy or the picture is

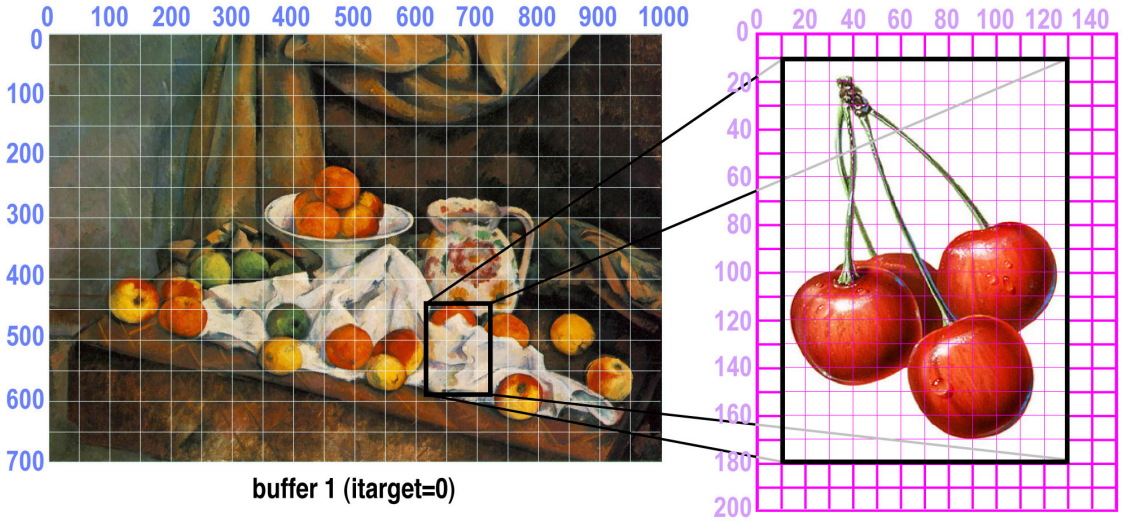
truncated (pixels too large to fit are lost). However, a more sophisticated copy can be carried out by using the following statement:

picture_write(isource, targetx, targety [, destwidth, destheight, sourcex, sourcey, sourcewidth, sourceheight])

draws picture number isource into the buffer. TargetX and TargetY are the upper left hand pixel coordinates. If no other parameters are included, then the entire picture is written into the target at targetx and targety. You can, if desired, include another six parameters to designate the size of the window and the portion of the picture you want to draw. Destwidth and destheight set the size of the window into which you want to write the picture. Sourcex, sourcey, sourcewidth and sourceheight set the upper left hand corner and size of the picture area you want to copy into the window previously defined. The sizes and aspect ratios need not be the same and thus the picture, or picture section, can be compressed or skewed to accommodate the target window.

The combination of optional transparency and the ability to control both location and size of a copy allows for high-level, complex graphics. The program above illustrates the process by placing a smaller picture of cherries onto a perfectly nice Cezanne painting using transparency.

```
// program_name: template_picture_transparency.txt
// This program places a few cherries in a Cezanne painting using
// transparency versus no transparency in a loop.
// Main
dim destheight,destwidth,filenumber,isource,itarget as integer
dim itime,sourceheight,sourcewidth,sourcex,sourcey as integer
dim targetx,targety as integer
dim filename as string
dim Qcreated,Qfound,Qtest as boolean
clear_text_output(0)
set_graphics_slider(70)
print(" We demonstrate adding some cherries to a Cezanne")
print(" We alternate using transparency and no transparency")
print(" Press stop if you get bored.")
buffer_create(0,1001,700)
filename = "famous_artists\cezanne_fruit.jpg"
filenumber = 0
Qfound = open_user_picture_file(filenumber,filename)
if not Qfound then
  print(" cant find "+filename+"!!!")
  return
end if
Qcreated = picture_create(1,150,200,false)
filename = "famous_artists\cherries_transp_32.tiff"
Qfound =open_user_picture_file(filenumber,filename,1)
if Qcreated and Qfound then
  for itime=1 to 50
    buffer_create(0,1001,700)
    filename = "famous_artists\cezanne_fruit.jpg"
    filenumber = 0
    Qfound = open_user_picture_file(filenumber,filename)
    pictures_clear_all
    Qcreated = picture_create(1,150,200,false)
    filename = "famous_artists\cherries_transp_32.tiff"
    filenumber = 0
    Qfound =open_user_picture_file(filenumber,filename,1)
    if itime mod 2=0 then
      picture_make_transparent(1)
      show_progress_line("White is transparent")
    else
      picture_make_transparent(-1)
      show_progress_line("White is Not transparent")
    end if
    isource = 1 // we are using picture no. 1
    targetx = 620
    targety = 440 // upper left corner of subspace
    destwidth = 120
    destheight = 170
    sourcex = 0
    sourcey = 0
    sourcewidth = 150
    sourceheight = 200
    picture_write(isource,targetx,targety,destwidth, _
    destheight,sourcex,sourcey,sourcewidth,sourceheight)
    buffer_copy_to_canvas(active_canvas,0)
    if check_for_stop_button then exit
    pause(1000)
  next
end if
// End Program
```



targetx = 630, targety = 450
destwidth = 120*0.8, destheight = 170*0.8

picture_write(itarget,isource,
targetx,targety,destwidth,destheight,
sourcex,sourcey,sourcewidth,sourceheight)

sourcex = 10, sourcey = 10
sourcewidth = 120
sourceheight = 170

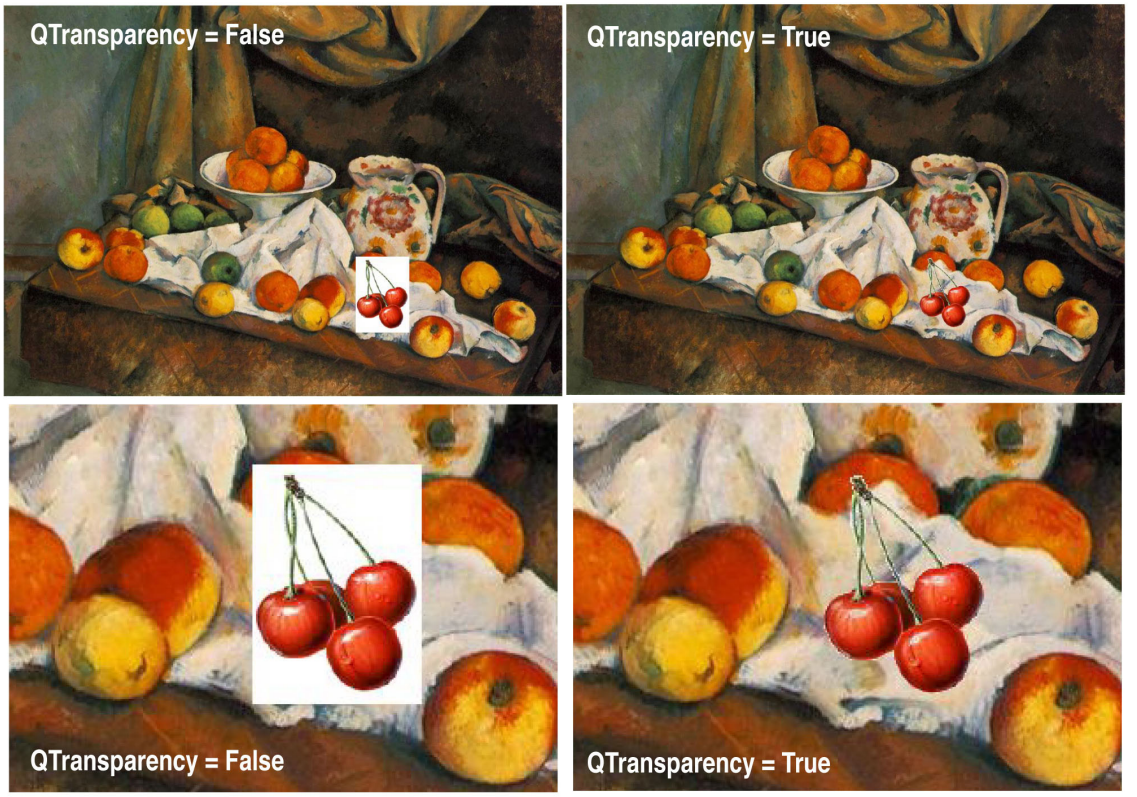


Figure 3.7.1. Demonstration of copying pictures onto pictures using transparency.

3.8. Plotting

Plotting is the process of presenting data in a format which uses two or three dimensions to represent the data in a spatially relevant format. Scriptor has a full set of plotting functions which can handle x(1..npoints), y(1..npoints) data sets or a2(1..nx, 1..ny) data sets. The ability to take a two-dimensional array and present it graphically has value for both scientific and graphic arts applications. We will emphasize graphic arts in this section, and reserve scientific plotting discussions for the numerical methods discussion in Chapter 4. The following plotting functions and parameters are available:

plot3D(z(), theta, phi, xzoom, yzoom, [ioption] or [cz()]) plots a three dimensional representation of the 2D double array z(1..ubound1, 1..ubound2) from a view direction of theta and phi degrees (20, 20 usually works) with size options determined by xzoom and yzoom (start with 1, 1). The last parameter is either ioption or a 2D array of colors the same size as z(,). Ioption selects transparent mesh (0), wire frame (1), gray scale (2) or color (3).

plot3d_xshift assigns as integer assigns the x axis offset shift of the 3D plot. A positive value shifts the plot to the right.

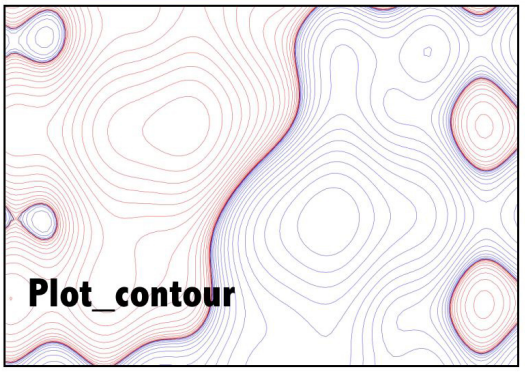
plot3d_yshift assigns as integer assigns the y axis offset shift of the 3D plot. A positive value shifts the plot up.

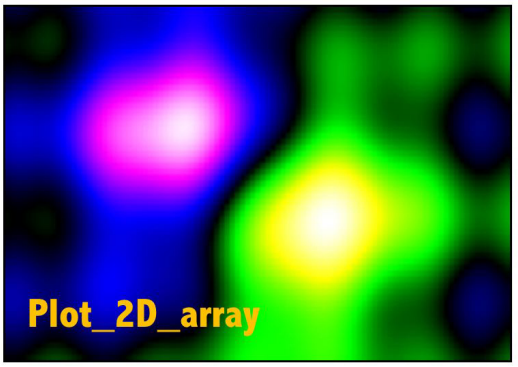
plot_2d_array(a2(), n1, n2, ioption, imod, Qzero) convert data in a2(1..n1, 1..n2) or if Qzero then a2(0..n1, 0..n2) into a filled contour plot. ioption provides for the option of apodization (ioption=0, 1=none, 2=triangular (linear) ww = 1 - (rxy/rxymax), rxymax = (nxbasis-center) = center, 3=lorentzian (quadratic) ww = 1 - rxy^2/rxymax^2 = 1 - rxy^2/center^2, >3=gaussian (exponential) with fwhm = 1/ioption (1/4, 1/5, etc.), if negative, the apodization is based on abs(ioption) but the absolute value is plotted. The parameter imod selects the modulus of the display (min=1, max=4). Add 100 to ioption for grey scale.

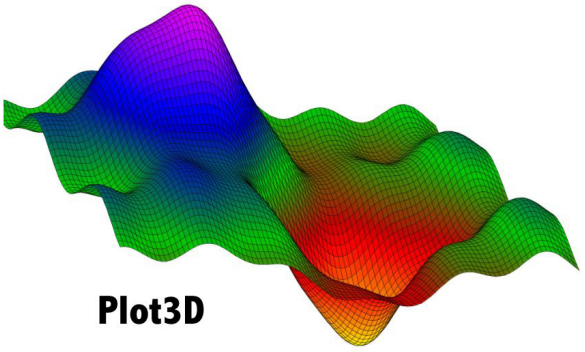
plot_contour(a2(), nx, ny, ncontours, linewidth, ioption) convert data in a2(1..nx, 1..ny) into a line contour plot with ncountours lines. Linewidth = width of the contour lines. Ioption = 0 (plot on top of white), 1 (plot on top of background color), 2 (plot on top of whatever was already there). Use ioption=2 to plot a contour on top of a plot_2d_array display. The contour levels are printed out in the Main text editfield.

A two-dimensional contour plot is sometimes called a three-dimensional plot because the position on the surface is determined by the two variables x, y and a third variable, z, define the contour lines. Despite the x, y, z nature of a contour plot, it is called two-dimensional for the same reason a map is two-dimensional. A three-dimensional plot is different in that it projects the z data out from the surface. Consider the following examples, which all plot the z(100, 60) matrix generated in the short code segments shown at right.

```
mxz = 100
nyz =60
redim z(mxz,nyz)
h1 = mxz/2.5
h2 = nyz/2.5
h3= mxz/1.75
h4 = nyz/1.75
istep3 =  3*mxz/100.
istep5 = 5*mxz/100.
istep10=10*mxz/100.
jstep6 = 6*nyz/80.
// if you have a long formula or expression you can use continuation
lines to make it more readible.  A continuation line is indicated by
ending the previous line with an underline(shift-dash on most
keyboards)
for i=1 to mxz
  for j=1 to nyz
    z(i,j)=1000*exp(-(pow((i-h1),2)+pow((j-h2),2))/500.0) _
    -1000*exp(-(pow((i-h3),2)+pow((j-h4),2))/500.0)  _
    + 50*cos(i/istep3) - 100*sin(j/jstep6) _
    +50*cos(i/istep10)*sin(i/istep5)
  next
next
```







The choice of plotting method depends upon the goal. If rigor is the key goal, the contour plot is the best choice. If beauty is the goal, then it is a matter of taste. These plotting methods can also be used to "plot" pictures, generating some interesting effects. For example, the following program loads a picture and then plots it using plot_2D options.

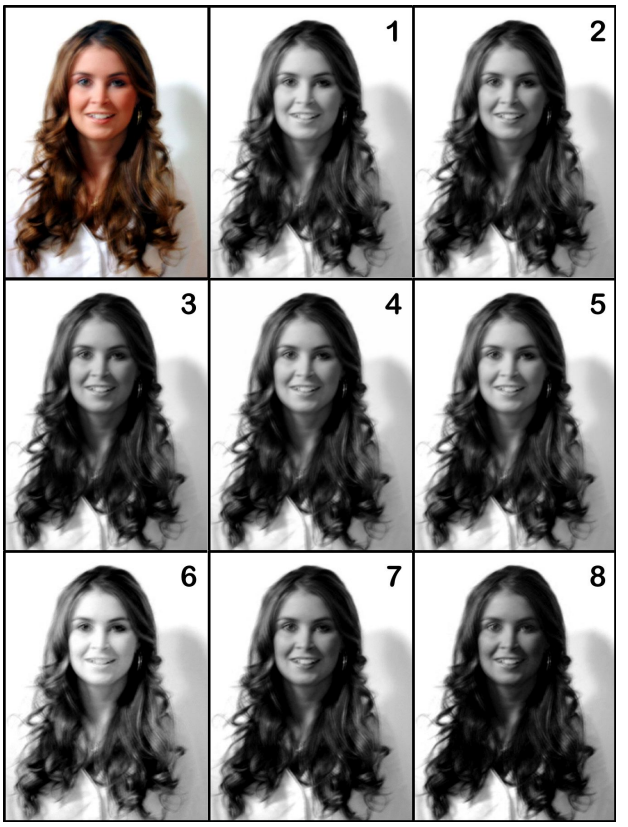
When the program at right is run, it generates the four pictures shown above. The picture in the upper-left quadrant is the original picture. The upper-right quadrant shows the same picture plotted using plot_2d_array() with ioption=100 (grey-scale) and imod=1 (no modular display). The lower-left picture uses color to display the intensity, and generates a false-color picture. Finally, the picture in the lower-right quadrant is created by using plot_contour to draw contours. This approach simulates an artists line drawing. The example also demonstrates the use of the command "set_to_graphics" which when executed, selects the graphics panel under program control.

```
// Program Name: template_picture_to_2D_array.txt
// we open a picture and display it using an apodized 2D display
// demonstrates multiple buffers.
// Main
dim filenumber,i,imod,ioption,j,k,nx,ny as integer
dim a2(1,1) as double
dim filename as string
dim Qappodize,Qfound as boolean
nx = 1200
ny = 800
redim a2(nx,ny)
set_graphics_slider(80)
buffer_create_multiple(-4,0,nx,ny) // create 4 buffers
filename = "famous_artists\caillebotte_rainy.jpg"
filenumber = 0
Qfound = open_user_picture_file(filenumber,filename)
if Qfound then
   set_to_graphics // move observer to the graphics panel
   show_progress_line("picture --> 1200x800 matrix")
   buffer_to_array(a2(),4) // buffer to 2D array
   buffer_copy_to_canvas(active_canvas,0)
   buffer_copy_to_buffer(2) // move to buffer 2
   buffer_copy_to_canvas(active_canvas,0)
   show_progress_line("Creating contour plot of the array")
   plot_contour(a2(),nx,ny,30,1,0)
   buffer_copy_to_buffer(4) // move to buffer 4
   buffer_copy_to_canvas(active_canvas,0)
   ioption=0 // color represents intensity
   imod=1
   plot_2d_array(a2(),nx,ny,ioption,imod,false)
   show_progress_line("Plotting the data based on intensity")
   buffer_copy_to_canvas(active_canvas,0)
   buffer_copy_to_buffer(3) // move to buffer 3
   buffer_copy_to_canvas(active_canvas,0)
   show_progress_line("Plotting the data in grayscale mode")
   ioption=100 // force grayscale
   plot_2d_array(a2(),nx,ny,ioption,imod,false) //
   buffer_copy_to_canvas(active_canvas,0)
   buffer_flip_buffers(1,2)
   buffer_copy_to_canvas(active_canvas,0)
   show_progress_line("Flipping the buffers")
else
   print("Could not find "+filename+", so we stop")
end if
// End Program
```

3.8.1. Color to Grey Scale Options

To convert a picture that has been loaded into the buffer into a single two-dimensional array, the following statement is used:

buffer_to_array(a2(), ioption)



which converts the pixels in the buffer to data in a2(1..nx, 1..ny) based on ioption (see example at right where the number at upper right is ioption and the original picture is in the upper left canvas).

If ioption=0, then each RGB value is given equal weight (not shown);
if ioption=1, then red is given double the weight;
if ioption=2, then green is given double the weight;
if ioption=3, then blue is given double the weight;
if ioption=4, then use Adobe standard (red*0.7, blue*0.89, green*0.41);
if ioption=5, then use grey scale (same as ioption 1 in reality);
if ioption=6, then use only the red channel;
if ioption=7, then use only the green channel;
if ioption=8, then use only the blue channel;
if ioption is negative, the picture is inverted using a weighting based on abs(ioption).
If there are multiple buffers, only the picture in buffer 1 is worked on. The array, a2(), is redimensioned by this statement to match the buffer size. Because the user has created the buffer, the size is known, but if more convenient, the user can always (re)discover the size by including the following statements following the buffer_to_array statement:

nx=**ubound**(a2, 1)
ny=**ubound**(a2, 2)

3.8.2. Working with Color Pictures

Scriptor provides bit level control of RGB images by using the buffer_to_arrays() and buffer_fill_from_arrays() methods:

buffer_to_arrays((ired(), igreen(), iblue())

where the red channel is stored into ired(1..buffer_width, 1..buffer_height), the green into igreen(1..buffer_width, 1..buffer_height) and the blue into iblue(1..buffer_width, 1..buffer_height). The three arrays are redimensioned to the size of the buffer. One can then manipulate these arrays and load them back into the buffer using:
buffer_fill_from_arrays(ired(), igreen(), iblue()).

```
Q = picture_create(1,nx,ny,false)
Q = open_user_picture_file(zero,filename,1)
for i=1 to 8
  picture_copy_to_buffer(1)
  buffer_to_arrays(ir(),ig(),ib())

  select case i
  case 1
    clear_arrays(ig(),ib())
  case 2
    clear_arrays(ir(),ib())
  case 3
    clear_arrays(ir(),ig())
  case 4
    sqrt_arrays(ir())
  case 5
    sqrt_arrays(ig())
  case 6
    sqrt_arrays(ib())
  case 7
    sqrt_arrays(ir())
    sqrt_arrays(ig())
    sqrt_arrays(ib())
  case 8
    square_arrays(ir())
    square_arrays(ig())
    square_arrays(ib())
  end select

  buffer_fill_from_arrays(ir(),ig(),ib())
  buffer_copy_to_buffer(i+1)
next
```

The program segment at upper left generates the collection at right (the creation of the multiple buffer and subroutines are not shown). The insert at upper left is the original picture. Inserts 1, 2 and 3 are generated by zeroing out all the arrays except red (1), green (2) and blue (3). Thus, these three examples simply show the individual components which make up the final, full color RGB picture. Inserts 4, 5, and 6 have the red, green and blue arrays replaced with their square root and then renormalized to yield the same max intensity. Insert 7 has all three channels handled in the same way

which decreases contrast. Insert 8 has the intensity in all three channels squared and then renormalized. This process increases contrast.

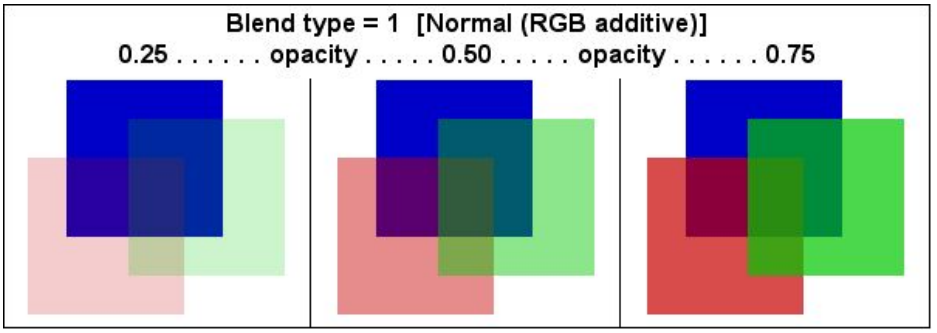
3.9. Color Blending and Variable Transparency

There are two types of transparency in graphics manipulations. Type I transparency is all or nothing. We investigated this concept in Section 3.7, where the white portions of the picture can be made invisible (transparent) so that a picture underneath is visible. Type I transparency is very fast because it is supported by the internal graphics commands. Type II transparency allows a graphics object to be partially transparent so that objects underneath are visible, but have their colors blended (modified) by the colors and transparency level of the top object. Those familiar with high-level graphics programs (Adobe Illustrator, ACD Systems Canvas X, Freeverse Lineform, etc.) are likely familiar with the nature of transparency. Transparency is specified not only by the opacity (0% is completely transparent, 100% is solid) but also the type of interaction between the colors of the overlapping objects. By convention the blend color is the original color of the object on top, the base color is the underlying color in the artwork and the resulting color is the color resulting from the blending process.

Transparency blending is implemented by invoking the following internal method, which modifies a single buffer pixel based on one of six blendtypes:

```
buffer_pixel_blend(kx, ky, opacity [, blendtype])=blend_color
```

where **kx** and **ky** reference the x and y pixel location within the buffer, **opacity** ranges from 0.0 (100% Transparency) to 1.0 (0% Transparency) and the optional integer parameter, **blendtype**, determines the method of blending. If this parameter is not present, a default value of 1 is assumed, and normal blending is carried out (see below). The **blend_color** is the color of the object that is on top and which is blended into the pixel color found at position **kx** and **ky**. This pixel at location **kx** and **ky** is thus replaced by the result of the blending process. The following paragraphs provide an overview of the various types of transparency blends. Each example was created by using a white background, a solid blue square of opacity=1, a red overlapping square with variable opacity followed by a green overlapping square with identical opacity. The displayed opacities are 0.25 (75% transparency, left), 0.5 (middle) and 0.75 (25% transparency, right).



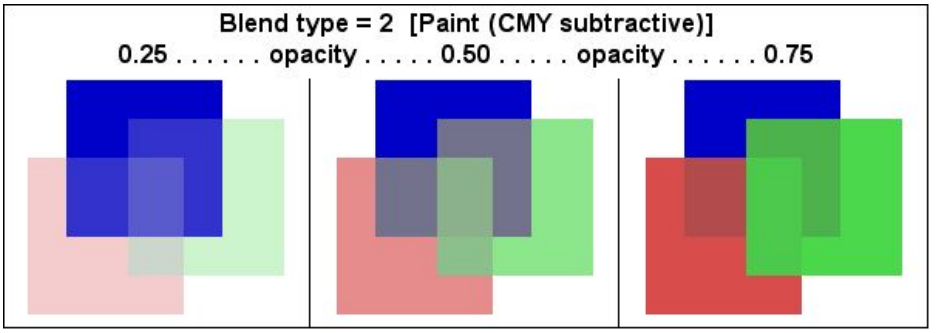
Normal (blend type = 1): The colors are blended as if the objects are transparent filters with lighting from underneath. This is the blending that the human visual cortex finds the most pleasing because it is familiar and expected, and provides intuitive depth perception. Thus, normal helps the eye figure out which objects are behind and which are in front. The formula that represents this process is:

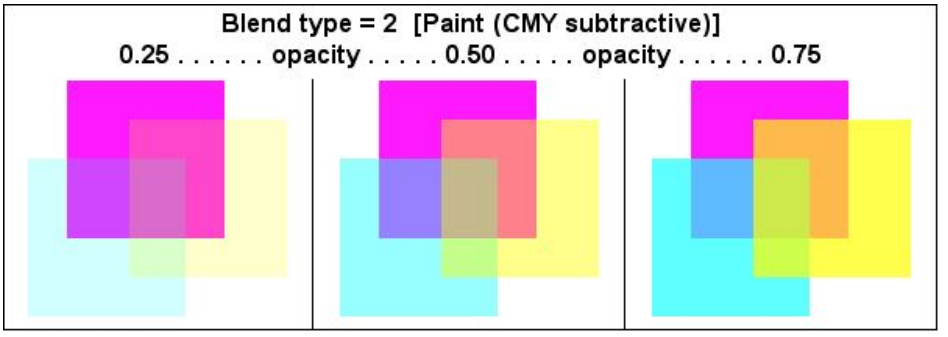
```
result.red   = opacity*blend.red   + (1 – opacity)*base.red
result.green = opacity*blend.green + (1 – opacity)*base.green
result.blue  = opacity*blend.blue  + (1 – opacity)*base.blue
```

where result is the color that results from blending the blend (front) color into the base (back) color. Each component is calculated separately using the same weighting scheme. A subroutine that carries out this transparency is presented below for demonstration purposes.

```
sub pixel_blend1(kx as integer, ky as integer, opacity as double, assigns c0 as color)
  ' reads the color of buffer pixel at kx, ky and blends the color c0
  ' into that pixel based on the opacity (0.0 to 1.0) using
  ' the adobe "normal" convention (weighted rgb averaging)
  dim jr, jg, jb as integer
  dim tp as double
  dim c1 as color
  tp=1.0-opacity // transparency equals one minus opacity
  c1=buffer_pixel(kx, ky) // get buffer BASE pixel color
// calculate RESULT color based on weighted RGB averages
  jr =  opacity*c0.red+tp*c1.red
  jg =  opacity*c0.green+tp*c1.green
  jb =  opacity*c0.blue+tp*c1.blue
  buffer_pixel(kx, ky)=rgb(jr, jg, jb)
end sub
```

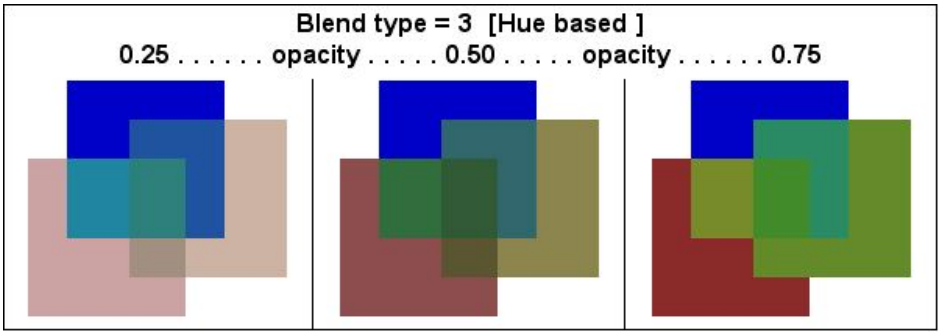
The above subroutine is presented for demonstration purposes. There is no need to use it because the internal buffer_pixel_blend routine implements the identical algorithm. Note that the blending math is done using the extensions .red, .green and .blue to rapidly extract the rgb components from a color. The template variable_transparency.txt can be used to explore transparency blends in more detail.





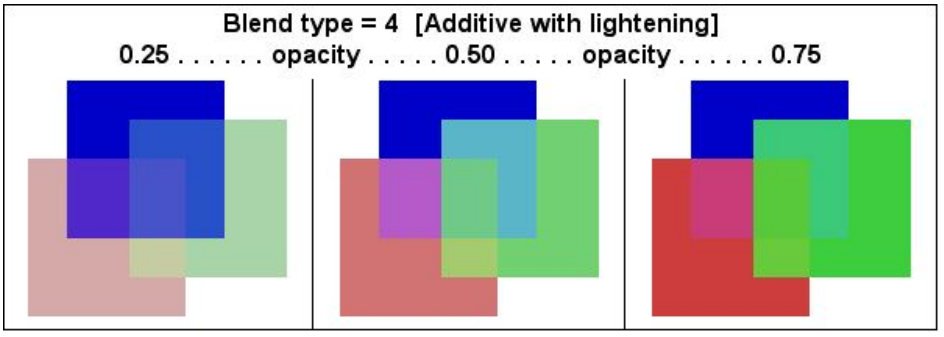
Subtractive Blending (blend type = 2). Normal blending simulates transparent filters with backlighting, and is the most natural of the blends. Subtractive blending simulates the properties of painting one or more objects on top of previous objects with variable opacities. The implementation is similar to that used by a color printer that uses three ink colors (cyan, magenta and yellow) to create a colored picture on a white background (white paper). However, because the inks have various levels of opacity, and the colors are by their nature added on top of each other, the result is different. The results are shown above for both RGB and CMY color sets. The algorithm is shown below:

```
tp=1.0-opacity
c1=buffer_pixel(kx, ky)
pc=    max(opacity*c0.cyan, tp*c1.cyan)
pm =   max(opacity*c0.magenta , tp*c1.magenta)
py =   max(opacity*c0.yellow , tp*c1.yellow)
buffer_pixel(kx, ky)=cmy(pc, pm, py)
```

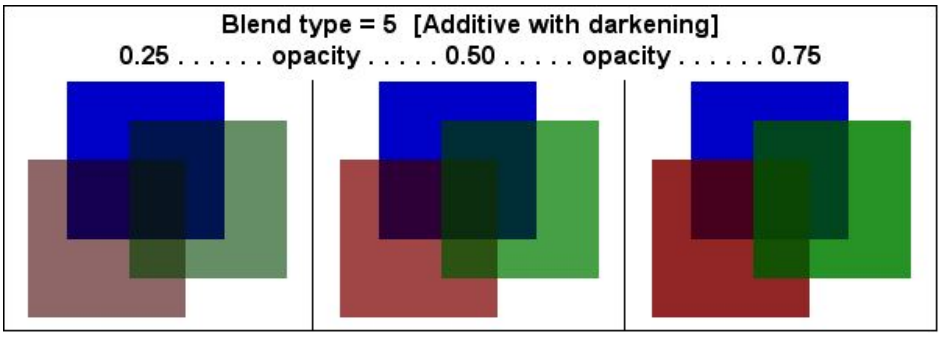


Hue-based Blending (blend type = 3). Averaging the hue while manipulating the saturation and opacity to enhance the overlap regions provides the artistic blending that is shown above. This type of blending is sometimes called hue-shifting, but it also involves manipulation of the saturation and value to enhance the visibility of the overlap regions.

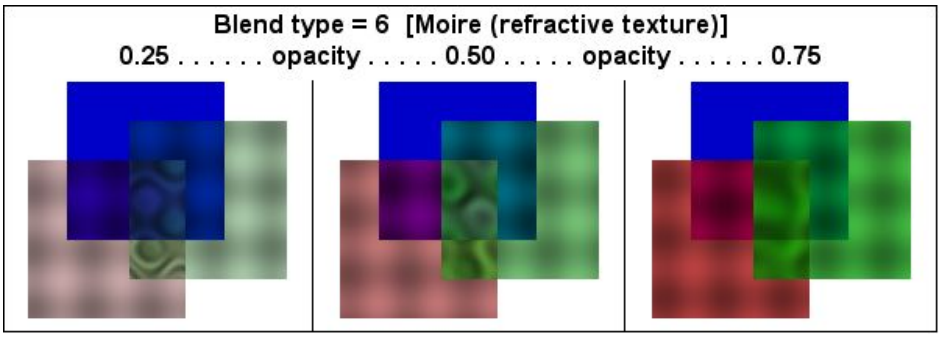
```
tp=1.0-opacity
c1=buffer_pixel(kx, ky)
ph =    opacity*c0.hue+ tp*c1.hue
ps =  max(opacity*c0.saturation, tp*c1.saturation)
v0=min(opacity*c0.value, tp*c1.value)
if v0>0.5 then
  pv=v0
else
  pv=max(opacity*c0.value, tp*c1.value)
end if
buffer_pixel(kx, ky)=hsv(ph, ps, pv)
```



Additive Blending with lightening (blend type = 4). If one carries out a simple RGB average as in normal blending, but lightens the overlap regions, one gets the results shown above. This kind of blending is more artistic than realistic, but should be considered when the number of overlapping objects is large and a majority have opacities greater than half.



Additive Blending with darkening (blend type = 5). If one carries out a simple RGB average as in normal blending, but darkens the overlap regions, one gets the results shown above. This kind of blending should be considered when the number of overlapping objects is large and a majority have opacities less than half. Using blend types of 4 or 5 is to be avoided if one can use blend type=1, the default, with satisfactory results. The default (blend type = 1) is more realistic and will provide the viewer with a better perspective on the objects.



Additive Blending with Moiré Refractive Texture **(blend type = 6).** If one carries out a simple RGB average as in normal blending, but mediates the resulting color with an artificial texture that simulates the refraction and interference of the light by two nearly parallel plates, one gets the interesting results shown above. The blending looks realistic to the eye, is accurate in terms of resultant color, and the moiré patterns help delineate the overlapping regions.

Chapter 4
Numerical Methods

MathScriptor provides a significant number of internal functions designed for numerical methods. The purpose of this chapter is to provide a listing of those internal functions that are relevant to numerical methods, as well as a brief discussion of usage. Many of the functions described in this chapter require MathScriptor mode, which means the program must be registered and MathScriptor mode selected under the compiler menu.

4.1. Matrix Methods

Before listing all of the matrix methods, it is important to provide a perspective on MathScriptor for those users who have experience with MatLab. Matrices in MathScriptor are explicit, not implicit, and manipulation must be carried out on the individual elements. Thus, if **A** and **B** are both matrices, you cannot use a statement like the following:

$$\mathbf{C = A*B}$$

to multiply matrix **A** and **B**. However, this operation can by done in a single operation by using a function, e.g.

```
Function matmult(a2(,) as double,b2(,) as double) As double(,)
  // multiply c2(i,k) = sum(j)[a2(i,j)*b2(j,k)]
  dim i,j,k as integer
  dim m1,m2,n1,n2 as integer
  dim c2(1,1) as double
  m1=ubound(a2,1)
  m2=ubound(a2,2)
  n1 = ubound(b2,1)
  n2 = ubound(b2,2)
  redim c2(m1,n2)
  for i=1 to m1
    for j=1 to n2
      for k=1 to n1
        c2(i,j)=c2(i,j)+a2(i,k)*b2(k,j)
      next
    next
  next
  return c2()
End Function
// end program
```

This function is called by using the following:

```
c()=matmult(a(), b())
```

A few things to notice. First, a function can return an entire matrix by assigning a function to have type **As double(,).** Then the function must return the entire matrix using the statement **return** c2() [or if you prefer, just **return** c2]. Programmers are

invariably puzzled about the protocols of when to include versus omit a comma in the parentheses. For example, a comma was included in assigning the data type but not in the return statement. This seems to be inconsistent because both statements represent the identical matrix. The rule is as follows: Only include a comma when the matrix is first being defined, and never again. Thus, in the function definition, the properties of the return matrix were defined. The compiler, when it finds the return statement, already knows by the function definition that a two-dimensional matrix is to be returned. The use of "(,)" following the variable is redundant. The compiler generates a syntax error which is handled by either removing the comma, or the comma and parentheses. The clever reader will recognize that the parentheses are also redundant, and by these rules, should also generate a syntax error. However, parentheses are ignored by the compiler when they are redundant.

The following matrix methods are available in MathScriptor. The subroutines return the matrix results as a ByRef parameter, whereas the functions return the matrix via assignment. For example, b2=matinv(a2) or matrix_invert(a2, b2) both invert a2(,) and place the result in b2(,). In most cases matrices can be used without including the parens when the entire matrix is to be passed or is to be assigned.

matdup(a2()) **as double(,)**
copy the matrix a2(n1, n2) into a new matrix b2(n1, n2).
Usage example: b2 = matdup(a2()) where b2 is a two-dimensional array.
Note that the (0, 0), (0, 1), (1, 0) elements are included in duplication. It is essential that the a2(,) matrix be dimed or redimed to the desired size prior to calling matdup.

matidn(nsize) **as double(,)**
initialize a two-dimensional identity matrix of dimension nsize by nsize with all elements equal to zero except diagonals which equal 1. Usage example: a2=matidn(10) where a2 is a two-dimensional array. Note that the a2(0, 0) element is also created and set to one.

matinv(a2()) **as double(,)**
invert the two-dimensional square matrix a2()
Usage example: b2=matinv(a2()) where b2 is a two-dimensional array. Note that the (0, 0), (0, 1), (1, 0) elements of a2 are ignored. It is essential that the a2(,) matrix be square and be dimed or redimed to the desired size prior to calling matinv.

matmult(a2(), b2()) **as double(,)**
multiply a2(n1, n2) by b2(n2, n3) to create c2(n1, n3). Usage example: c2=matmult(a2(), b2()). Note that the (0, 0), (0, 1), (1, 0) elements are ignored. It is essential that the a2(,) and b2(,) matrices be dimed or redimed to the correct sizes prior to calling matmult.

matrand(n1, n2) **as double(,)**
initialize a two-dimensional matrix of dimension n1xn2 with random elements from -1 to 1. Usage example: a2=matrand(10, 10) where a2 is a two-dimensional array.
Note that the (0, 0), (0, 1), (1, 0) elements of a2 are also created and randomized.

matrix_diagonalize(h2(), v2(), e1(), nsize, nroots)
diagonalize the square matrix h2(1..nsize, 1..nsize) to generate the lowest nroot solutions. Place the eigenvectors in v2(vector_number, root_number) and the eigenvalues in e1(1..nroots), e1(1) is the smallest or most negative eigenvalue.

matrix_gauss_jordan(a2(), b12(), nsize [, ms])
use Gauss Jordan elimination to solve the set of linear equations in the square matrix a2(1..nsize, 1..nsize) and replace it with its inverse and return the ms solution vectors (ms<=nsize) in the matrix b2(1..nsize, 1..ms). If ms=1 (a single solution vector), use a single array b1() and leave out last parameter. Less efficient than matrix_invert but provides solution vector(s).

matrix_invert(a2(), det, nsize)
use LU decomposition to replace the square matrix a2(1..nsize, 1..nsize) with its inverse and return the determinant in det. Provides the determinant but no solution vectors. This method is more efficient than matrix_gauss_jordan.

matrix_svd(a2(), v2(), w1(), m, n)
given the matrix a2(1..m, 1..n) compute its singular value decomposition. Upon return, the a2() matrix is replaced by the u2() matrix with the v2() matrix and the w1 diagonal elements are returned in the parameters so designated.

matrix_svd_backsubstitute(u2(), v2(), w1(), m, n, b1(), x1())
This subroutine is called after matrix_svd is executed and the values of u2(), v2() and w1() are exactly those returned in the a2(), v2(), w1() matrix_svd parameter sequence. What is needed first, however, is to go through the w1() array and zero out those that have a magnitude that is significantly smaller than the largest weight. Because all calculations are done in double precision, one can carry out this zeroing process with confidence whenever w1(k) is less than 10^{-6} of wmax. Some trial and error is required for values above this cut-off. Then one needs to supply a value for the b(1..m) vector to extract the x(1..n) vector, which is the desired result.

mattran(a2()) **as double(,)**
transpose the matrix a2(n1, n2) into the matrix b2(n2, n1). Usage example: b2=mattran(a2()) where b2 is a two-dimensional array. Note that the (0, 0), (0, 1), (1, 0) elements are included in transposition. It is essential that the a2(,) matrix be dimed or redimed to the exact size prior to calling mattran.

matzero(n1, n2) **as double(,)**
initialize a two-dimensional matrix of dimension n1xn2 with all elements equal to zero. Usage example: a2=matzero(10, 10) where a2 is a two-dimensional array.
Note that the a2(0, 0) element is also created and set to zero, and that the matrix created is redimensioned to n1 by n2 [redim a2(n1, n2)].

4.2. Singular Value Decomposition

Linear algebra is a field of mathematics that deals with the use of matrices to solve problems in both pure and applied math. Matrix decomposition is the process of factorizing a matrix into a more useful (canonical) form. The term canonical has many different meanings (normal, standard form, differential form), but in the present case, it means that each entry or component has a direct relationship to an observable. Thus, when a matrix is decomposed, the result is a matrix which has vectors or components that have been simplified so that the result is more easily evaluated or assigned to observables.

Singular value decomposition (SVD) is the most general, and powerful, of the decomposition methods. The method is used extensively within MathScriptor as the method of choice in fitting polynomials of all types. The significant advantage of SVD is that it helps eliminate degrees of freedom that are redundant or irrelevant, and prevent these additional degrees of freedom from damaging the quality of the analysis. Singular value decomposition (SVD) solves the matrix problem shown below:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} = \begin{pmatrix} u_{11} & \cdots & u_{1n} \\ \vdots & \ddots & \vdots \\ u_{m1} & \cdots & u_{mn} \end{pmatrix} \bullet \begin{pmatrix} w_1 & & 0 \\ & \ddots & \\ 0 & & w_n \end{pmatrix} \bullet \begin{pmatrix} v_{11} & \cdots & v_{1n} \\ \vdots & \ddots & \vdots \\ v_{n1} & \cdots & v_{nn} \end{pmatrix}^T$$

where the T indicates the transpose of matrix v(1..n, 1..n). In our adaptation, the diagonal matrix **w** is returned as a one-dimensional array containing the diagonal elements. An alternative view of the above matrix equation is the following:

$$A_{ij} = \sum_{k=1}^{n} w_k U_{ik} V_{jk}$$

SVD takes the matrix A and decomposes it into a set of singular values with weighting factors w_k. One way of viewing SVD is a method of solving a set of m linear equations involving n unknowns. There is no more reliable method of doing a linear least squares fit, and SVD is used in many of the internal fit methods. SVD is also used extensively to extract component spectra from time resolved spectra. In many cases, SVD will yield a set of weighting factors dominated by only a small number of weighting factors, w(), with the other weighting factors near to zero. Then it is important to zero out these values and use backsubstitution to extract vectors that are physically realistic and not contaminated by noise.

The **matrix_svd_backsubstitute()** method is called after the matrix_svd is run and the values of U(), V() and W() are exactly those returned in the A(), V(), W() matrix_svd parameter sequence. Note that you do not need to take the transpose of v(). What is needed first, however, is to go through the w() array and zero out those that have a magnitude that is significantly smaller than the largest weight. Because all calculations are done in double precision, you can carry out this zeroing process with confidence whenever w_k is less than 10^{-6} of w_{max}. Some trial and error is required for values above this cut-off. Then you need to supply a value for the b(1..m) vector to extract the x(1..n) vector, which is the desired result:

$$x = V \bullet \begin{pmatrix} w_1 & & 0 \\ & \ddots & \\ 0 & & w_n \end{pmatrix} \bullet U^T \bullet b$$

There are times when it is not advisable to zero out any of the w's, regardless of relative magnitude. It is sometimes better to simply allow the small values to propagate through, and for that reason, it is important to test both options and determine which one yields the best solution. Small values of w_i usually signal that the working equation has too many degrees of freedom and requires modification. Using the weights to adjust the working equation often yields a far superior result than simply zeroing out the weights below a given threshold.

The subroutine at right demonstrates the use of SVD to carry out a least squares fit based on a polynomial function. But this same method can be used to fit any linear function and all that is needed is to replace the function poly() with another which returns afunc(1..ma) as a function of the x(i) values. Any linear function is allowed, which means that afunc(i)=x(i)^nexp(i), where nexp(i) is a positive or negative integer.

Going through this subroutine in detail should provide the interested reader with an excellent perspective on the meaning of the individual vectors and matrices that are used in the SVD methods. Note that this subroutine is part of a module which declares the following variables

u(ndata, nparams)
v(nparams, nparams)
w(nparams)

as doubles, where ndata is the number of data pairs [x(1..ndata), y(1..ndata)].

Note that both matrix_svd() and matrix_svd_backsubstitute() are used in this subroutine.

The subroutine also returns calculated values of y in the array ycalc(1..ndata), where each value returns the value ycalc(i) = a(1)+a(2)*x(i) +a(3)*x(i)^2+… a(ma)*x(i)^ndegree.

```
private sub svdfit(x() as double, y() as double, _
  sigma() as double, ndata as integer,  a() as double, _
  ma as integer, byref chisq as double,ycalc() as double)
  // this subroutine does an SVD fit using a polynomial of degree
  ndegree the polynomial is evaluated in the subroutine poly(...) the
  data points to which the polynomial is to be fit are in x(1:ndata),y
  (1:ndata) and the standard deviations are in sigma(1:ndata) if there
  are no standard deviation data, set all sigma(i)=1 the dimensions of
  u(mp,np), v(np,np), w(np) are such that mp must be >= ndata, the
  number of data points np must be >= number of parameters fit
  (normally much smaller than mp) the equation to be fit is a(1)+a(2)*x
  + a(3)*x^2 + ... + a(ndegree+1)*x^ndegree
  dim i,j,ndegree as integer
  dim sum,thresh,tmp,wmax,b(1),afunc(1) ,tol  as double
  redim b(ndata+1)
  tol = 1.0E-12
  // ma = number of parameters to fit
  ndegree = ma-1
  redim afunc(ma)
  // accumulate coefficients of the matrix describing the fit into u(i,j)
  iterate over each data pair x(i),y(i) with sigma(i) evaluate the naked
  polynomial at x(i) returning the values in afunc() such that afunc(1)
  =1, afunc(2)=x(i), afunc(3)=x(i)^2, to afunc(ma)=x(i)^ndegree
  for i=1 to ndata
    poly(x(i),afunc(),ma)
    tmp=1.0
    // sigma(i) is the error in y(i).  The variable tmp weights each value
    by the inverse of the uncertainty set all sigma(i) to 1.0 for
    unweighted fit
    if sigma(i)>0.0 then tmp = 1.0/sigma(i)
    // generate the columns of u(i,j)
    for j=1 to ma
      u(i,j) = afunc(j)*tmp
    next
    b(i) = y(i)*tmp
  next
  matrix_svd(u(),v(),w(),ndata,ma)
  wmax=0.0
  for j=1 to ma
    wmax = max(w(j),wmax)
  next
  thresh =  tol*wmax
  for j=1 to ma
    if w(j)<thresh then
      // set w(j) to zero if below threshold
      w(j)=0.0
    end if
  next
  matrix_svd_backsubstitute(u(),v(),w(),ndata,ma,b(),a())
  chisq=0.0
  for i=1 to ndata
    poly(x(i),afunc(),ma)
    sum=0.0
    for j=1 to ma
      sum = sum+a(j)*afunc(j)
    next
    ycalc(i)=sum
    chisq = chisq + pow(((y(i)-sum)/sigma(i)),2)
  next
end sub
```

The covariance matrix can be generated by calling the subroutine svdvar() which places the covariance matrix into the matrix cvm(1..nparams, 1..nparams). The error in each of the coefficients is given by the square root of the diagonal elements:

$a_j = a(j) \pm \text{sqrt}(cvm(j, j))$

The covariance matrix provides a quantitative measure of the relationship between the individual coefficients. Consider the two coefficients a(r) and a(s) and the covariant matrix elements connecting them, c_{rs} = cvm(r, s). If c_{rs} is zero, it means the two coefficients are invariant to changes in the other one. For a fit, this is a good situation. If c_{rs} is positive, then a change in a(r) will generate a corresponding change in a(s) in the same direction. Conversely, if c_{rs} is negative it means as a(r) increases, a(s) decreases, and vice versa. The larger the value, the more correlated is the correspondence. If two coefficients are highly correlated, it is likely that both are not needed and the function can be simplified. High correlation also means it is less likely that the values of the coefficients can be assigned with accuracy.

```
private sub svdvar(ma as integer, cvm(,) as double)
  // this subroutine evaluates the covariance matrix which is returned
  in cvm(1..ma,1..ma). The error in the jth coefficient is equal to +/-
  sqrt(cvm(j,j))
  dim i,j,k as integer
  dim wti(1),sum as double
  redim wti(ma)
  for i=1 to ma
    wti(i)=0.0
    if w(i)>0.0 then wti(i)=1.0/(w(i)*w(i) )
  next
  for i=1 to ma
    for j=1 to i
      sum=0.0
      for k=1 to ma
        sum = sum + v(i,k)*v(j,k)*wti(k)
      next
      cvm(i,j)=sum
      cvm(j,i)=sum
    next
  next
end sub
```

4.3. Fitting Methods

MathScriptor provides a number of very powerful fitting routines based on singular value decomposition. These methods provide for very rapid fitting of linear functions to a data set, and all use SVD to provide accurate and stable fits.
The following standard polynomial fits are available:

fit_polynomial(x(), y(), ndata, a(), nparams, rmserror, rsquared)
fits the data in the x(1..ndata), y(1..ndata) arrays using SVD to assign the parameters a(1..nparams) via least squares regression. The equation to be fit is:

$$y = a(1)+a(2)*x + a(3)*x^2 + a(4)*x^3 + \ldots a(\text{nparams})*x^{(\text{nparams}-1)}$$

Fit_lanczos(x(), y(), ndata, a(), nparams, rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to a Lanczos-type polynomial with nparams parameters returned in a(1..nparams):

$$y = a(1)+a(2)/x + a(3)/x^2 + a(4)/x^3 + \ldots a(nparams)/x^{(nparams-1)}$$

The RMS (root-mean-squared) error is returned in rmserror. The function that is fit is credited to the Hungarian mathematician and physicist, Cornelius Lanczos, who made many contributions to relativity theory and numerical methods. This function was originally proposed as a method of fitting the loggamma function, but has now been generalized to any fit of this type. The name Lanczos is pronounced "lan-sosh".

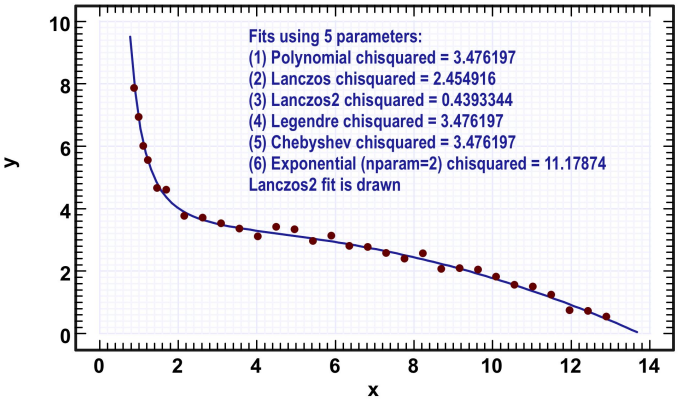
Fit_lanczos2(x(), y(), ndata, a(), nparams, rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to the non-standard (or mixed) Lanczos-type polynomial

$$y=a(1)+a(2)/x+a(3)*x+a(4)/x^2+a(5)*x^2+a(6)/x^3+a(7)*x^3 + \ldots$$

The programmer can evaluate this polynomial by using the function lanczos2(iterm, x) as follows:

```
for j=1 to nparams
ypi=ypi+a(j)*lanczos2(j, x0)
next
```

lanczos2(i, x) **as double**; returns the ith term of the type2 Lanczos polynomial for argument x.



There are times when a Lanczos2 fit is significantly better than all the others. One such example is shown in the figure at right. In general, any function that appears to be a mixture of an exponential that approach infinity as x approaches 0, but has a broad tail that extends to high x, is a perfect candidate for a Lanczos2 fit. Such are often encountered in science and engineering.

4.3.1. Fitting to Orthogonal Polynomials

Polynomials $p_m(x)$ and $p_n(x)$ are said to be orthogonal over the range x=a to x=b if they obey the following integral relationship:

$$\int_a^b w(x)\,p_m(x)\,p_n(x)\,dx = 0 \quad (m \neq n)$$

where w(x) is a weighting function, normally assumed to be unity. Fitting a data set to orthogonal polynomials instead of standard polynomials has significant advantages in certain circumstances. The key advantage is that truncation error is reduced significantly. Second, in some cases, fewer parameters are necessary to achieve a given goodness of fit. Third, there are situations in science and engineering where a solution in terms of orthogonal polynomials can be transferred to a physical solution in a straight-forward fashion. One example is the design of electronic filters based on the fit of the desired response function to Chebyshev polynomials.

MathScriptor includes two internal orthogonal polynomials which can be calculated or used in least-squares fits. The simplest orthogonal polynomial is the Legendre polynomials given by the function (m = 0, 1, 2, …):

$$P_m(x) = \frac{1}{2^m\, m!}\frac{d^m}{dx^m}\left[\left(x^2-1\right)^m\right]$$

and provided by the function **legendre**(n, x) **as double,** where n = m+1 = 1, 2, 3, … and x is the argument. The first ten functions are listed below:

n	
1	$P_0 = 1$
2	$P_1 = x$
3	$P_2 = \frac{1}{2}(3x^2 - 1)$
4	$P_3 = \frac{1}{2}(5x^3 - 3x)$
5	$P_4 = \frac{1}{8}(35x^4 - 30x^2 + 3)$
6	$P_5 = \frac{1}{8}(63x^5 - 70x^3 + 15x)$
7	$P_6 = \frac{1}{16}(231x^6 - 315x^4 + 105x^2 - 5)$
8	$P_7 = \frac{1}{16}(429x^7 - 693x^5 + 315x^3 - 35x)$
9	$P_8 = \frac{1}{128}(6435x^8 - 12012x^6 + 6930x^4 - 1260x^2 + 35)$
10	$P_9 = \frac{1}{128}(12155x^9 - 25740x^7 + 18018x^5 - 4620x^3 + 315x)$

and shown in Figure 4.3.1. The other orthogonal polynomial is the well-known Chebyshev polynomial given by the generating equations:

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

and the internal function that provides the polynomials is given by **chebyshev**(np, x) **as double,** where np = n+1 = 1, 2, 3, … The first seven solutions are listed below:

Chebyshev(1,x) = $T_0(x) = 1$
Chebyshev(2,x) = $T_1(x) = x$
Chebyshev(3,x) = $T_2(x) = 2x^2 - 1$
Chebyshev(4,x) = $T_3(x) = 4x^3 - 3x$
Chebyshev(5,x) = $T_4(x) = 8x^4 - 8x^2 + 1$
Chebyshev(6,x) = $T_5(x) = 16x^5 - 20x^3 + 5x$
Chebyshev(7,x) = $T_6(x) = 32x^6 - 48x^4 + 18x^2$

These polynomials are compared to the Legendre polynomials in Fig. 4.3.1. The following two functions are available:

Fit_chebyshev(x(), y(), ndata, a(), nparams, rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to orthogonal Chebyshev polynomial with weights returned in a(1..nparams). The following program segment generates a value of y (ypi) for a given value of x (x0):

```
for j=1 to nparams
ypi=ypi+a(j)*chebyshev(j, x0)
next
```

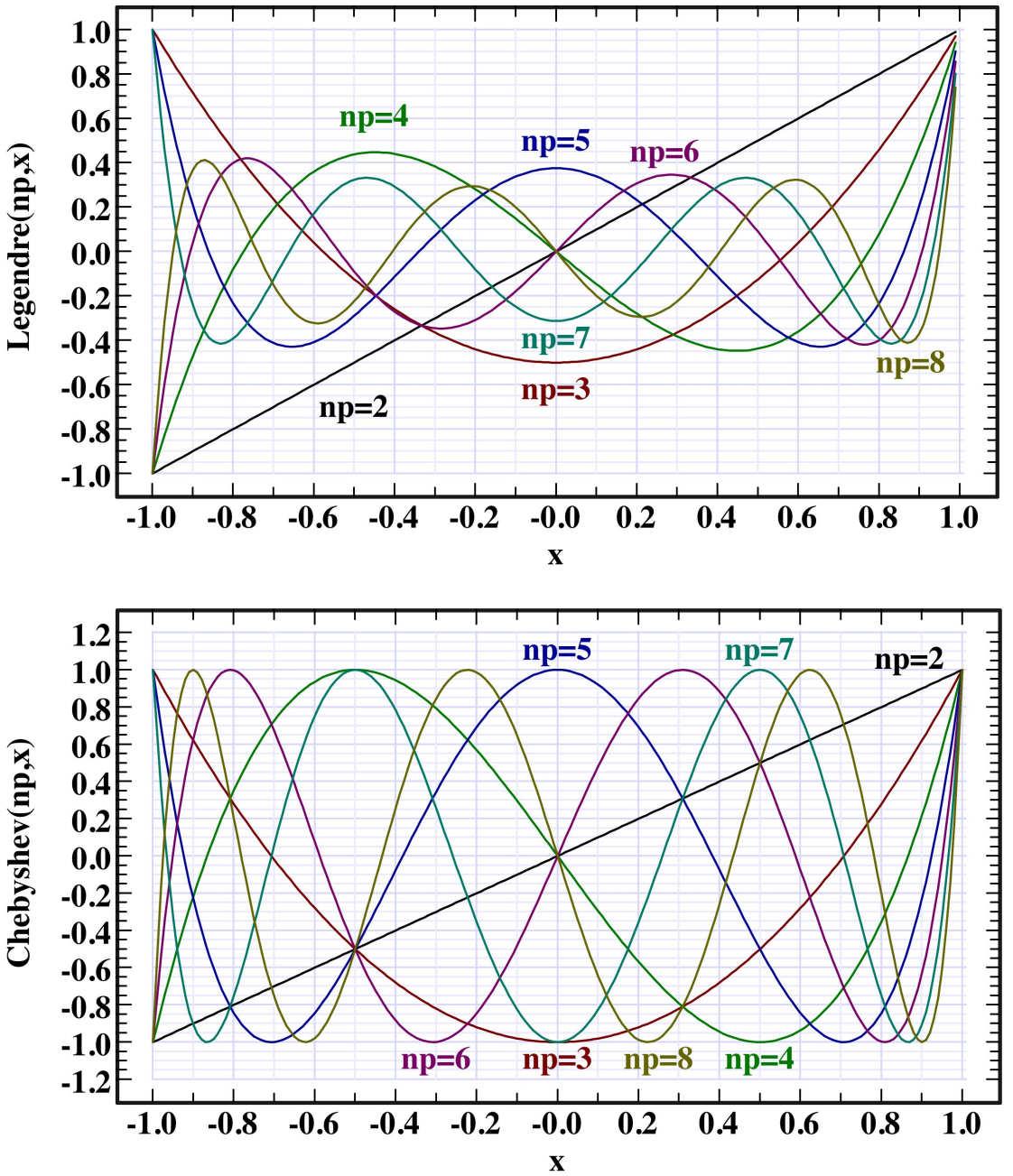


Figure 4.3.1. Comparison of the Legendre (top) and Chebyshev (bottom) polynomials for np = 2 – 8, where np is the argument for the functions Legendre(np, x) and Chebyshev(np, x).

Fit_legendre(x(), y(), ndata, a(), rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to orthogonal Legendre polynomial with weights returned in a(1..nparams). The following program segment generates a value of y (ypi) for a given value of x (x0):

```
for j=1 to nparams
ypi=ypi+a(j)*legendre(j, x0)
next
```

We should clear up one common source of confusion now. Although these functions are specified to be orthogonal only for the x region from –1 to 1, these polynomials are actually nearest neighbor orthogonal over any symmetric region centered at zero. Furthermore, one can use these functions to fit a data set over any range desired if one is willing to give up some of the advantages inherent in using orthogonal polynomials. If, on the other hand, it is important to optimize the fit based on an orthogonal basis set, the solution is to transpose the data so that the fit is carried out over the range –1 to 1. This should be done linearly so that:

$$x_{new} = a_{tran} + b_{tran} * x_{old}$$

In practice, this rarely is worth doing because the SVD routines that are used to carry out the fitting are very stable and well-behaved.

4.3.2. Other Fitting Options

There are a number of additional fitting routines available in MathScriptor and these are presented below:

Fit_exponential(x(), y(), ndata, a(), rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to the equation y = a(1)*exp(a(2)*x). Note that unlike the other linear fits, nparams is not a parameter because it is fixed at 2. The chi-squared goodness of fit is returned in chisqr.

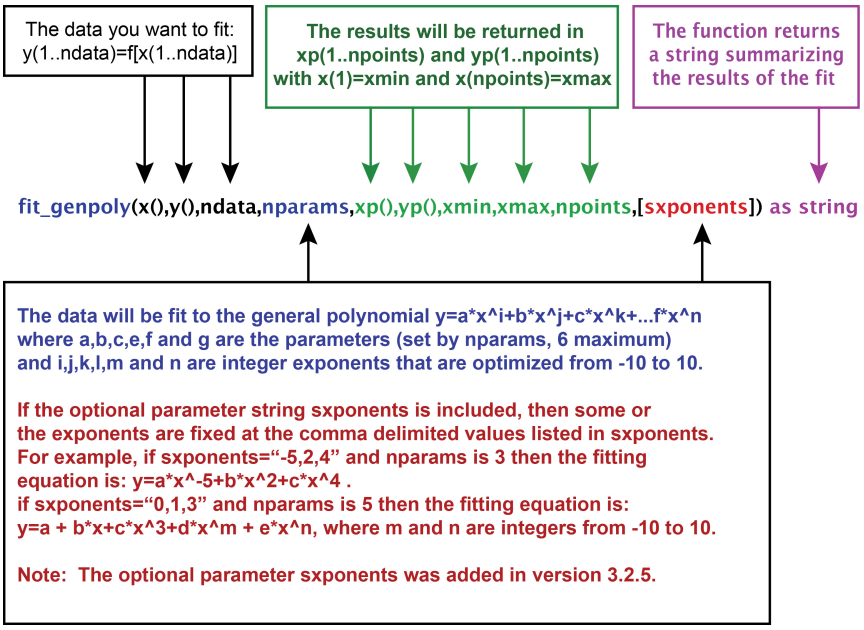
Fit_genpoly(x(), y(), ndata, a(), b(), nparams) **as string**
returns a string summary of the best fit general polynomial for the data x(1..ndata), y(1..ndata) using a maximum of nparam fit parameters. Nparam must be less than 6. The equation that is fit is a(1)*x^b(1) + a(2)*x^b(2)+ ... +a(nparams)*x^b(nparams), where b(1)..b(nparams) are integers in the range -10 to 10. This function is overloaded and exists in two forms. Above method returns the coefficients instead of the best fit line. Both forms are iterative and are CPU intensive.

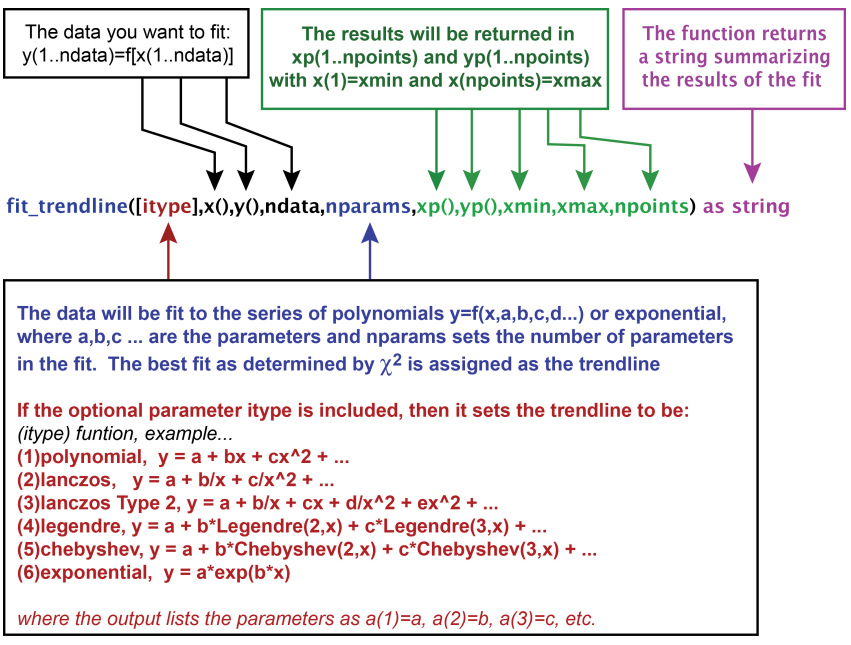
An alternative (overloaded) form is as follows:

Fit_genpoly(x(), y(), ndata, nparams, xp(), yp(), xmin, xmax, npoints) **as string**
returns a string summary of the best fit general polynomial for the data x(1..ndata), y(1..ndata) using a maximum of nparam fit parameters. Nparam must be less than 6. The equation that is fit is a(1)*x^b(1) + a(2)*x^b(2)+ ... +a(nparams)*x^b(nparams), where b(1)..b(nparams) are integers in the range -10 to 10. This function then returns calculated values in xp(1..npoints), yp(1..npoints) for x in the range xmin to xmax.

Fit_trendline([itype], x(), y(), ndata, nparams, xp(), yp(), xmin, xmax, npoints) **as string** returns a string summary of the best fit trendline for the data x(1..ndata), y(1..ndata) using a maximum of nparam fit parameters. This function then returns calculated values in xp(1..npoints), yp(1..npoints) for x in the range xmin to xmax. If itype is included, it selects the trendline fit to be: (1)polynomial, (2)lanczos, (3)lanczos2, (4)legendre, (5)chebyshev, (6)exponential.

The last two functions are the most useful of the fitting functions as they cover all the options and return a set of predicted values ready for plotting in xp(1..npoints), yp(1..npoints). Although genpoly uses SVD, it is also iterative and therefore slow. A full nparams=5 fit can take many minutes. The following graphics provide a convenient summary of these functions. These are also available inside the program via the Tips menu.


The data you want to fit:
y(1..ndata)=f[x(1..ndata)]
The results will be returned in
xp(1..npoints) and yp(1..npoints)
with x(1)=xmin and x(npoints)=xmax
The function returns
a string summarizing
the results of the fit
fit_genpoly(x(),y(),ndata,nparams,xp(),yp(),xmin,xmax,npoints,[sxponents]) as string
The data will be fit to the general polynomial y=a*x^i+b*x^j+c*x^k+...f*x^n
where a,b,c,e,f and g are the parameters (set by nparams, 6 maximum)
and i,j,k,l,m and n are integer exponents that are optimized from -10 to 10.
If the optional parameter string sxponents is included, then some or
the exponents are fixed at the comma delimited values listed in sxponents.
For example, if sxponents="-5,2,4" and nparams is 3 then the fitting
equation is: y=a*x^-5+b*x^2+c*x^4 .
if sxponents="0,1,3" and nparams is 5 then the fitting equation is:
y=a + b*x+c*x^3+d*x^m + e*x^n, where m and n are integers from -10 to 10.
Note: The optional parameter sxponents was added in version 3.2.5.


The data you want to fit:
y(1..ndata)=f[x(1..ndata)]
The results will be returned in
xp(1..npoints) and yp(1..npoints)
with x(1)=xmin and x(npoints)=xmax
The function returns
a string summarizing
the results of the fit
fit_trendline([itype],x(),y(),ndata,nparams,xp(),yp(),xmin,xmax,npoints) as string
The data will be fit to the series of polynomials y=f(x,a,b,c,d...) or exponential,
where a,b,c ... are the parameters and nparams sets the number of parameters
in the fit. The best fit as determined by χ2 is assigned as the trendline
If the optional parameter itype is included, then it sets the trendline to be:
(itype) funtion, example...
(1)polynomial, y = a + bx + cx^2 + ...
(2)lanczos, y = a + b/x + c/x^2 + ...
(3)lanczos Type 2, y = a + b/x + cx + d/x^2 + ex^2 + ...
(4)legendre, y = a + b*Legendre(2,x) + c*Legendre(3,x) + ...
(5)chebyshev, y = a + b*Chebyshev(2,x) + c*Chebyshev(3,x) + ...
(6)exponential, y = a*exp(b*x)
where the output lists the parameters as a(1)=a, a(2)=b, a(3)=c, etc.

4.3.3. Special Fitting Functions

There are two fitting functions that are available in MathScriptor which solve unique fitting goals.

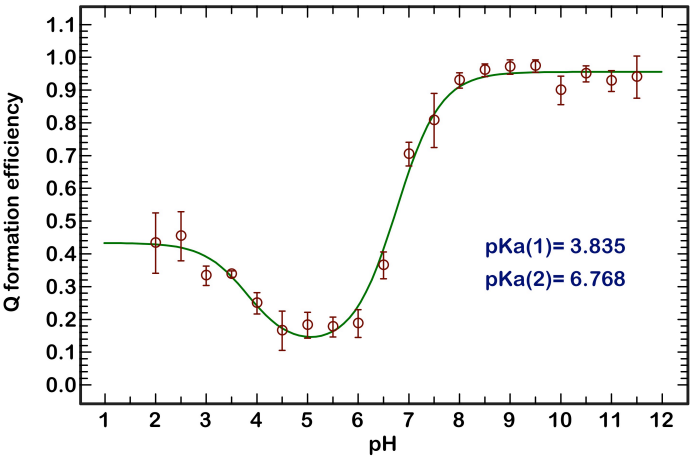
fit_henderson_hasselbalch(pH(), y(), n, a(), pka(), nterms, xp(), yp(), x1, x2, np) **as string** carries out a one (nterms=1), two (nterms=2) or three (nterms=3) term Henderson Hasselbalch fit to some numerical property of an ionizable molecule or protein. The measurements in y(1..n) as a function of pH(1..n) are fit to the equation:

$$\text{Protein Property} = a_0 + \frac{a_1}{1+10^{(pH-pK_a^{(1)})}} + \frac{a_2}{1+10^{(pH-pK_a^{(2)})}} + \frac{a_3}{1+10^{(pH-pK_a^{(3)})}}$$

where the full three term expression is shown. The function returns the fit parameters in a(0..2) and pka(1..2) and the result in xp(1..np) and yp(1..np) from x1 to x2 where np is the number of points which should be set by the user. If np=0, no curve is returned. The function returns a string summarizing the results of the fit. An example of a two component fit is shown below along with the text output.

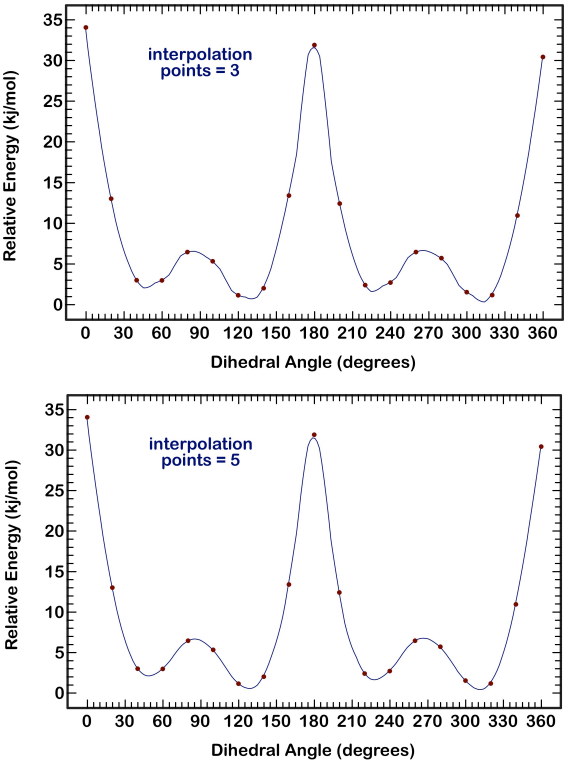
```
Fit to a0+a1/(1+10^(ph-
pka1))+a2/(1+10^(ph-pka2))

rmserror*nparams(=5) =
0.1974552
Renormalized to original Y
values:
a0=0.9534774
a1=0.3214927
pka1=3.83506
a2=-0.8434066
pka2=6.767612
```



Numerical_fit_to_gaussians(xkk(), y(), n, nbands, niter, xp(), yp(), x1, x2, np [, sharpen]) **as string** is a method that takes spectral data in wavenumber space and fits it to a collection of Gaussian bands. The spectrum should be normalized in y(1..n) and must have the energy axis, xkk(1..n) be in kiloKaysers (kK = 1000 cm^{-1}). The initial number of Gaussians is set by nbands and one should seek to use a minimum number of Gaussians to fit the spectrum. Trial and error is necessary. The total number of iterations is set by niter where niter=500 is usually adequate. The spectrum based on the sum of gaussians is returned in xp(1..np) and yp(1..np) from x1 to x2, which also must be in kK. If the optional parameter sharpen is included, the spectrum in xp(), yp() can be sharpened (0.0<sharpen <0.75) or broadened (-0.9<sharpen <0.0). The string that is returned is a program that can be used to regenerate the spectrum and includes a list of all the Gaussian bands.

Numerical_interpolate_points(x(), y(), n, xp(), yp(), byref np, nlevel) This method uses Lagrangian interpolation to transfer the sparse x(1..n), y(1..n) arrays into dense xp(1..np), yp(1..np) arrays which permits smooth plotting the results. The level of the interpolation is set by nlevel (=2, 3, 4 or 5). Note that the number of points in the returned arrays may be modified slightly to accommodate the Lagrangian fit, and hence the np variable is passed by reference and should be examined before plotting the result. The examples shown at right demonstrate two and five point interpolation. In general, five point interpolation is recommended, but if the data in x() and y() are very sparse, lower values of nlevel may be necessary to avoid false maxima.



Numerical_fraction(f1, ndigits) **as string** converts a real number to its equivalent fraction, or a fraction that reproduces the number to an accuracy of ndigits. The string that is returned includes the fraction as well as the expansion of the fraction to a precision of 32 digits. A maximum of 15 digits of precision is allowed. An example of finding a fraction that approximates π is shown below:

```
s1=numerical_fraction(const_pi, 8)
print(s1)
93343/29712 =
3.14159262250942380183091007000054
```

Numerical_generate_expression(target, ndepth) **as string** return an expression, f(...), that provides the best fit to the target. This function not only assigns the function but the values of parameters of that function, and minimizes the error given by abs(f(...)-target)/abs(target). The value of ndepth controls the depth of the search and assigns the maximum value for the integers in the expressions. If ndepth is negative, the depth of search is equal to abs(ndepth) but the best expression for each expression is listed for comparison. This function is useful whenever a calculation yields a value which you suspect is represented by one of the above expressions. Values of ndepth=15 provide a fairly rapid search and are the minimum ndepth used regardless of input. Ndepth values above 15 take progressively longer such that ndepth=100 will take many hours. The target should be in the range ±(0.01 to ndepth) for this method to explore all of the expressions listed above. An example is shown below revealing the expression responsible for a complex variable analysis. The number in curly-brackets "{}" at the end of each expression gives the degrees of freedom. If two results yield identical error, one should choose the expression with the smallest degrees of freedom. For the convenience of the user, the result for numerical_fraction(target, 8) is printed at the bottom of the output.

```
target=value(real(pow("0, 1", "0, 1"))) // i^i
s1=numerical_generate_expression(target, -15)
print(s1)

Best result for target = 0.2078795763507619
exp((-1/2)*const_pi) = 0.207879576350762 (err~0)[0.54s]{8}

Options for target = 2.078795763507619E-1
exp((-1/2)*const_pi) = 0.207879576350762 (err~0)[0.54s]{8}
(3/11)*log((7/2)*const_pi^(-3/7)) = 0.207863030724132 (err=7.959E-5)[0.14s]{34}
9*sin(const_pi/136) = 0.20788102487701 (err=6.968E-6)[0.45s]{147}
-9*cos(69*const_pi/136) = 0.207881024877009 (err=6.968E-6)[0.45s]{215}
(1/8)*const_pi^(4/9) = 0.207905276692377 (err=1.236E-4)[0.05s]{23}
(5/2)*(factorial(2)/factorial(4)) = 0.208333333333333 (err=2.183E-3)[0.05s]{15}
(11/10)*sqrt(1/28) = 0.207880460155075 (err=4.252E-6)[0.05s]{51}
... [search time in seconds] {degrees of freedom}
For reference, numerical_fraction(target, 8) returns 4944/23783 =
0.20787957784972459319682125888239 (err=7.211E-9)
```

4.3.4. On the Goodness and Quality of a Fit

The commonly defined measure of goodness of fit is chisqr:

$$X^2 = \sum_{i=1}^{n} \frac{\left(y_i^{obsvd} - y_i^{calc}\right)^2}{\sigma_i} = \sum_{i=1}^{n} \left(y_i^{obsvd} - y_i^{calc}\right)^2 \; if \; no \; \sigma_i \; data$$

where the sum is over all n measurements and σ_i is the error in the ith measurement. The smaller the chisqr, the better the fit. When this is not known, or is ignored as in our fitting functions, then σ_i is set equal to 1, not 0, for all i. Some definitions replace σ_i with y_i^{obsvd}, a valid practice provided there are no values of $y_i^{obsvd} = 0$, which generates infinity. Similarly, the error term σ_i must never equal zero for any i. For a variety of reasons, the safest definition of chisqr is as follows:

$$X^2 = \sum_{i=1}^{n} \left(y_i^{obsvd} - y_i^{calc}\right)^2$$

Given the above ambiguity regarding the assignment of X^2, the internal fitting programs return the RMS (root-mean-square) error as defined by the following equation:

$$RMS\ error = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i^{obsvd} - y_i^{calc})^2}$$

The best fit is determined by minimizing the RMS error times the number of parameters included in the fit:

$$Minimize\ for\ best\ fit = N_{parameters} \times RMS\ error$$

This equation follows from a recognition that if a fit requiring fewer parameters generates the same error, the fit involving fewer parameters must be superior because it achieved the same result with fewer degrees of freedom. From another perspective, the simpler the solution, the better (e.g. Occam's razor, lex parsimoniae). The only time the above criterion for best fit should not be used is when an equation with a fixed number of parameters is known to best represent the physical process under study.

4.3.5. Coefficient of Determination

Another commonly used measure of the quality of fit is the coefficient of determination, which is given the symbol, R^2.

$$R^2 = \frac{\sum_{i=1}^{n} \left(y_i^{calc} - y_{ave} \right)^2}{\sum_{i=1}^{n} \left(y_i^{obsvd} - y_{ave} \right)^2}$$

where y_{ave} is the average of the original data. As the fit gets better, R^2 approaches 1.0. The advantage of the coefficient of determination is that it typically ranges in a well defined range from 0.5 (poor fit) to 1.0 (a perfect fit). However, if the data are dominated by random fluctuations, R^2 does not provide a very useful measure of the goodness or quality of the fit. Nevertheless, it is so commonly used that reporting this value is nearly as common as reporting rmserror or X^2. And everyone agrees that an R^2 value of 1.0 means a perfect fit.

4.4. Numerical Integration

Numerical integration is the process of approximating an integral by replacing the formal equation with a sum, e.g.

$$\int_a^b f(x)dx = \sum_{i=1}^{n} w_i f(x_i) \qquad (4.4.1)$$

The above formula is known as a quadrature rule or quadrature approximation, and is carried out by assigning values for n (the number of points), x_i (the abscissas, the values of x at which the function is evaluated) and w_i (the weights at each evaluation point). Virtually all numerical methods of integration are based on the above quadrature formula, but they differ in how the weights and abscissas are selected.

The simplest approach is to assign the values of x_i starting at a and ending at b in small equal increments of Δx. Then all the weights are the same and equal Δx. The accuracy of this method increases as the number of points increases (and Δx decreases). The method improves in accuracy as the number of points increase, although truncation error generally causes the accuracy to decrease at some point. Another problem with this simple approach is that it is inefficient, and quite impossible if the integration is from $-\infty$ to $+\infty$. To provide greater accuracy and permit extension of the numerical methods to

handle indefinite integrals, Gaussian-quadrature methods have been developed. There are two types supported by MathScriptor.

4.4.1. Gauss-Legendre Quadrature

Gauss-Legendre methods handle the integral shown in Eq. 4.4.1. The weights and abscissas (x_i values) are determined by using the solutions of the Legendre polynomial to determine both values to yield maximum accuracy. As a simple example, if the integral is from –1 to 1, then the weights are given by the following equation,

$$w_i = \frac{2}{\left(1-x_i^2\right)\left(P_n'\left(x_i\right)\right)^2} \tag{4.4.2}$$

and the abscissas are the roots of the Legendre polynomials. Any finite range can be renormalized to make use of these abscissas and weights, so Gauss-Legendre quadrature is completely general. The following MathScriptor method provides the abscissas and weights for any definite integral evaluated from x1 to x2.

gauss_legendre(x1, x2, x1(), w1(), n), which generates the Gauss-Legendre abscissas x(1:n) and weights w(1:n) for n-point quadrature for integration from x1 to x2. For example, if the goal is to integrate the equation:

$$f(x)=\exp(-x)\sin(8x^{2/3})+1$$

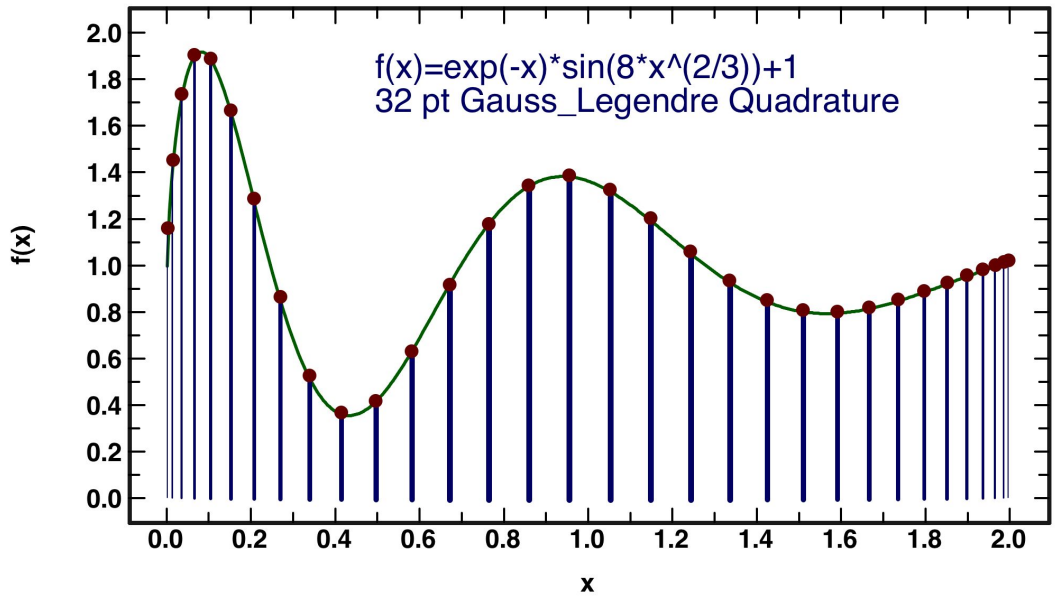
from 0 to 2, then the following code segment will carry out the integration.

```
nmax=12 // usually adequate(n=4096)
// you can go as high as nmax=16(n = 65536)
// or higher if you have the memory and time
twothirds = 2/3 // used in function
for j=2 to nmax
  n=pow(2,j) // number of points=2^j
  x1=0
  x2=2
  redim x(n) // abscissas - redim prior to call
  redim w(n) // weights - redim prior to call
  // the key function
  gauss_legendre(x1,x2,x(),w(),n)
  ss=0.0
  for i=1 to n
    xx=x(i) // the x value
    fx = exp(-xx)*sin(8*xx^twothirds)+1 // the function
    ww = w(i) // gauss legendre weight for above x value
    ss = ss + ww*fx // sum into ss
  next
  print(" Integral(n = "+str(n)+") =  "+format(ss,16,12))
  pause(0) // shows intermediate results
next
```

with the following printout:

```
Integral(n = 4) =        2.106663539467
Integral(n = 8) =        2.017911793792
Integral(n = 16) =       2.016456684779
Integral(n = 32) =       2.016298066544
Integral(n = 64) =       2.016281583851
Integral(n = 128) =      2.016279906892
Integral(n = 256) =      2.016279738313
Integral(n = 512) =      2.016279721466
Integral(n = 1024) =     2.016279719780
Integral(n = 2048) =     2.016279719621
Integral(n = 4096) =     2.016279719605
Integral(n = 8192) =     2.016279719600
```

The value of printing out the results as a function of n in the fashion shown is that each doubling of the value generates, on average, another significant digit of accuracy. This general rule is very approximate, by watching the change in value provides a rough idea of the accuracy of the last number. If the integral values are jumping up and down, rather than monotonically converging on a value, then the method is failing and one cannot have much confidence in the result for high n. But the vast majority of integrals can be handled by this method and the value at n=8192 is often accurate to 8 or more significant digits. The example above yielded 12 significant digits.



One of the strengths of Gauss-Legendre quadrature is the excellent handling of edge effects. As shown in the graph above, where the abscissas are marked with dots and the weights are indicated by the thickness of the vertical lines, the ends are measured with higher resolution than the values near the center of the span. This approach is optimal for a majority of functions.

4.4.2. Gauss-Laguerre Quadrature

Gauss-Laguerre quadrature is designed to handle definite integrals for zero to infinity, -infinity to zero, or –infinity to infinity:

$$\int_0^{\infty} f(x)\,dx = \sum_{i=1}^{n} w_i f(x_i) \tag{4.4.3}$$

$$\int_{-\infty}^{0} f(x)\,dx = \sum_{i=1}^{n} w_i f(x_i) \tag{4.4.4}$$

$$\int_{-\infty}^{\infty} f(x)\,dx = \sum_{i=1}^{n} w_i f(x_i) \tag{4.4.5}$$

All three of these cases are handled by using the same function, however, to generate the weights and abscissas. The function generates these values for Eq. 4.4.3 only, and the user must multiply the abscissas by –1 to handle 4.4.4 or carry out a two part calculation with normal and reflected abscissas to handle 4.4.5:

$$\int_{-\infty}^{\infty} f(x)\,dx = \sum_{i=1}^{n} w_i f(-x_i) + \sum_{i=1}^{n} w_i f(x_i) \tag{4.4.6}$$

If it is known that the function to be integrated has inversion symmetry, then one only needs to carry out the summation of Eq. 4.4.3 and multiply it by two. The MathScriptor function is:

gauss_laguerre(x1(), w1(), nterms) which returns the Gauss-Laguerre abscissas x1(1:n) and weights w1(1:n) for integration from 0 to infinity where n is used to set the number of points. If n is less than 14, the number of points is equal to 2^n such that n =2(4), 3(8), 4(16), 5(32), 6(64), 7(128), 8(256), 9(512), 10(1024), 11(2048), 12(4096), 13(8192). If n is greater than 13 but less than 257, n sets the number of points.

We demonstrate the use of Gauss-Laguerre Quadrature to evaluate the integral:

$$A_a = \frac{r_1 r_2}{2} \int_0^{\infty} \left(s + r_1^2\right)^{-(3/2)} \left(s + r_2^2\right)^{-(1/2)} \left(s + 1\right)^{-(1/2)} ds \tag{4.4.7}$$

which is an important integral in electrostatics and solvent effect theory. This integral has no closed form solution for most values of r_1 and r_2, and has been the subject of many studies due to the difficulty of solving the integral numerically. If we select r_1 and r_2 = 1, then this integral is known to equal 1/3 exactly. The following program evaluates this integral:

```
dim  x(1),w(1) ,ww ,ww1,a,b,c,r1,r2,ss as double
dim xx,aa,integral as double
dim c1,td2,onh,r1r2d2 as double
dim i,j,k,l,m,n as integer
dim s0,s1 as string
clear_text_output(0)
print(" Use Gauss-Laguerre Quadrature to calculate")
s0="Aa = r1*r2/2*Integral[(0,infinity)(s+r1^2) ^-(3/2)"
s0=s0+"*(s+r2^2)^-(1/2) *(s+1)^-(1/2) ds]"
print(s0)
print("-------------------------------------------------")
print("set R1=1 and R2 = 1, then Aa = 1/3 exactly")
print("")
r1 = 1
r2 = 1
r1r2d2=r1*r2/2
for j=1 to 8 // scan over 2^(j+5) values of n
  gauss_laguerre(x(),w(),j+5) // the key function
  n=ubound(x)
  ss=0.0
  for i=1 to n
    xx=x(i)
    ww = w(i)
    aa =r1r2d2*pow((xx+r1*r1),-3./2)*pow(xx+r2*r2,-1./2)*pow(xx+1,-1./2)
    ss = ss+ww*aa
  next
  s0=" n = "+format(n,"0000")+" , integral = "
  s0=s0+ format(ss,"#.00000000")
  s0=s0+", error = "+format(ss-1/3,"#0.00000000")
  print(s0)
next
```

and the following is the output:

```
Use Gauss-Laguerre Quadrature to calculate
Aa = r1*r2/2*Integral[(0, infinity)(s+r1^2) ^-(3/2)*(s+r2^2)^-(1/2) *(s+1)^-(1/2) ds]
-------------------------------------------------
set R1=1 and R2 = 1, then Aa = 1/3 exactly

 n = 0064 , integral = 0.33324729, error = 0.00008605
 n = 0128 , integral = 0.33330346, error = 0.00002988
 n = 0256 , integral = 0.33332170, error = 0.00001163
 n = 0512 , integral = 0.33332968, error = 0.00000366
 n = 1024 , integral = 0.33333205, error = 0.00000129
 n = 2048 , integral = 0.33333288, error = 0.00000046
 n = 4096 , integral = 0.33333317, error = 0.00000016
 n = 8192 , integral = 0.33333326, error = 0.00000007
```

In general, Gauss-Laguerre quadrature is rarely capable of yielding accuracy that is comparable to that achieved by using Gauss-Legendre simply because the span of the abscissas is so much greater. And there are many examples of functions which totally confound the method. These include those with sharp features that occur at large x

values, and those that oscillate with high frequency over the entire range. It is thus a good idea to plot portions of the function and determine whether it is well-behaved.

4.5. Arbitrary Precision Arithmetic

One of the key technological accomplishments of the past decade was the development of methods and procedures for routinely masking sub-micron features on mass-produced semiconductor chips. The current feature size (2014) of 20 nm allows Intel, IBM and other CPU manufacturers to make processors that are many orders of magnitude more powerful than those in the mainframe computers of the 1980s. A typical personal computer now has a processor operating at 2 GHz (or more), and the processor includes an integrated floating point unit capable of handling both single and double precision math. The numerical routines in MathScriptor take advantage of this capability to carry out two-dimensional Fourier transforms in milliseconds instead of the minutes that were typical a decade ago.

The speed of double precision arithmetic and the fact that most computers have more than a gigabyte of memory suggests that numerical calculations can now be routinely performed in double precision (15-16 significant digits). Although double precision is adequate for most calculations, there are times when additional precision is needed. One such example is in the generation of the abscissas and weights for Gauss-Laguerre quadrature (see Section 4.4.2). Any attempt to calculate more than 1024 of these values using double precision arithmetic will generate underflow and the calculation will fail. The coefficients generated by **gauss_laguerre**(x1(), w1(), nterms) for nterm values that generate more than 256 values are calculated by using 32 significant digits of precision, which yields about 14 significant digits of accuracy in the abscissas and weights. Whenever a numerical calculation involves the subtraction or division of two large numbers of comparable magnitude, the Arprec methods provided by MathScriptor may be needed.

Arbitrary precision in MathScriptor uses strings to represent the numbers and all arbitrary precision functions have string parameters and return strings. It is implemented using Robert Delaney's elegant fp plugin (http://delaneyrm.com). The Arprec functions also work on complex numbers identified by separating the real and imaginary parts with a comma (do not explicitly include I). The following functions are Arprec savvy: plus(s1, s2), minus(s1, s2), mult(s1, s2), div(s1, s2), real(s1), imag(s1), pow(s1, s2), log(s1), loggamma(s1), exp(s1), abs(s1), sin(s1), asin(s1), cos(s1), acos(s1), tan(s1), atan(s1), sinh(s1), asinh(s1), cosh(s1), acosh(s1), tanh(s1), atanh(s1). Exponents as large as ±58, 000, 000 are allowed.

The first thing to do before starting any Arprec calculation is to assign the number of digits of precision that is desired by using the method **Arprec_set_precision**(idigits). The value of idigits can be any number from 8 to 2, 147, 483, 648. Although one can change the working precision in the middle of a calculation, if the precision has been increased, previous calculations of relevance will need to be redone. Thus, it is best to assign the maximum precision required at the very beginning, remembering to add a few extra digits to handle anticipated truncation error. Note that the argument, idigits, is assigning the precision of the mantissa (also called the coefficient or significand), and large exponents do not reduce the precision in the mantissa as is the case with some arbitrary precision implementations.

To format Arprec numbers there are two options. A temporary result can be rounded to lower precision by using round_to_precision(s1, ndigits). One can also format numbers for printing by using Format(s1, nwidth, ndecimal), which also works on complex numbers (total width=2*nwidth+3 for comma delimiter). Comparisons using Arprec strings can be carried out using Q_greater_than(s1, s2) and Q_less_than(s1, s2), and both functions return true if the stated comparison is true. To test for equal, one can use s1=s2 as the conditional, but such comparisons are dangerous as the two numbers must be identical in both value and precision for this to work. It is best to write code that avoids an equals test. If this is impossible, then use round_to_precision() to use a lower precision comparison for improved reliability.

Following is a list of Arprec savvy functions as well as functions that are useful for Arprec calculations.

abs(s1) **as string** absolute value of x

acos(s1) **as string** returns arc cosine, or inverse of the cosine, of x in radians
divide acos() by const_degree to convert to degrees.

acosh(s1) **as string** inverse of the cosh() function

Arprec_degree as string calculates and returns radians per degree to an arbitrary precision.

Arprec_e as string, calculates and returns e to an arbitrary precision.

Arprec_euler as string returns Euler's constant to the requested precision, or 10000 digits, whichever is smaller.

Arprec_factorial(n **as integer) as string** returns the string representation of the factorial of n, for values of n from 0 to 8, 600, 000. This function is unique in that previous values are cached to speed up subsequent determinations.

Arprec_pi as string calculates and returns pi to an arbitrary precision.

Arprec_precision as integer a read only integer that returns the current precision set for the Arprec internal arithmetic. The user sets this value by executing **Arprec**_set_precision(ndigits). Do not change this value- it is read only.

Arprec_random_float[(seed_string)] **as string** returns an arbitrary precision floating point number between 0 and 1. An Arprec string representing a positive number between 0 and 1 can be included to seed the generator. This seed, if used again, will produce an identical set of pseudo-random numbers. Subsequent references should not include any seed string, i.e. just **Arprec_random_float().**

Arprec_random_integer(ndigits) **as string** returns a random integer of length ndigits. Make sure you set **Arprec**_precision() to a value larger than ndigits prior to doing math.

Arprec_set_precision(idigits) set the precision in number of digits for subsequent arbitrary precision calculations.

Arprec_variational_min(sx(), sy(), n) **as string** returns the value of x for which y is a minimum for a data set sx(1), sy(1), sx(2), sy(2) ... sx(n), sy(n) where n=3 or 4. Precision should be set at 3 times the desired number of significant digits, and the range of sx(1)...sx(n) decreased with each iteration. More on this function below.

asin(s1) **as string** returns arc sine (or inverse of the sine) of x in radians.

Note that any Arprec functions that return radians can be convert to degrees by using the following expression:

div(**asin**(s1), **Arprec_degree**)

Similarly, when one has a string in degrees (e.g.. s1 = “45”) and wishes to convert it to radians, one multiplies the string by **Arprec_degree**:

mult(s1, **Arprec_degree**)

asinh(s1) **as string** inverse of the sinh() function

atan(s1) **as string** returns arc tangent in radians of x

atanh(s1) **as string** inverse of the tanh() function

cos(s1) **as string** returns cosine of x assuming x is in radians if using degrees, multiply by const_degree e.g. cos(mult(**Arprec**_degree, s1))

cosh(s1) **as string** returns the hyperbolic cosine.

div(s1, s2) **as string** s1/s2 using arbitrary precision string arithmetic.

exp(s1) **as string** returns e to power of x = pow(e, x) = e^x

format(s1, nwidth, ndecimal) **as string** returns a formatted Arprec string number of width nwidth showing ndecimal digits to the right of the decimal point. If s1 is complex, then both the real and imaginary components are formatted and separated by a comma.

imag(s1) **as string** returns the imaginary part of an arbitrary precision complex string number.

log(s1) **as string** natural log of s1 (e.g. log(exp(12))=12)

log10(s1) **as string** log based 10 of s1 (e.g. log10(1000)=3)

logGamma(s1) **as string** natural log of Gamma() for x>0. Note that n! = Gamma(n+1) = exp(logGamma(n+1)). For example, 12! = 479, 001, 600, which equals Exp(Gamma(12 + 1)) = 479, 001, 600. Truncation error makes this method approximate for large n, thus 19! = 121, 645, 100, 408, 832, 000 but exp(loggamma(19 + 1)) = 121, 645, 100, 410, 059, 440.

minus(s1, s2) **as string** s1 - s2 using arbitrary precision string arithmetic.

mult(s1, s2) **as string** s1*s2 using arbitrary precision string arithmetic.

plus(s1, s2) **as string** add two strings using arbitrary precision string arithmetic.

pow(s1, s2) **as string** returns s1^s2.

primeQ(candidate_prime, ntrials) **as Boolean** Arprec application of the Miller-Rabin prime number test on candidate_prime based on ntrials. False means definitely not prime. True means prime with an error probability of (0.25^ntrials). The

candidate_prime integer can be either an int64 or an Arprec string with the number of digits less than ndigits set via **Arprec**_set_precision(ndigits).

Q_greater_than(s1, s2) **as boolean**
returns true if s1>s2, where s1 and s2 are arbitrary precision string floats or integers.

Q_less_than(s1, s2) **as boolean**
returns true if s1<s2, where s1 and s2 are arbitrary precision string floats or integers.

real(s1) **as string**
returns the real part of an arbitrary precision complex string number.

round_to_precision(s1, nsd) **as string** Round the real number x to nsd significant digits. This function works on Arprec strings as well as string complex numbers. In the latter case, the real and imaginary components are returned rounded and separated by a comma.

sin(s1) **as string** returns sin of x assuming x is in radians

sinh(s1) **as string** returns the hyperbolic sine.

tan(s1) **as string** returns tangent of x assuming x is in radians

tanh(s1) **as string** returns the hyperbolic tangent.

It is worth reminding the reader that functions like **Arprec_random_float**() can be written as **arprec_random_float()** or **Arprec_Random_Float()**, and all are treated as identical by the compiler. Capitalization is ignored.

4.5.1. Arbitrary Precision Variational Optimization

Variational optimization is based on the use of the first derivative to assign the location of a minimum numerically. This method in a slightly different form is called Newton's method, but in our implementation, requires the use of high precision math to achieve a reliable solution. The variational method is so useful in practice that it is included as an internal Arprec method in which all of the internal math is carried out using Arprec functions. To understand the approach, examine Fig. 4.5.1.

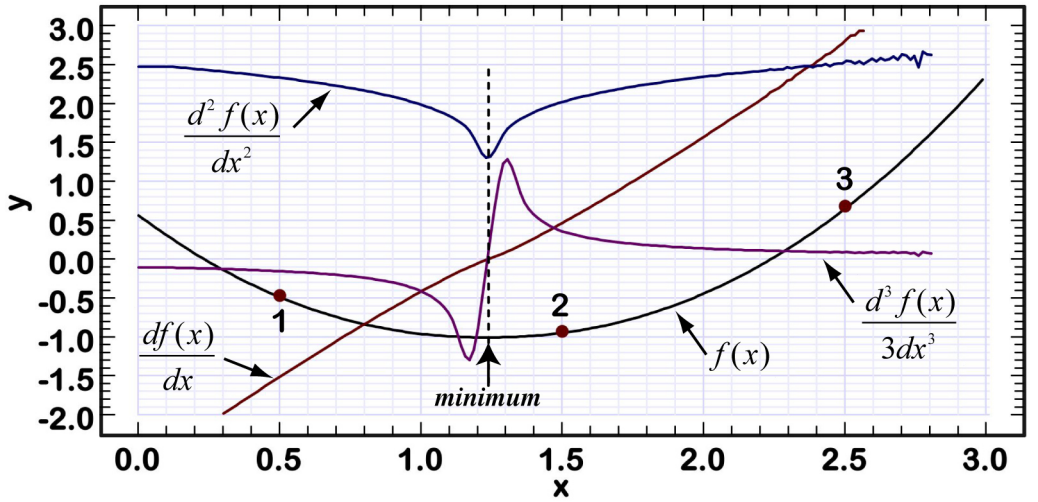


Figure 4.5.1. Example of using the first derivative zero point to assign the minimum in a function using the function Arprec_variational_min(sx(), sy(), n).

The first step in using **Arprec_variational_min** is to select three points of equal separation with the second point as close to the minimum as possible. A course search is the best approach and when such was carried out on the function, f(x), point 2 was found to be the minimum of the search. Points 1, 2 and 3 now represent the first three values to be provided to **Arprec_variational_min** as follows:

```
sx(1)="0.5" , sy(1)="-0.48331905"
sx(2)="1.5" , sy(2)="-0.94145388"
sx(3)="2.5" , sy(3)= "0.65516388"
arprec_variational_min(sx(), sy(), 3) → "1.2229635"
```

The first iteration returns a better estimate of the minimum, and the program should then take the minimum and add and subtract an amount that is half the separation that was used previously. Hence, iteration two should be:

```
sx(1)="0.9729635" , sy(1)="-0.94330835"
sx(2)="1.2229635" , sy(2)="-0.99992874"
sx(3)="1.4729635" , sy(3)="-0.95347537"
arprec_variational_min(sx(), sy(), 3) → "1.2352933"
```

Note that sx(2) equals the last result returned by **Arprec_variational_min.** The above example rounded all values to 8 significant digits to make it more easily followed. The true minimum is exactly 1.2345, and with the range halved each iteration, the user can decide when to discontinue iterations by assuming that the error is conservatively 10% of the total range investigated. Taking the process too far can result in truncation error

dominating, despite the use of Arprec methods. To understand the process, and the potential problems, we examine the internal workings of this function.

The internal workings of this function for n=3 are quite simple. The three data points, {x1, y1}, {x2, y2} and {x3, y3} are fit to a simple polynomial $a + bx + cx^2$ via closed form assignment of b and c:

b = -((sqr(x2)*y1 - sqr(x3)*y1 - sqr(x1)*y2 + sqr(x3)*y2 +sqr(x1)*y3 - sqr(x2)*y3)/((x1 - x2)*(x1*x2 - x1*x3 - x2*x3 + sqr(x3))))

c = -((-(x2*y1) + x3*y1 + x1*y2 - x3*y2 - x1*y3 + x2*y3)/(sqr(x1)*x2 - x1*sqr(x2) - sqr(x1)*x3 + sqr(x2)*x3 + x1*sqr(x3) - x2*sqr(x3)))

where sqr(x) equals x^2. The basic premise of variational analysis is a recognition that the first derivative of a function passes through 0 when the function is at a minimum (e.g., Fig. 4.5.1). If $f(x) = a + bx + cx^2$ then

$$\frac{df(x)}{dx} = b + 2cx$$

Setting the first derivative to zero and solving for x we get

$$x_{min} = -b/(2c)$$

which represents the abscissa at which the function has reached a minimum. This is the value returned by **Arprec_variational_min.** Although the math does not require that the function be analyzed at fixed Δx [i.e. $(x_2-x_1) = (x_3-x_2)$], truncation error is reduced when such is the case. Furthermore, best results are obtained when x_2 approaches or equals the actual function minimum. Hence, as previously stated, the next iteration should be carried out by setting x_2 to the previous variational min.

Arprec_variational_min also allows for the use of four point variational minimization which involves the closed form analysis of $a + bx + cx^2 + dx^3$. This option is selected by setting the third parameter to 4, instead of 3. The assumption that a four-point analysis is always better than a three-point analysis is not true unless the function that is being analyzed has higher order terms. A four point fit requires that each iteration have values assigned with equal Δx values with the most recent minimum bounded by x_2 and x_3. The use of a three-parameter fit is faster and is preferred under a majority of circumstances.

4.6. Plotting Numerical Data

There are a variety of plotting functions available to handle scientific and engineering data. The following list provides an overview of these functions. Additional details may be found in Appendix 1.

plot_data(x(), y(), npoints, x1, x2, y1, y2, xlabel, ylabel, gridlevel)
plot x(1..npoints), y(1..npoints) using a standard plot format. This function must be called prior to plot_more_data or plot_data_point. The axes and labels are black. Adjust the plot color using graphics_forecolor(0, color) and the stroke width of the plotted line by using graphics_stroke_width(0, iwidth).

plot_data_point(x, y, s0, ilocation, symbol_type, symbol_size, symbol_color)
plot single data point at position x, y. The size in pixels is controlled using symbol_size and the color is set using symbol_color. You can label each data point using the string s0 and the ilocation integer symbol 1(upper left), 2(above), 3 (upper right)4 (at left), 5 on top, 6 (at right) 7 (lower left), 8 below, 9 (lower right). The font name and font size are specified by graphics_font. Each data point is plotted using one of 12 symbol_types [symbol_type=1 (square solid), 2 (triangle down solid), 3 (circle solid), 4 (triangle up solid), 5 (square open), 6 (triangle down open), 7 (circle open), 8 (triangle up open), 9 (square open thick), 10 (triangle down open thick), 11 (circle open thick), 12 (triangle up open thick)].

plot_data_points(x(), y(), npoints, symbol_type, symbol_size, symbol_color)
plot individual data points x(1..npoints), y(1..npoints). Each data point is plotted using one of 12 symbol_types [symbol_type=1 (square solid), 2 (triangle down solid), 3 (circle solid), 4 (triangle up solid), 5 (square open), 6 (triangle down open), 7 (circle open), 8 (triangle up open), 9 (square open thick), 10 (triangle down open thick), 11 (circle open thick), 12 (triangle up open thick)].

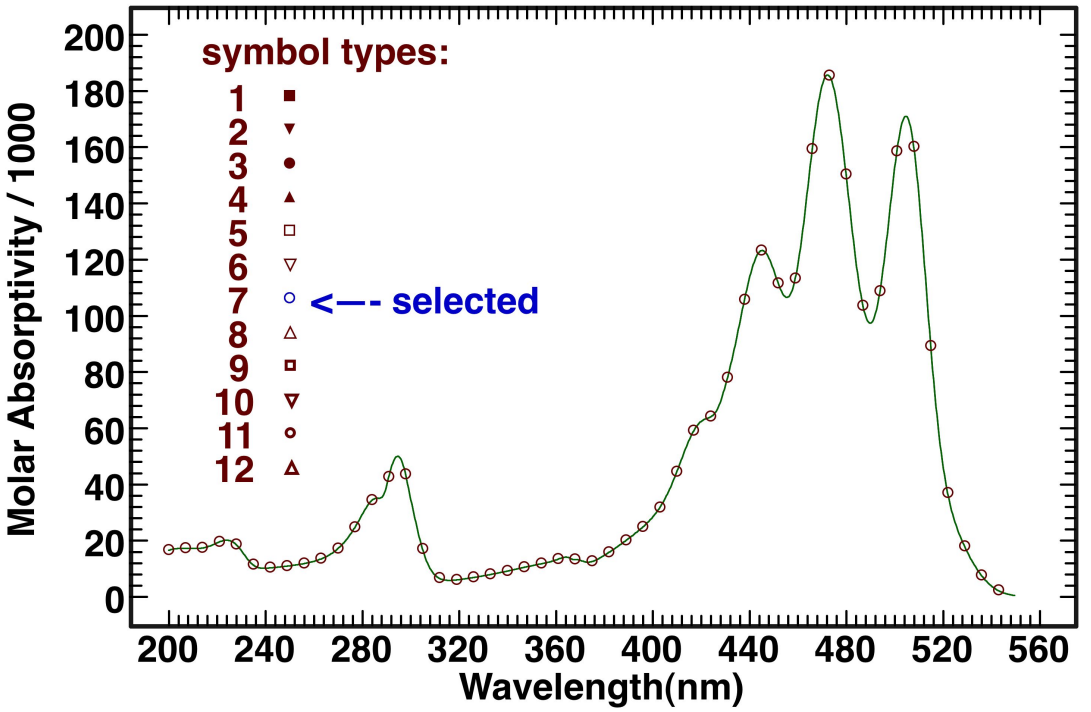


Figure 4.6.1. Sample output from the template_plot_data_points program, which illustrates the use of the plot_data_points() function. This function provides 12 different symbols as shown in the insert, and when this program is run, it allows the user to select the symbols using the arrow keys.

plot_data_points_with_errors(x(), y(), yerror(), npoints, symbol_type, symbol_size, symbol_color, error_bar_type) This method will plot data points including a vertical error bar. All parameters defined as in plot_data_points except yerror(1..npoints) gives the total length of the errorbar in units of y, and error_bar_type specifies the type of error bar (0=single line, >0=width of horizontal lines at top and bottom of error bar in pixels).

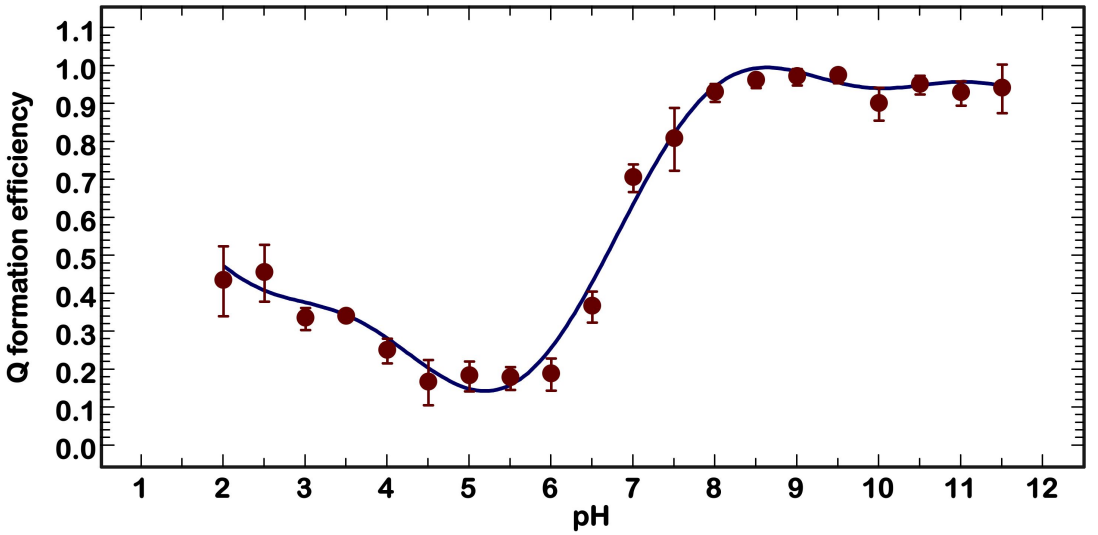


Figure 4.6.2. Example of using **plot_data_points_with_errors()** to include error bars in the plot. The height of the error bars are equal to the values provided by the parameter array, yerror(1..npoints).

plot_fontname = system_fontname_label or
plot_fontname = name of any font in the font folder on your computer

call one of the above statements prior to calling plot_data to set font for use in the plot.

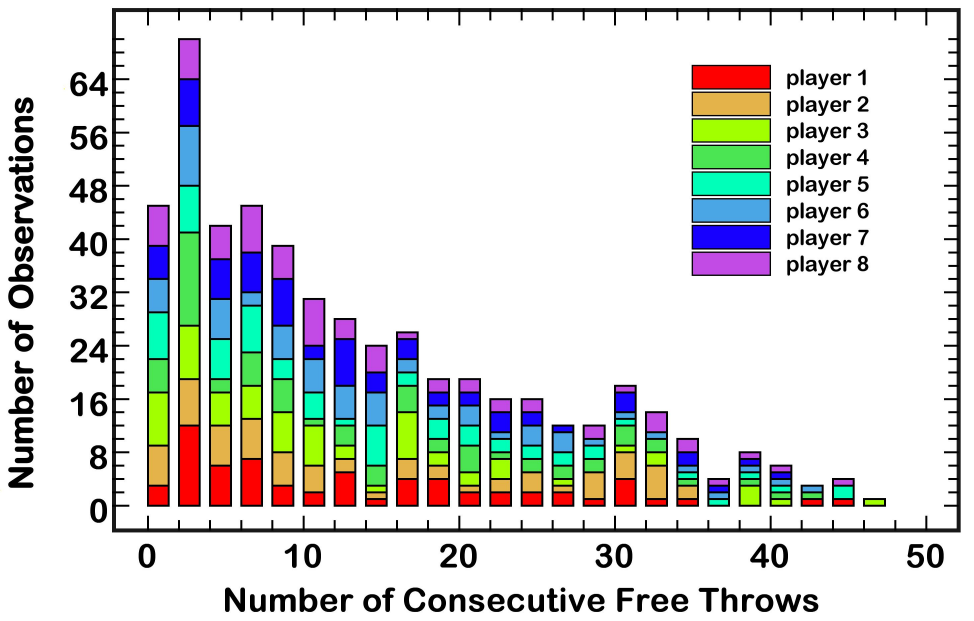
plot_histogram(harray(), icolor(), nh, dfwhm, barwidth, xp1, xp2 [, region_colors()])
generate and plot a histogram of the data in harray(1..np) using the integer array icolor(1..nh) to designate the color to be assigned to each individual point in harray(). The maximum number of different values is 32, but numbers less than 12 work best for clarity. The sampling width is given by dfwhm, and a barwidth of dfwhm/3 is nominal. The plot is from xp1 to xp2. Set both to zero if you wish the program to select the range. The optional parameter, region_colors(), is a byref array, redimmed automatically, which returns the region colors corresponding to the integer values in carray(). The requirements of doing bit-level manipulations in this function requires that it create its own buffer of size 3000 x 2000 and to set plot_fontsize=32. If one chooses to add an x axis label using draw_string(), this buffer size should be kept in mind and plot_set_options() used to shift the x axis up to make room.

Histograms are a special type of graphic presentation which provide information on distributions or sets of data for which there are numerous observations with the same or similar values and which may come from different sources. The version supported here plots the number of observations in the Y axis and the values along the X axis. The

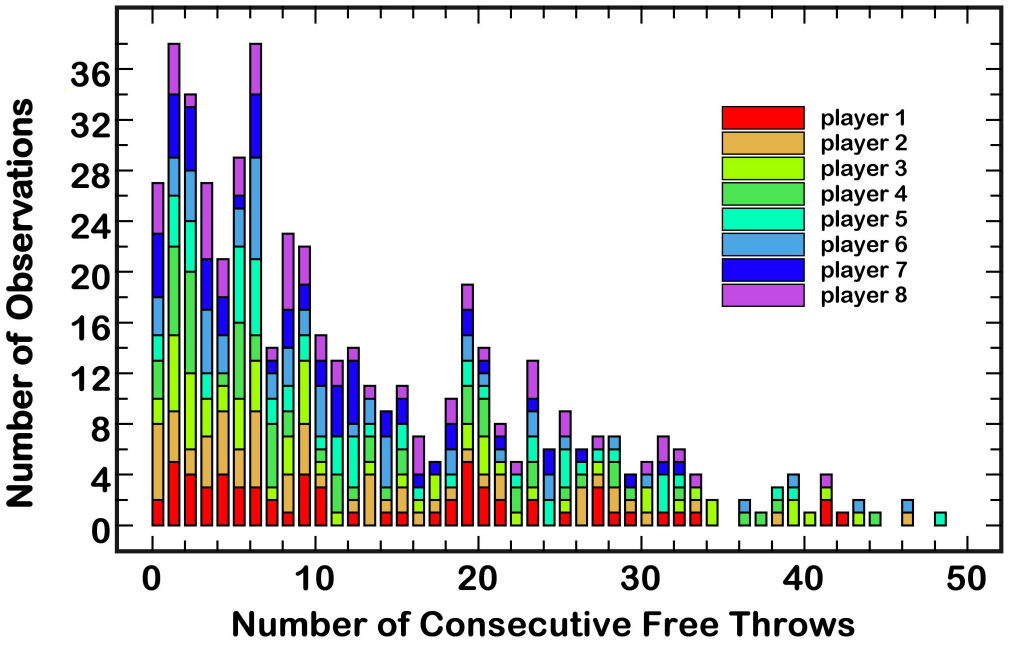
values are passed to the function in harray(1…nh) and the sources (represented by colors, as many as 16) are indicated by integer (1..16) in icolor(1..nh). It is best to keep the number of colors (sources) to 8 or less so that the colors can be distinguished by the viewer. It is very important to keep in mind that this function is unique in that it creates its own buffer of size 3000x2000.

Consider a basketball team with 8 players. Each player is told to make as many free throws in a row as possible. The players continue until 512 sets are collected. The program below generates the histogram that follows:

```
// data are in harray() and carray()
dfwhm=4
barwidth=dfwhm/3
plot_set_options(100,0)
// following creates a 3000x2000 buffer, nh=number of data sets in
harray(1..nh) and carray(1..nh). The colors are returned in hcolor(1-8)
Making nh negative signals no histogram spectrum.
plot_histogram(harray(),carray(),-nh,dfwhm,barwidth,0,50,hcolor())
s0 = "Number of Consecutive Free Throws"
draw_string(0,s0,1600,1900)
call graphics_font(0,"Arial rounded mt bold",64,false,false)
for j=1 to 8 // 8 players only
   plot_rectangle(35,70-4*j,40,70-4-4*j+0.5,4,rgb(0,0,0),hcolor(j))
   plot_string("player "+str(j)+const_eol,41,66-4*j+1 ,rgb(0,0,0))
next
buffer_copy_to_canvas(active_canvas,0)
```



Note that each vertical bar represents an x range of two so that the first vertical bar represents all players who sunk 0 or 1 baskets in a row; the second bar represents all the players who sunk 2 or 3, and so on. The last bar represents the rare case of a player singing 46 or 47, and the color indicates that it was player 4. If the variable dfwhm in the above program were changed to 2, then each of the vertical bars would mark a single value, and provide greater resolution at the expense of readability (too many bars).
An example of a different data set presented using dfwhm=2 is shown in the histogram below:



It is a subjective matter regarding the choice for dfwhm. In general this variable generates histogram slots which are half the size of dfwhm, but the program will mediate the analysis so this rule is not reliable for all data sets. The programmer is responsible for adding the x-axis label, which is optional. The programmer is also responsible for inserting the legend, when there is sufficient space within the plot. When a legend is desired, the plot_histogram() call should include the optional color array as the last parameter. This array will return the colors that are displayed to coincide numerically with the integers passed in carray(1..nh). Thus, hcolor(3) is the color representing all instances of carray() equal to 3.

Histograms provide an excellent method of presenting complex data sets involving multiple sources of identical measurements. The alternative is a table, which is much more difficult to appreciate without careful study.

The following three methods provide additional options for adding plots to a previous figure, adjusting the labels or the tick marks. Plot_set_options and plot_set_ticks must be called prior to executing plot_data.

plot_more_data(x1(), y1(), npoints, line_thickness, line_color)
plots x1(1..npoints), y1(1..npoints) in graph previously created by calling plot_data

plot_set_options(nx_axis_shift, ny_axis_shift)
Inserts additional space between the axes and the labels when positive values are used. Negative values decrease the space between the numerical and text data. Set all values to 0 to return control of the plot spacing control to MathScriptor.

plot_set_ticks(xsmall, xbig, ysmall, ybig)
when executed prior to a plot_data statement, manually sets the major (xbig, ybig) and minor (xsmall, ysmall) tick separations. If the values are all integers, and the subsequent plot ranges are integers, then ticks are assigned using modular arithmetic. If the values are real, the ticks are separated from x1 and y1, and for this reason, x1 and y1 must be assigned on a major tick mark for aesthetic reasons. If the user wants to return to automatic tick assignment, execute plot_set_ticks(0, 0, 0, 0) and the internal plot methods will auto-assign ticks as best they can.

4.7. Fourier Series and Fourier Transforms

A function of t, f(t), where t is measured from –1 to 1 radians can be represented exactly by the Fourier series:

$$f(t) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty}\left[a_n \cos(nt) + b_n \sin(nt)\right] \qquad (4.7.1)$$

where the weighting functions, a_n and b_n are given by the equations:

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\cos(nt)dt \qquad (4.7.2)$$

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\sin(nt)dt \qquad (4.7.3)$$

In practice, a Fourier series is truncated after a certain number of terms to make the summation computationally tractable. Thus, a Fourier series rarely reproduces the function rigorously, but the feature that makes this process so useful is that the a and b coefficients can be calculated readily using methods that have been developed to make the computational effort highly efficient. The process is known as the Fast Fourier Transform and it is implemented within MathScriptor using the following methods:

FFT1(a1(), c1(), s1(), n) one-dimensional fast Fourier transform of the linear array a1(0….n) placing cosines into c1(0..n) and sines into s1(0..n)
FFT1_inverse(a1(), c1(), s1(), n) inverse 1D fast Fourier transform. Takes the cosine c1(0..n) and sine s1(0..n) arrays and returns the inverse transform in a1()
FFT2(a2(), c2(), s2(), n) two-dimensional fast Fourier transform of the two-dimensional array a2(0….n, 0...n) placing cosines into c2(0….n, 0...n) and sines into s2(0….n, 0...n)
FFT2_complex_association(c2a(), s2a(), c2b(), s2b(), n) **as double** returns the complex association between a pair of transforms of the same size (n by n).
FFT2_inverse(a2(), c2(), s2(), n) inverse 2D fast Fourier transform

where we use the arrays a1() and a2() to represent the one or two-dimensional functions to be transformed and the cosine terms (Eq. 4.7.2) are in c1() or c2() and the sine terms (Eq. 4.7.3) are in s1() or s2(). All the Fast Fourier Transform (FFT) methods require that n, the number of points, be a multiple of 2 (i.e. equal to 2^k, where k is an integer). Valid values of n are 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288 or higher. The coefficients are stored in a folded format so that the lowest frequencies are on the "edges" and the highest frequencies are in the middle. The 0th element is the offset, or frequency=0, component. The **FT_linearize**

statement is available to translate the coefficients to more manageable form where the 0th to highest frequency components are stored from low to high in subscript. These can be refolded by using **FT_fold**.

FT_fold(c1(), s1(), c1lin(), s1lin(), npoints) transform the linearized coefficients in c1lin(1..npoints) and s1lin(1..npoints) into a folded pair in c1(1..npoints) and s1(1..npoints). When folded, the lowest frequencies are at 1 and npoints and the highest frequency is in the middle (npoints/2). The zeroth frequency components are transferred such that s1lin(0)=s1(0) and c1lin(0)=c1(0).

FT_linearize(c1(), s1(), c1lin(), s1lin(), npoints) transform the folded Fourier coefficients in c1(1..npoints) and s1(1..npoints) into a linearized pair in c1lin(1..npoints) and s1lin(1..npoints). When linearized, the frequency increases linearly from low subscript to high subscript. The zeroth frequency components are transferred such that s1(0)=s1lin(0) and c1(0)=c1lin(0).

Fourier transforms are often discussed in terms of real and imaginary components because of the following relationships:

$$f(t) = \sum_{n=-\infty}^{\infty} \left[c_n \exp(-int) \right] \tag{4.7.4}$$

where

$$c_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(t) \exp(-int)\, dt = \frac{1}{2}\left(a_n - ib_n\right) \tag{4.7.5}$$

To better understand why these two forms are identical, we recall the following three equations from the Math review of Appendix 4.

$ae^{\mathbf{i}b} = a \cos b + \mathbf{i} a \sin b$ (A4.2.12)

$\mathrm{Re}(ae^{\mathbf{i}b}) = a \cos b$ (real part) (A4.2.13)

$\mathrm{Im}(ae^{\mathbf{i}b}) = a \sin b$ (imaginary part) (A4.2.14)

If the reader is not familiar with complex numbers, a review of the relationships in Appendix 4 is recommended. For the purposes of the present discussion, however, one can operate with the more conventional forms of the equations shown in 4.7.1 - 4.7.3 without formally dealing with complex numbers at all. Furthermore, if the function is symmetric so that

$$f(t) = f(-t)$$

then the Fourier series representing this function will only have cosine (real) components.

4.7.1. Fourier Analysis of Infrared Absorption

As a scientifically interesting demonstration of one-dimensional Fourier analysis, we examine the classical treatment of infrared absorption of light by a molecule. This process is associated with the interaction of the electromagnetic field of the

$$\alpha(\omega)=\frac{4\pi\omega\tanh(\beta\hbar\omega/2)}{3\hbar c\, n(\omega)V}\int_{-\infty}^{\infty}\left\langle \mathbf{M}(t)\bullet\mathbf{M}(0)\right\rangle e^{-i\omega t}dt \tag{4.7.6}$$

where $\alpha(\omega)$ is the absorption coefficient per unit path length, the homogeneous sample occupies the volume V, $n(\omega)$ is the frequency dependent refractive index, the frequency of the light that is absorbed is related to the angular frequency by the relationship ($\omega=2\pi v$), **M**(0) is the dipole moment on average and **M**(t) is the dipole moment as a function of time. Applying Euler's formula (see Eq. A4.2.9)

$$e^{-i\omega t}=\cos(\omega t)-i\sin(\omega t) \tag{4.7.7}$$

and rewriting the Fourier Transform of the function **F**, in terms of the integral over time, we get

$$\alpha'(\omega)=\frac{\omega}{n(\omega)}\int_{-\infty}^{\infty}\left\langle \mathbf{M}(t)\bullet\mathbf{M}(0)\right\rangle e^{-i\omega t}dt=\frac{\omega}{n(\omega)}\,\mathbf{F}\left\{\left\langle \mathbf{M}(t)\bullet\mathbf{M}(0)\right\rangle\right\} \tag{4.7.8}$$

where $\alpha'(\omega)$ is the relative absorptivity. Thus, the Fourier transform of the dipole moment as a function of time will yield the absorption spectrum. In specific,

$$\alpha'(\omega_k)=\frac{\omega_k}{n(\omega_k)}\left|a_k+ib_k\right| \tag{4.7.9}$$

where a_k is the kth cosine term and b_k is the kth sine term from a Fourier transform of the dipole moment vector with time where ω_k is the angular frequency associated with the kth term. This value is a function of the separation of the measurements. An example of this type of analysis is shown in Fig. 4.7.1.

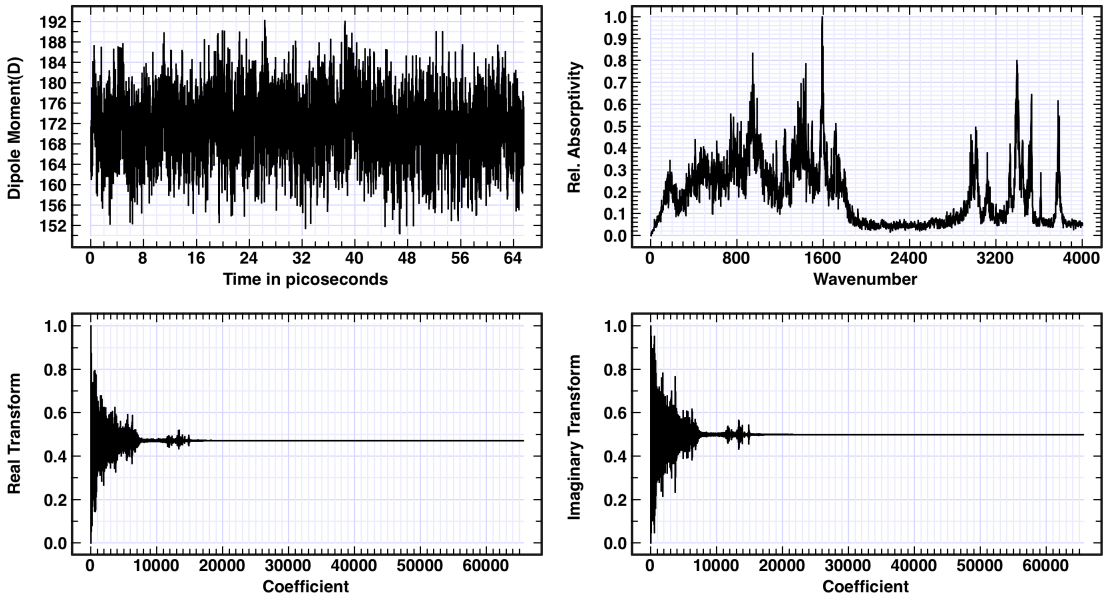


Figure 4.7.0. Example of using Fourier analysis to convert a time resolved dipole moment into a vibrational spectrum. The dipole moment is calculated for bacteriorhodopsin and was calculated using molecular dynamics and the Charmm force field.

4.7.2. Apodization

The Fourier coefficients, when plotted in the linear fashion shown in Fig. 4.7.0, represent components of the spectrum that increase in frequency as one goes from left to right (low number to high number). A significant advantage of working in Fourier space is that one can preferentially adjust the intensity of the components by reference to the frequency, which is linearly proportional to the coefficient number (if the coefficients have been ordered from low to high frequency). The process of manipulating the intensities of the Fourier coefficients is called apodization, which is a general term used for changing the shape of a mathematical function. One generic type of apodization is

$$apod(i) = \left(\frac{(n-i)}{n} \right)^k \tag{4.7.10}$$

where n is the number of coefficients, i is the coefficient number, and the level of apodization is given by the integer k , where the larger the value, the more rapid is the cut-off of the higher frequencies. The effects of apodization are explored in Fig. 4.7.1.

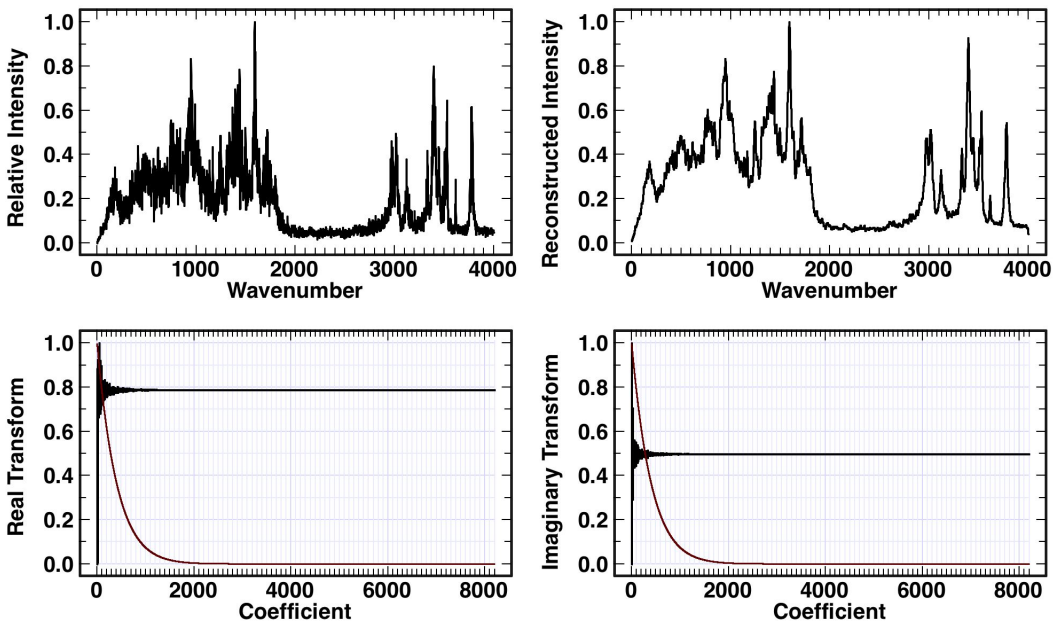


Figure 4.7.1. Example of low-pass apodization of the Fourier transform on the spectrum shown at upper left. Upon the inverse Fourier transform of the apodized coefficients, the spectrum at upper right is generated. The apodization function is the smooth curved line shown in the two panels at the bottom, and the Fourier coefficients are plotted post apodization. The apodization function is shown in Eq. 4.7.10 and is based on the assignment of k=20.

The above example of apodization is analogous to using a low-pass filter on a periodic function or signal. One can also use apodization to selectively enhance high frequency components. Such will increase the intensity of narrow bands but at the expense of enhancing noise, which is usually high frequency as well.

4.7.3. Fourier Self-Deconvolution

Narrowing a spectral line in the frequency (ω) domain is equivalent to stretching the corresponding signal in the time (t) domain, so that it does not decay as quickly. For example, a cosine wave that oscillates continuously in the time domain will generate an infinitely sharp spectral feature in the frequency domain. Conversely, if the cosine wave is apodized to generate a faster decay, the spectral feature broadens (see Fig. 4.7.3). Fourier-self-deconvolution (FSD) is in effect a reversal of the apodization process depicted in Fig. 4.7.3.

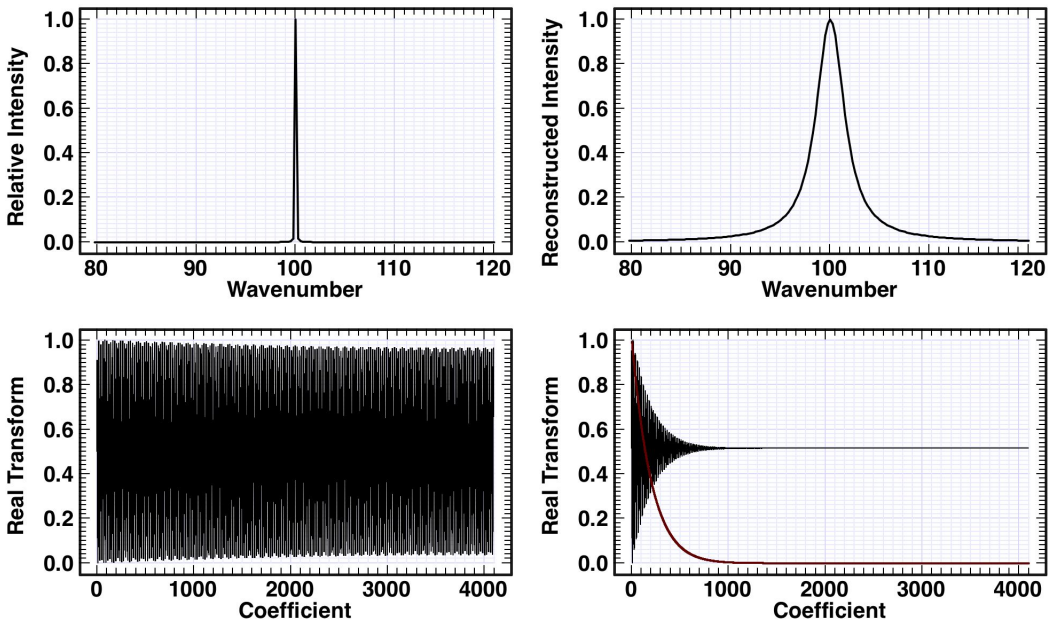


Figure 4.7.2. A very sharp spectral line (upper left), when Fourier transformed, generates a cosine wave that decays very slowly (lower left). If the line were infinitely sharp (a delta function), the cosine wave would be continuous without decay. If the cosine wave is apodized (lower right), the spectral feature broadens in proportion to the extent of apodization. Fourier-self-deconvolution represents a mathematical reversal of this process.

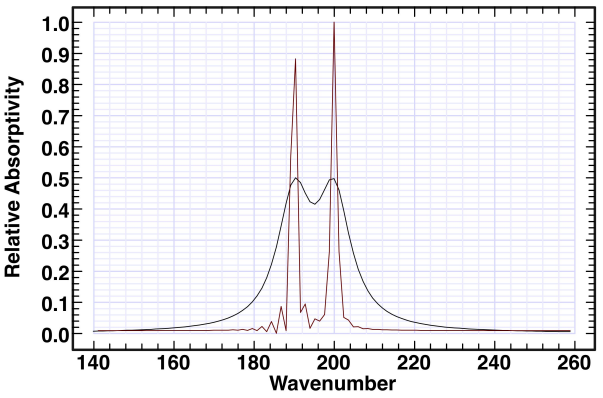
Standard implementation of FSD is based on the following algorithm:

1. The spectrum in frequency space is reflected about the origin to produce a symmetric version. This reflection doubles the amount of data to transform, but by making the spectrum symmetric, the Fourier-transform is real (no sin() components).

2. The Fourier transform is divided by the Fourier transform of the line shape function centered at the origin. This process is equivalent to deconvolving the spectrum by the line shape of the spectrum. The line shape function must have an integral of unity so that the oscillator strengths of the individual bands remain invariant to deconvolution. Thus, as the bands become sharper, their intensities will increase to maintain the same integral.

3. The resulting time domain signal is smoothed by using an apodization function that optimizes the line shape and prevents the creation of spurious features. Selection of the apodization is equally important to the selection of the line shape function.

An example of using FSD on two overlapping Lorentzians is shown at right. The original spectrum was generated using two Lorentzians at 190 and 200 cm^{-1} with full-widths at half-maxima of 5 cm^{-1}. This spectrum was then deconvolved by using a Lorentzian line shape with a FWHM of 3.5 cm^{-1}, a value which produced the optimal result. In principle, a value of 5.0 cm^{-1} would be the optimal choice, but in practice one must balance the choice with the apodization that is used to remove the noise. In this case, no apodization (smoothing) was used, and spurious features are observed on either side of the line shape. This signal had no noise.



This process can also be done using complex FSD, which is expedient in MathScriptor because of the optimized FFT methods. A simple program that carries out Fourier-Complex-Self-Deconvolution with apodization is shown below. Two types of line shape functions are demonstrated in this program. The first is an exponential function:

$$G_d(i) = \exp\left(-i\,k_d / n\right) \tag{4.7.11}$$

where i is the coefficient of the transform set, n is the total number of coefficients and k_d is the deconvolution line-shape factor. The larger the value of k_d, the more narrow line-shape. The function in Eq. 4.7.11 is appropriate for either a Lorentzian or Gaussian line shape. The (artificial) alternative is the use of a quadratic decay function:

$$L_d(i) = \left(\frac{2\,n\,k_d}{\left(2\,i\,k_d + n\right)^2} \right) \tag{4.7.12}$$

where i is the coefficient of the transform set and k_d is the deconvolution line-shape factor, where large values are associated with a more narrow line-shape. It is rare for the optimal values of k_d in Eqs. 4.7.11 and 4.7.12 to be the same. Furthermore, the optimal line shape function is strongly coupled to the apodization which means both must be optimized simultaneously. The deconvolution is carried as follows:

$$c_i^{new} = c_i^{old} \exp\left(\frac{i\,k_d}{n}\right) \times \left(\frac{(n-i)}{n}\right)^{k_a} \quad and \quad s_i^{new} = s_i^{old} \exp\left(\frac{i\,k_d}{n}\right) \times \left(\frac{(n-i)}{n}\right)^{k_a} \tag{4.7.13}$$

where c_i are the real (cosine) terms and s_i are the imaginary (sine) terms. The new Fourier coefficients are then inverse Fourier transformed to generate the resolution-enhanced Fourier-self-deconvoluted spectrum. There are analogous expressions for $L_d(i)$. A program that demonstrates complex FSD is shown at right. The program first generates a sample spectrum in the array y(1..npoints) consisting of three Lorentzians centered at 95, 105 and 115 wavenumbers. These bands are overlapping, and the goal of FSD is to sharpen these bands so that they are differentiated. This program demonstrates both Gaussian and Lorentzian deconvolution as well as the presence or absence of noise. The user must select Gaussian and Noise options at the top of the program using the two Booleans provided. The inherent sharpness of the deconvoluted line shape and the degree of apodization is set by entering two integers separated by a comma in the input field. Trial and error is required, and increasing the first parameter increases resolution, but also noise, and the second parameter adjusts the rate of apodization. As this rate increases, noise is diminished at the expense of resolution. Simultaneous optimization of both parameters is required to achieve optimal resolution enhancement. Because enhancement of resolution invariable enhances

```
// program name: complex_fourier_self_deconvolution.txt
// demonstrates a two-variable FSD enhancement in complex space.
Options for noise and type are set at top.
dim xmin,xmax,xdel,test,ymax,emax as double
dim x(4096),y(4096),c1(4096),s1(4096),ysave(4096),xs(4096) as double
dim c1lin(4096),s1lin(4096),decon,x0,anoise,apos as double
dim i,j,k,k0,k1,k2,k20,nbands,npoints,ioption,n2 as integer
dim s0 as string
dim Q,Qgaussian,Qnoise as boolean
Qgaussian=true
Qnoise = false
anoise=0.0
if Qnoise then anoise=0.002 //180,151 works
clear_text_output(0)
set_graphics_slider(65)
buffer_create_multiple(-4,0,1000,600)
npoints=4096
for i=1 to npoints
  x0=i/2
  x(i)=x0
  y(i)=2/(30 + 0.5*(x0-95)^2)+1/(30 + 0.5*(x0-105)^2) _
  +1.5/(30 + 0.5*(x0-115)^2)+anoise*(rnd-0.5)
next
numerical_normalize(y(),npoints,false,true)
plot_fontsize = 32
plot_fontname = "Arial"
fft1(y(),c1(),s1(),npoints)
ft_linearize(c1(),s1(),c1lin(),s1lin(),npoints)
for i=1 to npoints
  ysave(i)=y(i)
  y(i)=c1lin(i)
  xs(i)=i
next
numerical_normalize(y(),npoints,false,true)
plot_set_ticks(100,1000,0.02,0.2)
plot_data(xs(),y(),npoints,0,npoints,0,1,"Coefficient","Initial Transform",0.5)
buffer_copy_to_buffer(3)
n2 = npoints
s0=input("")
k1=val(s0)
if instr(s0,",")>0 then
  k2=val(nthfield(s0,",",2))
else
  k1=180
  k2=145
  if Qnoise then k2=151
end if
if Qgaussian then
  k20=k2
else
  // softer apodization with Lorentzian deconvolution
  k20=sqrt(k2)
end if
if k1>0 then //
  k=n2/k1
  k0 = n2/(2*k1)
  for i=1 to npoints
    if Qgaussian then // gaussian decay
      decon = exp(-i/k)
    else // lorentzian decay
      decon = k0/((k0 + i)^2)
    end if
    // next two statements do both deconvolution and apodization
    c1lin(i) =  (c1lin(i)/decon)*pow(abs(i-n2)/n2,k20)
    s1lin(i) =  (s1lin(i)/decon)*pow(abs(i-n2)/n2,k20)
  next
  print(" Fourier transform has undergone "+str(k1)+","+str(k2)+" FSD")
else
  print(" Fourier transform has undergone no FSD")
end if
for i=1 to npoints
  y(i)=c1lin(i)
next
numerical_normalize(y(),npoints,false,true)
plot_set_ticks(100,1000,0.02,0.2)
plot_data(xs(),y(),npoints,0,npoints,0,1,"Coefficient","Deconvolved Transform",0.5)
buffer_copy_to_buffer(4)
ft_fold(c1(),s1(),c1lin(),s1lin(),npoints)
fft1_inverse(y(),c1(),s1(),npoints)
numerical_normalize(y(),npoints/2,false,true)
plot_set_ticks(2,50,0.02,0.2)
plot_data(x(),y(),npoints,0,200,0,1,"Wavenumber","Reconstructed Intensity",0.5)
buffer_copy_to_buffer(2)
plot_data(x(),ysave(),npoints,0,200,0,1,"Wavenumber","Relative Intensity",0.5)
print(" Select Gaussian and Noise options at top of program")
print(" enter FSD level,apodization level in input window")
buffer_copy_to_canvas(active_canvas,0)
```

noise, trial and error is the best approach. The following figures provide insight into the process as well as the problems associated with noise.

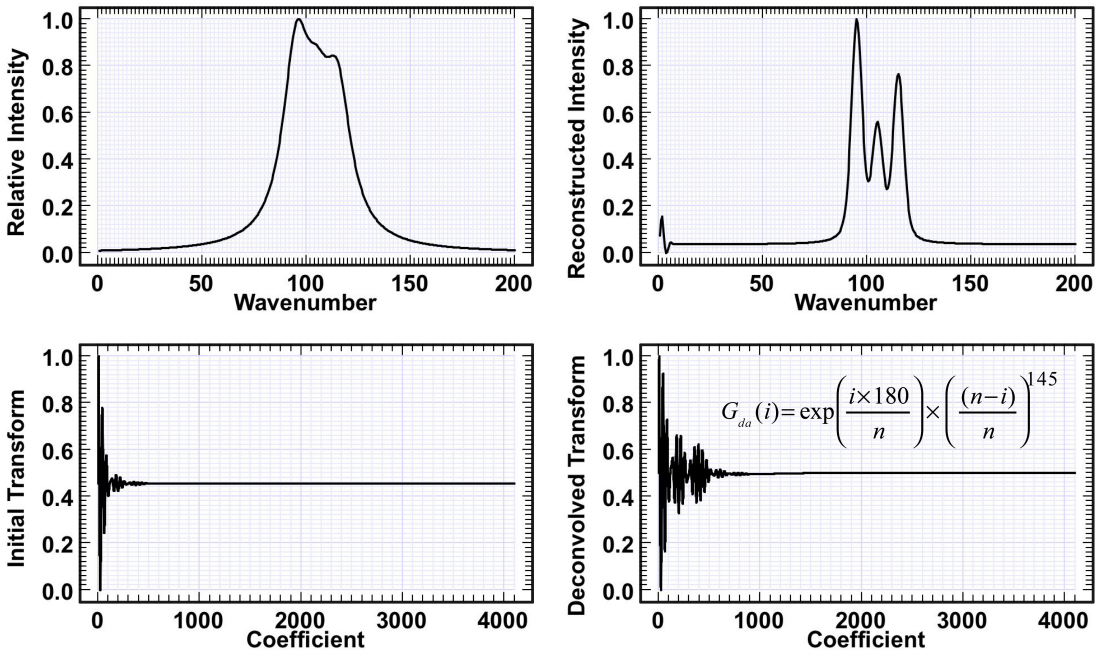


Figure 4.7.3. Fourier-Complex-Self-Deconvolution using the lineshape function of Eq. 4.7.11 and the apodization function of Eq. 4.7.10. The combined function is shown in the insert in the lower right panel. The original spectrum and its Fourier transform is shown in the two left panels, the deconvolved coefficients are plotted in the lower right panel and the resulting spectrum generated by inverse Fourier transformation is shown in the upper right panel. The features near 0 wavenumber are edge noise artifacts associated with slight over convolution. These spurious features are a common consequence of CFSD and provided they are outside the region of interest, of no concern.

As can be seen by examination of Fig. 4.7.3, the deconvolution process essentially dampens (prolongs) the decay of the oscillations in Fourier space. This process is to a first approximation the reverse of the apodization process shown in 4.7.2. Given that, one questions why it is necessary to use both a deconvolution and an apodization in the treatment. If no apodization is carried out, the line shape function which appears in the denominator starts to approach zero fairly quickly for narrow lines. The apodization clamps the Fourier coefficients to prevent spurious features of the type observed at low wavenumber in the upper right panel of Fig. 4.7.3. The best results are obtained when the line shape function and the apodization function are carefully balanced.

The example shown in Fig. 4.7.4 is an example of using a quadratic decay (Eq. 4.7.12) to represent the line-shape function. This function is not as realistic as the exponential decay (Eq. 4.7.11), but is better behaved when noise is present (see below).

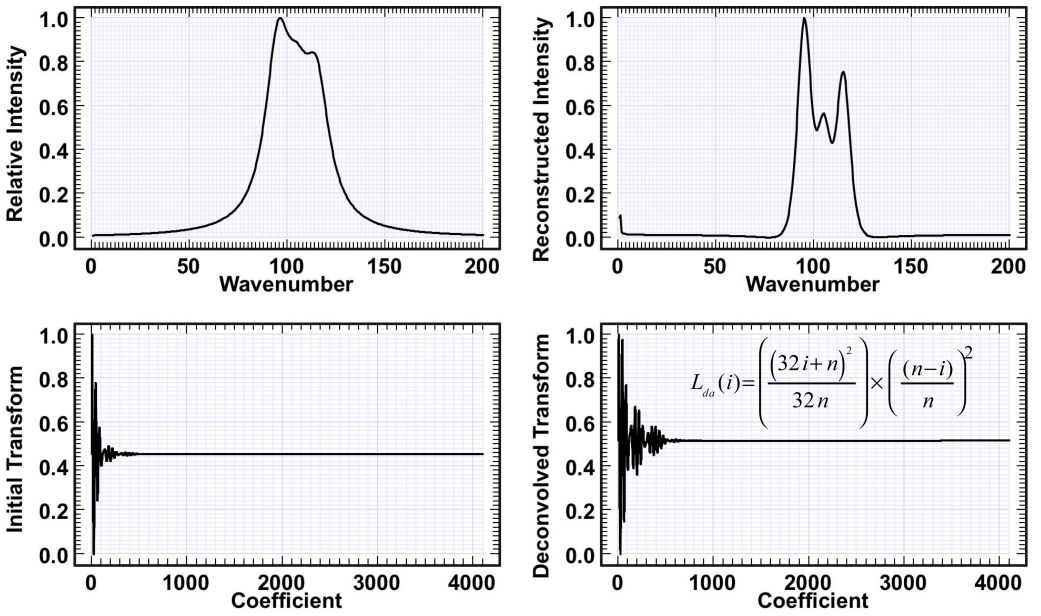


Figure 4.7.4. Fourier-Complex-Self-Deconvolution using the lineshape function of Eq. 4.7.12 and the apodization function of Eq. 4.7.10. Other details as in Fig. 4.7.3.

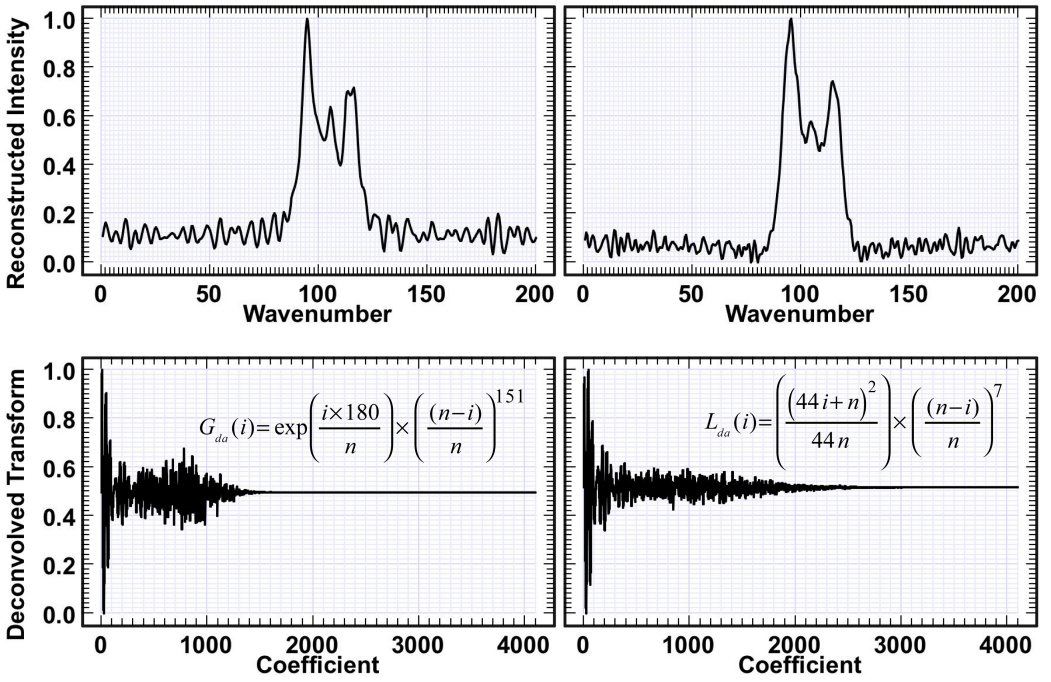


Figure 4.7.5. The effect of noise and lineshape function on Fourier-Complex-Self-Deconvolution of the spectrum shown in Figure 4.7.4. The noise level was one-tenth of the noise present at upper right.

The FCSDs shown in Fig. 4.7.5 provide a perspective on how the two lineshape functions behave when there is noise present. The exponential function has the advantage of providing a realistic representation of a true lineshape, whether Lorentzian or Gaussian. This observation is a source of confusion for students, so it is important to recognize that the Fourier transform of symmetric band, whether Lorentzian or Gaussian in shape, is an exponential function. Thus, Eq. 4.7.11 is invariably the best choice if one is dealing with a pure lineshape. However, the quadratic function shown in Eq. 4.7.12 is preferable if significant noise is present because it distributes the noise more evenly and provides some fortuitous cancellation of the noise. This observation is demonstrated by comparing the performance of these two functions when there is modest noise present in the spectrum (Fig. 4.7.5). The exponential function enhances the noise in frequency regions well below where the original noise was present. In contrast, the quadratic function tends to keep the frequency of the noise background closer to the original, and while it is enhanced, the enhancement is only about 60% of that observed for the exponential function. Thus, the quadratic function, while artificial, is a better choice when there is a great deal of noise present.

4.8. Maximum Entropy and Linear Prediction

We explore in this section one of the more interesting numerical methods known as maximum entropy. This concept is based on the premise the in both thermodynamics and in predicting events based on probability, the systems under study will naturally seek a state of maximum entropy. Indeed, the first implementations of this concept were made by E.T. Jaynes in 1957 by demonstrating that the concept of entropy in thermodynamics can be generalized to information theory. One important application of maximum entropy is in the prediction of future events based on a set of measurements of the process. In that regard, it is important to recognize that the principle of maximum entropy requires prior information to make useful predictions. This principle has had a profound impact on the mathematical technique called linear prediction.

Linear prediction is the process of predicting future values of a time-dependent signal as a linear function of previous samples of the signal:

$$p(n) = \sum_{i=1}^{m} a_i p(n-i) + x_n \qquad (4.8.1)$$

where p(n-i) is the value of the periodic function at point n-i, p(n) is the extrapolated value of the function at point n, a_i is the ith linear prediction coefficient and x_n is the discrepancy between the actual value and the extrapolated value at point n. Linear prediction as implemented in MathScriptor requires that the periodic function, p, be measured at equidistant points in time.

Maximum entropy turns out to be the most reliable method of calculating the linear prediction coefficients provided the data to be extrapolated has adequate length and has been sampled to avoid aliasing. The methods and procedures are based on a coupling of Levinson-Durbin recursion methods with Burg's maximum entropy formula. A detailed discussion of these methods is beyond the scope of this book, and the interested reader is referred to chapter 13 of Ref. 4 for an excellent discussion. Our goal in this book is to explain how to use these methods. And there are many uses of these formulas which are interesting and, in some cases, financially viable.

```
// njk = number of days of past history to analyze
njk=23
valstock=0.0
for i=1 to ns
  x(i)=i
  // y(i) holds the actual stock price for the ith day
  y(i)=value(spreadsheet_cell(i,8))
  yinvest(i)=0.0
  if i>njk then
    k=0
    for j=njk downto 1
      k=k+1
      // yp(1..njk) holds the stock prices for the last njk days
      yp(k)=y(i-j)
    next
    // generate the linear prediction coefficients to fit the last njk days
    numerical_maxent_lpc(yp(),lpc(),xms,k,k-5)
    // use the coefficients to predict the stock price tomorrow
    numerical_maxent_extend(yp(),ynew(),lpc(),k,k-5,2,0)
    if ynew(1)>y(i-1)  or xms>0.0002 then
      // if tomorrow 's price is going to be greater or if the linear
      prediction is ambiguous, hold on to the stock
      valstock=valstock+y(i)-y(i-1)
    else
      // otherwise sell it and then buy it back
      valstock=valstock+y(i-1)-y(i)
    end if
    yinvest(i)= valstock
    ylongterm(i)=ylongterm(i-1)+y(i)-y(i-1)
  end if
  show_progress_bar(100*i/ns)
next
```

4.8.1. Maximum Entropy and the Stock Market.

There are two ways of investing money in the Stock Market. One method is to actually know what you are doing, analyze the financial and strategic characteristics of the company involved, and purchase the stock based on sound economic principles. This approach is to be recommended, but it doesn't always work because there are too many variables outside of the control of the company and the investor. An alternative method is to use linear prediction and maximum entropy to predict where the stock is headed and make decisions based on these predictions. The reader may be surprised to learn that virtually all investment firms use this method as well, particularly to make short term decisions. I discovered this fact from a former graduate student, who after obtaining a Ph.D. in theoretical chemistry decided that predicting stock market trends was more rewarding (financially) than predicting how electrons respond to electromagnetic radiation inside a molecule. This young scientist is now writing maximum entropy linear prediction algorithms to predict stock trends and programming a computer to make buy/sell decisions within seconds when a reliable trend is discovered.

Let us now explore how this might be done using the internal methods in MathScriptor. The relevant portion of the program is shown above, and the application to an analysis of the stock of General Electric Corporation is shown in Fig. 4.8.1. The routine is fully commented and is mostly self-explanatory. The basic idea is that 23 days of stock

prices are fed into the numerical_maxent_lpc() method to generate a set of linear prediction coefficients using Levinson-Durbin recursion and Burg's maximum entropy approach. The method returns not only the coefficients but also the root-mean-square error associated with the linear prediction, xms. Following this analysis, the coefficients are used to make a prediction for the next days stock. If the stock is predicted to go up, the stock is held (not sold). However, if the stock is predicted to go down, the stock is sold immediately and then bought at the end of the next day. This is done whether or not the stock actually went up or not. The only additional feature of the method is to use the mean-square error, xms, to determine whether the linear prediction algorithm has established a reliable set of coefficients. If not, then the stock is not sold regardless of the prediction, which reflects the assumption that when in doubt, revert to the long term investment strategy.

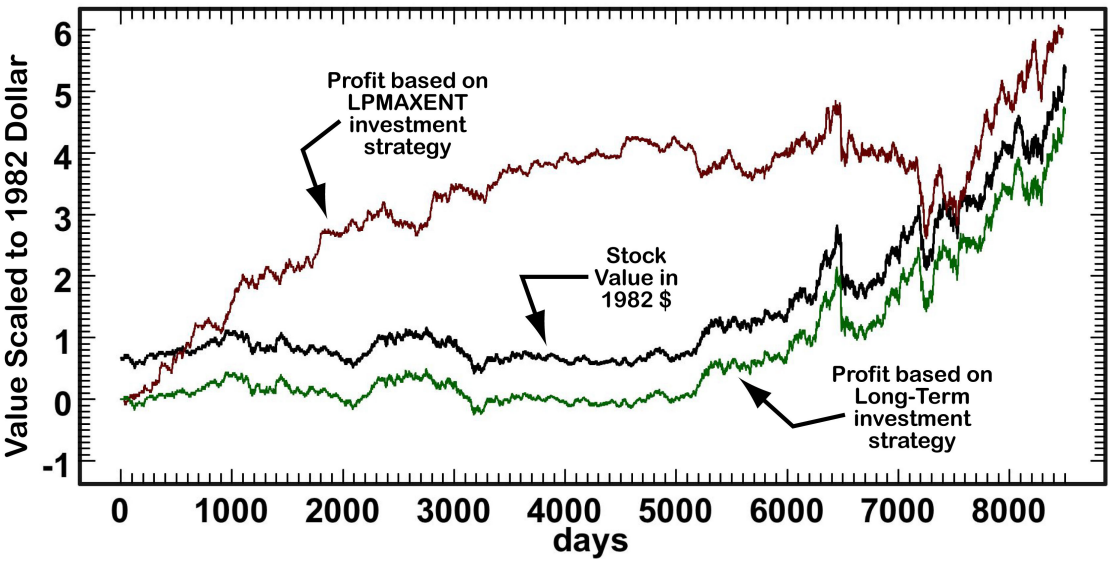


Figure 4.8.1. Comparison of profits made by two investors. One uses linear-prediction maximum entropy to make buy-sell decisions (top curve) and the second buys the stock on day 1 and keeps it for the full 8500 days (bottom curve). The middle curve shows the actual stock price. Note that the value of the stock and the investment income has been scaled to 1982 dollars, and is plotted for a single share.

Although the analysis shown in Fig. 4.8.1. suggests that one would do well to use maximum entropy linear prediction to make buy/sell decisions in the stock market, the simulation assumed that there were no brokerage fees involved in the numerous buy-sell decisions made during the course of this simulation. If realistic brokerage fees were included in this simulation, the maxent-LP investment strategy would not beat the long-term strategy of simply holding on to the stock. During the early stages, the maxent-LP method did extremely well, but then failed during the latter portion of the simulation to hold on to the gains. The reason for this particular failure is unknown, but is typical of

maxent-LP investment strategies. Sometimes they work really well and sometimes they work very poorly. The use of the mean-square error to predict when failure is likely helps significantly, but is unable to provide a-priori prediction of abrupt external events which impact the stock price.

Thus, our simple program should not be used to make investment decisions. If one is serious about doing this, one must use maxent-LP methods to follow multiple interrelated variables relevant to the stock. The professional programs that have been developed by the investment firms will typically monitor thousands of stocks as well as other variables simultaneously and use these trends to make correlated decisions. Not surprisingly, none of these investment firms make their programs public, which precludes further discussion.

4.8.2. Maximum Entropy Linear Prediction Resolution Enhancement.

We return to the use of Fourier-Complex-Self-Deconvolution and demonstrate that the combination of FCSD with maxent-LP provides an optimal method of increasing the resolution of a spectrum. The best-known example of this method is called LOMEP, which stands for Line Shape Optimized Maximum Entropy Linear Prediction [J. K. Kauppinen, D. J. Moffatt, M. R. Hollberg, and H. H. Mantsch, "Characteristics of the LOMEP Line-Narrowing Method, " Appl. Spectrosc. **45**, 1516-1521 (1991)]. The approach we demonstrate here is a simplified version, but it maintains the key elements of the LOMEP method.

```
// k3 is the number of points to use to generate the linear prediction
coefficients as well as the starting point for the extrapolated data
iap=3 // select concave apodization
ncoef=min(k3-10,npoints-k3+1) // number of lp coefficients
redim lpc(ncoef)
numerical_maxent_lpc(c1lin(),lpc(),xms,k3,ncoef)
print("Maxent xms (real) = "+str(xms)+" for "+str(k3)+" pts")
numerical_maxent_extend(c1lin(),y(),lpc(),k3,ncoef,npoints-k3,iap)
k=0
for i=k3+1 to npoints
  k=k+1
  c1lin(i)=y(k) // add new points after k3
next
numerical_maxent_lpc(s1lin(),lpc(),xms,k3,ncoef)
print("Maxent xms (complex) = "+str(xms))
numerical_maxent_extend(s1lin(),y(),lpc(),k3,ncoef,npoints-k3,iap)
k=0
for i=k3+1 to npoints
  k=k+1
  s1lin(i)=y(k)
next
```

The first step is to carry out FCSD using the methods described in section 4.7.3. The second step is to use maxent-LP to expand the Fourier series thereby enhancing the resolution (see program insert). Recall that the sharper the spectral line, the slower is the decay of the cosine wave associated with the spectral feature (see Fig. 4.7.2). In the LOMEP method, and the adaptation explored here, maxent-LP is used to extrapolate the Fourier series to longer time thereby decreasing the line-widths of the constituents features.

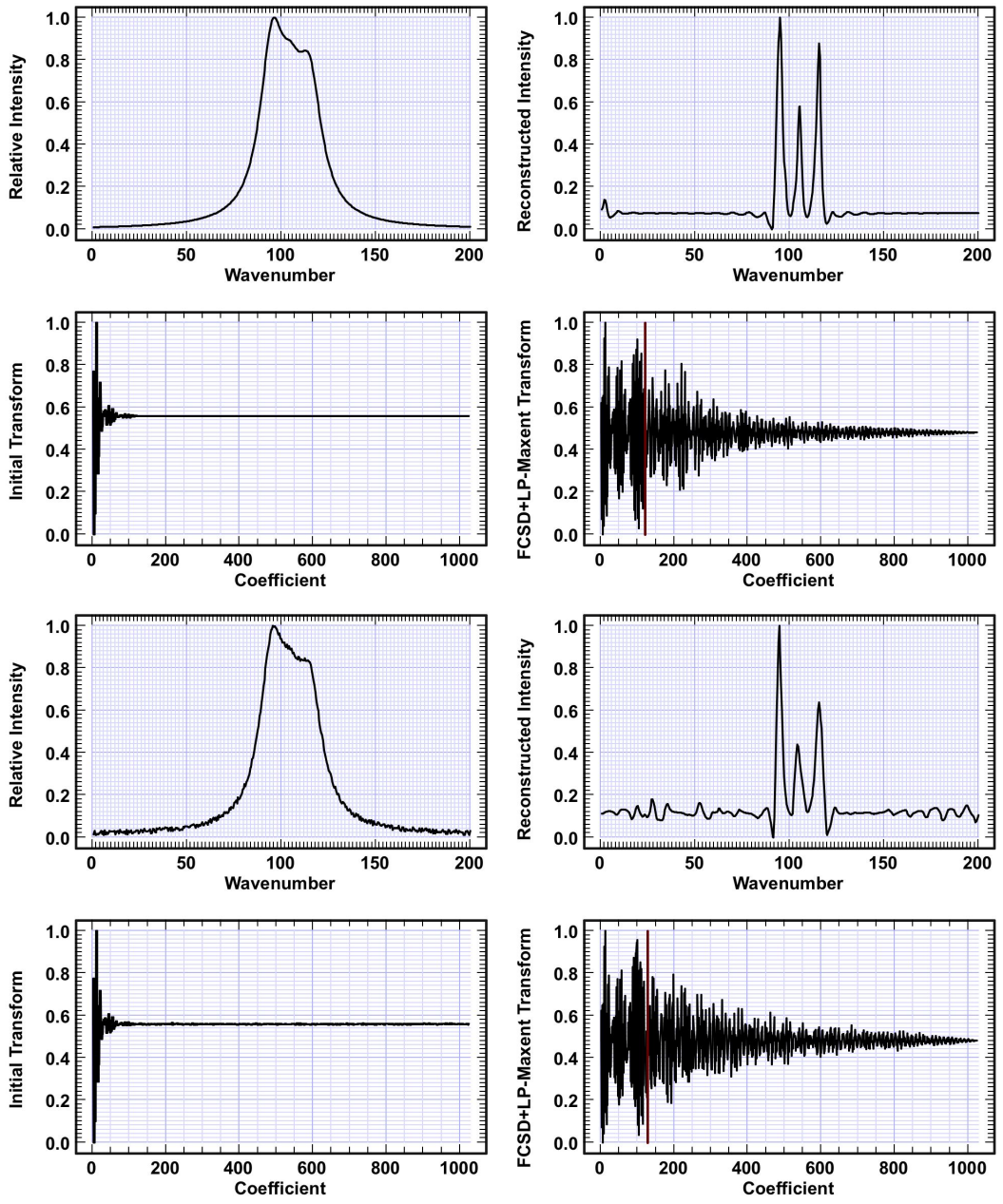


Figure 4.8.2. Lineshape narrowing by using maxent-LP to extrapolate and amplify the FCSD results based on the use of an exponential line-shape function (Eq. 4.7.11). The top set of four graphs show enhancement of the FCSD results shown in Fig. 4.7.3. The bottom set of four graphs show enhancement of the FCSD results shown in Fig. 4.7.5, which include a small amount of noise in the original spectrum. Note that maxent-LP does an excellent job of handling noise.

References for Chapter 4

1. A First Course in Numerical Analysis, Anthony Ralston and Philip Rabinowitz. Dover (2001). ISBN: 978-0486414546

2. Numerical Methods for Scientists and Engineers, R. W. Hamming. Dover Publications (1973, 2nd Edition, 1987). ISBN: 978-0486652412

3. Numerical Methods for Engineers and Scientists, Joe D. Hoffman. CRC Press (2001) ISBN: 978-0824704438

4. Fourier Transforms in Spectroscopy, J. Kauppinen and J. Partanen, Wiley VCH (2001). ISBN: 978-3527402892

Chapter 5
Classes

This chapter investigates classes, which are powerful objects that allow the programmer to expand or customize the capabilities of the extended basic language. Classes were briefly introduced in Section 2.9.1, and the purpose of this chapter is to revisit this topic in sufficient detail to permit the reader to write classes that are useful, powerful and encapsulated.

Classes are available in all modern object oriented languages, and the most common use of classes is to create new types of variables and establish the methods and attributes for manipulating those variables. Indeed, all variable types are represented by classes, and if you type in a declaration such as the following:

```
dim fubar as boolaboolean
```

that makes reference to an unsupported variable type, many compilers display the following error: There is no class with this name. Some of the more modern compilers use a less obtuse error statement such as: Can't find a type with this name. Regardless of the error message, the key point is that each variable type is represented by a class which provides not only the types of variables available, but the rules necessary to manipulate those variables.

A class can be defined to do just about any task that is needed. Because a class can fundamentally alter or extend a language, classes represent the highest objects in the object oriented programming family. Professional programmers spend a majority of their time writing or modifying classes, because classes (when properly written) are fully encapsulated and independent of the other objects in a program. Thus, a programmer can use a class written by another competent programmer with confidence that it will not interfere with other objects within their program. The structure of classes is designed to guarantee that classes are encapsulated, but there are ways a programmer can write a class improperly so that it leaks or manipulates objects outside of its domain. This chapter seeks to not only explain how to create and optimize classes, but also what problems to avoid when writing a new class.

5.1. The Class Structure

Classes are identified by using the "**class** classname" statement and the end of the class is identified with the "**end class**" statement. Each class can have constructor and destructor subroutines which serve to initialize or close down the class. The best approach to understanding classes is to explore examples. The first example is a circle class which does nothing more than return the area and circumference of a circle (see

below). Each class must have a unique name that is used by the program. One or more variables can be assigned to represent the class by using a declaration statement such as:

```
dim c1, c2 as circle
```

which indicates that these variables can be used in the program to represent the class. But these variables cannot be used until they are instantiated, a process that is unique to classes and has the appearance of a standard assignment statement. For example, to instantiate the variable c1 to hold the circle class use:

```
c1 = new circle(3)
```

This statement carries out the following two processes. First, it associates the variable c1 as a holder of the class called circle and copies the entire class into this variable, including all of the class variables. Second, it fires the constructor. If there are multiple constructors, one is selected based on the arguments. There are two constructors in this class. One does not have any parameters and the second has a single parameter. The above instantiation seeks a constructor that can use the parameter 3, so it selects the second of the two constructors. When executed, this constructor takes the parameter rad, which now equals 3, and executes the statement radius=rad. Now the private variable radius is assigned the variable rad (which is in this case 3). There is a natural inclination to write the constructor in a more simple form such as sub constructor(radius as double) in the hopes that upon instantiation the passed variable will go directly into the class variable of the same name declared at the top. This does not work because the two variables are different. One is local to the constructor and the other is global to the entire class. The local assignment takes precedent, and the global class variable never gets assigned at all. Thus, you need to pass the parameter into a local variable that is subsequently assigned to the global variable. An alternative is to instantiate without assignment by executing the statement,

```
class circle
  dim radius as double
  dim status as string
  sub constructor()
    // allows instantiation without properties assignment
    status="undefined"
  end sub
  sub destructor()
    // this is fired when the class goes out of scope
    if status="undefined" and radius=0 then
      print("Destructor for circle with undefined radius has fired")
    else
      print("Destructor for circle of radius="+str(radius)+" has fired")
    end if
  end sub
  sub constructor(rad as double)
    // instantiation with assignment of radius
    radius = rad
    status="defined"
  end sub
  sub assign_radius(assigns rad as double)
    // assigns the radius
    radius=rad
    status="defined"
  end sub
  function area() as double
    // returns the area for radius
    return  const_pi*radius^2
  end function
  function circumference() as double
    // returns the area for radius
    return  2.0*const_pi*radius
  end function
end class
```

```
c2 = new circle()
```

Now the variable c2 is associated with a new copy of the class, but the radius is undefined. We include a string called status which is set equal to "undefined" to allow the user to monitor the status and could be used within the class to alert the user not to trust the area or circumference yet. No need because the radius is assigned to be zero when the class is instantiated automatically, so in fact, the status variable is really not needed in this case. It is included for demonstration purposes. But now the question is, how do we assign the radius to the class variable c2? A subroutine called assign_radius has been included to do this assignment:

```
c2.assign_radius=1.2345678
```

This works just fine, but there is an even simpler mechanism available. This subroutine is not needed at all because we can access any of the class variables declared at the top of the class by using the dot extension method. Thus, to assign the radius to c2 we execute:

```
c2.radius=1.2345678
```

and the same goal is accomplished directly. However, we need to know that this variable is available, which is why many classes are written using subroutines to make all variable assignments. Indeed, if one is concerned that a user will mangle a class variable by accident or due to some personality disorder, one has the option to make all class variables private as follows:

```
private dim radius as double
private dim status as string
```

Private variables are not available outside of the class and are fully protected. The use of private class variables allows an additional level of encapsulation. But if one is writing classes for a larger project where others are making use of it, it is more common to keep many of the class variables public because the only way they can be interrogated or modified is via extension such as c1.radius or c2.radius, etc. Note also that because the entire class along with its variables are copied into the c1 and c2, c1.radius is not equal to c2.radius. They are independent variables stored in different locations in memory.

The area and circumference data are available by executing c1.area, c1.circumference, c2.area, and c2.circumference. While it might appear to the program that these are variables, in this case we are executing a method within the class. Both variables and methods can be made private. If we had assigned the function **area** to be private:

```
private function area() as double
```

then the statement `c1.area` would have generated the following error:

```
Err(54) = This method is protected. It can only be called from within its class.
```

The programmer can help make a class more fully encapsulated by making all functions not intended to be accessed out side the class private.

We close this example by examining the destructors, which are subroutines that fire when the class goes out of scope, and is no longer available to any program element. Destructors must be subroutines because no value can be returned and destructors can accept no arguments. The primary use of destructors historically was to release the memory which was allocated for scratch space by the class. Modern memory management has largely made this process unnecessary, but destructors still have their use. For example, versions of mathscriptor above 2.0 include functions for interfacing with phidgets, which are electronic boards which connect to the computer via usb and which can monitor voltages and control servos, stepper motors, relays, LEDs and many other devices. Classes can be written that control the phidgets and automatically go out of scope when the process, which can be carried out independent of the program, is complete. The class can then alert the program by setting a global variable that the process is done. A destructor can also be used for classes that are operating within separate threads to alert the calling program that the thread process is complete. Students rarely need to use destructors, and their inclusion within a class is optional.

```
// Program Name: demo_class_inheritance.txt
// This program demonstrates a geometric_object
// class and a square class with inheritance.
class geometric_object
   dim radius as double
   dim length as double
   dim width as double
   dim object_type as string
   sub constructor(rad as double)
      radius = rad
      object_type="sphere"
   end sub
   sub constructor(ln as double,wd as double)
      object_type="rectangle"
      length=ln
      width=wd
   end sub
   function area() as double
      if object_type="sphere" then
         return  const_pi*radius^2
      else
         return length*width
      end if
   end function
end class

class square
   inherits geometric_object
   sub constructor(side as double)
      object_type="square"
      length=side
      width=side
   end sub
end class
// Main
dim g1,g2 as geometric_object
dim g3 as square
g3=new square(4)
print(g3.object_type +" with sides = "+str(g3.length))
print("has an area of "+str(g3.area))

// end program
```

5.2. Class Inheritance

One of the important attributes of modern object oriented languages is inheritance based on class hierarchy. In general, objects inherit the properties, methods and events of the superclass upon which they are based. A simple example is that functions and subroutines inherit all of the properties that were declared in the Main Program. In contrast, the Main Program does not have access to the properties declared inside a function or subroutine because the Main Program has a higher position in the object oriented hierarchy than the functions and subroutines that serve it.

A class inherits the properties of another class by using the **inherits** keyword as shown in the example at right. By default, all classes that you write are given an equal status in the object hierarchy until an **inherits** keyword is found, and then that keyword serves to designate the target as the superclass for that pair of objects. In the example at right, the class square adds a new constructor, takes advantage of all the methods and properties of the parent class geometric_object.

```
function plus(c1 as complex, c2 as complex) as complex
   // return c1 + c2
   dim a,b,c,d as double
   dim cw as complex
   a = c1.real
   b = c1.imag
   c = c2.real
   d = c2.imag
   cw = new complex(a+c,b+d)
   return cw
end function
```

5.3. Creating New Variables Using Classes

One of the most powerful uses of classes is to create new variables and define the mathematical functions that operate on the new variables. MathScriptor provides a majority of the variable types that are needed to do object oriented scientific programming. However, MathScriptor does lack a double precision complex variable which forces programmers who need to use complex numbers to work with string complex numbers via the Arprec classes (see section 2.8.1). For most applications, Arprec string complex numbers serve the purpose. But for more extensive complex work where speed is more important than access to high precision arithmetic, the class that we introduce in this section is to be preferred. The primary goal of this section, however, is to introduce the methods and procedures of using classes to create new variables. Thus, our initial discussion will concentrate on the programming aspects rather than the mathematical procedures. We will follow with a detailed discussion of the ancillary functions that are needed to make the class complete.

Complex numbers consist of a real and an imaginary part. Thus, a double precision complex number requires twice as much storage as a double precision real number.

```
function minus(c1 as complex, c2 as complex) as complex
   // return c1 - c2
   dim a,b,c,d as double
   dim cw as complex
   a = c1.real
   b = c1.imag
   c = c2.real
   d = c2.imag
   cw = new complex(a-c,b-d)
   return cw
end function
```

```
function mult(c1 as complex, c2 as complex) as complex
   // return c1*c2
   dim a,b,c,d as double
   dim cw as complex
   a = c1.real
   b = c1.imag
   c = c2.real
   d = c2.imag
   cw = new complex(a*c - b*d,b*c + a*d)
   return cw
end function
```

```
function div(c1 as complex, c2 as complex) as complex
   // return c1/c2
   dim a,b,c,d,denom as double
   dim cw as complex
   a = c1.real
   b = c1.imag
   c = c2.real
   d = c2.imag
   if abs(c)>=abs(d) then
      denom = c + d*(d/c)
      cw = new complex((a + b*(d/c))/denom,(b - a*(d/c))/denom)
   else
      denom = d + c*(c/d)
      cw = new complex((a*(c/d) + b)/denom,(b*(c/d) - a)/denom)
   end if
   return cw
end function
```

```
function abs(c1 as complex) as complex
   // return abs(c1) as a complex number (imaginary part is 0)
   dim a,b  as double
   dim cw as complex
   a = c1.real
   b = c1.imag
   if abs(a)>=abs(b) then
      cw = new complex(abs(a)*sqrt(1 + pow(b/a,2)),0.0)
   else
      cw = new complex(abs(b)*sqrt(1 + pow(a/b,2)),0.0)
   end if
   return cw
end function
```

```
function dabs(c1 as complex) as double
   // return double of absolute value (no imaginary part, which is zero)
   dim a,b  as double
   dim cw as complex
   a = c1.real
   b = c1.imag
   if abs(a)>=abs(b) then
      cw = new complex(abs(a)*sqrt(1 + pow(b/a,2)),0.0)
```

The class shown at right implements a complex number as a pair of double precision variables, one assigned to the real part and the second assigned to the imaginary part. At the top of the class we define two double precision variables. Note that it is necessary to have each declaration on a separate line, a requirement that is peculiar to class variables. We declare cr and ci as doubles to provide a complex number in the form cr + ci*I. These two variables are assigned during instantiation via the constructor. The two functions inside the class allow the real and imaginary parts to be returned as double precision numbers. The complex class is really very simple, but it facilitates all of the external functions that operate on complex numbers to function on a single entity called complex. The entity behaves just like a variable, but it is not declared but rather instantiated. Thus, to create a new complex number 3 + 4i, one must do two things. First declare a variable as complex and second instantiate that variable:

```
dim c1 as complex
c1 = new complex(3, 4)
```

What remains is to write the functions that carry out the math. This is not a trivial process because many of the complex functions have unusual behavior that is unique to complex arithmetic. Shown at right are the functions that provide access to addition, subtraction, multiplication and division as well as absolute. The internal symbols +, -, * and / are not available for use by us because the extended basic language does not allow these symbols to be overloaded. This is a common limitation that many languages impose on programmers to avoid the many problems that might occur if the capabilities of these symbols were overloaded improperly. Thus, we use plus(c1, c2), minus(c1, c2), mult(c1, c2) and div(c1, c2) to replace c1+c2, c1-c2, c1*c2 and c1/c2, where c1 and c2 are arbitrary complex numbers.

The function abs(c1) returns the absolute value of the complex number c1, and takes on two forms. The nominal form, abs(c1), returns a

```
function sqrt(c1 as complex) as complex
   // return sqrt(c1) as complex
   dim a,b,c,d,w  as double
   dim cw as complex
   c = c1.real
   d = c1.imag
   if c=0 and d=0 then
      w = 0
   elseif abs(c)>= abs(d) then
      w =sqrt(abs(c))*sqrt((1+sqrt(1+pow(d/c,2)))/2)
   else
      w = sqrt(abs(d))*sqrt((abs(c/d)+sqrt(1+pow(c/d,2)))/2)
   end if
   if w=0 then
      a = 0
      b = 0
   elseif c>=0 then
      a = w
      b = d/(2*w)
   elseif c<0 and d>=0 then
      a = abs(d)/(2*w)
      b = w
   else
      a = abs(d)/(2*w)
      b= -w
   end if
   cw = new complex(a,b)
   return cw
end function

function arg(c1 as complex) as double
   // return arg(c1) as complex where arg[a+b*i]=atan(b/a)
   dim a,b  as double
   dim cw as complex
   a = c1.real
   b = c1.imag
   return  atan2(b,a)
end function

function log(c1 as complex) as complex
   // return log(c1)  to the base e [ ln(c1) in math notation]
   // Log[w] = Log[|w|]+I*(Arg[w]+2*Pi*k), for any integer k.
   // let k=0 and Arg[w]=imag/real
   dim a,b  as double
   dim cw as complex
   cw = new complex(log(dabs(c1)),arg(c1))
   return cw
end function

function exp(x0 as complex) as complex
   // return exp(x0) as complex
   // exp(a + b i) = exp(a) Cos[b] + I*exp(a)*Sin[b]
   dim xx,sinx,ff,cc as complex
   dim a,b,yr,yi,ex as double
   a=x0.real
   b=x0.imag
   ex=exp(a)
   yr=ex*cos(b)
   yi=ex*sin(b)
   cc=new complex(yr,yi)
   return cc
end function

function pow(x0 as complex, y0 as complex) as complex
```

complex number with an imaginary component of 0. It is sometimes useful to a program to provide only the real portion as a double, so dabs(c1) is included, which returns a double.

The other functions are shown on the next page. All of these functions have overloaded versions that handle strings or doubles, and the compiler is capable of handling functions that reference a new variable that is created by a class. One might question why these functions can be overloaded while the +, -, * and / functions cannot. In principle, the compiler could allow all of these objects to be overloaded. But to allow the basic symbols to be overloaded represents a significantly higher level of compiler sophistication than to overload simple functions which can be parsed in a single pass. Thus, one should not be surprised to find that the compiler allows for standard function overloading but does not allow operator overloading of the basic math symbols.

In general, each of these functions creates and instantiates a new complex number which is returned by the function to the calling program. The value is returned by the function, and with the exception of the dabs() function, the statement needs to be prepared to receive a complex number. The most straightforward approach is to instantiate a complex variable to receive the result. But, one can use the complex value returned in any function that can handle a complex. For example, the following statement is allowed:

```
function sin(x0 as complex) as complex
  // return sin(x0) as complex
  // sin(a + b i) = sin(a)cosh(b) + cos(a)sinh(b) i
  dim xx,sinx,ff,cc as complex
  dim a,b,yr,yi as double
  a=x0.real
  b=x0.imag
  yr = sin(a)*cosh(b)
  yi = cos(a)*sinh(b)
  cc=new complex(yr,yi)
  return cc
end function

function cos(x0 as complex) as complex
  // return cos(x0) as complex where
  // cos(a + b i) = cos(a)cosh(b) - sin(a)sin(b) i
  dim cc as complex
  dim a,b,yr,yi as double
  dim i,j,nterms as integer
  a=x0.real
  b=x0.imag
  yr = cos(a)*cosh(b)
  yi = -sin(a)*sinh(b)
  cc=new complex(yr,yi)
  return cc
end function

function tan(x0 as complex) as complex
  // return tan(x0) as complex (sine over cosine)
  return div(sin(x0),cos(x0))
end function
```

```
ctot=  div(mult(plus(c1, c2), abs(c1)), minus(c1, sqrt(c2)))
```

To be clear, then, one can use complex variables and the results of the complex functions in any statement or equation just like doubles, provided a function has been written to handle the target math.

```
class student
  // class variables available via extension method
  dim name as string // full name
  dim level as string // 1=fresh,2=soph,3=junior,4=senior
  dim honors as boolean // true if honors student
  dim sid as integer // student ID number
  dim semester_gpa(24) as double // grades by semester

  sub constructor(s1 as string)
    name=s1
  end sub
  sub constructor(inum as integer)
    sid=inum
  end sub

  function gpa() as double
    dim i,n as integer
    dim w as double
    w=0.0
    n=0
    for i=1 to ubound(semester_gpa)
      if semester_gpa(i)>0.0 then
        w=w+semester_gpa(i)
        n=n+1
      end if
    next
    return w/n
  end function
end class
// Main
dim s(10) as student
dim i,j,k,n as integer
dim s1 as string
dim gpa,sum,w as double

s(1)= new student(1713268)
s(1).name="Clark Gable"
s(1).honors=false
s(1).level="junior"
s(1).semester_gpa=array(0.0,2.9,2.97,3.21,_
3.22,4.31,4.07,4.22)

s(2)= new student("Raymond Chandler")
s(2).honors=true
s(2).level="senior"
s(2).sid=1735542
s(2).semester_gpa=array(0.0,3.3,3.5,3.56,_
3.44,3.78,3.92,4.55,4.35,4.22)

for i=1 to 2
```

5.4. Database Classes

Some languages include records or structured declarations which allow the programmer to create a collection of variables of different types that are referenced by using a single name. Such structures (records) are invaluable for creating databases. Unfortunately, our version of extended basic does not include records or structures. Fortunately, classes can do the same thing, and provide additional flexibility and processing capabilities.

The example at right shows a student class which stores information about a student in a single variable declared and instantiated as a class. The example includes only a small number of variables compared to a true student record, but it will serve the purpose. First note that the class declares a set of variables at the top which are of different types. Each time the class is instantiated, all of these variables are copied into the variable declared as type "student". Here we demonstrate using an array, S(1..10), to allow for multiple students. Each array element represents a student, and all the data associated with the student. A new student can be instantiated either by name or student id number, which provides flexibility. After the gpa semester data are entered, the average gpa over all semesters can be accessed by using the `s(i).gpa` extension. Note that the variable gpa is also declared in the main program. This is not the same variable, and in fact, `gpa` is a function that returns the value. There is no conflict, and indeed one could use gpa (the variable in main) to store the `s(i).gpa` value. Outside of the class, it is not possible to tell whether a dot extension is accessing a class variable or a class function as both behave identically. While the above example demonstrates a vastly oversimplified student record, it should be clear how one can expand this class to handle additional data and carry out additional analyses via function calls.

Chapter 6
Advanced Topics

This chapter investigates object oriented programming in more detail, with an emphasis on the optimization of a program to enhance code reliability, maintenance and execution speed. Although code reliability is an important goal from the very first program one writes, it is not realistic to learn the elements of programming while simultaneously optimizing the ability to maintain the program in the future. In fact, most programs written in the course of learning programming will be disposed of after the grade is received. Coding elegance is sacrificed for expedience and the desire to simply finish the assignment.

This approach does not work well when one is writing a program in a research or commercial environment. Programs written under these conditions now must satisfy three goals: reliability, speed and maintainability. This chapter addresses these three goals from various perspectives, and in a depth that is beyond the scope of first semester courses in programming.

6.1. Optimizing Transparency, Maintainability and Reusability

The term transparency refers to code that is written so that the purpose of each line of code is easily deciphered based on standard coding logic. The term maintainability is a reference to code that can be modified or improved by both the original writer of the program and other future programmers. Finally, reusability refers to the ability to use the entire program, or segments of the program, in other applications or for other purposes. The goals of this section are to explain why these goals are important, and to provide insights into how these three goals can be achieved.

These three goals are complimentary, and rarely does an improvement in one area not enhance the other two. The issue of maintainability, however, deserves a brief introduction. It may seem that writing code to be maintained by the original writer, and code to be maintained by other programmers, requires a different approach. This assumption is not correct. One should always write code as if someone else will be responsible for maintaining the code. No human has perfect memory, and a section of code that appears transparent during the initial coding process may have obscure function when examined a few weeks later by the original programmer. Thus, the additional effort of implementing the following concepts represents an insurance policy that the code written today will have utility tomorrow.

6.1.1. Variable names should reflect function

During the early days of programming, when memory was expensive and compilers were in the early stages of development, variable names were restricted by length. Early implementations of Basic often limited variables to a letter followed by a number. Early Fortran compilers limited variables to six characters, and constrained the first letter to follow variable type rules (e.g. integers must begin with i, j, k, l, m or n). The most significant problem created by this constraint was the inability to use variable names that conveyed function and purpose. Scriptor allows variable names to be 64 characters in length, and that flexibility allows the programmer to assign variables names that are meaningful. And while this might appear to be a superficial capability, the use of properly named variables can make a significant difference in generating code that is transparent (e.g. easy to understand). Lets compare two programs.

```
// Program 1
n1=0.0
k=0.0
do
  i=0
  do
    xx=rnd
    yy=rnd
    i=i+1
    if xx*xx+yy*yy<1.00 then
      n1=n1+1
    end if
  loop until i>10000
  k=k+i
  api = 4.0*n1/k
  print("Current value of pi = "+str(api))
loop until check_for_user_action("user")

// Program 2
hits_inside_circle=0.0
total_trials=0.0
do
  ntrials=0
  do
    x_random=rnd
    y_random=rnd
    ntrials=ntrials+1
    if x_random^2+y_random^2<1.00 then
      hits_inside_circle=hits_inside_circle+1
    end if
  loop until ntrials>10000
  total_trials=total_trials+ntrials
  pi_calc = 4.0*hits_inside_circle/total_trials
  print("Current value of pi = "+str(pi_calc))
loop until check_for_user_action("user")
```

Note that the second program is much easier to follow in terms of program logic. The use of descriptive variables makes the process of deciphering the logic not only faster but also less prone to misinterpretation. The use of comments would further help this process, but even without any comments at all, the use of descriptive variable names makes the program easy to follow.

6.1.2. Optimizing Comments

The idea of adding commentary to a program is as old as programming, and all languages include the option of adding a line of code that is a comment line. As the concepts of reusability and maintainability became more important, languages added the ability to add a comment at the end of a line of code. The programmer should include comments in the following areas, and in the order and/or location listed.

Program Overview and Input Requirements (top, #1). The program should have comments at the beginning which include the name and date of the programmer followed by an overview of what the program does. Following this brief introduction, the comments should include a detailed list of what input is required, and the formatting that the program expects. The importance of a discussion of the input protocols cannot be overemphasized because the future utility of the program depends upon the information provided.

Timing and Quit Options (top, #2). An estimate of the execution time of the program should be provided along with methods available to the user for stopping execution. Programs that run in a few seconds or less do not need any Quit options, but any program that takes more than 15 seconds should have a method of stopping the execution. Time is valuable, and nothing is more frustrating than to be forced to wait for a program to end knowing that there is some problem that will prevent the output from being useful (e.g. missing or incorrect input data). Scriptor provides many different methods of monitoring escape/quit situations, and the user needs to know whether to push Q, escape, command+period, the mouse button, the stop button or some other key.

External Objects (top, #3). If a program relies on the availability of external objects (methods or classes), these objects should be named and the local paths and filenames should be given. This information should be provided even if the program is loading these objects automatically. The comments in this area should include an overview of what the external objects do in terms of the overall functionality of the program.

External Files (top, #4). If a program relies on the availability of external data files, these should be listed and the local paths and filenames should be given. This information should be provided even if the program is loading the files under program

control, because if these files are not present, the program will likely require user intervention. In that regard, the nature of the data within the required file should be described. For example,

```
// This program opens a text file called "dictionary.txt" inside the folder "user_files". This file should contain a list of correctly spelled words separated by end_of_line markers (const_eol). The program can adjust to any length file, so larger or smaller dictionaries can be substituted.
```

Section Function (place prior to relevant section). Comments should precede any section of code that performs a specific task. The section of code can be short or long depending upon how effectively the variables have been named to indicate the meaning or function of these variables. If is very helpful to reusability if the section of code is indicated to be CPU intensive and important to the overall timing of the program. Future programmers may wish to target this section for optimization.

Line Function (following code on same line). One can add a short comment at the end of a line of code to indicate what that line does. The necessity for such comments is often alleviated when the programmer uses descriptive terms for the variables.

Comment Markers. There are three comment markers available. The standard C-type comment marker, //, and the Basic or Fortran marker (the single quote '), are interchangeable. These markers indicate the start of a comment, and the comment continues until the end-of-line marker is encountered. If a long comment is desired, and the comment will be divided up into paragraphs, a new comment marker is required at the beginning of each paragraph.

The Rem Marker. The "Rem" marker is also available, but is designed to be used by the instructor to make comments inside a student program. The comments are marked in bright orange, easy to find, and easy to remove by using the "Remove Instructor Comments" menu item under Debug. The programmer is welcome to make use of Rem statements, of course. Rems are commonly used to mark sections where further coding or optimization is required.

6.1.3. Optimizing Structure

The entire concept of structured programming is to enhance the readability of the code and improve the ability of the programmer to maintain and optimize the code. Some programmers operate under the assumption that languages such as extended Basic and C++ provide automatic structuring by virtue of having a full compliment of structured loops and conditionals, which virtually eliminates the need for the GOTO statement. And this assumption is to a certain extent true. But there are things that one can do to optimize structure and readability. The following are presented in descending order, so the most important concepts are presented first.

Use Methods to Enhance Structure, Readability and Reusability. The classical argument for using methods is to eliminate unnecessary coding by reusing code segments in functions and subroutines. This process reduces program size, but it also makes the program easier to read and understand. By making a program modular, and using comments to fully explain function, a program is enhanced in terms of readability and reusability. Furthermore, grouping functions and subroutines together if they share a common purpose enhances readability. Thus, modules should be explored whenever this condition is met, even if there is no need to create a module. (The common situation that prompts the use of modules is a collection of functions or subroutines which use shared variables that are declared as private to the module.)

Exception GOTOs. Despite all the rhetoric in the computer science literature and in this book against the use of goto statements, there are times when the use of a goto statement is not only useful, but wise. Although one can almost always find a structured solution that avoids the use of a goto, the added complexity that results can sometimes obscure rather than enhance program structure.

An example of a program that uses a goto statement to enhance readability is shown at right. This program asks the user a series of questions, and analyzes the response to assign a Jungian personality profile. Parts of this program, along with the questions and analysis section, have been removed to shorten the example. The goto statement is used to handle a situation where a user decides to exit the questionnaire prematurely. It is more logical to place the code that handles this situation at the bottom of the program, rather than insert it inside the code that handles the questionnaire. In this example, the goto statement is handling what is known as a user exception, a non-normal situation

```
// Program Name: goto_example.txt
// this program demonstrates the use of a goto statement that
improves the clarity of program structure
dim squestion(4),sresponse,sresponses(4),s1,sdata(10) as string
dim i,j,k as integer
//  sdata(1..4) hold the questions
s1="Remember: Press / to indicate that you have answered the question."
s1=s1+const_eol+"You can quit at any time by pressing stop or /Q"
for i=1 to 4
   squestion(i)="Question("+str(i)+"): "+sdata(i)+const_eol+s1
next
for i=1 to 4
   print(squestion(i))
   pause(0)
   sresponse=""
   show_progress_line(sresponse) // clear user input area
   do
      // using a / to indicate completion allows use of return inside
      answer which has advantages if the answer is detailed and is in
      paragraph form
      sresponse= input("") //the input remains in its entirety
      if instr(sresponse,"Q/")>0 or check_for_stop_button then
         goto end_with_error_statement //  handle quitter
      end if
      pause(0)
   loop until instr(sresponse,"/")>0
   sresponse = replace(sresponse,"/","")
   print(sresponse)
   sresponses(i)=sresponse // save the responses
   print(string_repeat("-",38))
   // code to analyze the response .... ....
next
print("You have completed the questionaire successfully:")
pause(0)
// carry out detailed analysis and print results .....
return
// following section handles premature exits
end_with_error_statement: // the goto target
s1="You have elected to quit before completing the questionaire.  " _
+"It is not possible to provide a Jungian profile of your personality," _
+" other than to point out that you are a quitter."
print(string_repeat("-",38)+const_eol+s1)
pause(0)
// end program
```

that is created by a user doing something that is outside the normal expected pattern. Using goto statements to handle user exceptions improves the flow of the main portion of the program and makes the program easier to understand and follow. A more general suggestion is that goto statements should be considered whenever the following conditions apply:

1. Exception conditions when the exceptions will require premature exit from a highly structured code segment, and no reentry into this code segment is desired.

2. The goto statement is sent to a label which clearly defines the purpose.

3. Only one or at most two goto statements are being used.

6.2. The Nature and Optimization of Objects

Objects are components of a program that add a new behavior or capability. An object oriented programming environment provides objects that are encapsulated so that interaction of these objects with the other components of the software is fully controlled. Scriptor provides four types of external objects: Functions, Subroutines, Modules and Classes. To this set we add Main, which represents the primary object which orchestrates the flow of the program and interacts with the other objects to carry out the function of the program.

It is worth noting that programming objects take many forms. An object can be as small as a single variable or as large as a 120, 000 line module containing hundreds of methods. Big objects are made up of smaller objects, and all objects need to be able to communicate with each other as appropriate while not contaminating other objects in the process. Contamination can take many forms, some obvious and some subtle. An obvious contamination would be a variable changing the value of another variable when its value is changed. In the early days of Fortran, and the use of common memory blocks to transfer information between methods, it happened accidentally all the time. Such problems were hard to diagnose and led computer scientists to develop languages that prevent, or nearly eliminate, accidental contamination. Indeed, Fortran 90 (one of the more recent versions of this popular scientific language), common blocks are eliminated. In that regard, Fortran 90 is considered to be an object oriented version of Fortran 77. An object oriented language has many built-in features that constrain the programmer to write object oriented code. Much of the work is done by the language itself. One prominent example relevant to Scriptor is the requirement that every variable be declared. Early versions of Fortran autodeclared variables in a “you use it, you own it” approach that made programming quite fast and sometimes disastrous. The problem with autodeclaration is that a simple misspelling of a variable would not

generate an error, but simply create a new variable with the misspelled name. But this would happen automatically and the programmer would not be notified. Programmers could (and often did) spend hours tracking down simple spelling mistakes involving variables. Thus, autodeclaration is not allowed in object oriented languages. A programming language, however, can only do half the job of creating an object oriented program. The programmer must do their part, and the purpose of this section is to more fully examine the methods and procedures of object oriented programming. We must first describe the concept of encapsulation which mediates the interaction between objects. We start our discussion with an allegory.

The watch maker, the surfer and the watch. Imagine an island in the South Pacific where the key industry is fishing. A young watch maker has opened up a new shop to sell his unique watches which are designed with the fisherman in mind. He makes watches which not only tell the time but show the phase and amplitude of the ocean tides. He makes the watch so that the fishermen can know when it is optimal to go fishing (fish behavior is closely related to the tidal rhythms). But one day a surfer buys a watch because he knows that the size and structure of the waves is best when the tide is going out. But the surfer is adventurous and often gets spilled onto the sand, and the watch stops working because the water resistant housing had not been designed to withstand the battering of the watch against the sand. Although the watch still works, it is no longer accurate.

The above story may appear to have no useful connection to object oriented programming, but it demonstrates one of the more important aspects. Assume that the watches represent objects and the purchasers are programmers. The watch has been designed to be encapsulated so that its inner workings are protected from the environment, and yet it must return information to the owner. Thus, the hands of the watch must present data to the wearer. Because the watch is designed by the watch maker to present time following a standard code, the owner can read the time by looking at the face of the watch. He can also follow the instructions to read the fourth hand which points to the tide level markers, from low to high. The human does not need to know how the watch works, only how to read the markings on the face of the watch. What makes the watch function properly is encapsulation. The complex inner workings are protected from the outside world by the case. And yet the surfer has taken the watch into an environment where it was not designed to work, and the encapsulation was found to be inadequate. This kind of problem applies to both watch makers and to programmers. One must make watches and write programs that can handle untested and unanticipated programming environments. Programmers are human and it is nearly impossible to anticipate all of the possible applications of a given program, just as the watch maker did not anticipate the surfer. Object oriented programming provides a set of rules to force rigorous encapsulation that when applied properly guarantees that a programmer using your software will not damage the inner workings of the methods or

classes that you have written. Thus the first goal of this section is to describe the methods of encapsulating objects (from methods to classes) so that the objects are fully protected from other objects, and equally important, these methods will not contaminate code in other objects (Main, other methods or classes). In this regard, the programmer and the watch maker have the same goal. To provide an object that works, interacts with the user following well-defined rules, provides the desired information, and is impervious to outside perturbations.

6.2.1. Encapsulation.

Scriptor, and all modern programming languages, are designed to optimize the encapsulation of objects. But the programmer needs to follow a few rules to achieve this goal. The following rules help optimize encapsulation.

Avoid using external globals. A common trick that most programmers tend to abuse is to declare a variable in the Main program, and then manipulate that variable within a function or subroutine rather than pass the variable as an argument. This approach is popular because few programming tasks are more laborious than typing in a long list of parameters. There are multiple problems with this approach. First, the use of globals means the code is not reusable. Second, the code is not transparent, because the use of globals, even when documented, creates an interdependence that is hard to decipher. And finally, the reliance on global variables breaks the encapsulation of both Main and the methods that share them.

Use ByRef variables only when required. Although the ability to pass a variable by reference provides flexibility, this capability should only be used when necessary. ByVal variables are much safer because they enforce encapsulation.

Classes are the highest form of object. All objects are not created equal. The highest form of object is the class, which provides a combination of power and encapsulation that is not available to any other object. These objects were introduced in Section 2.9.1 and are discussed in detail in Chapter 5. What makes classes so powerful is the availability of constructors and destructors, as well as the process of instantiation, which makes a unique copy of the code within the class so that it has the properties defined by the constructors during the instantiation process. Thus, a single class can provide multiple behaviors depending upon how they were instantiated. In addition, all of the functions within a class are available by using the following calling protocol (notice the “dot”):

```
instantiated_name.method_name(optional parameters) or
variable = instantiated_name.function_name(optional parameters).
```

The process of instantiation provides an additional layer of encapsulation by forcing the programmer to assign a temporary variable to represent the class. We call this the instantiated name. For example,

```
dim instantiated_name as classname  // declaration
instantiated_name = new classname  // instantiation
```

creates an instance of the class called classname. A key advantage of classes is that the programmer has control over the assignment of the instantiated_name. Thus, there is no possibility of name conflicts within the program, which is a dimension of encapsulation that is often overlooked. Furthermore, the instantiated_name-dot-method_name requirement provides further protection from naming conflicts.

In addition to naming advantages, classes also provide sophisticated control over memory usage. This accrues from the availability of a destructor which can carry out memory management prior to the class going out of scope. Because the destructor is called automatically and called only once, the process is easy to implement. We remind the reader that each time a class is instantiated, the entire class is copied into memory and variable memory assigned to the class is allocated. When the class goes out of scope, this memory is returned to the operating system. Hence, the destructor does not need to manipulate the memory allocated to the class, but may need to redim global arrays that were used by the class. Another use of destructors is to release semaphores that were set to prevent other objects from using variables being manipulated by the class in question.

Because classes can be instantiated within classes, it is noted that destructors are called from the inside out. Hence the last destructor that is called is the destructor of the parent class. This arrangement needs to be kept in mind when redimming global arrays that may have been shared among the various classes. Furthermore, destructors take no arguments and cannot return values. The destructor subroutine does have access to all the class variables. Finally, recognize that destructors are optional.

Modules as method encapsulators. The syntax of modules was introduced in Section 2.9. When properly used, modules provide excellent encapsulation of the methods that are enclosed, but because encapsulation is not automatic, some discussion is necessary. The power of modules derives in large part from the ability to declare variables inside the module that can be shared among the methods enclosed. These are called module variables and are not to be confused with the variables declared inside the methods enclosed by the module. The reason modules are not automatically good encapsulators is that all variables and methods default to public. This means that code outside of the module has access to these variables and methods by default, and this is a poor recipe for encapsulation. Many programmers view classes as good object oriented constructs and modules as poor (or dangerous) object oriented constructs, precisely because all class variables are private whereas module variables default to public. But there are times when modules are very useful, and one can write modules that are well encapsulated. The outline shown at right presents a module that is designed to be properly encapsulated. First, there are no public module variables declared. Second, all public methods are named by reference to the name of the module followed by a descriptor ("_uniquename"). The goal is to facilitate code transparency. This approach turns a module into an object that has superficial behavior reminiscent of a class. Modules do not require instantiation, and operate in a simpler fashion. There are times when modules are the best choice.

```
module modlname
  // the module name, modlname, is chosen to reflect the
  common capability of the enclosed methods.

  // declare variables that are shared among the methods as
  private. Do not declare any public variables as these will
  damage encapsulation

  private dim var1 as double
  private dim ic1 as integer

  // private methods are defined next and are available only to
  methods inside the module.  Any naming convention is
  allowed because they are invisible to outsiders

  private sub a1(i as integer, byref x1() as double)
    // This subroutine is available to only that code that is
    within the module. Any properties declared are local to the
    subroutine
    ....
  end sub

  // public methods are defined next and because they are
  available to all code, they are named following the
  convention:  modlname_methname

  public sub modlname_subname1(x2() as double)
    // this subroutine is available to all code any properties
    declared are local to the subroutine the name, subname1,
    should explicitly express what it does (e.g. complex_invert)
    ....
  end sub

  public function modlname_funcname1(x3 as double) as double
    // this function is available to all code (default is public) any
    properties declared are local to the function the name,
    funcname1, should explicitly express what it will return
    (e.g. complex_abs)
    ....
  end function

end module
```

Object Errors. Most of the compiler errors that one encounters are easy to understand, and as such, are easy to fix. But there is one type of error that is a source of confusion for new programmers, and this error involves the nature and implementation of ByRef and ByVal parameters. This is an example of an object error created when the properties of an object are violated by the programmer.

Consider the following error:

Error number = 21 near line number nnnn
Err(21) = Can't pass an expression as a ByRef parameter.

This error is often encountered when using the following statement:

```
open_user_text_file(byref ifilenumber, byref filename, byref filecontents) as boolean
```

when the programmer tries to replace one or more of the variables ifilenumber and filename with explicit variables. For example,

```
filename="unknown"
Q = open_user_text_file(0, filename, filecontents)
```

The question that invariably arises when this statement generates Error 21 is "why can't the compiler figure out what I am trying to do?" The answer is the compiler could indeed figure this out, but refuses to accept the usage because it violates object oriented rules, and could lead to serious problems. We explore this issue in more detail in the next section, but for now lets consider what could happen if the compiler allowed the above usage. We are submitting the number 0 as the first parameter and the filename has been assigned to "unknown". This combination directs the function open_user_text_file() to open up a dialogue so that the user can select a text file. The replacement of the variable ifilenumber with "0" will work just fine. But what if the user assigns filename to equal a null string (filename=""). That combination directs the subroutine to return the number of files in the folder "user_files" in the variable ifilenumber. So what is the compiler to do with this variable when the user has place the number 0 in this slot. The compiler has three options:

1. Never allow this situation to happen. If a number has been used in place of a ByRef variable, the compiler throws an error condition and refuses to run the program. This is the option that Scriptor selects.

2. Noting that a number has been used instead of a variable, the compiler simply ignores those lines of code which attempt to assign a value to it.

3. The compiler replaces the number that was passed to the subroutine with the number of files that were found in the folder. Assume the number of files was 6. From this point on, any use of the number "0" will be replaced with the number "6".

The reader will no doubt conclude that option 3 is a terrible idea, but many early Fortran and C compilers did exactly that. One could redefine the character sets via these procedures. Very few modern compilers allow such draconian manipulations because they violate the fundamental principles of object oriented programming. Indeed, the option 2 listed above also violates object oriented rules because the behavior of the subroutine is, under certain circumstances, undefined.

The above examples provide a perspective on why strict object oriented compilers do not provide flexibility in replacing ByRef variables with expressions or values. And for a computer environment that is to be used by students learning object oriented programming, strong typing is the only logical choice. We explore what this means in more detail in the next section.

6.3. Strong versus Weak Typing and Variable Conversions

Strong typing describes a programming language characteristic which imposes formal restrictions on a user's ability to mix variables of different types in expressions or in method parameter lists. In contrast, weak typing describes a programming language characteristic that allows users to freely mix variables of different types and do other types of variable conversions and replacements implicitly. Virtually all languages fall somewhere in between which makes the use of these terms rather subjective. In general, the term strong typing is used to describe a language that allows mixed variables in math expressions where conversion is well defined, while throwing a compiler error for less common conversions or conversions which reduce precision. In that regard, Scriptor is strongly-typed, and floating point and integer variables are allowed to be mixed in expressions and assigned to one another. The user needs to remember that if an integer is equated to a floating point number, the number is truncated (not rounded). Hence, the expression i=4.8, evaluates to 4 (not 5 as would seem more logical). The decision to truncate rather than round a real number during conversion to an integer is common to scientific languages, a tradition established by the early Fortran compilers.

6.4. Variants and Polymorphism

There are times when weak typing is desired, and Scriptor versions 1.8.10 and higher do allow the use of variants. Variants are untyped variables which can hold any one of the variable types listed in Table 6.4.1. After assignment, these variables hold not only the value assigned but information as to what type of variable it is. This is done by using an integer ID number, as shown in the second column of the Table 6.4.1.

Table 6.4.1. Properties and Identifiers of Variants in Scriptor

Type	ID	Example	Convert_to_string()	Str()
Nil	0	nil	nil	0
integer	2	1234	1234	1234
int64	3	9223372036854775800	9223372036854775808	9.223372e+18
single[(a)]	4[(a)]	12345E25	1.234500E32	1.23450e32
double	5	1.2345E297	1.23450000000000E+301	1.23450e301
currency[(a)]	6[(a)]	1234567.89	1, 234, 567.89	1.234568e+6
Boolean	11	true	true	1
Boolean	11	false	false	0
color	16	RGB(100, 0, 0)	RGB(100, 0, 0)	6.553600e+6

(a) All real numbers, when assigned to a variant, will default to doubles. If a non-double is desired, it must be set by assigning the variant to a variable declared as the desired type (single or currency).

Note that the ID in Table 6.4.1. is returned by the following function:

variant_type(v0) **as integer**
returns the integer ID that identifies the properties of the variant, v0
ID = 0(nil), 2(integer), 3(Int64), 4(single), 5(double), 6(currency), 8(string), 11(boolean), 16(color), where the variable type or property is listed in parentheses.

The variant_type() function should be called whenever one needs to know which variable type should be used to accept the value stored within the variant. It is not recommended that any manipulation of variants be done other than to use an assignment statement to transfer the contents of the variant into a variable of known type. The example shown below illustrates this approach.

Polymorphic methods. The main reason variants are available is to simplify the process of creating polymorphic methods. Polymorphism is the ability of a computer language, or a method within a language, to handle multiple data types using a common interface. For example, the sin() function in Scriptor works with arguments that are singles, doubles as well as arprec real and complex string variables. This function is therefore polymorphic because the same function name is used for the three different argument types. A program in Scriptor can create a polymorphic function by using one of two methods. The first approach is method overloading, which involves creating two or more methods of the same name but with different argument types. The compiler couples a function call to the correct method by matching the parameter types, and if one or more parameters are Byref, the match must be perfect. What that means is if one seeks to provide a math function which works with single, double, integer and currency data types, then four different functions must be created to represent each data type. Method overloading was discussed in section 2.7.1. An alternative approach is to use variants in the parameters. The compiler will allow any parameter to couple with a

variant, and only one function needs to be written. Lets assume that we want to create a function called fabs() which returns a floating point (double) absolute value of an argument. But this function must be able to handle not only real and arprec string variables, but Booleans as well. When a Boolean is passed, the function will return 1 if true and 0 if false. This function can be implemented within a single method by using a variant, as shown in the fabs() function example at right. The use of variants to handle polymorphism is their primary reason for existence, and when implemented for this purpose, variant parameters provide transparency and code reusability. To guarantee transparency, the variant should be transferred into a typed variable prior to manipulation. This also speeds program execution.

```
function fabs(v1 as variant) as double
  // returns a floating point absolute value for integer,
  single, double, string and string complex values as
  well as booleans (1=true, 0=false)
  dim itype as integer
  dim s1,s2 as string
  dim a1 as double
  dim Q as boolean
  itype=variant_type(v1)
  if itype=8 then // v1 is a string
    s1=v1
    s2=abs(s1)
    return value(s2)
  elseif itype=11 then
    Q=v1
    if Q then return 1 else return 0
  elseif itype>1 and itype<7 then
    a1=v1
    return abs(a1)
  else
    return 0
  end if
end function
```

The use of variants can make a program hard to follow, and to many computer scientists, the use of variants invariably violates the tenants of object oriented programming. Some argue than any language that includes variants is, by definition, weakly-typed. Because variants can be of any type, including no type, these variables are not really weakly typed. More accurately, they are untyped until they have been assigned a type via usage. It is thus a matter of semantics whether the availability of a variant class makes a language weakly-typed. We suggest that if one seeks a strongly-typed environment, it is best to avoid the use of variants. (Future versions of Scriptor will include a preference that allows variants, goto statements and other non-object-oriented constructs to be turned off.)

6.5. The Extends Keyword

Scriptor includes a few internal methods that operate via extensions. These including array sorting extensions (.sort, .sortwith), array manipulation extensions (.pop, .append) and color manipulation extensions (.red, .green, .blue, .hue, .saturation, .value, .cyan, .magenta and .yellow). One can create extension methods by using the extends keyword, as shown in the example at right. The extends keyword must appear as the first element in the parameter list, and takes the form:

```
// Program Name: template_module_extends.txt
// this program demonstrates the declaration of functions which use
the extends keyword

function endswith(extends s as string, withwhat as string) as boolean
    // Return true if 's' ends with the string 'withWhat', doing a standard
    string comparison.
    return Right(s, withWhat.Len) = withWhat
end function
function beginswith(extends s as string, withwhat as string) as boolean
    // Return true if 's' begins with the string 'withWhat', doing a
    standard string comparison.
    return left(s, withWhat.Len) = withWhat
end function

// Main
dim s0,s1,qq as string
s0="Steve Martin has pizzaz."
qq=chr(34) // double quote
s1=qq+s0+qq
if s0.endswith("pizzaz." ) then
    print(" The string "+s1+" ends with `pizzaz.`")
else
    print(" The string "+s1+" does not end with `pizzaz.`")
end if
if s0.beginswith("steven" ) then
    print(" The string "+s1+" begins with `steven`")
else
    print(" The string "+s1+" does not begin with `steven`")
end if
if s0.beginswith("steve" ) then
    print(" The string "+s1+" begins with `steve`")
else
    print(" The string "+s0+" does not begin with `steve`")
end if
print("")
print(" Note that all string comparisons are case insensitive.")
// end program
```

extends variable_name **as** type

where variable_name is the name used to represent the variable within the function and type specifies the variable type. If additional parameters are included, these will be included in parentheses following the function name as shown in the example. One can add as many additional parameters as desired, following the same rules as are applied to non-extension functions. Some compilers require that all functions involving extends must be declared inside a module, and earlier versions of Scriptor used a compiler that imposed that constraint. Scriptor versions 1.8.14 and above, however, use a more sophisticated compiler that allows extends functions to be defined outside of modules.

6.6. The Call Statement

All functions, by definition, return a variable. There are times, however, when the programmer would like to ignore the returned variable and use the function as if it were a subroutine. Such situations can be handled by one of two approaches. The more common approach is to put the result into the appropriate variable, and then ignore the value of the variable. A cleaner, and more readable approach to this situation is to replace the variable return assignment with the keyword “Call”. This usage informs the compiler to ignore the returned variable and yet handle all the other aspects of the function as normal. Thus, any ByRef variables listed in the function parameter list can be manipulated by the function and returned as before.

Some argue that the Call Statement should not be used because it introduces ambiguity regarding the nature of the function. This position is not valid. The use of the Call statement is to be encouraged when the programmer intends to ignore the value returned, because it clearly signals that such is the case. The alternative, which is to ignore the contents of the returned variable, introduces potential confusion and ambiguity. Other programmers will wonder where in the program the variable is used, and this will make the program harder to understand, especially if the variable is used in another context later on in the program.

6.9 Optimizing Execution Speed

The last thing a student should worry about is execution speed. Nevertheless, one of the more common complaints the author (as teacher) has received from students is that speed optimization is only briefly discussed in class. So for all those students, and users, of Scriptor and MathScriptor, who seek faster code, here is a list of things to do to make programs run faster. The list is in approximate order of importance. That is, items listed first have the potential to increase execution speed more than those listed at the bottom. However, if one of these items is inside a loop, small changes can make a bid difference. Hence the first suggestion is to find out which sections of code are taking the most time, and optimize those first.

6.9.1. Gather Code Metrics

There is no point in spending time optimizing code segments that are not contributing significantly to the total execution time. Normally, one can find a code segment that is primarily responsible for the total execution time, and improve performance by carefully optimizing the key section of code.

```
module timer_metrics
  // module to carry out program timing analysis with results
  presented in the spreadsheet insert start_timer() above segment
  and stop_timer() below the segment designed to handle 10 or fewer
  segments. change nsegs to increase max segments user should
  press Clear Output to reset timers

  private dim time_at_start as double
  private dim time_at_end as double
  private dim nsegs as integer = 10

  sub start_timer(isegment as integer)
    dim icws(),i1,i2,i3 as integer
    dim hdrs() as string
    if isegment=1 then
      i1=value(spreadsheet_cell(1,1))
      i2=spreadsheet_maxcolumn
      i3=spreadsheet_maxrow
      if i1<>1 or i2<>3 or i3<>nsegs or isession=1 then
        hdrs=string_split("Segment,time(ms),Percentage",",")
        icws=array(0,100,200,200)
        spreadsheet_create(nsegs,3,hdrs(),icws,2)
        spreadsheet_cell(1,1)="1"
        print("new spreadsheet created and zeroed")
      end if
    end if
    time_at_start=microseconds
  end sub

  sub stop_timer(isegment as integer)
    dim dtnew,dtold,dt(1),dt0 as double
    dim i as integer
    redim dt(nsegs)
    time_at_end=microseconds
    dtnew=(time_at_end - time_at_start)/10^3
    dtold = value(spreadsheet_cell(isegment,2))
    dtnew = (dtnew + (isession-1)*dtold)/isession
    spreadsheet_cell(isegment,2)=format(dtnew,",0")
    dt0=0
    for i=1 to nsegs
      dt(i)=value(spreadsheet_cell(i,2))
      dt0 = dt0 + dt(i)
    next
    for i=1 to nsegs
      spreadsheet_cell(i,1)=str(i)
      spreadsheet_cell(i,3)=format(100*dt(i)/dt0,"#0.00")
    next
  end sub

end module
```

The best way to examine code is to insert timing analyses. The module shown at right collects and averages timing information and displays the results in the spreadsheet. It can be found in the modules folder.

The code shown at right provides an example of a program that has been divided into segments for analysis. This is a short, and artificial example to demonstrate how the segments are analyzed. This example loads the module shown in the previous example into the object panel. For this to work, one must turn off simple mode so that the program can instantiate external objects. Each segment is assigned a number (segment 1 must be the first one executed). Before the first statement of the segment, insert the start_timer(#) line and after the last statement of the segment, insert the stop_timer(#) line, where # is the segment number. The program at right has three segments so marked. When the program is run a few times, the averaged results shown below are presented in the spreadsheet:

```
// Program Name: utility_speed_metrics.txt
// This program demonstrates carrying out speed metrics on a program
using the timer_metrics module
dim i,j,k,ix(32),iy(32),xw,yh as integer
dim x,huevalue,xx,yy,pi as double
dim s1,s2 as string
//# load_module("timer_metrics.txt",1)

arprec_set_precision(128)

start_timer(1)
' first section of code to be tested
x=0
for xx=0 to 360 step 0.2
  for yy=0 to 360 step 0.2
    x=x + sin(xx)+cos(yy)
  next
next
stop_timer(1)

start_timer(2)
' second section of code to be tested
for i=1 to 1000
  s1=str(rnd)
  s2=sin(s1)
next
stop_timer(2)

start_timer(3)
' third section of code to be tested
xw=800
yh=400
pi=const_pi
buffer_background_color=rgb(0,0,100)
buffer_create(1,xw,yh)
for k=1 to 16
  for j=1 to min(220,256*(k/6))
    for i=1 to 24
      ix(i)=xw/2+((xw-2*j-k*10)/2)*sin(pi*(i*k/24))
      iy(i)=yh/2+((yh-2*j-k*10)/2)*cos(pi*(i*(k+3)/24))
    next
    huevalue= k/8.0-j/220
    if huevalue>1 then huevalue=huevalue-1
    if huevalue<0 then huevalue=huevalue+1
    graphics_forecolor(0,hsv(huevalue,1,1))
    draw_arrays(0,ix(),iy(),1,24)
  next
  buffer_copy_to_canvas(active_canvas,0)
next
stop_timer(3)
// end program
```

Segment	Time (ms)	Percentage
1	718	25.76
2	458	16.43
3	1, 611	57.80

This means that the third segment is responsible for ~58% of the of the execution time, and should be tackled first. We will now discuss the various methods and procedures of optimizing code. Then we will return to this particular example and see how to best optimize segment three so that it executes faster.

6.9.2. Loop Optimizations

Because loops are repeated multiple times, they offer significant opportunity for optimization. The following are methods of improving execution speed of loops.

Extract unnecessary statements. During a loop operation, only those statements that are affected by the loop parameters or are involved in the loop operations should be included inside the loop. Math that can be removed from the loop should be removed, even if that means creating a new variable to hold the results.

```
xw=800
yh=400
pi=const_pi
buffer_background_color=rgb(0,0,100)
buffer_create(1,xw,yh)
xwd2=xw/2.0
yhd2=yh/2.0
for k=1 to 16
  kd24=pi*k/24.0
  kp3d24=pi*(k+3)/24.0
  k10=k*10
  kd8 = k/8.0
  jmax = min(220,256*(k/6))
  for j=1 to jmax
    j2mk10=j*2+k10
    for i=1 to 24
      ix(i)=xwd2+((xw-j2mk10)/2)*sin(i*kd24)
      iy(i)=yhd2+((yh-j2mk10)/2)*cos(i*kp3d24)
    next
    huevalue=  kd8-j/220
    if huevalue>1 then huevalue=huevalue-1
    if huevalue<0 then huevalue=huevalue+1
    graphics_forecolor(0,hsv(huevalue,1,1))
    draw_arrays(0,ix(),iy(),1,24)
  next
next
buffer_copy_to_canvas(active_canvas,0)
```

Never calculate loop endpoint in loop statement. Never use a calculated value for the end point of a loop statement if it can be avoided. Loops carry a great deal of overhead, and the requirement that the statement evaluate the endpoint at each loop reset adds a significant burden. If the endpoint can be calculated once and placed in a variable, using this variable in place of the expression in the loop statement will increase execution speed. If the endpoint is changing during loop execution, use a conditional exit statement to handle the situation.

Example. The loop that was executing slowly in the above example has been modified at right to remove all possible calculations outside of the loops. Note the use of jmax, which is calculated only once, as the end point of the for j=1 to jmax loop. This change, and the move of the buffer_copy_to_canvas() statement outside the loop system, represented the most significant changes. However, by moving the buffer copy outside the outer loop, we cannot watch the drawing process in action.

All of these changes improved the performance of this code segment by 14% (1600 ms down to 1370 ms). While that represents a meaningful improvement, it involved a fair amount of code rewriting. That is why code optimization should be carried out first on the time critical segments, and one should not waste time on those segments of code that contribute little to the overall execution time.

6.9.3. String Optimizations

If strings are used in an application, it is likely that the manipulation of the strings is a major contributor to total execution time. Unlike math operations, which can take full advantage of hardware math functions, string functions are handled entirely in software. Thus, they are comparatively slow. Added to this is the fact that many string operations are carried out on very large strings. One of the most time consuming operations is to extract substrings from a long string-based list. Consider the timing test shown at right. The first segment opens up the dictionary.txt file, which contains 24, 259 words. The timing module indicates the following times for the three segments: 11 ms (#1), 22088 ms (#2) and 13 ms (#3). Thus, the standard method of extracting strings using nthfield() is not efficient when working with large files. Which brings us to the first recommendation.

```
start_timer(1)
// open the data file
fnumber = 0
fname="dictionary.txt"
Q = open_user_text_file(fnumber,fname,filecontents)
s0 =string_replace_lineendings(filecontents,const_eol)
s0=trim(s0)
stop_timer(1)

start_timer(2)
// extract words using standard method
ns=countfields(s0,const_eol)
redim words(ns)
for i=1 to ns
   words(i)=nthfield(s0,const_eol,i)
next
print("The "+str(ns)+"th (last) word is "+ words(ns) )
stop_timer(2)

start_timer(3)
// extract words using string_split
words=string_split(s0,const_eol)
ns= ubound(words)
print("The "+str(ns)+"th (last) word is "+ words(ns) )
stop_timer(3)
```

Use String_Split() for large lists. When extracting records from a large string list, string_split() is significantly faster than NthField() by more than a 1000 times. The reason is that nthfield, whenever it is called, starts counting from the beginning each time and then extracts the desired field. As the number of fields increases, the process gets slower and slower. If there are more than 1000 records, NthField is a bad choice. The only time NthField should be used on a large list is when only one or two fields are to be extracted and one does not want to use the memory necessary to hold the other fields.

Use String_join_quoted() for large lists. Generating a large list also can take a fair amount of computer time. For example, to recreate the dictionary list can be done in one of two ways. The normal method is shown in segment #4, and uses the + operator to append the words(i) string to the end of the list, s1. It takes 6550 ms. A far faster method (17 ms) is to use string_join_quoted() which takes the entire arrays and puts it into the string in one operation using the second parameter as the delimiter. The string_join_quoted() function puts quotes around the string in the event the string contains the delimiter. This

```
// regenerate the long list
start_timer(4)
s1=""
for i=1 to ns
   s1=s1+words(i)+const_eol
next
stop_timer(4)

start_timer(5)
s1= string_join_quoted(words,const_eol)
stop_timer(5)
```

is the preferred method of handling this situation, and if relevant, one can use string_split_quoted() to extract.

Use byte operations whenever possible. There are byte versions of many string functions, and these will speed up operations considerably. However, they will fail if the string you are working on is using multiple byte characters. If you are working in the US, Canada or Europe, then the chances are good that the strings you are working on are single byte encoded. If you are working in Japan or China, the chances are good you cannot use these functions. In general, byte operations are usually an order of magnitude or more faster that the corresponding multibyte capable string functions.

ascb(s0) **as integer** ascii value of the first byte of s0

chrb(i) **as string** returns single byte ith ASCII character

instrb([kstart], source, sfind) **as string** returns the position of sfind within the string source. The optional parameter kstart indicates the first character position at which to start the search. This function is case insensitive.

leftb(s0, n) **as string** returns the leftmost n bytes of string s0

midb(s0, istart[, nlength]) returns the nlength bytes starting at istart from s0
if nlength is not included, all bytes from istart to end are returned

replaceallB(source, substring, replacement) **as string** replace all occurrences of substring by replacement in source by byte. case sensitive.

replaceb(source, substring, replacement) **as string** replace substring by replacement in source by byte. case sensitive

rightb(s0, nbytes) **as string** returns the rightmost nbytes of characters from s0

Avoid RegEx functions. Regular expressions are very powerful, but two orders of magnitude more complicated than normal string operations. Regular expressions should be avoided when speed is an issue.

General recommendations on string speed enhancements. There are a number of small things that one can do to keep string manipulations efficient. In general, if there is a function that will do what you want, it is almost always faster to use the function than to write the code yourself. The string functions have all been highly optimized and use all of the tricks available for making the operation fast and memory efficient.

6.9.4. Graphics Optimizations

There are few more CPU intensive tasks than graphics. And making recommendations on how to speed up graphics is made difficult by the fact that the speed of many operations depend in large part on the type of graphics card that is present as well as the operating system. Nevertheless, there are some general observations that can be made to improve the speed of graphics code segments.

Use fast buffering. Scriptor is designed to use buffered graphics for two reasons. First, such an approach is required for flicker-free graphics. Second, such an approach is faster. All graphics writing to the buffer is absolute. No scaling or anti-aliasing is done, and the graphics functions can operate at full speed. Although the buffer copy to canvas is optimized heavily, it is still a CPU intensive operation, especially if scaling and anti-aliasing are carried out. The latter are done whenever the buffer size is different from the target canvas. So the single best approach to achieve fast graphics is to use a buffer that is the same size as the canvas for which the graphics are ultimately to be targeted. This will increase the speed by 10-30% depending upon the amount of graphics statements preceding the buffer copy process. If writing a game, where speed is of paramount importance, making the buffer the same size as the target canvas is an excellent idea.

Use small buffers. The time to execute a graphics statement is proportional to the size of the object being drawn. If a large buffer is being used, the objects drawn will necessarily be scaled to the size of the buffer. To a first approximation, graphics time, T_G, is proportional to the square of the buffer size, or:

$$T_G = ß*buffer\_width*buffer\_height$$

where ß is a scalar proportional to the number and nature of the graphics statements. The point of this equation is that ß is determined by the graphics that is being carried out and is invariant to the buffer size. But by simply dropping the buffer dimensions from 3000x2000 to 1500x1000 will speed up the graphics by a factor of ~4. Naturally, graphics tend to look better when a larger buffer is used, so quality and speed are mutually exclusive. The goal is to use as large a buffer as is necessary to achieve the desired quality, but no larger.

Use Sans Serif Fonts. When speed is an issue, and a great deal of text is being drawn, the complexity of the characters affects the speed. Using sans serif fonts will increase the speed of the text handling portion of the graphics by about 30%.

6.9.5. File Optimizations

There is very little a user can do to optimize file operations because virtually all of the work is done by functions that are dependent upon code within the operating system of the computer as well as how the operating system has been setup to access disk drives and buffer the data. There are, however, things that can be done to increase the efficiency of file manipulations. Here are a few suggestions.

Use binary files when possible. Text files are handled using sequential access. In contrast, binary files are handled using direct access and in general, these processes are much faster. Furthermore, binary files allow access to individual elements using the **binary_file_read**(filepath, variable, nthelement) **as integer** which is a highly efficient method of accessing a large database. Because the file data are buffered, each data request only initiates a disk access when the data are outside of the buffer. This makes the process extremely fast under most circumstances. Windows XP (Service Pack 2 and higher) and Mac OSX (10.3.9 and higher) handle the buffering process very efficiently and will dynamically increase the size of the buffer to improve efficiency.

6.9.6. Optimizing Math

There is not a lot that a programmer can do to speed up math other than to use improved algorithms. That is of course an important goal, but a detailed discussion of algorithm optimization is beyond the scope of this section. There are a few simple things that can be done to increase execution speed.

Avoid mixed operations. When integers and real variables are used extensively in an algorithm, try and do all of the integer operations first before combining with the floating point variables. Integer math is significantly faster. Mixing integers and reals adds additional latency due to the cost of the type conversions. The CPU time for various operations is shown in Table 6.9.6. Note that operations that require that the integers be converted into reals prior to carrying out the function take more time than those that are directly carried out on doubles. The one exception is pow(real, integer), where the internal function uses a faster algorithm to handle integer exponentiation which provides a 17% faster result than pow(real, real).

Avoid using trig, log and pow functions. Although the algorithm is invariably the primary determinant in what functions are used, it is often possible to write equations in forms that avoid or minimize the use of transcendental, logarithmic and exponentiation functions. These functions tend to be 2 – 4 times more CPU intensive than simple math operations. It is to the credit of the software and hardware developers that the complex math functions are so efficient. It should be noted that the hyperbolic functions sinh(),

cosh(), tanh() are carried out using Arprec arithmetic which is responsible for the high cost of these functions. If the user plans to make heavy use of hyperbolic functions, and lower precision is acceptable, it is best to replace these with their exponential equivalents (see Appendix 4).

Table 6.9.6. Relative CPU Time (Cost) for Math Operations on Integers and Doubles[(a)]

Operation	Cost	Operation	Cost	Operation	Cost
i - j	2	**a - b**	12	**log(a)**	123
i + j	1	**a + b**	12	**exp(a)**	64
i * j	1	**a * b**	23	**sin(a)**	62
i \ j	7	**a / b**	35	**logGamma(a)**	1, 620
i / j	52	**sqrt(a)**	24	**sinh(a)**	68, 700
sqrt(i)	33	**pow(a, i)**	95	**bessel(a, b)**	2, 083, 000
pow(i, j)	220	**pow(a, b)**	114	**zeta(a)**	4, 729, 000

(a) Cost is measured in CPU time for an average of Intel and G5 processors and based on one-million operations on random values. Random integers are indicated using the variables i and j and random doubles are indicated using the variables a and b.

6.9.7. Compiler Pragmas

The term pragma is an abbreviation representing a "pragmatic compiler directive", a request of the compiler to modify its methods and procedures for some "pragmatic" reason. The pragmas relevant to program speed are all handled by using the #pragma statement:

#pragma directive [boolean]/sets or clears compiler directives.
BackgroundTasks True (enables yielding to background threads, which is the default)
BackgroundTasks False (disables yielding to background threads and speeds up program)
DisableBackgroundTasks (disables yielding to background threads and speeds up program)
BoundsChecking True (enables monitoring of array subscripts to trap out_of_bounds exceptions, the default)
BoundsChecking False (disables monitoring of array subscripts, and speeds up program)
DisableBoundsChecking (disables monitoring of array subscripts, and speeds up program)
NilObjectChecking True (enables trapping of nil objects prior to usage, which is the default)
NilObjectChecking False (disables trapping of nil objects prior to usage, and speeds up program)
StackOverflowChecking True (enables monitoring of the stack size to prevent overflow, default)
StackOverflowChecking False (disables monitoring of the stack size, and speeds up program)

If the goal is to speed up a program as much as possible, the following four pragma lines will provide the maximum speed enhancement:

```
#pragma backgroundtasks false
#pragma boundschecking false
#pragma nilobjectchecking false
#pragma stackoverflowchecking false
```

These pragmas will have greatest impact when matrix operations on multi-dimensional arrays are involved.

It is recommended that these directives not be used until a program has been fully debugged and tested. Modern operating systems (Mac OSX, Linux, Windows XP and Vista) will not allow a program to damage the operating system or other programs, and thus these pragmas can be used safely. However, writing a program with a computationally intensive loop, and running this program with backgroundtasks turned off, can essentially prevent Scriptor from releasing time to the user interface or other programs. Multiprocessor computers will handle this situation with grace, and assign processor time to other programs and prevent lockout. Nevertheless, the Scriptor user interface can be locked out, and one might need to do a forced quit to stop the program.

6.9.8. Common Misconceptions

There are a number of logical steps that can be taken to improve the speed of a program. The above guidelines provide a starting point, but each program is different. The best way to optimize a program is to use both logical analysis, based on the information provided above, and trial and error. The purpose of this section is to clear up a few misconceptions that students often have in terms of execution speed.

compilation and speed. There is no significant advantage to compiling the program separately with regard to speed. The core routines in Scriptor and MathScriptor are already compiled and the overhead associated with compiling the users program is rarely more than a second. If the user has developed a very large program using MathScriptor, and compilation is generating noticeable latency, then it is time to switch to Xojo (www.xojo.com), C++ or MatLab. If some of the MathScriptor internal functions are needed, contact R.R.Birge (rbirge@uconn.edu).

ByRef is faster than ByVal. This statement is true, but the CPU difference is too small to measure under normal circumstances. Where this difference might be important under some languages would be in the passing of arrays, but in mathscriptor, arrays are always passed ByRef whether labeled as such or not.

Appendices:

Appendix 1
Scriptor and MathScriptor Language Reference
Version 3.6.0 (January 2015)

This appendix provides an overview of the language syntax of Scriptor and MathScriptor. Because all of the features of Scriptor are present in MathScriptor, reference will be made to Scriptor in most of this discussion. When a feature is present only in MathScriptor, that fact will be indicated by using the symbol |MS| next to the keyword. When a feature is present only in MathScriptor versions above 2.0, that fact is indicated by using the symbol |Ph| next to the keyword. The help screen is a shorter version of this appendix, and can be consulted within the program by pressing the Help button.

Scriptor is intended primarily as a learning platform while MathScriptor is intended primarily for scientific and engineering programming. Both Scriptor and MathScriptor reside within the same program and the user can switch between the two environments at will from the Compiler Menu. Access to MathScriptor requires that the program be registered. MathScriptor provides additional math capabilities (arbitrary precision, complex arithmetic, expanded integration, wavelet transforms, additional matrix operations), access to external objects, web access under program control and compilation with optimization. The Scriptor environment can run programs compiled by MathScriptor, however. The advantage of working in Scriptor during the learning process is that the additional features and complexity of MathScriptor can create problems that are difficult to diagnose. Thus it is recommended that all students learning how to program work in Scriptor during the early stages of their learning process, and if MathScriptor functions are needed, run MathScriptor in Simple Mode (first menu option under Debug). When Simple Mode is turned on, external objects are not available. External objects provide significant power and flexibility, but can cause confusion during the debugging process.

There are a few things to keep in mind when programming in Scriptor. First, capitalization is ignored. All variables and operators are converted to lowercase prior to compilation which means the compiler treats Adam, aDam and ADAm as identical variables and Sin(x), sin(x) and sIN(x) as identical functions. Second, spaces are important, but the number of spaces is rarely important except within quotes or strings or between methods and their parameter lists. Thus, the following statements: Sin(x) and Sin(x) are identical and there is no difference in how the compiler interprets the equation: 1+2*x+3*x*x versus 1 + 2 * x + 3 * x * x. However, it is important not to have a space between a function or subroutine name and the first paren that marks the start of the parameter list.

In the following examples, any variable that begins with an I, J, K, L, M or N is an integer, with a C is a color, with a Q is a Boolean, and with an S or T is a string unless

defined otherwise. All other variables are real. However, Scriptor requires that all variables be assigned and imposes no conventions on the first letter of any variable. The requirement that all variables be defined is sometimes viewed by students as an unnecessary burden on the programmer, but it makes code more readable and prevents typing mistakes from dominating the debugging process. This requirement is also in keeping with an object oriented structured environment, and quite different from Fortran or Basic, early languages which allowed the use of undeclared variables.

Although you must use a Dim or Const statement to declare each variable, you do not need to put these declarations at the top. We recommend you do so for clarity, but the compiler only requires that you declare each variable somewhere within the program. The one exception is a variable that is being redimensioned via the Redim statement. The Redim statement is carried out during runtime, and thus its location in the program is important and must precede any usage that depends upon the revised size. The debug menu in Scriptor provides the option to display all of your public and main variables and should be used frequently to monitor variable and confirm the correct type is declared.

Autostart Packages:

If you wish to share your program with other registered users, you should consider making an autostart package. This is also the most reliable way of submitting a program to your instructor if it requires external objects or data. This section provides directions to make an autostart package. It is simple, and requires duplicating the Scriptor environment to the extent necessary to run your project in a separate folder.

First, create a new folder and give it any name you desire. We will refer to this folder as the package folder. Move your program inside the package folder and change the filename of your program to one of the following:

autostart.txt (if you want the program to start up in the Main panel) or
automusicstart.txt (if you want the program to start up in the Music panel)

If your program needs to load a dataset prior to running, create a folder called "data sets" and place the spreadsheet file inside the "data sets" folder. Change the name of the spreadsheet file to "**autodata.cet**". The data set must have been created by using the save command in the data set panel of Scriptor. However, you can change the name by hand to " **autodata.cet** ".

If you have developed your program using Simple Mode, the instructions in this paragraph can be skipped. If your program needs to read any classes, modules or methods you must place copies of these text files inside new folders created inside and

named "Classes", "Modules" or "Methods". Make sure that you do not change the names of the files from those used by your program to reference these external objects and make sure that each file is placed in the correct folder.

If your program accesses pictures, include copies of these pictures inside a folder called "user_pictures". If your program uses text files, include copies of these files inside a folder called "user_files". Finally, if you would like to replace the Scriptor splash screen with one of your own, create a jpg file and place it in the top level of the package folder and call it **splash.jpg**. The image size should be roughly 600x 400, but the program will scale this image to fit so experiment.

Finally, to test your package, place a copy of Scriptor inside and double click on it. Scriptor should start up with your program loaded in the correct window. When you press run, all of the resources should be loaded normally. All that you have done is recreate a minimum environment with only the necessary resources present, but in locations identical to those used in your primary Scriptor folder. When you share your program with others, you should not send another copy of Scriptor along. If the recipient does not have a registered version of Scriptor, the autostart package will not run. If they do, all they will need to do is place a copy of Scriptor inside the folder and double click on it. If you plan to email your package, the creative ability of email programs to trash text files suggest that you should first make a zip, stuffit or suitable archive prior to emailing the package.

Valid Operators:
standard math operators: +, -, *, /, Mod
exponentiation uses either the standard basic operator ^: x^y = x^y
or the C-type pow(a, b) operator: x^y = pow(x, y)
comparison operators: <, =, >, <=, >=, <>
logical operators: And, Not, Or.
comments begin using: ' (a single quote), // or REM*

*It is recommended that the student only use the ' or // markers to identify comments and leave the REM keyword for the instructor to insert comments. The REM comments are highlighted in bright orange to make them easy to find. They can also be removed by selecting a menu item under Debug.

Data Types:
Integer (32-bit whole numbers in the range ± 2, 147, 483, 648)
Int64 (64-bit whole numbers in the range ± 9, 223, 372, 036, 854, 775, 807)
Single (32-bit positive or negative 7 digit real values between
1.175494×10^{-38} and $3.402823 \times 10^{+38}$)
Double (64-bit positive or negative 15-16 digit real values between
$2.2250738585072013 \times 10^{-308}$ and $1.7976931348623157 \times 10^{+308}$)
Boolean (1 bit: true, false)
String (ASCII characters of arbitrary length – see Appendix 3)
Color (32-bit color specification, e.g. RGB(255, 255, 255))
Const (any variable type, but once defined, cannot be changed via assignment in program)
Currency (A 64-bit fixed point variable for accounting and business applications involving money. The variable has 15 digits to the left of the decimal point and 3 digits to the right of the decimal. Only standard, non-transcendental arithmetic, is allowed.)

Data types are declared using the **Dim** or **Const** statements as follows:
Dim aa **as double**, ii **as integer**, qq **as Boolean**, reddish **as color**
Const Pi=3.14159265358979323846264338327 95

The compiler will figure out what data type to use for a constant, and in the above example, it will use a double. However, only the first 16 significant digits will be stored.

Arrays: Arrays can use any of these types and can by dynamically redimensioned in the program using the Redim statement. All arrays have a zeroth element or elements. That is, if the array a2 is dimension a2(1, 1) has four elements: a2(0, 0), a2(0, 1), a2(1, 0) and a2(1, 1).

Classes and Modules:

Modules provide a straightforward and flexible approach to providing a set of functions and subroutines that are available to other objects outside of the module, but can communicate between each other via private properties and private methods, if desired. Modules are defined using the following syntax:

```
module module_name

public and private properties are declared here with the requirement that
each variable is declared using a single line dimension statement.
Each private property is shared by all the methods within the module, but
is invisible outside of the module.  Public properties are available to all the code in your
application.  Examples include:

private dim u(10, 10) as double  // only available to code inside the module
private dim v(10, 10) as double  // only available to code inside the module
public dim w(10) as double  //  available to all code
dim pwr(10) as integer  // available to all code (default is public)

methods are defined next and are available outside of the module unless their name is
preceded with the word private.  Thus

private sub a1(i as integer, byref x1() as double)
// this subroutine is available to only code within the module
// any properties declared are local to the subroutine
….
end sub

public sub a2(i as integer, byref x2() as double)
// this subroutine is available to all code
// any properties declared are local to the subroutine
….
end sub

function a3(i as integer, byref x3() as double) as double
// this function is available to all code (default is public)
// any properties declared are local to the function
….
end function
```

You cannot have code outside of functions or subroutines within a module. Modules are never called by themselves but only serve as containers for properties and methods.

end module

The fact that the compiler requires that each variable be assigned in a separate (one-line) dimension statement is an inconvenience, but is a restriction that can be justified based on the significant amount of work that is required of the compiler when handling public and private variables within both modules and classes (see below). But help is available. You can collect all of your dimension statements into groups as normal and then press the clean code menu item, and the dimension statements will be expanded automatically. This saves time during the writing of your programs.

Classes are collections of methods that are available to objects outside of the class, but which must be "called" by using a different syntax than is used to call methods declared via modules. To use classes properly you need to learn three new programming terms, instantiation, constructors and destructors. Instantiation refers to the process of creating a "copy" of the class for use in your program. A variable is created to represent your class using a standard dimension statement. Lets say your class is called class1. To use this class, you would assign a variable to be of type class1 by using the statement dim variable_name as class1. Then a copy of the class is created by using the statement variable_name = **new** class1. This process is known as instantiation. Classes also need constructors.

Constructors. You must add one or more methods to the class with the name constructor. If two or more are present, they must have different types or numbers of parameters. Multiple methods with the same name but different parameters are called "overloaded". These methods are run when the class is instantiated. The process of instantiation uses the **new** keyword (a keyword that is sometimes called a constructor) to create a new "instance" of the class within your program. You should think of an "instance" as a copy but with properties defined by the constructor. Once created (or instantiated) you have access to the methods that have been defined by your class. The constructor is a critical part of instantiation because of the flexibility that it provides. You can use the constructors to initialize the class to behave differently, or have different properties. We use the term constructors because you can have more than one method of the same name which is selected based on the way the **new** statement is written. You can have multiple constructors and the constructors can be overloaded (see below). Some classes also need to carry out a cleanup operation when the program no longer needs them and they go out of scope. For this reason, classes have an additional but optional subroutine which is called **sub destructor()**. This subroutine is automatically called when the class is no longer available to the program. For example, if a class has been instantiated within a subroutine, after exiting the subroutine the class

has gone out of scope and the class destructor subroutine is executed if present. The destructor provides the programmer with an opportunity to do any necessary cleanup operations or redimensioning of variables that were, for example, increased in size within the constructor. However, the programmer need not worry about variables that were local to the class. The memory allocated to these variables is returned to the system automatically.

Classes are powerful but complicated objects that new programmers should avoid using until they have mastered modules and the concept of method overloading. Modules are easier to use because all public methods defined within a module are immediately available to the program just as if they had been defined within the main program. The following example illustrates the definition and use of a simple class that takes a number and multiplies it by π (the default) or a user assigned number. The example also illustrates an important aspect of classes. When they are instantiated (created by using the **new** keyword), the variable that you defined to be of type class1 is "filled" with the code associated with that class. It will not change even if another variable of type class1 is instantiated but instantiated using a different value for the internal private variables. The following example illustrates this important, but rather confusing aspect.

```
class class1  // creates a class called class1
  dim a1 as double  // all variables are private to the class
  dim k as integer // each variable must be declared separately

  public function mba1(x as double) as double
    // public not required because public is the default
    return x*a1
  end function

  sub constructor()
    // constructor uses default initialization of pi
    a1=const_pi
  end sub

  sub constructor(a1set as double)
    // allows user to select other options during new assignment
    a1 = a1set
  end sub

  function a1val() as double
    // this function returns the value of a1
    // although there are no parameters, we need ()
    return a1
  end function

  sub destructor()
    //  optional subroutine is executed when class goes out of scope
  end sub

end class

dim r1, a2 as double
dim blim, blam as class1

// create an instance of the class using default value of pi
blim = new class1
// create an instance of the class using a value of 4
blam = new class1(4)

r1 = 4.0
a2 = blim.mba1(r1)
print("a2 (based on blim.mba1)  = "+str(a2))
```

```
print("blim.a1val = "+str(blim.a1val))

r1=4.0
a2 = blam.mba1(r1)
print("a2 (based on blam.mba1)  = "+str(a2))
print("blam.a1val = "+str(blam.a1val))

// the following demonstrates that creating an instance of blam
// did not override the definition of blim.  Once an instance is
// created, it remains invariant to new instances and constructors.
r1 = 4.0
a2 = blim.mba1(r1)
print("a2 (based on blim.mba1)  = "+str(a2))
print("blim.a1val = "+str(blim.a1val))

// end program
```

The above program, when run, will generate the following output:

```
a2 (based on blim.mba1)  = 12.56637
blim.a1val = 3.141593
a2 (based on blam.mba1)  = 16
blam.a1val = 4
a2 (based on blim.mba1)  = 12.56637
blim.a1val = 3.141593
```

Note that when you want to call a class method, you need to access that method by using the syntax: class_variable.class_method where the class_variable is the variable that was declared (instantiated) to represent the class in your program and the class_method is the name of the method that resides within the class. If this seems like a lot of work without any obvious advantage, it is important to understand that a class provides some new flexibility. The flexibility is that when a class is instantiated, it can be instantiated with various assignments made at the time of instantiation. You can thus have many different variables representing the same class, but which have functions that are altered at the time of assignment to suit your needs. The above is a trivial example, designed to illustrate the concepts, but not the power, of this flexibility. The class instantiation and construction syntax may appear to be arbitrarily complicated, but once you get used to the syntax and explore the possibilities, you will appreciate the new flexibility the class structure provides.

After the **end class** statement, which is the last statement of a class definition, you can include a program that tests the class. You can also store this program with your class

definition. This test program is useful for testing and will be ignored if the class is placed into any of the four windows of the Objects Panel. You can use the following compiler directives to automatically load a method, module or class into the Objects Panel windows. You must keep track to make sure you use a different window number for each object.

|MS| **//# load_class(**class_filename.txt, iwindow**)** class file must be in classes folder
|MS| **//# load_method(**method_filename.txt, iwindow**)** method file must be in methods folder
|MS| **//# load_module(**module_filename.txt, iwindow**)** module file must be in modules folder
iwindow = 1, 2, 3 or 4 where the number identifies the object window destination. If you set iwindow=0 when loading a module, then the lowest numbered empty window is used. This only works with modules because of their special properties and capabilities.

|MS| Note that external methods accessed via the objects panel are only possible when running in MathScriptor mode. However, if the above statements are present in a Scriptor program, they will not generate an error if the program is registered, but will generate a warning statement in the output text editfield.

Functions and Subroutines:

Function name(parameter list) **As Type**
… … …
Return value
End Function

Sub name(parameter list)
… … …
End Sub

A function must return a value (using the Return statement), and when referenced in your program, the returned value must be assigned to the appropriate variable. You can have multiple return statements, but once a return statement is encountered, the function exits and returns the value. A subroutine cannot have a return statement, and if one exists, the compiler will return an error. The parameter list allows two types of variables to be passed: ByVal or ByRef. If passed by value (ByVal), the value of the variable is copied into the local variable. If the value of the variable is modified inside the subroutine, the calling variable remains unaltered. In contrast, if a variable is passed by reference (ByRef), the memory location is passed and upon exit, if a change in the value has occurred within the function or subroutine, the change is retained by the

variable upon exit. All parameters default to ByVal except for arrays, which are always passed ByRef. Accordingly, you normally only need to indicate ByRef as in the following example:

```
Sub sub_name(aa as double, ByRef bb as double, ic as integer, a2(, ) as double)
```

In this example, the variables bb and the two dimensional array a2(,) are passed ByRef while the variables aa and ic are passed ByVal. Again, it is important to remember that arrays are always passed ByRef so if you change an array element within the subroutine, that change will be preserved upon exit. If you want to pass an array Byval, you must do so programmatically by making a copy and placing the copy in the parameter list.

You can also load a method (function or subroutine) into one of the Object Panel windows using the following compiler directive:

```
//# load_method(method_filename.txt, iwindow)
```
iwindow = 1, 2, 3 or 4 where the number identifies the object window destination.

You cannot mix classes and methods or modules at the same time (have both instantiated) nor can you have test code at the bottom of a method. You can mix modules and methods.

A function can return an entire array if desired. For example, the following is an example of a function that generates an identity matrix.

```
Function matidn2(nsize as integer) as double(, )
  dim i, j as integer
  dim a2(1, 1) as double
  redim a2(nsize, nsize)
  for i=0 to nsize
    for j=0 to nsize
      a2(i, j)=0.0
    next
    a2(i, i)=1.0
  next
  return a2() // note a2() is used, not a2(, )
end function
```

Following is a short section of code that calls this function and demonstrates two important aspects of calling functions which return arrays. First, not only is the array elements returned, including the (0, 0;0, 1;1, 0 elements even if not assigned), but the array is redimensioned to correspond to the dimension that is assigned within the

function. Second, the dimension of the array is not reflected in the return statement (above) or the assignment statement in line 4 below.

```
dim a(10, 10) as double
dim n, n1 as integer
n1=4
a()=matidn2(n1) // note a() is used, not a(, )
n=ubound(a(), 1)
set_text_style("Courier", 12, rgb(0, 0, 0), false, false)
print(matrix_print(a(), n, n, n))
```

Method Overloading:

Scriptor allows functions and subroutines to be overloaded, which allows two or more methods to be defined with the same name but with different numbers or types of parameters. This capability provides important flexibility in programming and usage. The flexibility is demonstrated in the following example of a function max1, which can be called with two or three doubles or two string variable representations of numbers.

```
function max1(a as double, b as double) as double
  return max(a, b)
end function

function max1(s1 as string, s2 as string) as double
  return max(value(s1), value(s2))
end function

function max1(a as double, b as double, c as double) as double
return max(a, max(b, c))
end function
```

Loops:

Scriptor provides three types of loops, using the combinations: For…Next, Do…Loop and While…Wend. The most useful of these options is the For…Next loop.

```
For i = istart to iend step idelta
if testcondition then exit
Next

For i = istart downto iend step idelta
if testcondition then exit
Next
```

where the term testcondition is used to indicate any variable or expression that evaluates true or false. The conditional exit statement is optional (see below). The Next statement increments the value of i by idelta, which must be a positive value. Thus, if you want to decrement, you must use the downto statement. The step idelta is optional and if the step parameter is not present, an increment of +1 (or –1 with downto) is assumed. The For..Next loop can also operate on floating point numbers where rdelta can be any positive real number. The if testcondition then exit statement is available in all of the Scriptor loops, and can be used to exit the loop whenever the testcondition evaluates as true.

```
For r = rstart to rend step rdelta
Next

For r = rstart downto rend step rdelta
Next
```

The other two loops are more useful in situations where a test is to be carried out during the looping process and when the test condition has been satisfied, the loop is excited. When using the do loop, you have the option of testing before, during or after the loop has been executed:

```
Do Until testcondition
Loop

Do
Loop Until testcondition

Do
if testcondition then exit
Loop

Do Until testcondition1
if testcondition2 then exit
Loop Until testcondition3
```

The first three examples represent the most common usage, but it is valid and sometimes necessary to include conditional tests before, during and after the loop is executed as shown in the fourth example. Each testcondition can be different. The While…Wend statement provides an addition looping option that provides no additional flexibility but has the modest advantage of providing a more natural resonance with the English language.

```
While testcondition
Wend

While testcondition1
if testcondition2 then exit
Wend
```

Here, the test condition is only available at the beginning of the loop but you do have the option of exiting at any point in the loop based on the conditional exit statement.

Conditionals:

The If statement coupled with the Else or Elseif statements is the most commonly used conditional and exists in one form, or another, in all high level languages. The one-liner form of this statement is,

```
If  testcondition1 Then statement1 Else statement2
```

where statement1 is executed only if testcondition1 evaluates to true and statement2 is executed only if testcondition1 evaluates to false. The multiline approach is often easier to read and provides for additional options.

```
if testcondition1 then
  statements in here are executed if testcondition1 is true
Elseif testcondition2 then
  statements in here are executed if testcondition2 is true
  and testcondition1 was false when evaluated
Else
  these statements are only executed if all previous test conditions were false
End if
```

Although only one ElseIf statement is shown, you can have as many ElseIf sections as desired. It is important to keep in mind that once any of the test condition has evaluated to be true, the corresponding code is executed and the If statement exits to the line following the End If. A high level conditional is also provided by the Select Case statement, which is valuable in providing a highly readable but significantly slower version of the multiline conditional.

```
Select Case  testexpression
Case testvalue1
statements in here are evaluated if testvalue1=testexpression is true
```

Case testvalue2
statements in here are evaluated if testvalue2=testexpression is true
Case first_value **To** second_value
statements in here are evaluated if testexpression is in the range specified
Case is testvalue2
statements in here are evaluated if testvalue2 has an inequality
relative to testexpression that is true (i.e. <, <=, >, >= as specified)
Else
statements in here are evaluated if none of the above tests were true
End Select

Input and Output Functions:

Clear_Text_Output(ioption) Clears all of the text in 0(both), 1(Main Panel), 2(Text Panel)

Input(Prompt **as string**) **as string** Retrieves input from the user with an optional prompt

Print(s0) Sends string s0 to both the Main and Text panel output windows. If running in the Music Panel, this statement also sends output to the Music Output text buffer.

Format(number, n, m) **as string** Formats number into an n digit number with m digits after the decimal point.

Format(number, string_format) **as string** Formats output using rules discussed below

Show_progress_bar(ip) Displays the progress bar for ip=0(start) to ip=100(finished).

Show_progress_line(s0 [, fontname, fontsize]) Displays the string s0 in the input line using the default fontname and fontsize, but the user can override the defaults by explicitly specifying both the fontname and fontsize.

String_speak(Text, Qnow) Speaks the Text and if Qnow is true, immediately interrupts.

Not included in the above examples are the numerous graphics input and output capabilities that are discussed below.

Functions and Reserved Variables:

Arguments or parameters are defined using the conventions discussed at the top of this chapter: a variable that begins with I, J, K, L, M or N is an integer, with a Q is a Boolean and with an S is a string unless specified otherwise. All other variables are real. When an array is used, all single dimensioned array labels end in 1 [e.g. x1()] and two-dimensional arrays (or matrices) end in 2 [e.g. a2()]. RGB color values ired, igreen and iblue are integers constrained from 0 (off) to 255 (max). In contrast, all parameters

for CMY and HSV are real numbers between 0 and 1.0. If the variable is called color, then the color can be specified using RGB, CMY or HSV methods. If the variable is called buffer, then it refers to the internal graphics buffer. When multiple buffers are created, all operations default to buffer 1, which is the upper left-hand buffer. You can then copy this buffer to any of the other buffers by using the copy_buffer_to_buffer routine.

Keep in mind that Scriptor automatically upgrades variables so that you can often replace a double with an integer, and the integer will be converted to a double in the process of transferring the parameters to the subroutine. Thus, although sin(x) expects a double, it will work just fine when an integer parameter is encountered. That is, sin(i), where i is defined as an integer and assigned a value of 12, will generate a valid result (-0.5365729).

|MS| Arbitrary Precision Arithmetic

MathScriptor versions 1.8.2 and above include the capability of doing arbitrary precision (arprec) arithmetic as well as string based complex arithmetic. The precision of arprec arithmetic is controlled by the command arprec_set_precision(idigits), where idigits is equal to the number of digits of precision in the real number, not including those digits in the exponent. An added benefit of using arprec arithmetic is that exponents as large as ±58, 000, 000 are allowed.

Arbitrary precision functions have string parameters and return strings. These functions also work on complex numbers identified by separating the real and imaginary parts with a comma (do not include I, it is understood). The following functions are arprec savvy: plus(s1, s2), minus(s1, s2), mult(s1, s2), div(s1, s2), real(s1), imag(s1), pow(s1, s2), log(s1), loggamma(s1), exp(s1), abs(s1), sin(s1), asin(s1), cos(s1), acos(s1), tan(s1), atan(s1), sinh(s1), asinh(s1), cosh(s1), acosh(s1), tanh(s1), atanh(s1).

Output precision can be rounded to a lower precision by using the function round_to_precision(s1, ndigits). This function returns a string which can be inserted directly into a Print statement. Alternatively, one can format arprec strings by using the function Format(s1, nwidth, ndecimal), which also works on complex numbers where the total width of the output string equals 2*nwidth+3 for comma delimiter.

There are three comparison functions that can be used with arprec strings. The first is the standard equals ("=") which when used in a conditional statement returns true if two arprec strings are identical. This function, when combined with the round_to_precision function, allows identity to be established at lower precision if necessary. The two arprec specific functions, **Q_greater_than**(s1, s2) and **Q_less_than**(s1, s2), return true

if s1 is greater than, or less than, s2. These two functions will even work if s1 and s2 were calculated at different precision.

The flexibility and power provided by arbitrary precision arithmetic comes with a price. The most significant cost is in CPU time as a 32 digit arprec multiplication takes 4550 times longer than a 16 digit precision double multiplication. The reason for this significant difference is that double precision arithmetic can take advantage of floating point hardware that is designed to manipulate double precision numbers. In contrast, all of the arprec math must be done in software and despite use of extensive use of processor floating point arithmetic, arprec math invariably requires thousands of processor cycles. Additional latency is associated with the use of strings to receive and return the results.

The following table provides some typical execution times in microseconds for a 2.33 GHz Intel Core-2 Duo Processor:

Operation	double (16 digits)	arprec (32 digits)	arprec (128 digits)
addition [a+b or plus(s1, s2)]	0.01	89	225
multiplication [a*b or mult(s1, s2)]	0.02	91	235
division [/ or div(s1, s2)]	0.03	85	222
trigonometry [sin(x) or sin(s1)]	0.06	135	563
logarithms [log(x) or log(s1)]	0.08	181	1283
complex addition	77	112	350
complex multiplication	83	119	363
complex sin	155	250	1156
complex log	171	274	1649

The reason the 16 digit complex number manipulation is significantly more CPU intensive than the other 16-digit calculations is that all complex number operations are done using the arprec functions, even if the precision has been set to only 16 digits.

Despite the increased computation time associated with arprec math, there are times when high precision arithmetic is needed. Salient examples include situations where the relatively small IEEE exponent range of ±308 is inadequate for a given calculation. This limitation is often a problem in physics, chemistry and engineering calculations. Overflow or underflow problems are easily eliminated by switching to arprec arithmetic. Cryptography, numerical integration, perturbation theory and Monte-Carlo methods also benefit significantly from expanded precision. It is also useful to do a sample calculation using arbitrary precision arithmetic to verify that truncation error is not a problem, and then revert to double precision after verifying that it is adequate.

|MS| Compilation

When running in MathScriptor, it is possible to compile your program into an internal code format that provides a more compact program and often allows your program to run faster. Compiled programs carry the extension .mcc, which stands for MathScriptor Compiled Code. Code that is compiled in MathScriptor can be run within Scriptor even if functions reserved to the MathScriptor language are used. Your instructor may distribute problem sets that have been compiled for the sole reason of providing a demonstration of what the correct output should look like while not giving away how to write the program to create that output. You cannot convert compiled code back into readable source code. For this reason it is critical that you always save your original source code. To provide backup, each time you compile, your source code is saved into a folder called "saved_source_code" which is created by MathScriptor within your Programs folder. If you do a lot of compiling this folder can get fairly large and at some point it may be wise to copy this folder to a backup medium and remove the contents. The code is listed not by the name of the program, but by a unique identifier name that includes the date (or to be precise, the date stored as the current date within the computers operating system). For example, if you compile a program on January 8, 2007 the source code will be saved with a filename of s2007_01_08a1.txt. The last two characters increment from a1 to z9 allowing for 234 source code backup saves in a given day. If you need more than that, you are probably operating under the misconception that you need to convert your program into a compiled version to run the program. This is true, but simply pressing the Run button compiles and runs the program while leaving the source code in the programming window. This is the recommended mode. The Compile menu item is for generating compiled programs for distribution to others when you wish to prevent the recipient from seeing your source code, or you want to make sure a Scriptor user can run the program. Compilation will also allow you to optimize your compiled program so that it runs faster provided you are using an LLVM or XS version of Scriptor and operating in MathScriptor mode.

Internal Methods and Reserved Keywords

The following is a complete list of all the internal methods and keywords associated with Scriptor and MathScriptor**.** Methods only available in MathScriptor mode are preceded with a |MS| symbol. If a parameter, set of parameters or keyword is enclosed in brackets, it is optional or should be used only under certain conditions, which are specified. All trigonometric functions in Scriptor operate on (or in the case of the arc functions return) radians. To convert degrees to radians, multiply degrees by const_degree. To convert radians to degrees, divide radians by const_degree.

// Program name: my_program_name.txt First line of a program- this line will tell the save command the name to use when saving the program. This example will cause "my_program_name.txt" to be the suggested file name when saving the program to the programs folder. One always have the option to change this name during the save dialogue and change the location where the program is saved. If the program names starts with the phrase "Template_", then the program is saved into the Templates folder.

// end program The last statement of a program. This statement is automatically generated during either a Precompile (under Debug menu) or during a Structure and Mark Program operation (under the Debug menu). If all structural elements (loops, conditionals, etc.) of the program are not complete (satisfied) prior to this statement, a compiler error is generated and the program will not be run.

|MS| **//# load_class**(class_filename.txt, iwindow) load the class in file class_filename.txt into the iwindow editfield within the objects panel; Class files must be in classes folder; make sure there is a single space between # and load_class

|MS| **//# load_method**(method_filename.txt, iwindow) load the method or set of methods in file method_filename.txt into the iwindow editfield within the objects panel;Method or method set files must be in methods folder; make sure there is a single space between # and load_method

|MS| **//# load_module**(module_filename.txt, iwindow) load the module in file module_filename.txt into the iwindow editfield within the objects panel; Module files must be in modules folder; make sure there is a single space between # and load_module

|MS| **#pragma directive [boolean]** sets or clears compiler directives as listed below. Those that require a Boolean are listed with the Boolean as well as the meaning. Note that this statement is not preceded by a double slash (//).

BackgroundTasks True (enables yielding to background threads, which is the default)
BackgroundTasks False (disables yielding to background threads and speeds up program)
DisableBackgroundTasks (disables yielding to background threads and speeds up program)
BoundsChecking True (enables monitoring of array subscripts to trap out_of_bounds exceptions, the default)
BoundsChecking False (disables monitoring of array subscripts, and speeds up program)
DisableBoundsChecking (disables monitoring of array subscripts, and speeds up program)
NilObjectChecking True (enables trapping of nil objects prior to usage, which is the default)

NilObjectChecking False (disables trapping of nil objects prior to usage, and speeds up program)

StackOverflowChecking True (enables monitoring of the stack size to prevent overflow, default)

StackOverflowChecking False (disables monitoring of the stack size, and speeds up program)

Abs(x) **as double** returns the absolute value of the argument. This function, like most of the other standard math functions is arprec savvy which means you can feed it a string representation of a real or complex number, and it will operate in arprec mode and return a string representation of the absolute value.

Acos(x) **as double** returns the arccosine of the argument. This function is arprec_savvy, which means x can also be a string representation of a numerical value, either real or complex. Complex numbers are two string numbers separated by a comma (e.g. "1.234, 3.456" which represents 1.234 + 3.456I).

Acosh(x) **as double** returns the inverse of the hyperbolic cosine, cosh(x).

Active_canvas as integer returns the currently visible canvas (1 in Main, 2 in Graphics panel or 3 in Music panel.) This variable is useful when writing programs which are to be run in two or more different panels.

|MS| **arprec_degree as string** returns an arprec string representing the number of radians per degree with a precision of idigits as determined by the most recent arprec_set_precision(idigits). The |MS| label on this and subsequent statements indicates that it is only available in MathScriptor mode.

|MS| **arprec_e as string** returns an arprec string representing the value of the mathematical constant, e = 2.71828.., with a precision of idigits as determined by the most recent arprec_set_precision(idigits).

|MS| **arprec_euler as string** returns Euler's constant (= 0.5772156649…) to an arbitrary precision, but with a maximum accuracy of 1000 digits.

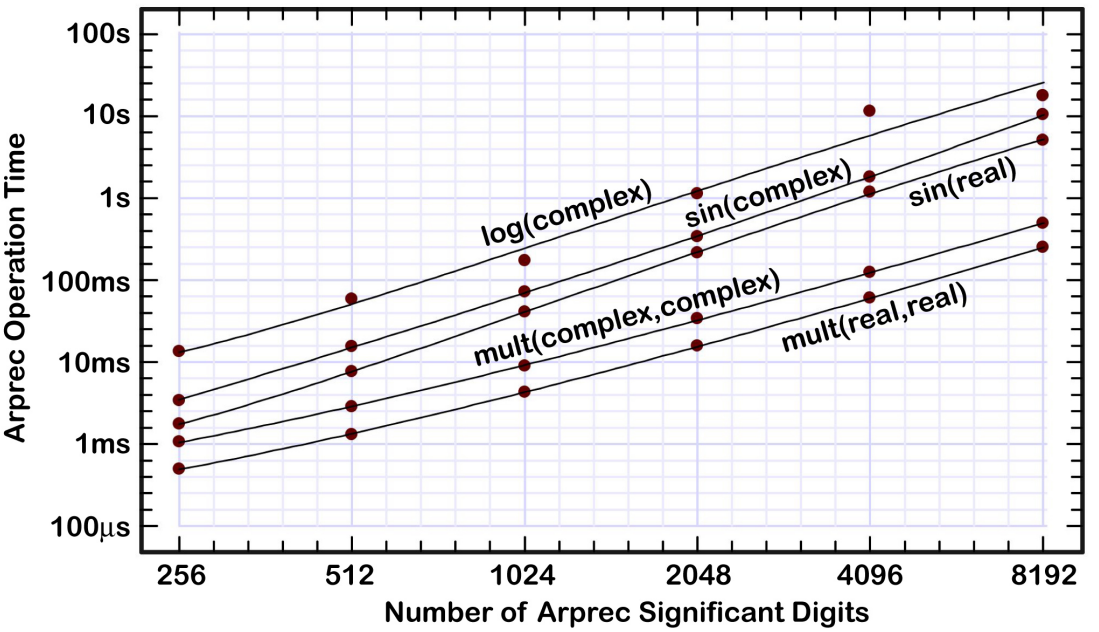
|MS| **arprec_factorial(n as integer) as string** returns an arprec string equal to the factorial of n, where n is a positive integer less than or equal to 8, 600, 000. This function saves results from previous evaluations to speed subsequent calculations.

|MS| **arprec_pi as string** returns the value of π as an arprec string. The number of significant digits is limited only by memory and is set by arprec_set_precision(idigits) (see below).

|MS| **arprec_precision as integer** returns the current precision of arprec calculations. Setting this variable has no effect. Use arprec_set_precision to set the precision.

|MS| **arprec_random_float([seed string]) as string** returns a random floating point number between 0 and 1 as an arprec string. The optional seed string, if included, will reseed the generator. The seeding process should only be done once, and a given seed will always generate an identical set of random numbers.

|MS| **arprec_random_integer(ndigits) as string** returns a random integer of length ndigits as an arprec string.



|MS| **arprec_set_precision(idigits)** sets the precision of the all subsequent arprec arithmetic in terms of the number of significant digits prior to the exponent. The choice of precision impacts the time necessary for an operation as shown above for selected operations. In general, complex math doubles the time of operations on single arguments and quadruples the time of operations involving two arguments (i.e. mult, div, plus, minus).

|MS| **arprec_variational_min(sx(), sy(), n) as string** returns the value of x for which y is a minimum for three (n=3) or four (n=4) pairs of x, y data points. If the minimum is outside the range of values defined by the data pairs, the x value on the edge that has the smallest y value is returned. The math is variational which means the function returns x for which the derivative dy/dx=0. Best results and highest computational speed are realized by using n=3, where the central (2nd) pair [sx(2), sy(2)] are close to the minimum.

|MS| **arprec_zeta_critical_root(n) as string** returns an arprec string identical to the zeta_critical_root(n), but to the designated arprec precision or 64 digit, whichever is smaller. If the arprec precision is greater than 64 digits, this function returns exactly 64 digits (i.e. zero padding is not done). At present, the maximum value of n is 60, 000 and requests for roots higher than 60, 000 returns 0 [arprec_zeta_critical_root(>60000) = 0]. The largest available root is returned by the function arprec_zeta_critical_root(-1). By convention, the 0^{th} root is set equal to 1 even though no such root exists.

Array(comma delimited list of any type variable) **as variant one-dimensional array** transfers the comma delimited set of variables into a one-dimensional array of the appropriate type starting at the 0th element. Usage example: a1 = array(1.0, 2.0, 3.0, 4.0), where a1 is a one-dimensional double array. If you want to start at the array(1) element, simply insert a null value (0, 0., or "") in the first position as shown in the following string example: string_array = array("", s1, s2, s3, ...). The recipient array is dynamically redimensioned to be of the exact size as required to hold the data. The compiler checks the first element of the comma delimited list and verifies that it is of exactly the correct type as the one-dimensional array. Thus, if the target is a double array, make sure the first element has a decimal point so the compiler doesn't assign it as an integer. Likewise, if it is an integer, make sure there is no decimal point. As of version 1.8.2, this function can also handle a comma delimited list of colors.

Asc(s0) **as integer** returns the ASCII value of the first character of the string s0.

AscB(s0) **as integer** returns the ASCII value of the first byte of the string s0, which is also the first character for a majority of strings. However, if the string represents Chinese or Japanese glyphs, this function returns nonsense. The reason this function exists is speed. If one knows that the string is a standard Roman character string, this function can be used to enhance execution speed by factors of 3-10.

Asin(x) **as double** returns the arcsine of the argument.

Asinh(x) **as double** returns the inverse of the hyperbolic sine, sinh(x).

Assigns, when placed in front of the last variable declared in a list of subroutine parameters, indicates that the variable is assigned by using an equals sign (see Section 2.7.1).

Atan(x) **as double** returns the arctangent of the argument.

Atan2(x, y) **as double** returns the arctangent of the point whose coordinates are x and y. This function is valuable because it provides access to all four quadrants.

Atanh(x) **as double** returns the inverse of the hyperbolic tangent, tanh(x).

Atomic_orbital_list is a string which contains a comma delimited list of the atomic orbitals utilized in the most recent SCF calculation in order of appearance in the LCAO expansion. For formaldehyde (O, C, H, H) the variable returns "O1(2s), O1(2px), O1(2py), O1(2pz), C2(2s), C2(2px), C2(2py), C2(2pz), H3(1s), H4(1s)". If you wish this list to be loaded into a string array, orbs(1..norbs), execute orbs=string_split(aromic_orbital_list, ", ").

Atom_properties_cndo(atomic_number, property_letter_code, n1, byref atom_name) **as double** returns an individual atom property based on the CNDO/2 parameterization. The n1 integer designates the orbital: 1(s), 2(p) or 3(d). This function returns the letter representing the atom in the byref parameter atom_name. The property_letter_code options are as follows:

A valence state electron affinity in eV (A_{ss} , A_{pp} or A_{dd}; n1=1, 2, 3)
B β_0 (atomic bonding parameter) in eV
E $(I_\mu + A_\mu)/2$ for s, p or d (n1=1, 2, 3) in eV
G gamma (one center repulsion integral) in eV
I valence state ionization potential in eV (I_{ss} , I_{pp} or I_{dd}; n1=1, 2, 3)
M Slater Exponent (ξ)
S put all this information into the spreadsheet (see example below).
U U_{ss}, U_{pp} or U_{dd} (n1=1, 2, 3) in eV
Z valence core charge (not a parameter but a fact of atomic structure)

The reason this method is called atom_properties_cndo() is because none of the additional INDO parameters are generated. The INDO parameters are fixed by evaluation of the slater integrals and cannot be modified. Set Q_use_external_parameters to true to read the parameterization from the spreadsheet. The program only reads the parameters in the columns 3-8. An example of the spreadsheet printout is shown below. The single line statement atom_properties_cndo(i, "S", n1, s1) where all but the second parameter are dummies.

atom	Z	Slater	gamma	-beta0	(Is+As)/2	(Ip+Ap)/2	(Id+Ad)/2	Uss	Upp	Udd	Iss	Ipp	Idd	Ass	App	Add
H	1	1.200	20.408547	9.00000	7.17610			-17.38037			17.38037			-3.02817		
He	2	1.700	28.912108													
Li	1	0.650	6.425503	9.00000	3.10550	1.25800		-6.31825	-4.47075		6.31825	4.47075		-0.10725	-1.95475	
Be	2	0.975	9.638255	13.00000	5.94557	2.56300		-20.40295	-17.02038		10.76470	7.38213		1.12644	-2.25613	
B	3	1.300	12.851007	17.00000	9.59407	4.00100		-41.72159	-36.12852		16.01957	10.42650		3.16857	-2.42450	
C	4	1.625	16.063759	21.00000	14.05100	5.57200		-70.27416	-61.79516		22.08288	13.60388		6.01912	-2.45988	
N	5	1.950	19.276510	25.00000	19.31637	7.27500		-106.06067	-94.01930		28.95463	16.91326		9.67811	-2.36326	
O	6	2.275	22.489262	31.00000	25.39017	9.11100		-149.08111	-132.80194		36.63480	20.35563		14.14554	-2.13363	
F	7	2.600	25.702014	39.00000	32.27240	11.08000		-199.33549	-178.14309		45.12341	23.93101		19.42139	-1.77101	
Ne	8	2.925	28.914766													
Na	1	0.733	5.151150	7.72030	2.80400	1.30200	0.15000	-5.37958	-3.87758	-2.72558	5.37958	3.87758	2.72558	0.22842	-1.27358	-2.42558
Mg	2	0.950	6.673081	9.44710	5.12540	2.05160	0.16195	-15.13502	-12.06122	-10.17157	8.46194	5.38814	3.49849	1.78886	-1.28494	-3.17459
Al	3	1.067	7.492582	11.30110	7.77060	2.99510	0.22425	-26.50206	-21.72656	-18.95571	11.51689	6.74139	3.97054	4.02431	-0.75119	-3.52204
Si	4	1.383	9.716943	13.06500	10.03270	4.13250	0.33700	-44.04200	-38.14180	-34.34630	14.89117	8.99097	5.19547	5.17423	-0.72597	-4.52147
P	5	1.600	11.238873	15.07000	14.03270	5.46380	0.50000	-64.60763	-56.03873	-51.07493	19.65214	11.08324	6.11944	8.41326	-0.15564	-5.11944
S	6	1.817	12.760804	18.15000	17.64960	6.98900	0.71325	-87.83402	-77.17342	-70.89767	24.03000	13.36940	7.09365	11.26920	0.60860	-5.66715
Cl	7	2.033	14.282735	22.33000	21.59060	8.70810	0.97695	-114.42838	-101.54588	-93.81473	28.73197	15.84947	8.11832	14.44923	1.56673	-6.16442
Ar	8	2.250	15.804666													

Bessel(n, x) **as double** returns $J_n(x)$, the value of the Bessel function of the first kind for argument x and integer values of n. Very fast algorithms are used for Bessel(0, x) and Bessel(1, x) calculations so that this function can be used for efficient Fourier transform apodization.

Bessel_general(itype, norder, x, byref stype) **as double**
returns the value (or zero point) of various Bessel functions for itype values as follows:
itype=1 Bessel J function of order = norder for argument = x
itype=2 Bessel Y function of order = norder for argument = x
itype=3 Bessel L function of order = norder for argument = x
itype=4 Bessel K function of order = norder for argument = x
itype=5 returns the x value for which Bessel J function of order = norder equals zero
itype=6 returns the x value for which Bessel Y function of order = norder equals zero
itype=7 Spherical Bessel J function of order = norder for argument = x
itype=8 Spherical Bessel Y function of order = norder for argument = x
The byref variable stype returns a string description of the Bessel function followed by the string equivalent in full precision.

Bin(i) **as string** returns the binary equivalent of the 32 bit integer i in a string. bin(15) = "1111", oct(15) = "17", hex(15) = "F"; bin(127) = "1111111", oct(127) = "177", hex(127) = "7F". If you want to enter an integer using a binary number use i = ivalue("&b1111111"). The largest argument is 2147483647 which returns bin(2147483647) = "11111111111111111111111111111111"

Binary_file_read(filepath, [filecontents] or [ii, nth] or [aa, nth] or [ia(), n1] or [a1(), n1] or [a2(), n1, n2]) **as integer** read the individual value or multiple values in the binary file designated by filepath. The filepath represents a path within the same folder as mathscriptor. For example, filepath="data_sets:bf12" reads the data in the file "bf12" inside the folder "data_sets". This routine can read a string, an integer array, or a double in the form of a one-dimensional array a1(0..n1) or a two-dimensional array a2(0..n1, 0..n2). Returns the file length in bytes (0=failure). For the one dimensional integer or double arrays, you have the option of only reading the nth value by replacing the array with a non-array variable and using n1 to select the nth array element.

Binary_file_write(filepath, [filecontents] or [ii, nth] or [aa, nth] or [ia(), n1] or [a1(), n1] or [a2(), n1, n2]) **as boolean** write the data in binary format to a disk file designated by the string filepath, where filepath references paths within the same folder as the program. For example, filepath="data_sets\bf12" stores the data in the file bf12 inside the folder "data_sets". Returns true if the operation was successful.

Binomial(n, k**) as double** returns the binomial coefficient for n over k [=n!/(k!(n-k)!)] where n>=k>=0.

BitwiseAnd(i, j) **as integer** returns the bitwise AND of the two arguments.
For example:
bitwiseand(5, 3) = 1
bitwiseor(5, 3) = 7
bitwisexor(5, 3) = 6
bin(5)=0101 & bin(3)=0011

BitwiseOr(i, j) **as integer** returns the bitwise OR of the two arguments.
example: bitwiseor(5, 3) = 7

BitwiseXor(i, j) **as integer** returns the bitwise XOR of the two arguments.
example: bitwisexor(5, 3) = 6

Buffer_Backcolor(ired, igreen, iblue) sets the background color of the buffer.

Buffer_Background_color = color alternative method to set the buffer backcolor.

Buffer_clear This statement clears the buffer to buffer_background_color. If multiple buffers have been created, this statement only clears buffer 1. The default buffer_background_color is pure white.

Buffer_copy_to_buffer(itarget) copy buffer 1 to itarget=2, 3, 4...<= nbuffers.

Buffer_copy_to_buffer(isource, itarget) copy buffer isource to buffer itarget.

Buffer_copy_to_canvas(icanvas, [ioption]) icanvas=1, 2, 3 or active_canvas; ioption=0, 1, 2. if ioption is left out, then a pixel-to-pixel fast copy is done starting at upper left (0, 0). If present, scaling is done based on ioption where ioption =0 (center graphic and preserve aspect ratio), 1 (upper left, preserve aspect ratio), 2 (fill completely and ignore aspect ratio). icanvas=1(main), 2(graphics), 3(music).

Buffer_copy_to_picture(ipicture) copies the contents of the entire buffer into picture slot ipicture. If ipicture has not been created, it will be created and the size of ipicture will be set equal to the size of the buffer.

Buffer_create(ioption, iwidth, iheight) creates a single buffer of size iwidth by iheight with a white background (ioption=0) or a colored background (ioption=1). If the latter, assign the color using graphics_background_color prior to this statement.

Buffer_create_multiple(nbuffers, ioption, iwidth, iheight) as above but creates multiple buffers each of which has a size of iwidth and iheight with a white (ioption=0) or colored (ioption=1) background. The sign of nbuffers determines placement:
2 (1 2 --- left right); -2 (1 above 2 -- up down) ;
3 (1 2 3 --- left middle right) ; -3 (1 above 2 above 3 ---- top middle bottom) ;
4 (1 2 3 4 --- left to right in line) ; -4 (1 2 on top, 3 4 underneath);
6 or -6 (1 2 3 on top 4 5 6 underneath) ;
9 or -9 (1 2 3 on top 4 5 6 in middle and 7 8 9 on bottom);
40 (1 2 3 4 --- top to bottom).

buffer_draw_dashed(x0, y0, x1, y1, L1, S1, L2, S2, line_thickness, dashcolor) draw a dashed line from x0, y0 to x1, y1 of width line_thickness and color dash_color. The dash lengths are L1 and L2 and the intervening spaces are S1 and S2. For a simple dash, L1=S1=L2=S2. All values are in buffer pixels.

Buffer_draw_odometer(value, ndigits, ndecimal, ix, iy, relative_size) draw an odometer into the buffer at position ix, iy, value = the value to be drawn in the odometer window (double), ndigits = total number of digits to display including the decimal place, which takes up one of the digit slots, ndecimal = number of digits to be displayed to the right of the decimal point, ix, iy are the x and y positions of the upper left corner of the odometer within the buffer window and relative_size gives the relative size of the odometer, 1.0=full size, 0.5=half size, etc.

Buffer_fill_from_array(a2(,)) fill the buffer from the double array a2(nx, ny) where nx<=buffer_width and ny<=buffer_height. Each value ranges from 0(black) to 1(white).

Buffer_fill_from_arrays(ired(), igreen(), iblue()) fill the buffer from the three integer arrays where each color is represented by a two-dimensional array as in ired(1..nx, 1..ny) where nx<=buffer_width and ny<=buffer_height. Each value ranges from 0 to 255.

Buffer_flip_buffers(i, j) flips the graphics contents of any two buffers within the range of buffers created using buffer_create_multiple.

Buffer_gaussian_blur(ilevel) carries out a high-quality, but slow, Gaussian blur on the buffer (or buffer 1 if multiple buffers are present). [A quick but less accurate blur function is available in buffer_quick_blur (see below).]

Buffer_height as integer returns the current height of the buffer in pixels. If a multiple panel buffer has been created, this variable returns the height of the individual buffer. Note that when there are multiple buffers, each buffer is the same size.

Buffer_pixel(ix, iy) **assigns as color** or **as color** reads or sets the color at buffer pixel ix, iy.

Buffer_pixel_blend(kx, ky, opacity [, iblendtype])=blend_color. blends the blend_color into the color found at pixel location kx, ky based on the opacity of the blend_color (which is on top) and the optional integer, iblendtype (0-6):

0 or 1 = normal additive transparency (RGB based, simulates colored transparent glass, default)
2 = subtractive transparency (CMY based) (simulates paint)
3 = hue based transparency (HSV based)
4 = additive transparency with enhanced lightening
5 = additive transparency with enhanced darkening
6 = moiré (artistic, textured) blend

Buffer_quick_blur(ilevel) carries out a high-speed, anti-aliased blur on the buffer (or buffer 1 if multiple buffers are present). Algorithm uses the internal graphics hardware. Not as accurate as buffer_gaussian_blur, but 100 times faster with adequate results for most applications.

Buffer_rotate(degrees [, fillcolor] or [, ix0, iy0]) rotate the picture in the buffer counterclockwise (degrees>0) or clockwise(degrees<0). If fillcolor is present, regions outside the rotated graphic are filled with this color. Rotation is about the center of the buffer if the second parameter is the fill color or around ix0, iy0 if the latter two integer variables are passed. Rotation around the center with a fillcolor is significantly faster.

Buffer_save_to_jpeg(filename as string) **as boolean** saves the buffer (of buffer 1 of a multiple buffer) to the user_pictures folder using the filename that is passed as a parameter with the extension, .jpg, added (i.e. filename.jpg). One can store a series of pictures into a folder inside the user_pictures folder by including the folder name in the filename, as shown below for Mac OSX:

```
filename="wavepackets:wave"+str(n)+str(v2)+format(ifile, "000")+".jpg"
```

where the individual files are named "wave001.jpg", "wave002.jpg" and are all place inside the folder wavepackets. This folder MUST be created prior to calling this method. This function will not create a folder. This works on other platforms, but the Max OSX separator ":" needs to be replaced with one appropriate for the operating system (i.e. "\" on Windows XP). This function requires that QuickTime (either Standard or Pro) be installed on the computer. Quicktime Standard is free and is available from the Apple Web Site (http://www.apple.com/quicktime/download/).

buffer_save_to_photoshop(fname) **as boolean** (Mac Only) Copy the current buffer to a photoshop file inside user_pictures with the path and filename given by the string fname. Returns true if successful.

buffer_save_to_tiff(filename) **as boolean** Save the buffer (or buffer 1 of a multiple buffer) to the user_pictures folder as an uncompressed tiff file with name filename.tif. Returns true if successful.

Buffer_to_array(a2(), ioption) converts the graphics pixels in the buffer to corresponding numerical values placed in the two-dimensional double array a2(nx, ny) with dimensions set by the size of buffer based on ioption, which determines how the RGB color channels are weighted:
ioption=1, then red is given double the weight;
ioption=2, then green is given double the weight;
ioption=3, then blue is given double the weight;
ioption=4 use Adobe standard (red*0.7, blue*0.89, green*0.41);
ioption=5 use grey scale;
ioption=6, then use only the red channel;
ioption=7, then use only the green channel;
ioption=8, then use only the blue channel;
if ioption is negative, the picture is inverted prior to extraction.

Buffer_to_arrays((ired(), igreen(), iblue()) copy the buffer into the three arrays where each color is represented by a two-dimensional integer array as in ired(1..buffer_width, 1..buffer_height). The three integer arrays are redimensioned to the size of the buffer. The values range from 0 to 255.

Buffer_trim(nleft, nright, ntop, nbottom) reduce the size of the buffer by cropping the buffer from the left, right, top and bottom by the number of pixels specified.

Buffer_width as integer returns the current width of the buffer in pixels. If a multiple panel buffer is created, this returns the width of the individual panel.

Buffer_write_paintbrush(ix, iy, ired, igreen, iblue) the pixel at position ix, iy and all adjacent pixels of the same color are replaced with the color RGB(ired, igreen, iblue)

ByRef when preceding a parameter, establishes a call by reference. This means that a pointer to the variable being passed is used to directly access that variable. Any changes in the parameter inside the method will alter the value of the variable passed. Note that all arrays are called ByRef under all circumstances.

ByVal when preceding a parameter, establishes a call by value. This means the value is passed to a local variable, and if the local variable is changes inside the method, the passed variable remains unchanged. If used in front of an array variable, it is ignored as all arrays are automatically called by reference.

Call this keyword, when placed in front of a function call, indicates the return variable is to be ignored.

Canvas_clear(icanvas) clears the canvas to pure white. Should be used prior to calling buffer_copy_to_canvas() if one is not filling the canvas. However, if one is generating a dynamic movie involving repeated buffer_copy_to_canvas() calls, one should avoid using canvas_clear except at the very beginning of the process. This statement causes flicker on Windows platforms. Repeated use of this statement slows down graphics on both Windows and Macintosh platforms. This statement is deprecated. It is better to clear the canvas by creating a buffer of the same size as the canvas and then doing a copy_buffer_to_canvas(..).

Canvas_height(icanvas) **as integer** height of canvas specified by icanvas= 0 (buffer), 1 (small canvas in main), 2 (large canvas in graphics).

Canvas_update (update_time) requests activation of the paint event of the visible canvas, and hands time over to the event loop to carry out the canvas update. Update_time is in milliseconds, and for most applications, a value between 10 and 30 will suffice. This number will need to be increased for slower computers. The user should make sure that buffer_copy_to_canvas() has been executed prior to calling canvas_update. This method is rarely needed as the computer will usually automatically update the canvas automatically in the background.

Canvas_width(icanvas) **as integer** width of canvas specified by icanvas = 0 (buffer), 1 (small canvas in main), 2 (large canvas in graphics).

Case [**is**] part of the Select Case statement. Use "Case is" before < or > conditionals.

```
select case expression
case 1
case 2, 3, 4
case 5 to 8
case 9 to 11, 15, 19
case is [<[=]][>[=]] value
...
else
...
end select
```

Ceil(x) **as double** rounds x up to the nearest integer (but returns it as a double)

if x=4.952, floor(x)=4, ceil(x)=5
if x=-4.95, floor(x)=-5, ceil(x)=-4
if x=6, floor(x)=6, ceil(x)=6

Chebyshev(n, x) **as double** returns the numerical value of the nth Chebyshev polynomial for argument x. The first possible value of n is 1, not zero, so the "zeroth" Chebyshev polynomial is referenced by n=1. The first seven polynomials are listed below:

Chebyshev(1,x) = $T_0(x) = 1$
Chebyshev(2,x) = $T_1(x) = x$
Chebyshev(3,x) = $T_2(x) = 2x^2 - 1$
Chebyshev(4,x) = $T_3(x) = 4x^3 - 3x$
Chebyshev(5,x) = $T_4(x) = 8x^4 - 8x^2 + 1$
Chebyshev(6,x) = $T_5(x) = 16x^5 - 20x^3 + 5x$
Chebyshev(7,x) = $T_6(x) = 32x^6 - 48x^4 + 18x^2$

Check_and_clear_input as string Each time this function is called, it returns the character equivalent of the keyboard key that was pressed. If the string is empty, a key may have been pressed, but does not translate into a valid character. The carriage return "<CR>" and backspace or delete keys "<BS>" and escape "<ESC>" are returned using the indicated bra-ket abbreviations. All other non-printing characters generate null strings. The function is cleared each time it is called so if a new character is found, then a unique key event was trapped. On Windows, it is more reliable to use the key_down_ascii_value and only use this command to update the value of key_down_ascii_value.

Check_for_mouse_action(imilliseconds, x1, x2, y1, y2) **as Boolean** returns true when the mouse has been pressed and released within a rectangle defined by x1, y1 and x2, y2 in the graphics canvas within the graphics panel. The variable imilliseconds gives the number of millisecond latency allowed for user mouse movement and should be set to 0 and then increased if the user is not getting enough time allocated to the mouse manipulation by the program.

Check_for_stop_button as Boolean returns true if the stop button has been pressed at any time during the current Run. For example, do .... loop until check_for_stop_button will continue to loop until the stop button has been pressed. This approach is somewhat slower than do ... loop until check_for_user_action("...") which monitors keyboard presses via interrupt rather than threading. If you select the menu item, "activate debug stops via mouse click" under the debug menu, the testing is very fast, but any mouse click will set this Boolean variable to true. If the above menu item is not checked, you will need to press the stop button.

Check_for_user_action(action_string) **as Boolean** returns true when the user has pressed the appropriate keyboard key specified by the action_string as follows: "mouse down", "mouse down now", "control key", "option key", "shift key", "command key", "control key now", "option key now", "shift key now", "command key now" or "user cancelled". In the case of user cancelled the Mac expects comman+period and Windows expects escape, but the Mac will usually respond to either. If none of the above are found, the action_string is converted to a number using val(action_string) and that number is compared to the current keycode that is being pressed. If so, it returns true. For example, keycodes 123 through 126 are associated on most keyboards with the four arrow keys.

chop(x) **as double** Chop the argument to nine significant digits and set to zero if the absolute value is less that 10^-15. This function is Arprec savvy (handles both string reals and complex strings).

Chr(i) **as string** returns the character whose ASCII value is passed.

ChrB(i) **as string** returns a single byte character representing the value i.

Class classname marks the start of a class definition.

Clear_graphics(icanvas) clears icanvas canvas [1(Main), 2(Graphics)].

Clear_mouse_data resets mouse_down and mouse_up data.

Clear_text_output(itarget) clears target panel text output [0(both), 1(Main), 2(Text)].

CMY(cyan, magenta, yellow) **as color** specifies a color using CMY system. This is the system used for color ink rather than additive color on a CRT. Values of the three variables range from 0 (none) to 1.0 (maximum amount).

Color_modify(starting_color, dark_bright, color_shift) **as color** returns a modified version of starting_color which has its lightness modified via dark_bright (0.1(much darker)...0.5(darker)...1(no change)...1.5(brighter/lighter)...2(twice as bright)) and its hue shifted by color_shift (-1.0 to 1.0, 0=no shift).

Color_selection_window(color, text_prompt) **as Boolean** opens the color selection window and allows the user to select a color using whatever methods are available via the operating system. The text_prompt is displayed on some operating systems and the color is returned in color. The routine returns false if the user pressed cancel.

Color_value(color, s0) **as double** returns the requested color value where s0 can equal "hue", "saturation", "value", "red", "green", "blue", "cyan", "magenta", "yellow". Only the first letter is looked at so you can simplify s0 as h, s, v, r, g, b, c, m, y. Although red, green and blue values are integers from 0 to 255, the function returns a double and needs to be handled as a double upon return. An alternative to using this function is to use the .red, .green, .blue, .cyan, .magenta, .yellow, .hue, .saturation or .value extensions to return or assign individual color components.

Const variable_name = string, integer or real value defines a constant. An attempt to change a constant will generate a run-time error, so dont.

Constructor At least one subroutine called constructor is required in each Class, and will be run when the class is instantiated. These subroutines, all with the name constructor, can be overloaded so that a class can be instantiated with different properties based on the selection of parameters. A destructor subroutine, which is run when the class goes out of scope, is optional.

Const_degree as double internally defined constant = radians/degree = 0.017533…

Const_e as double internally defined constant = 2.71828... (no longer supported, but note that an arprec_e is available).

Const_eol as double end of line character for current computer.

Const_pi as double internally defined constant = π = 3.14159....

Const_tab as string the tab character.

Convert_to_string(any_variable) Converts single, double, integer, Boolean, or colors into their string representation for printing while maintaining full accuracy. This function is very useful for converting doubles to strings for use in Arprec math without losing precision.

Cos(x) **as double** returns the cosine of x assuming x is in radians.

Cosh(x) **as double** returns the hyperbolic cosine of x.

CountFields(source, separator) **as integer** returns the number of fields within the source string delineated by the string separator.

Currency a variable of use in business calculations designed to store and manipulate money with maximum accuracy. Three digits to the right of the decimal point and fifteen digits to the left of the decimal point are available. Adequate to handle the US budget deficit created under the second Bush administration.

Date_seconds_to_SQL(total_seconds **as double**) **as string** returns the SQL date and time as a function of the number of seconds that have elapsed since January 1, 1904. SQL format is YYYY-MM-DD HH:MM:SS

Date_SQL_now as string returns the SQL date and time as of the moment during which the statement is executed. SQL format is YYYY-MM-DD HH:MM:SS

Date_SQL_to_seconds(SQL_string) **as double** returns the number of seconds that have elapsed since 1904-01-01 00:00:00. A date and time prior to 1904-01-01 00:00:00 will yield a negative value. Dates prior to Jan 1, 1601 are assigned based on the modern calendar and may be different from historical record. The SQL date is a string in the format YYYY-MM-DD HH:MM:SS

Date_to_seconds(string_date) **as double** returns the number of seconds that have elapsed since January 1, 1904. A date prior to Jan 1, 1904 will yield a negative value. Dates prior to Jan 1, 1601 are assigned based on the modern calendar.

Debug_save_workspace when placed in a program, this statement saves both the program and the current output in Main inside the programs folder with a filename of "debug_workspace"

Destructor name header for an optional subroutine within a class definition that is executed when the class goes out of scope. Can be used to release memory, remove confidential information in a spreadsheet or handle other tasks that are only needed when the class is no longer active (in scope).

Dim assign space for variable of any type, one line for each type.

Dim pie **as double** = 3.14159265 assign variable and its initial value. If a constant variable is desired, use the Const expression.

|MS| **div**(s1, s2) **as string** returns s1/s2 using arbitrary precision string arithmetic.

const_Degree as double internal constant that defines the ratio of radians divided by degrees. Multiply angle in degrees by const_degree to convert degrees to radians. const_degree = π/180 = const_pi/180.0 = 0.017453292519943.

Do [until]... **Loop [until]**... see loop discussion above

Downto used in For loop to decrement counter. When downto is used with the optional step keyword, the variable or numerical value following step should be positive, not negative as intuition would suggest. The step keyword assigns an increment, not the direction. When a negative step size is encountered, the loop will not execute at all.

Draw_arrays(ix(), iy(), n1, n2) draws a set of lines connecting the points ix(n1), iy(n1) to ix(n1+1), iy(n1+1) … ix(n2-1), iy(n2-1), ix(n2), iy(n2) to the buffer. The line color is set using Graphics_forecolor(color) and the line width is set using Graphics_stroke_width(npixels).

Draw_arrow(ix0, iy0, ilength, iwidth, theta, fill_color, draw_color) draw an arrow of the specified length and width, with origin at ix0, iy0 at an angle of theta (degrees). The outline is drawn using the draw_color and the arrow is filled with fill_color. Thus this function serves to both draw and fill the arrow, so there is no fill_arrow command. The thickness of the outline (draw portion) is controlled by a prior call to graphics_stroke_width.

Draw_axes(x1, x2, y1, y2, line_thickness) draws unlabeled axes into the buffer and enclosing the entire buffer inside. This is designed to be used with plot_contour() or plot_filled_contour() which also uses the entire buffer.

Draw_circle(ix_center, iy_center, radius) draw a circle with center at ix_center, iy_center of radius = radius. The thickness of the circle is assigned via a prior assignment of graphics_stroke_width.

Draw_line(ix1, iy1, ix2, iy2) draw a line from ix1, iy1 to ix2, iy2. The thickness of the line is assigned via a prior assignment of graphics_stroke_width.

Draw_oval(ix_center, iy_center, iwidth, iheight)

Draw_paragraph(string_paragraph, sfontname, ifontsize, paragraph_pixel_width, byref line_separation, ix, iy) **as boolean** Draws the string_paragraph to the buffer with upper left coordinates ix, iy and using the font sfontname of size ifontsize. The maximum paragraph_pixel_width should be less than the width of the buffer because it is only checked in between words. The line_separation should be assigned an initial variable, but if the paragraph extends beyond the buffer height, this method returns false and a smaller line_separation. A loop can be used to continue trying until the line_separation has been reduced to a value that fits. Make sure there is at most one end-of-line character in the string_paragraph, and if it is present, it is at the end of the string_paragraph. Feeding a string_paragraph parameter with multiple paragraphs inside (as delineated with the end-of-line character) generates unreadable results.

Draw_rect(ix_center, iy_center, iwidth, iheight)

Draw_rotated_string(text, ix_center, iy_center, angle) draw a string centered at ix_center, iy_center that has been rotated by the angle in degrees (not radians). The nature of the string object is assigned by prior execution of graphics_font.

Draw_string(text, ix_center, iy_center) Draw the single line string text centered at ix_center and iy_center. If multiline (string contains one or more end-of-line characters), ix_center and iy_center designate the left edge of the first line of a left-aligned paragraph. One can force left alignment of a single line string by making ix_center negative (x_left = abs(ix_center)). All "<*p*>" substrings found will be replaced with endofline characters.

Else optional component of a conditional (if then .. elseif then… else… end if)

Elseif optional component of a conditional (if then .. elseif then… else… end if)

End class last statement of a class definition

End function last statement of a function

End if last statement of a conditional (if then .. elseif then… else… end if)

End module last statement of a module

End select last statement of a select case statement

End sub last statement of a subroutine

Erf(x) **as double** returns the error function for real values of x.

Erfc(x) **as double** returns the complimentary error function for real values of x.

Exit jumps to the first line of code following the last statement of the current loop

Exp(x) **as double** returns the natural exponent of x = pow(e, x) = e^x

Extends Used in a method declaration to indicate that the method is to be called using object syntax, i.e., as if it were a method belonging to the object itself. Extension methods should be inside a module as shown in the example below, but the newer versions of MathScriptor allow functions with extends outside of modules. To call, simply use dot syntax: if s0.endswith("s") then…

```
module string_extends
function endswith(extends s0 as string, withwhat as string) as boolean
    // Return true if string s0 ends with the string 'withWhat',
    // doing a standard string comparison.
    return right(s, withWhat.Len) = withWhat
  end function
  function beginswith(extends s as string, withwhat as string) as boolean
    // Return true if 's' begins with the string 'withWhat',
    // doing a standard string comparison.
    return left(s, withWhat.Len) = withWhat
  end function
end module
```

Factorial(ix) **as double.** returns ix!, the factorial of ix for values of ix from 0 to 170. If factorials above 170 are required, use arprec_factorial.

$$n! = \prod_{k=1}^{n} k \qquad \forall n \in \mathbb{N}.$$

Factorial_double(ix) **as double**. returns ix!!, the double factorial of ix for values of ix from -1 to 300.

$$n!! = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1; \\ n \times (n-2)!! & \text{if } n \geq 2. \end{cases}$$

n	n!	n!!
1	1	1
2	2	2
3	6	3
4	24	8
5	120	15
6	720	48
7	5040	105
8	40320	384
9	362880	945
10	3628800	3840
11	39916800	10395
12	479001600	46080
13	6227020800	135135
14	87178291200	645120
15	1307674368000	2027025
16	20922789888000	10321920
17	355687428096000	34459425
18	6402373705728000	185794560
19	121645100408832000	654729075
20	2432902008176640000	3715891200
21	51090942171709440000	13749310575
22	1124000727777607680000	81749606400
23	25852016738884976640000	316234143226

False is the opposite of true, and is one of the two possible states of a Boolean.

Fourier-Transform methods are listed below. All the Fast Fourier Transform (FFT) methods require that n, the number of points, be a multiple of 2 (i.e. equal to 2^k, where k is an integer). Valid values of n are 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288 or higher. The coefficients are stored in a folded format so that the lowest frequencies are on the "edges" and the highest frequencies are in the middle. The 0th element is the offset, or frequency=0, component. The ft_linearize statement is available to translate the coefficients to more manageable form where the 0th to highest frequency components are stored from low to high in subscript. These can be refolded by using ft_fold.

FFT1(a1(), c1(), s1(), n) one-dimensional fast Fourier transform of the linear array a1(0….n) placing cosines into c1(0..n) and sines into s1(0..n).

FFT1_inverse(a1(), c1(), s1(), n) inverse 1D fast Fourier transform. Takes the cosine c1(0..n) and sine s1(0..n) arrays and returns the inverse transform in a1().

FFT2(a2(), c2(), s2(), n) two-dimensional fast Fourier transform of the two-dimensional array a2(0….n, 0...n) placing cosines into c2(0….n, 0...n) and sines into s2(0….n, 0...n).

FFT2_complex_association(c2a(), s2a(), c2b(), s2b(), n) **as double** returns the complex association between a pair of transforms of the same size (n by n). A value of 1 means the images or data transformed have near perfect association. This function mimics Fourier transform holographic association and is useful for simulation human associative processes.

FFT2_inverse(a2(), c2(), s2(), n) inverse 2D fast Fourier transform.

FT_fold(c1(), s1(), c1lin(), s1lin(), npoints) transform the linearized coefficients in c1lin(1..npoints) and s1lin(1..npoints) into a folded pair in c1(1..npoints) and s1(1..npoints). When folded, the lowest frequencies are at 1 and npoints and the highest frequency is in the middle (npoints/2). The zeroth frequency components are transferred such that s1lin(0)=s1(0) and c1lin(0)=c1(0).

FT_linearize(c1(), s1(), c1lin(), s1lin(), npoints) transform the folded Fourier coefficients in c1(1..npoints) and s1(1..npoints) into a linearized pair in c1lin(1..npoints) and s1lin(1..npoints). When linearized, the frequency increases linearly from low subscript to high subscript. The zeroth frequency components are transferred such that s1(0)=s1lin(0) and c1(0)=c1lin(0).

Fill_arrays(ix(), iy(), n1, n2) fills a set of lines connecting the points ix(n1), iy(n1) to ix(n1+1), iy(n1+1) … ix(n2-1), iy(n2-1), ix(n2), iy(n2) with the current forecolor. The result is sent to the buffer.

Fill_circle(ix_center, iy_center, radius) fill a circle of a given radius with forecolor (set first) and place the center of this circle at ix_center, iy_center.

Fill_oval(ix_center, iy_center, iwidth, iheight) fill an oval with forecolor.

Fill_rect(ix_center, iy_center, iwidth, iheight) fill a rectangle with forecolor.

Fit_chebyshev(x(), y(), ndata, a(), nparams, rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to orthogonal Chebyshev polynomial with weights returned in a(1..nparams). Numerical values of the Chebyshev polynomials are given by the function chebyshev(iterm, x), and the resulting fit evaluated using:

```
for j=1 to nparams
ypi=ypi+a(j)*chebyshev(j, x0)
next
```

Fit_exponential(x(), y(), ndata, a(), rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to the equation y = a(1)*exp(a(2)*x). Note that unlike the other linear fits, nparams is not included as a parameter because it is fixed at 2. The chi-squared goodness of fit is returned in chisqr.

Fit_fourier_transform(x(), y(), n, xp(), yp(), np, wapp) **as string** return the result of a FFT fit to the data in x(1..n), y(1..n) in the plotting arrays xp(1..np), yp(1..np). The value of np is equal to the number of coefficients used in the FFT, and thus must be equal to 2^n, where n=4 to 16. High frequency noise can be reduced by setting the apodization constant, wapp, to values above 0. Wapp is a double between 0.0 and 2.0, but values above 0.5 may generate spurious edge effects.

Fit_genpoly(x(), y(), ndata, nparams, xp(), yp(), xmin, xmax, npoints, [sxponents]) **as string** returns a string summary of the best fit general polynomial for the data x(1..ndata), y(1..ndata) using a maximum of nparam fit parameters. The equation that is fit is a(1)*x^b(1) + a(2)*x^b(2)+ ... +a(nparams)*x^b(nparams), where b(1)..b(nparams) are integers in the range -10 to 10. This function then returns calculated values in xp(1..npoints), yp(1..npoints) for x in the range xmin to xmax. If the string sxponents is included, it lists one or more exponents of x that will be included in the fit (e.g."0, 1, -2" and nparams=4 will fit to y=a+bx+c*x^-2+d*x^n) where a, b, c, d and n are optimized by the fit.

An alternative form of genpoly is the following…

Fit_genpoly(x(), y(), ndata, a(), b(), nparams) **as string** returns a string summary of the best fit general polynomial for the data x(1..ndata), y(1..ndata) using a maximum of nparam fit parameters. The equation that is fit is a(1)*x^b(1) + a(2)*x^b(2)+ ... +a(nparams)*x^b(nparams), where b(1)..b(nparams) are integers in the range -10 to 10. This function is overloaded and exists in two forms. Above method returns the coefficients instead of the best fit line. Both forms are iterative and are CPU intensive.

Fit_henderson_hasselbalch(pH(), y(), n, a(), pka(), nterms, xp(), yp(), x1, x2, np) **as string**

carries out a one (nterms=1), two (nterms=2) or three (nterms=3) term Henderson Hasselbalch fit to some numerical property of an ionizable molecule or protein. The measurements in y(1..n) as a function of pH(1..n) are fit to the equation:

$$\text{Property}\left(pH\right)=a_0+\frac{a_1}{1+10^{(pH-pK_a^{(1)})}}+\frac{a_2}{1+10^{(pH-pK_a^{(2)})}}+\frac{a_3}{1+10^{(pH-pK_a^{(3)})}}$$

The function returns the fit parameters in a() [a_0=a(0), a_1=a(1), etc.] and pK_a(1..3) and the result in xp(1..np) and yp(1..np) from x1 to x2 where np is the number of points which should be set by the user. If np=0, no curve is returned. The function returns a string summarizing the results of the fit. An alternative slow but safer method is available using fit_henderson_hasselbalch_safe(..).

Fit_lanczos(x(), y(), ndata, a(), nparams, rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to a Lanczos-type polynomial with nparams parameters returned in a(1..nparams):
y=a(1)+a(2)/x+a(3)/x^2+…+a(nparams)/x^(nparams-1).
The chi-squared goodness of fit is returned in chisqr. The function that is fit is credited to the Hungarian mathematician and physicist, Cornelius Lanczos, who made many contributions to relativity theory and numerical methods. This

function was originally proposed as a method of fitting the loggamma function, but has now been generalized to any fit of this type. The name Lanczos is pronounced "lan-sosh".

Fit_lanczos2(x(), y(), ndata, a(), nparams, rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to a non-standard Lanczos-type polynomial with nparams parameters returned in a(1..nparams): y=a(1)+a(2)/x+a(3)*x+a(4)/x^2+a(5)*x^2+a(6)/x^3+a(7)*x^3 +…
The programmer can easily evaluate this polynomial by using the function lanczos2(iterm, x) as follows:

```
for j=1 to nparams
ypi=ypi+a(j)*lanczos2(j, x0)
next
```

Fit_legendre(x(), y(), ndata, a(), nparams, rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to orthogonal Legendre polynomial with weights returned in a(1..nparams). Numerical values of the legendre polynomials are given by the function legendre(iterm, x), and the resulting fit evaluated using:

```
for j=1 to nparams
ypi=ypi+a(j)*legendre(j, x0)
next
```

Fit_polynomial(x(), y(), ndata, a(), nparams, rmserror, rsquared) fits via SVD least squares methods the data in x(1..ndata), y(1..ndata) to a polynomial with nparams parameters returned in
a(1..nparams): y=a(1)+a(2)*x+ a(3)*x^2+… +a(nparams)*x^(nparams-1). The chi-squared goodness of fit is returned in chisqr.

Fit_scan_polynomials(x(), y(), ndata, max_number_of_params, xp(), yp(), xmin, xmax, npoints) **as string** returns a string summary of the best polynomial fits for the data x(1..ndata), y(1..ndata) using between 2 and max_number_of_params free parameters. Note that this function includes all the polynomials from fit_trendline as well as fit_genpoly. In the latter case, the max number of parameters is held to 5 unless the user enters a negative value for max_number_of_params. If so, then genpoly can will allow a scan of up to six parameters. This option will add quite a bit of time to the search, however. After the search the best fit is selected based on the smallest value for rmserror*nparams and this function returns calculated values in xp(1..npoints), yp(1..npoints) for x from xmin to xmax.

Fit_trendline([itype], x(), y(), ndata, nparams, xp(), yp(), xmin, xmax, npoints) **as string** returns a string summary of the best fit trendline for the data x(1..ndata), y(1..ndata) using a maximum of nparam fit parameters. This function then returns calculated values in xp(1..npoints), yp(1..npoints) for x in the range xmin to xmax. This function is the best choice if the goal is to plot a trendline through a set of data points. It will also provide a quick analysis of which of the above fits provides the best fit to a set of data because the string that is returned not only defines the best fit but provides a list of the parameters. If itype is included, it selects the trendline fit to be: (1)polynomial, (2)lanczos, (3)lanczos2, (4)legendre, (5)chebyshev, (6)exponential.

Floor(x) **as double** next integer value smaller than x.
if x=4.952, floor(x)=4, ceil(x)=5
if x=-4.95, floor(x)=-5, ceil(x)=-4
if x=6, floor(x)=6, ceil(x)=6

For ... **Next** loops through the statements incrementing [or decrementing if "downto" is used] the loop counter i by i3 until it reaches i2. The step is optional, and if left out, the default is 1. It is important to remember that the step parameter is always positive, and the direction is set entirely by the presence or absence of "downto". Loops can also be constructing using real variables: e.g. For r = 0.1 to 0.7 step 0.1 is allowed and generates a loop with r=0.1, 0.2, 0.3, 0.4, 0.5, 0.6 and 0.7.

For i=i1 to [downto] i2 [step i3] ... next loops through the statements incrementing [or decrementing if "downto" is used] the loop counter i by i3 until it reaches i2. The step is optional, and if left out, the default is 1. It is important to remember that the step parameter is always positive, and the direction is set entirely by the presence or absence of "downto". Loops can also be constructing using real variables: e.g. For r = 0.1 to 0.7 step 0.1 is allowed and loops with r=0.1, 0.2, 0.3, 0.4, 0.5, 0.6 and 0.7.

Format(x, s0) **as string** returns a formatted string representation of the variable x where the characters in s0 produce the formatting described below:

#	placeholder that displays the digit if present (e.g. +###, ###, ###.00)
0	placeholder the displays either 0 or a digit if applicable (e.g. +0.000)
.	decimal location, when present, decimal is always shown
,	placeholder to indicate thousands separators using commas
%	multiplies the number by 100
(	displays an opening paren
)	displays a closing paren
+	forces display of sign, either + or – as required
-	displays a minus sign only when the number is negative
E, e	forces scientific notation (e.g. –0.000000000000E)
\char	displays the character that follows the backslash (e.g. \$-#, ###, ###.00)

Format(x, nwidth, ndecimal) **as string** returns a formatted string of width nwidth representing the number x. If ndecimal is 0, then x is shown as a rounded integer and the result is padded so that the number is right-justified with blanks. If ndecimal>0, a decimal point is shown and ndecimal digits to the right of the decimal point are shown. If the number will not fit, the function uses exponential (0.00…E) format. You can force exponential format by setting nwidth equal to the negative of the width desired.

Franck_condon_overlap(nu1, nu0, d10, rmass1, rmass0, v1, v0) **as double** returns the absolute value of the Franck-Condon overlap integral. Parameters are nu1 = wavenumber of state1 vibration; nu0 = wavenumber of state0 vibration; d10 = dimensionless shift of oscillators; rmass1 = reduced mass of the vibration nu1 (in amu); rmass0 = reduced mass of the vibration nu0 (in amu); v1 = vibrational quantum number in state 1; v0 = vibrational quantum number in state 0; If d10=0.0, the dimensionless shift is calculated approximately based on the other variables, and the calculated value is returned in d10.

FT_fold(c1(), s1(), c1lin(), s1lin(), npoints) transform the linearized coefficients in c1lin(1..npoints) and s1lin(1..npoints) into a folded pair in c1(1..npoints) and s1(1..npoints). When folded, the lowest frequencies are at 1 and npoints and the highest frequency is in the middle (npoints/2). The zeroth frequency components are transferred such that s1lin(0)=s1(0) and c1lin(0)=c1(0).

FT_linearize(c1(), s1(), c1lin(), s1lin(), npoints) transform the folded Fourier coefficients in c1(1..npoints) and s1(1..npoints) into a linearized pair in c1lin(1..npoints) and s1lin(1..npoints). When linearized, the frequency increases linearly from low subscript to high subscript. The zeroth frequency components are transferred such that s1(0)=s1lin(0) and c1(0)=c1lin(0).

Function defines the start of a function but must include parameters and type, for example, Function blim(x **as double**) **as double**. Close using **end function**.

|MS| **FWT2**(a2(), nbasis) replace the two-dimensional square matrix a2(1..nbasis, 1..nbasis) with its wavelet transform.

|MS| **FWT2_inverse**(a2(), nbasis) replace the two-dimensional square matrix a2(1..nbasis, 1..nbasis) with its inverse wavelet transform

Gamma(x) **as double** returns gamma(x), where x >=0. Note that x does not have to be an integer. This function is arprec savvy, and if the parameter is a string, gamma(s1) returns a string. The arprec version accepts complex string numbers, and returns complex string gamma(s1). Note that factorial(x-1)=gamma(x).

|MS| **Gauss_laguerre**(x1(), w1(), n0) returns the Gauss-Laguerre abscissas x1(1:n) and weights w1(1:n) for integration from 0 to infinity where n0 is used to assign the value of n, the number of pairs. The arrays x1() and w1() are returned automatically redimensioned to the appropriate values. If 3<n0<14 then n equals 2^n0 which provides for systematically larger values of 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 409, and 8192. If 13<n0<276 then n=n0. For compatibility with earlier versions, you can also set n0 equal to 512, 1024, 2048, 4096 or 8192. You can use the same values for –Infinity to 0 by simply multiplying the x() values by –1. Integration from –Infinity to Infinity is then simply the sum of –Infinity to 0 plus 0 to Infinity.

Gauss_legendre(x1, x2, x1(), w1(), n) generate the Gauss-Legendre abscissas x(1:n) and weights w(1:n) for n-point quadrature for an integral from x1 to x2.

GOTO string_label jump to label identified using an end colon (label:) Use of this function violates the rules of structured programming, but sometimes it is necessary.

Graphics_font(font_name, isize, Qitalics, Qbold) **as Boolean** sets the graphics font to the string font_name using isize to set the integer size and the booleans Qitalics and Qbold to set italics and bold options, respectively. Returns true if a valid font was found in the font folder.

Graphics_forecolor(color) sets the color for all graphics operations to follow.

Graphics_forecolor(ired, igreen, iblue) as above but uses specific RGB values.

Graphics_stroke_width(npixels) sets the stroke width in pixels.

Graphics_use_quickdraw. Sets via program command the preference "Use QuickDraw rather than Quartz graphics engine (OSX only)." This option is recommended when picture_write is running under MacOSX and transparency is involved. This command is ignored on Windows computers.

Harmonic_eigenvalue(wavnum, re, mu, v, byref ewav, byref r1, byref r2) **as double** returns the energy of the vth vibrational level of a Harmonic oscillator in eV as a function of the fundamental frequency, wavnum, in wavenumbers, the equilibrium bond length, re, in Angstroms and reduced mass mu in amu. Function returns the additional parameters ewav (level energy in wavenumbers), r1 (the minimum classical bond length (Å) in level v) and r2 (the maximum classical bond length (Å) in level v).

Harmonic_eigenvector(xr(), psi(), epot(), eshift, r1, r2, v, npoints) returns the eigenvectors of the quantum mechanical harmonic oscillator based on the properties assigned via a previous call to harmonic_eigenvalue(...). The wavefunction [psi(1..npoints)] in units of 1/Sqrt[Å] (assuming eshift=0.0) and classical potential function [emorse(1..npoints)] in eV are returned as a function of the r values in xr(1..npoints). If eshift is non-zero, the wavefunction is returned shifted in energy space by eshift so that the wavefunction can be displayed on top of the harmonic potential. Once shifted, the wavefunction can no longer be used to derive matrix elements of the harmonic oscillator.

Hermite(n, x) **as double**. returns the Hermite polynomial Hn(x) where n =0 to 200 and x is the real argument.

Hermite_function(n, x) **as double**. returns the Hermite Function representing the wavefunction of a harmonic oscillator of quantum number n and coordinate x.

$$\psi(x)=\sqrt{\frac{m\omega}{\pi\hbar}}\frac{1}{\sqrt{2^n\, n!\pi^{1/2}}}\exp\left(\frac{-x^2}{2}\right)Hermite(n,x)\quad n=0,1,2,....$$

$$\psi(x)=\sqrt{\frac{m\omega}{\pi\hbar}}\,Hermite\_function(n,x)$$

Hex(i) **as string** returns the string representing the hexadecimal value of integer i. bin(15) = "1111", oct(15) = "17", hex(15) = "F"
bin(127) = "1111111", oct(127) = "177", hex(127) = "7F"
If you want to enter an integer using a hexadecimal number use i = ivalue("&h7F")

HSV(hue, saturation, value) **as color** returns the color designated by the hue, saturation and value where each parameter is a double from 0 to 1.0. The values of hue from zero to one span the range from red(0.0)-orange(0.125)-yellow(0.17)-green(0.33)-aqua(0.5)-blue(0.66)-purple(0.8)-red(1.0). The saturation adjusts the amount of color versus gray scale with 1.0 providing full color. The value adjusts the brightness and goes from 0.0 (black) to 1.0 (brightest).

|MS| **http_bytes_received as integer** The number of bytes received from the last transfer.

|MS| **http_bytes_total as integer** The number of total bytes you should have received. This number is provided by the web site but is often unavailable or set to zero. Either way, when this number is zero you will have to examine the page_received data to verify that the downloaded page is correct. However, if it is nonzero and the bytes_received number is equal, you can have confidence that the download was successful.

|MS| **http_download_file**(string_file_address_as_url) **as Boolean** downloads the file using http or ftp protocols. The file is automatically placed in the folder called "downloads" which must be created and placed inside the Scriptor folder prior to using a program with this routine. The string_file_address_as_url should be of the form "http://some_web_address.abc/filename.xyz".

|MS| **http_error_code as integer** An error code that may help you diagnose why the download failed. For details see the RealBasic Language reference manual for the SocketCore Class. Some common error are: 1 Download file could not be

created (make sure there is a "downloads" folder in the same folder as Scriptor);
102 Lost connection in middle of transfer- also could mean a slow internet;
103 Bad or misspelled internet address, or the address is no longer valid;
108 Ran out of memory during transfer so the transfer was terminated.

|MS| **http_get_page**(string_url) **as Boolean** downloads the page from the internet using http protocols. The page address in the string_url should be of the form "http://some_web_address.abc". The progress is displayed in the basic_input field and if the download is successful, the page is placed into http_page_received. If a connection was made, the routine returns True, if a problem is discovered, it returns False. However, slow internet behavior will sometimes provide True but no page so the user needs to verify that there is a valid page by examining http_page_received.

|MS| **http_page_received as string** The page retrieved using the http_get_page routine.

|MS| **http_url_received as string** The actual url from which the http_page_received was retrieved. This may be different from the one you requested if the site automatically redirects you. This is updated after the page has been fully downloaded and can also be checked to verify a completed transfer.

If ... **Then** ... **Else** Basic form of the If conditional statement. Note that one-liners are accepted but one-liners do not end in end if. Multiline if statements must use the following format.

If ... **Then** ... **Elseif** ... **Else … End if** multiline conditionals must end with End if

|MS| **Imag**(s0 **as string**) **as string** returns the imaginary part of a complex arprec number.

|MS| **Imult**(s1 **as string**, s2 **as string**) **as string** returns the result of multiplying two arprec string integers.

Inherits is used in a class definition to designate the name of a class from which the current class is to inherit definitions (see discussion of classes above).

Input(s0 **as string**) **as string** Retrieves input from the user input line with an optional prompt (s0).

InStr([kstart,] source_string, find_string) **as integer** Find find_string in source_string starting at kstart (optional, if missing start at the first letter of source_string).

InStrB([kstart,] source_string, find_string) **as integer** as above but works on byte strings only.

|MS| **interface_serial_close**(icontrol) close serial software control = icontrol (1, 2 or 3). Whatever hardware port to which it was attached is now available to other systems.

|MS| **interface_serial_data_received**(icontrol, Qsave) **as string** returns the contents of the data received buffer of the software serial control = icontrol. If Qsave is true, the buffer is not cleared and the same data will be included in the next read.

|MS| **interface_serial_data_send**(icontrol)=string_data_to_send send the string data via serial control = icontrol(1, 2 or 3). The software serial control must have been initialized prior to sending data.

|MS| **interface_serial_initialize**(icontrol, iport, [ibaudrate, iparity, ibits, istopbits]) **as string** initializes a software data control (icontrol=1, 2 or 3) and attaches it to a hardware port (iport=1, 2, ..nports). The subroutine returns a string object which contains the information on the process and the default or assigned parameters. The first two parameters are required. The remaining four must either be absent or all four must be defined: ibaudrate = 0(300 bps), 1(600), 2(1200), 3(1800), 4(2400), 5(3600), 6(4800), 7(7200), 8(9600 bps=default), 9(14400), 10(19200), 11(28800), 12(38400), 13(57600), 14(115200), 15(230400);
iparity= 0 (none=default), 1 (odd) or 2 (even);
ibits = 0(5 data bits), 1(6), 2(7), 3(8 data bits = default);
istopbits = 0 (1 stop bit = default), 1 (1.5 stop bits), 2 (2 stop bits).

|MS| **interface_serial_list**(byref slist()) returns a list of serial ports available on the computer the list is loaded into slist() and the user should read the data by using ns=ubound(slist) to find how many ports are present

|MS **interface_serial_status_check**(icontrol, status) **as integer** based on what is passed in the string, status, this function returns the status of various lines in an integer. True/false results are returned as zero (false) or one (true). The string status can take the following values:
"baud" returns the baud rate in bps (300, 600, ..., 115200, 230400)
"bits" returns the number of bits (0=5, 1=6, 2=7, 3=8)
"cleartosend" returns 1 if cleartosend is true, 0 otherwise
"datacarrierdetect" returns 1 if true, 0 otherwise
"parity" returns parity set: 0=none, 1=odd, 2=even
"port" returns the port attached to control
"stop" returns the number of stop bits: 0=1, 1=1.5, 2=2 stop bits.

|MS| **interface_serial_status_set**(icontrol, status, ivalue) sets the status of serial control icontrol based on the following status strings and integer values: "baud" sets the baud rate --> ivalue = 0(300 bps), 1(600), 2(1200), 3(1800), 4(2400), 5(3600), 6(4800), 7(7200), 8(9600 bps=default), 9(14400), 10(19200), 11(28800), 12(38400), 13(57600), 14(115200), 15(230400). You can also set that baud rate by assigning to the variable, ivalue, a valid baud rate. If this option is used, ivalue must be equal to one of the rates listed above.
"bits" sets the number of bits [ivalue=0 (5), 1(6), 2(7), 3(8)]
"cleartosend" sets cleartosend to true (ivalue=1) or false (ivalue<>1)
"dataterminal" sets dataterminalready to true (ivalue=1) or false (ivalue<>1)
"parity" set parity to ivalue=0(none), 1(odd), 2(even)
"stop" sets the number of stop bits where ivalue=0(1 stop bit), 1(1.5), 2 (2)

iSession as integer returns the number of run sessions since the user manually cleared the output by pressing the “clear output” button. Useful for running a program with different user inputs or various options based on the number of times the program is run. If writing a program for others to use, remember to include instructions for the user to press “clear output” to reset the program.

iValue(s0) **as integer** returns the integer value of the string.

keyboard_keycode_decipher(keycode) returns the name of the key that is associated with the keycode (0 to 128). This function provides a cross-platform capability which works with the majority of keyboards. Run the program template_keycodes to verify that the current keyboard can be monitored.

keyboard_monitor_activity(ithkeypress, ascii_value, keycode, total_text_so_far) Provided Q_monitor_keyboard has been set to true, this method returns the ascii_value of the ithkeypress. If the ascii_value is outside of the range of printable characters the keypress is analyzed and the keycode associated with the pressed key is returned in keycode. The text that has been typed since Q_monitor_keyboard has been set to true is returned in total_text_so_far. The only value you should modify is ithkeypress as follows:
if ithkeypress is set equal to -1, reset all monitors and clear total_text_so_far.
if ithkeypress is set equal to 0 (which is the normal choice), return number of the last keypress in ithkeypress.
if ithkeypress is set >0, return that incident provided ithkeypress is equal to or less than the actual number of key presses.

keyboard_monitor_input(imilliseconds) **as string** an alternative method of monitoring input typing. The function returns all text that has been typed by the user since the Run was started. Only ascii printable text is returned. Use keyboard_monitor_activity() to monitor the entire keyboard.

key_down_ascii_value as integer the ascii value of the key that was last pressed. Each time a key is pressed on the keyboard, the key_down_ascii_value is set by the function check_and_clear_input to hold the ascii decimal equivalent of that key. The user should set this value to -1 after extracting the number to make sure each individual key press is treeated as a unique event. It is essential that you execute check_and_clear_input immediately prior to checking key_down_ascii_value as that is the function that traps and assigns this variable.

LaguerreL(n, alpha, x) **as double** returns the associated Laguerre polynomial for integer order n, where alpha and x can both be real.

LaguerreR(n, alpha, x) **as double** returns the associated Laguerre polynomial for floating point n, where alpha and x can both be real. The solution is obtained via interpolation and the results are not as accurate as LaguerreL(n, alpha, x).

Lanczos2(n, x) **as double** returns the numerical value of the nth Lanczos type 2 polynomial for argument x. The first possible value of n is 1, not zero.

Left(s0, n) **as string** returns the n characters from the left of s0.

LeftB(s0, n) **as string** returns the n bytes from the left of s0.

Legendre(n, x) **as double** returns the numerical value of the nth Legendre polynomial for argument x. The first possible value of n is 1, not zero, so the "zeroth" Legendre polynomial is referenced by n=1 [e.g. $P_n(x)$ =Legendre(n+1, x)] . The first ten polynomials are listed below.

n	
1	$P_0 = 1$
2	$P_1 = x$
3	$P_2 = \frac{1}{2}(3x^2 - 1)$
4	$P_3 = \frac{1}{2}(5x^3 - 3x)$
5	$P_4 = \frac{1}{8}(35x^4 - 30x^2 + 3)$
6	$P_5 = \frac{1}{8}(63x^5 - 70x^3 + 15x)$
7	$P_6 = \frac{1}{16}(231x^6 - 315x^4 + 105x^2 - 5)$
8	$P_7 = \frac{1}{16}(429x^7 - 693x^5 + 315x^3 - 35x)$
9	$P_8 = \frac{1}{128}(6435x^8 - 12012x^6 + 6930x^4 - 1260x^2 + 35)$
10	$P_9 = \frac{1}{128}(12155x^9 - 25740x^7 + 18018x^5 - 4620x^3 + 315x)$

Len(s0) **as integer** returns the number of characters in s0.

LenB(s0) **as integer** returns the number of bytes in s0. The only difference between using the byte string functions versus the normal versions is that the byte versions are much faster but only work with strings restricted to byte characters, such as the ASCII character strings in standard English. Languages such as Japanese and Chinese require multiple byte characters and require the use of the normal string operators.

|MS| **load_class**(filename, window) use with compiler directive //# (see above)

|MS| **load_method**(filename, window) use with compiler directive //# (see above)

|MS| **load_module**(filename, window) use with compiler directive //# (see above)

Log(x) **as double** returns the natural log = ln(x). In the event a different base is desired, use (Log(x))/Log(base) to calculate the value (see Log10(x) for log base 10).

Log10(x) **as double** returns $\log_{10}(x)$, the log base ten of x.

|MS| **LogGamma**(x) **as double** natural logarithm of the gamma function = LogΓ(x). Note that n! = Gamma(n+1) = exp(logGamma(n+1)). For example, 12! = 479, 001, 600, which equals Exp(Gamma(12 + 1)) = 479, 001, 600. Note that truncation error makes this method approximate for large n, thus 19! = 121, 645, 100, 408, 832, 000 but exp(loggamma(19 + 1)) = 121, 645, 100, 410, 059, 440

Loop [Until] ending keyword of a do [until] loop [until] construct

Lowercase(s0) **as string** converts the string s0 to all lower case

LTrim(s0) **as string** trims all leading (left) blanks from the string s0

|MS| **matdup**(a2()) **as double**(,) copy the matrix a2(n1, n2) into a new matrix b2(n1, n2). Usage example: b2=matdup(a2()) where b2 is a two-dimensional array. Note that the 0x0, 0x1, 1x0 elements are included in duplication. The recipient array will be redimmed to the same size as a2() in the process.

|MS| **matidn**(nsize) **as double**(,) initialize a two-dimensional identity matrix of dimension nsize by nsize with all elements equal to zero except diagonals which equal 1. Usage example: a2=matidn(10) where a2 is a two-dimensional array that is set to a size a2(0:10, 0:10). Note that the a2(0, 0) element is also created and set to one.

|MS| **matinv**(a2()) **as double**(,) invert the two-dimensional square matrix a2(). Usage example: b2=matinv(a2()) where b2 is a two-dimensional array. Note that the 0x0, 0x1, 1x0 elements of a2 are ignored. It is essential that the a2(,) matrix be square and be dimed or redimed to the proper size prior to calling matinv, as this method inverts the entire matrix.

|MS| **matmult**(a2(), b2()) **as double**(,) multiply a2(n1, n2) by b2(n2, n3) to create c2(n1, n3)Usage example: c2=matmult(a2(), b2()). Note that the 0x0, 0x1, 1x0 elements are ignored. The a2(,) and b2(,) matrices should be dimed or redimed to the correct sizes prior to calling matmult.

|MS| **matrand**(n1, n2) **as double**(,) initialize a two-dimensional matrix of dimension n1 x n2 with random elements from -1 to 1. Usage example: a2=matrand(10, 10) where a2 is a two-dimensional array. Note that the 0x0, 0x1, 1x0 elements of a2 are also created and randomized.

|MS| **mattran**(a2()) **as double**(,) transpose the matrix a2(n1, n2) into the matrix b2(n2, n1). Usage example: b2=mattran(a2()) where b2 is a two-dimensional array. Note that the 0x0, 0x1, 1x0 elements are included in transposition. It is essential that the a2(,) matrix be dimed or redimed to the exact size prior to calling mattran.

|MS| **matzero**(n1, n2) **as double**(,) initialize a two-dimensional matrix of dimension n1xn2 with all elements equal to zero. Usage example: a2=matzero(10, 10) where a2 is a two-dimensional array. Note that the a2(0, 0) element is also created and set to zero. The recipient array is redimmend to size n1xn2 in the process.

|MS| **matrix_complex_diagonalize**(ar(,), ai(,), vr(,), vi(,), er(), ei(), n [, smat, svec, sresult]) diagonalize the matrix a(1..n, 1..n) to produce eigenvectors v(1..n, 1..n) and eigenvalues e(1..n). The real and imaginary parts are in the _r and _i matrices or vectors, respectively. The variables ar(,), ai(,), vr(,), vi(,), er() and ei() are doubles and the matrices must be symmetric, Hermitian and dimensioned to nxn (ar, ai, vr, vi) or n (er, ei). The optional byref string variables return the input matrix in smat, the eigenvectors in svec and the result of the diagonalization in sresult. The latter should have very small off-diagonal elements and the eigenvalues along the diagonal.

|MS| **matrix_diagonalize**(h2(), v2(), e1(), nsize, nroots) diagonalize the square matrix h2(1..nsize, 1..nsize) to generate the lowest nroot solutions. Place the eigenvectors in v2(vector_number, root_number) and the eigenvalues in e1(1..nroots), e1(1) is the smallest or most negative eigenvalue.

|MS| **matrix_gauss_jordan**(a2(), b2(), nsize, ms) use Gauss Jordan elimination to solve the set of linear equations in the square matrix a2(1..nsize, 1..nsize) and replace it with its inverse and return the ms solution vectors (ms<=nsize) in the matrix b2(1..nsize, 1..ms). Less efficient than matrix_invert but provides solution vectors. If ms=1 (a single solution vector), use a one-dimensional array b1() and leave out the last parameter.

|MS| **matrix_invert**(a2(), det, nsize) use LU decomposition to replace the square matrix a2(1..nsize, 1..nsize) with its inverse and return the determinant in det. Provides the determinant but no solution vectors. More efficient than the gauss_jordan method.

|MS| **matrix_print**(a2(), nrows, ncols, ncols_per_set) **as string** return a formatted matrix a2(1..nrows, 1..ncols) with ncols_per_set at a time. You can then print out the matrix with the rows and columns aligned provided you have selected a proportional font using Set_text_style or by the menu.

|MS| **matrix_svd**(A(), V(), W(), m, n) given the matrix a(1..m, 1..n) compute its singular value decomposition. Upon return, the A() matrix is replaced by the U() matrix with the V() matrix and the W diagonal elements are returned in the parameters so designated. Thus, this subroutine solves the matrix problem shown below:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} = \begin{pmatrix} u_{11} & \cdots & u_{1n} \\ \vdots & \ddots & \vdots \\ u_{m1} & \cdots & u_{mn} \end{pmatrix} \bullet \begin{pmatrix} w_1 & & 0 \\ & \ddots & \\ 0 & & w_n \end{pmatrix} \bullet \begin{pmatrix} v_{11} & \cdots & v_{1n} \\ \vdots & \ddots & \vdots \\ v_{n1} & \cdots & v_{nn} \end{pmatrix}^T$$

where the T indicates the transpose of matrix v(1..n, 1..n). In our adaptation, the diagonal matrix w is returned as a one-dimensional array containing the diagonal elements. An alternative view of the above matrix equation is the following:

$$A_{ij} = \sum_{k=1}^{n} w_k U_{ik} V_{jk}$$

SVD takes the matrix A and decomposes it into a set of singular values with weighting factors w_k. One way of viewing SVD is a method of solving a set of m linear equations involving n unknowns. There is no more reliable method of doing a linear least squares fit (see demo_svd_fit.txt). SVD is also used extensively to extract component spectra from time resolved spectra (see demo_svd.txt). In many cases, and SVD will yield a set of weighting factors dominated by only a small number of weighting factors, w(), with the other weighting factors near to zero. Then it is important to zero out these values and use backsubstitution to extract vectors that are physically realistic and not contaminated by noise.

|MS| **matrix_svd_backsubstitute**(U(), V(), W(), m, n, b(), x()) This subroutine is called after the matrix_svd is run and the values of U(), V() and W() are exactly those returned in the A(), V(), W() matrix_svd parameter sequence. Note that you do not need to take the transpose of v(). What is needed first, however, is to go through the w() array and zero out those that have a magnitude that is significantly smaller than the largest weight. Because all calculations are done in double precision, you can carry out this zeroing process with confidence whenever w_k is less than 10^{-6} of w_{max}. Some trial and error is required for values above this cut-off. Then you need to supply a value for the b(1..m) vector to extract the x(1..n) vector, which is the desired result:

$$x = V \bullet \begin{pmatrix} w_1 & & 0 \\ & \ddots & \\ 0 & & w_n \end{pmatrix} \bullet U^T \bullet b$$

max(a1, a2, ..., aN) **as double** returns the variable that has the largest value (2 or more variables are possible).

Max(a(), n1, n2) **as double** returns the maximum value in the array in the range a(n1)…a(n2).

Microseconds as double returns the number of microseconds that have elapsed since your computer was turned on. This number will roll over if the computer has been on for a long time. This function is not very accurate on some Windows platforms.

Mid(s0, istart [, nlength]) **as string** return the substring from s0 starting at istart of length = nlength. If nlength is not included, the string from istart to the end is returned.

MidB(s0, istart [, nlength]) **as string** return the substring from s0 starting at istart of length nlength in bytes. If nlength is not included, the string from istart to the end is returned.

Midi_notes_off Turn off all midi notes that are on to produce silence.

Midi_play_note(ivoice, ipitch, ivelocity, ioption) play instrument ivoice at pitch ipitch (=60 at middle C) with volume ivelocity (0=off, max=127) and show the note being played on the keyboard if ioption=1

Midi_play_real_note(ivoice, realpitch, velocity, ioption) as above but realpitch can be any real number from 0.0 (off) to 126.9999. Allows playing of fractional pitches not conforming to well-tempered western tradition

Midi_set_polyphony(npoly) set the number of notes that can be played simultaneously from 1 to 64. Some computers can only handle 32 so experiment.

Min(a1, a2, …, aN) **as double** returns the minimum value within the set a1..aN. An alternative form of this function is shown below, which works with arrays.

Min(a(), n1, n2) **as double** returns the minimum value in the array in the range a(n1)…a(n2).

Minus(s1, s2) **as string** returns s1 - s2 using arbitrary precision string arithmetic

Mod operator for calculating the integer remainder (modulus) of two integers using the syntax iremainder = ia mod ib. This operator will accept real numbers, but rounds them to integers before carrying out the calculation.

Module module_name first line of a module- the last line is end module.

Molecules and Quantum Mechanics. The following molecular statements provide methods of plotting out a molecule as well as the electrostatic field calculated approximately based on the charges in Q(). All manipulations assume Cartesian coordinates [x(1..natoms), y(1..natoms) and z(1..natoms)] are in Ångstroms and that the charges [Q(1..natoms)], when included, are Mulliken charges. The dipole moment function is only rigorously defined for a neutral molecule. Dipole moments of charged molecules are, by convention, calculated by assuming center of mass coordinates, and some suggest meaningless.

Molecule_calculate_overlap(atom(), x(), y(), z(), natoms, S(,) [, gamma(,)]) **as string** return the overlap matrix for the molecule in S(nbasis, nbasis) and a printout in the returned string. The printout provides a description of the orbitals. The optional two-dimensional array, gamma(1..natoms, 1..natoms) returns the repulsion integrals.

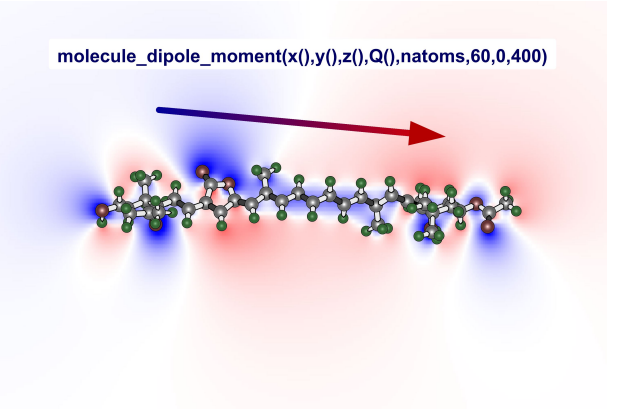
Molecule_charge as integer sets the charge on a molecule in preparation for a quantum mechanical calculation (CNDO or INDO).

Molecule_COM_coords(satom(), x(), y(), z(), natoms) translates the coordinates in x(1..natoms), y(1..natoms), z(1..natoms) to a center of mass representation.

Molecule_configuration_moment(i, j, mx, my, mz) **as double** returns the transition moment for the singly excited configuration j←i, where j is an unfilled molecular orbital and i is a filled molecular orbital in the ground state system. mx, my and mx return (byref) the transition moment components in the x, y and z directions in Debyes for the j←i configuration. [note that oscillator strength is given by f = 0.0037927 *dE(eV)*Mtot(D)^2, where dE is in electron volts and Mtot is the transition moment in Debye. Mtot = sqrt(mx^2 + my^2 + mz^2)]

molecule_correlate_ground_state(numclosed, numopen [, ddom], byref ecorrelation) **as string** returns an analysis of the correlation of the ground state based on full single and double configuration interaction within a basis set of numclosed by numopen molecular orbitals. The ByRef parameter ecorrelation returns the total CISD ground state correlation energy in Hartrees. The optional parameter ddom assigns a multiplier for double-double off-diagonal matrix elements. Values less than 1.0 enhance ground state correlation by transferring correlation from the excited state manifold into the ground state manifold. 0.0 sets the number to unity (default).

molecule_dipole_moment(x(), y(), z(), Q(), natoms, arrow_size, xshift, yshift) **as double**
returns the molecule dipole moment in Debyes based on the coordinates (x(1..natoms), y(1..natoms), z(1..natoms)) in Ångstroms and the atomic charges, Q(1..natoms). This function will also print out an arrow showing the dipole moment of size = arrow_size on the COM of the molecule. Set arrow_size=0 to only calculate the value. Shift the arrow by entering non-zero values of xshift (positive shifts right) and yshift (positive numbers shift up). Note that this method does not include the dipole moment associated with hybridization (for example, the sp contribution). To include that information, it is necessary to execute molecule_run_scf(..) and use molecule_scf_dipole_moment(…) to access the resulting dipole moment components.



molecule_draw(satom(), x(), y(), z(), natoms, iatomsize, Qshow_numbers) draw the molecule defined by the atom list satom(1..natoms)[which lists the atom labels as in C, N, O, etc.] and with cartesian coordinates x(1..natoms), y(1..natoms) and z(1..natoms) using a heavy atom size of iatomsize. The heavy atoms are numbered if Qshow_numbers=true. The molecule fills the canvas unless the range is defined by molecule_plot_space(x1, x2, y1, y2).

molecule_draw_simple(atom(), x(), y(), z(), natoms, brightness, bondwidth, Qdraw_hydrogens) draw the molecule defined by the atom list satom(1..natoms)[which lists the atom labels as in C, N, O, etc.] and with cartesian coordinates x(1..natoms), y(1..natoms) and z(1..natoms) using the integer bondwidth to set both single and double bond thickness. The molecule fills the canvas unless the range is defined by molecule_plot_space(x1, x2, y1, y2). Hydrogens are included if Qdraw_hydrogens is true. The double brightness varies from 0 (darkest) to 1 (all white).

molecule_draw_vector(vx, vy, vz, arrow_length[, arrow_width], ioption, xshift, yshift) draw the vector vx, vy, vz relative to the center of the buffer, with size arrow_size. Color is black (ioption=0), red to blue (ioption=1) or blue to red (ioption=2). Shift the arrow by entering non-zero values of xshift and yshift. If the arrow_width parameter is not included, the width is automatically calculated from the length.

molecule_external_charges(x(), y(), z(), Q(), ncharges) assigns the cartesian coordinates and values of external charges. Each ith charge has an assigned charge value, Q(i), in electron units, and cartesian coordinates x(i), y(i) and z(i), where i=1...ncharges. These values influence subsequent SCF calculations and field plots.

molecule_g(i, j, k, l) Returns the general two electron integral <ij|kl> in Hartrees based on the quantum chemist's notation. In this notation, coulomb integrals are <ii|kk> [=g(i, i, k, k)] and exchange integrals are <ik|ik> [=g(i, k, i, k)]. This function will also return eigenvectors v(iao, jmo) = g(iao, jmo, 0, 0) by setting the last two parameters to zero. Eigenvalues (in Hartrees) are returned by setting the first three parameters to zero, e.g. eigenvalues(jmo) = g(0, 0, 0, jmo). All values are those from the most recent SCF calculation. If no calculation has been performed, 0.0 is returned along with an error message. This function provides integrals, vectors and values relevant for configuration interaction calculations

molecule_generate_gaussian_gjf(satom(), x(), y(), z(), natoms) **as string** returns a gaussian input file based on the atom list in satom(1..natoms) and the cartesian coordinates in x(), y() and z(). Note that gaussian input files use an extension of .gjf or .com, and the gjf extension is used to differentiate between a gaussian input file and a gif picture.

molecule_internal_coord(satom(), x(), y(), z(), natoms, R43, A432, T4321, n4, n3, n2, n1) **as string** convert the cartesian coordinates in x(), y(), z() into internal coordinates involving byref bond length R43, byref bond angle A432, and byref dihedral angle T4321 for the set of atoms designated by n4, n3, n2 and n1. If n1=0, then only R43 and A432 are returned. If n1=0 and n2=0 then only R43 is returned. A string representation of the selected internal coordinates is also returned by the function for printout.

molecule_internal_coordinates(satom(), x(), y(), z(), natoms) **as string** returns a list of the internal coordinates of the molecule based on the x(1..natoms), y(1..natoms) and z(1..natoms) cartesian coordinates. This method provides additional redundancy in returning the internal coordinates relative to the minimal list generated by molecule_xyz_to_internal().

molecule_internal_to_XYZ(satom(), x(), y(), z(), natoms, r43(), a432(), t4321(), n4(), n3(), n2(), n1()) **as string** convert the internal coordinates involving bond lengths (r43), bond angles (a432), dihedral angles (t4321) and n4(), n3(), n2(), n1() atom pointers to cartesian coordinates in X(), Y() and Z(). The string that is returned as a formatted list of the cartesian coordinates.

molecule_name as string assigns the name of the molecule prior to running a CNDO, INDO or other molecular calculation with printout.

molecule_merge_hydrogen_charges(atom(), x(), y(), z(), Q(), natoms) Merges the hydrogen charges into the heavy atom charges of those atoms to which the hydrogens are bonded. The new charges are returned in Q(1..natoms).

molecule_multiplicity assigns as integer sets both the method and the multiplicity to be used by subsequent SCF calculations. If set to 0, a standard closed shell calculation is carried out on the closed shell ground state singlet state. If set to 1 or greater an open shell calculations is carried out for a multiplicity state equal to molecule_multiplicity. Note that a closed shell ground state (molecule_multiplicity=0) and an open shell singlet state (molecule_multiplicity=1) should generate identical observables, except for differences due to truncation error.

molecule_one_electron_hamiltonian(imo) **as double** returns the one-electron Hamiltonian in Hartrees for an electron in molecular orbital imo based on the previous SCF calculation. The electronic energy of a ground state molecule can be calculated as etotal = 2*Sum[H1] + Sum*Sum[2*J - K] where the sums are over all the occupied orbitals, and H1 is the one electron Hamiltonian over molecule orbitals. This method returns the energy of an electron in the *mo* imo associated with the interaction of that electron with the core nuclei. Note that this integral is over molecular orbitals, not atomic orbitals.

molecule_opt_scf(atom(), x(), y(), z(), [Q(),] natoms, maxiter, dx, dy, dz) **as double** returns the total energy after carrying out a single iteration of CNDO/2 (molecule_set_scf_method=2) or INDO (molecule_set_scf_method=1) optimization based on changing the x coordinates by ±dx Angstroms, the y coordinates by ±dy and the z coordinates by ±dz. If any delta is zero, that coordinate is not optimized (i.e. make dz=0.0 if the molecule is rigorously in the plane). This is a slow routine and should not be used for molecules with more than about 12 atoms unless run on a very fast computer, or Q() is used to select only some of the coordinates to optimize. It is appropriate to use smaller values for maxiter (10-20) during course optimizations (dx>0.01) but increase to 100 near the minimum. The optional Boolean array, Q(1..natoms), allows the user to specify which atoms are to be optimized. If Q(i) is true, the ith atom is optimized, otherwise it is fixed at its current position. Set the molecule name by assigning molecule_name. Set the molecule charge by assigning molecule_charge. INDO optimizations can be carried out on hydrogen through fluorine. CNDO/2 geometry optimizations can be carried out on molecules containing the following atoms:
H, (1s orbital)
Li, Be, B, C, N, O , F (2s, 2px, 2py, 2pz)
Na, Mg, Al, Si, P, S, Cl (3s, 3px, 3py, 3pz, $3dz^2$, 3dxz, 3dyz, 3dx^y, 3dxy)

molecule_opt_scf_fast(atom(), x(), y(), z(), natoms, maxiter, eiter(), [[dr] or [dx, dy, dz]]) **as string** returns the optimized coordinates from a fast analytical SCF optimization which is not as accurate as molecule_opt_scf(...) but is significantly faster for large molecules. Set dr (or dx, dy and dz) to the first iteration atom shift. The atom shift values will be decreased as the method finds a variational local minimum, where all atoms are optimized in full cartesian space, or subspace based on dx, dy and dz. For example, if only dx and dy are assigned (dz=0), then the optimization is limited to the x, y plane. Any combination is allowed. The molecule will be rotated to optimize convergence, but the rotation will be constrained by the choice of dx, dy and dz. The final coordinates are returned in x(), y() and z() and the energies for each iteration placed in eiter(1..maxiter). Set maxiter to values from 20 to 120, where lower numbers are faster, but yield less optimal coordinates. The string that is returned gives a list of the iterations, the shifts and the energy at each iteration.

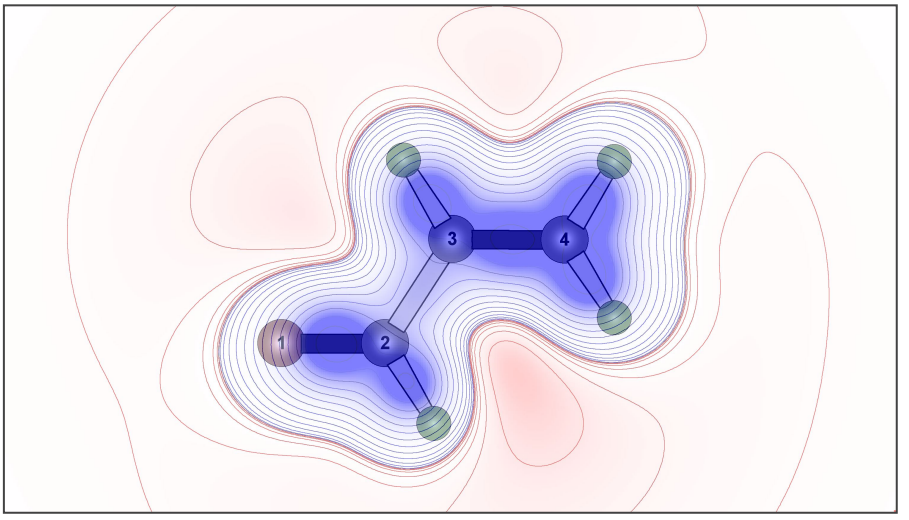
molecule_orient(n1, n2, n3, x(), y(), z(), natoms) orient the cartesian coordinates x(1..natoms), y(1..natoms) and z(1..natoms) by moving atom n1 to the origin, atom n2 along the x axis, and atom n3 into the XY plane with positive y value. If n3 is negative, perform a 180 rotation around x after orientation. If n2 is negative, perform a 180 rotation around y after orientation.

molecule_plot_bonding(x(), y(), z(), eab(,), natoms, nx, ny, ioption, color_contrast)

The resolution of the analysis is defined by nx (horizontal points) and ny (vertical points). Ioption=1, generate a plot2D representation of the bonding where red is antibonding or ionic repulsion and blue is bonding.
abs(Ioption)=2, plot both a 2D (as above) and a contour plot with 30 contours, Abs(ioption)=3, plot just a contour plot. color_contrast sets the contrast of the plot2D coloring, where larger values enhance color density. bond energy colors are forced below the contour lines. Following are examples:
// charge colors without contours:
molecule_plot_bonding(x(), y(), z(), Q(), natoms, 300, 100, 1, 4)
// charge colors with contours (30 contours assumed):
molecule_plot_bonding(x(), y(), z(), Q(), natoms, 300, 100, 2, 4)
// 30 contour lines without fill colors:
molecule_plot_bonding(x(), y(), z(), Q(), natoms, 300, 100, 30, 0) .

The following program section plots out the bonding analysis of propenal (acrolein) using the function pictures_blend_to_buffer to superimpose the molecular diagram and the bonding plot.

```
sout= molecule_run_cndo_energy_analysis(atom(),x(),y(),z(),natoms,100,e0,Q(),eab())
print(sout) // print out analysis
buffer_create(0,3000,1700)
molecule_plot_space(0,4,-2,4)
molecule_plot_bonding(x(),y(),z(),eab(),natoms,400,300,2,6)
buffer_copy_to_picture(1)
buffer_create(0,3000,1700)
molecule_plot_space(0,4,-2,4)
molecule_draw(atom(),x(),y(),z(),natoms,50,true)
buffer_copy_to_picture(2)
call pictures_blend_to_buffer(2,.5,1,.5)
```



molecule_plot_efield((x(), y(), z(), Q(), natoms, nx, ny, ioption, color_contrast)

plot the electrostatic field of the molecule defined by the coordinates (x(1..natoms), y(1..natoms), z(1..natoms)) in Angstroms and the atomic charges, Q(1..natoms).

The resolution of the analysis is defined by nx (horizontal points) and ny (vertical points).

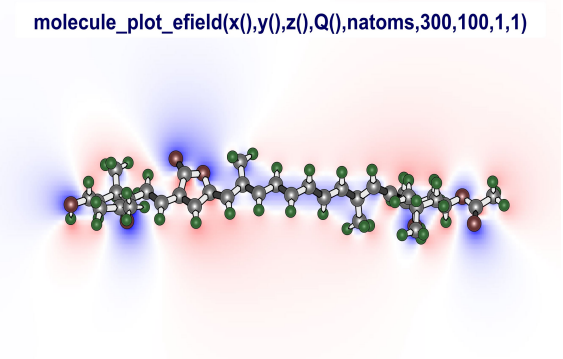
if abs(Ioption)=1, generate a plot2D representation of the field where red is excess positive charge and blue is excess negative charge,

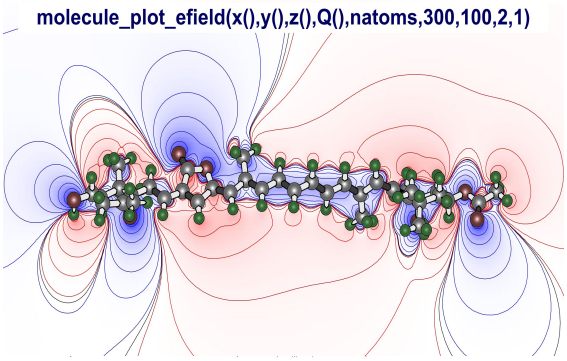
if abs(Ioption)=2, plot both a 2D (as above) and a contour plot with 30 contours,

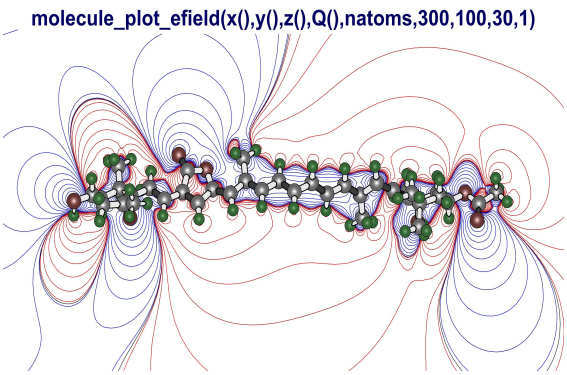
if abs(ioption)>2, plot just a contour plot where abs(ioption) assigns the number of contours.

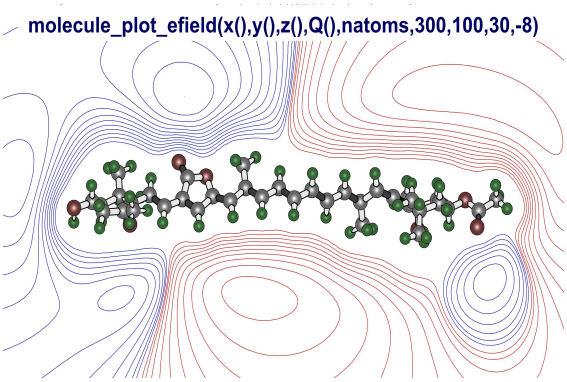
If ioption is negative, then field contours are calculated for atoms in the XY plane only. abs(color_contrast) gives the contrast of the plot2D coloring, where larger values enhance color density.

If color_contrast is negative, the contour lines are plotted to approximate the electrostatic field outside the molecule.









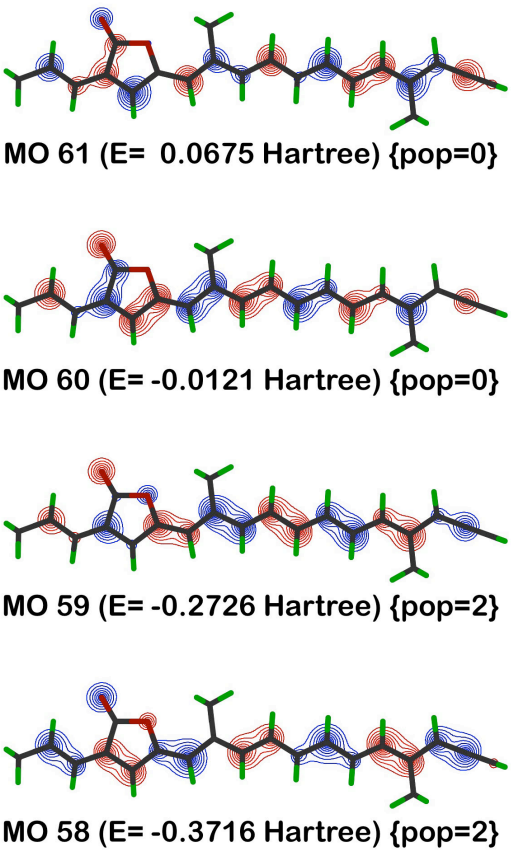
molecule_plot_eigenvector(imo, kspin, nx0, ny0, ncontours, color_fill_intensity)

Generate a 2D plot of molecular orbital number imo from the last closed shell calculation (kspin=0) or the last open shell calculation in which case kspin selects the alpha (kspin=1) or beta (kspin=2) manifold. The resolution of the plot in pixels is given by nx0 and ny0. If nx0=ny0=0, default values of 300 and 150 are used. The number of contours is set by using ncountours, and the depth of color in the fill is assigned by setting color_fill_intensity. The resulting plots are approximate and rendered in the XY plane. The d orbitals are all rendered as 3s orbitals. Note that pz orbitals are not visible. Thus, to see the pi system, either rotate the molecule rotate the molecule to transform the pz orbitals into px or py orbitals, or use molecule_plot_pz_vector() A nice plot can be obtained by using the following parameters:

molecule_plot_eigenvector(imo, kspin, 300, 150, 40, 2.0)

molecule_plot_pz_vector(imo, kspin, nx0, ny0, ncontours, color_fill_intensity) Generate a 2D plot of molecular orbital number imo from the last closed shell calculation (kspin=0) or the last open shell calculation in which case kspin selects the alpha (kspin=1) or beta (kspin=2) manifold. The resolution of the plot in pixels is given by nx0 and ny0. If nx0=ny0=0, default values of 300 and 150 are used. The number of contours is set by using ncountours, and the depth of color in the fill is assigned by setting color_fill_intensity. The resulting plots are approximate and rendered in the XY plane. This method will only show orbitals which have pz components, and will plot them by showing a slice through the pz lobes. An example is shown at right. A nice plot can be obtained by using the following parameters:

molecule_plot_pz_vector(imo, kspin, 300, 150, 30, 2.0)



molecule_plot_space(x1, x2, y1, y2) assign the dimensions of the canvas in units of the molecule coordinates such that the horizontal axis (x axis) goes from x1 (left) to x2 (right) and the vertical axis goes from y1 (bottom) to y2 (top). It is important to use this method to define the plot region prior to calling molecule_plot_efield(). To revert to autodraw, execute molecule_plot_space(0, 0, 0, 0).

molecule_rotate(x(), y(), z(), natoms, Tx, Ty, Tz) rotates the coordinates in x(1..natoms), y(1..natoms), z(1..natoms) returning values for rotation by Tx, Ty and Tz rotations in degrees. The angles refer to rotation around axes, that is, Tx=degrees of rotation around the x axis.

molecule_run_bond_energy_analysis(atom(), x(), y(), z(), natoms, maxiter, energy, Q(), eab(,)) **as string** Run a standard SCF calculation and return the atomic charges in Q(1..natoms) and the bond energy analysis in eab(1..natoms, 1..natoms). All other features and requirements as in molecule_run_scf() If a plot of the bond analysis is desired, use molecule_plot_bonding().

molecule_run_psdci(numpiclosed, numpiopen, numsigmaclosed, numsigmaopen, piatoms(), ioption [, ddom]) **as string** Run a configuration interaction calculation on the molecule most recently run via the molecule_run_scf() method. The SCF results are automatically transferred. The piatoms are listed in the integer array piatoms(1..natoms) by setting piatoms(i)=0 (not pi atom) or piatoms(i)=1 (pi atom). numpiclosed = the number of the highest energy closed shell pi molecular orbitals to include in the CISD, and numpiopen = the number of the lowest energy unfilled pi *mo*s included in the CISD, numsigmaclosed = the number of the highest energy closed shell sigma mos to include in the CIS, and numsigmaopen = the number of the lowest energy unfilled sigma orbitals included in the CIS, ioption=1 (CIS only), 2=(CISD without spin coupled triplets), 3=(CISD including spin-coupled triplet-triplet double excitations). Add 10 to ioption to request additional output (the LSD determines the CI option). The optional parameter ddom assigns the off-diagonal double-double multiplier. This value is normally 1.0, but can be made smaller (for aromatic or small conjugated systems) or larger (for large conjugated systems) to simulate coupled-cluster methods including triple and quadruple CI. The variables psdci_max_singles and psdci_max_doubles can be used to limit the number of singles and doubles, respectively. Also, if psdci_max_doubles is set to a negative number, the max number of doubles is automatically set equal to the number of singles for a full CIS expansion.

molecule_run_scf(atom(), x(), y(), z(), natoms, maxiter [, energy, Q()] [, fock(,), puv(,)]) **as string** calculate the SCF self-consistent-field for molecules, where atom(1..natoms) gives the atom label (C, H, N, O etc.) x(1..natoms), y(1..natoms) and z(1..natoms) are the cartesian coordinates in Angstroms, maxiter sets the maximum allowed number of SCF iterations, and energy and Q(1, , natoms) are optional (either both or neither). If the energy and Q() parameters are left out, or if natoms is multiplied by -1, this function returns a string which represents the complete output from the SCF calculation. If energy and Q() are included, the total energy of the molecule in Hartrees is returned in energy and the atom charges are returned in Q(1..natoms).

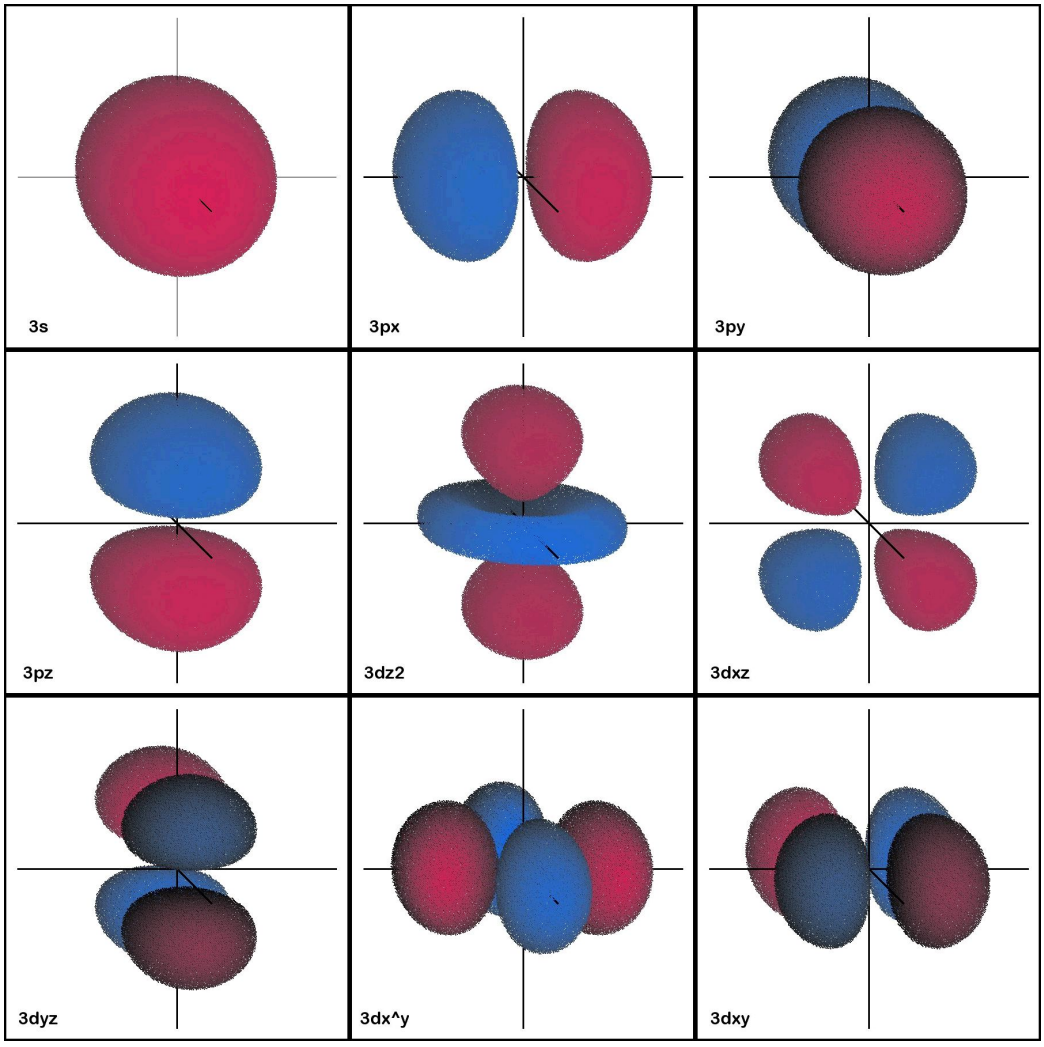
Set the molecule name by assigning molecule_name. Set the molecule charge by assigning molecule_charge. Set **molecule_set_scf_method**=1 to use INDO, =2 to use CNDO/2, and =0 to select INDO where possible and CNDO/2 otherwise. Set **Q_afaos_excited_singlet_state** to true to calculate the electronic properties of the first excited singlet state by using Averaged Field Approximate MO Theory. Set **molecule_multiplicity** to a value larger than 0 to use open shell methods. The default for this parameter is 0, which selects a closed-shell singlet state calculation. Setting molecule_multiplicity to 1 (singlet) will force open shell methods to calculate the lowest singlet state (use for testing, as the results should be identical to a closed shell singlet). Set **Q_use_external_parameters** to true to read the parameterization from the spreadsheet. To load the standard parameters into the spreadsheet, execute **atom_properties_cndo**(0, "S", 0, S1) and then modify by hand. One can then store the modified parameters as a data set for future use. The program only reads the parameters in the first 7 columns. The values of U, I and A are ignored by the SCF calculation. Occasionally, the SCF method fails to converge due to charge oscillation during the variational process. Set **Q_damp_scf** to true prior to running the calculations, and this will average the current with the previous Fock matrix prior to diagonalization. Also, SCF failures can occur for other reasons. When they fail, try using **molecule_com_coords**(), rotating the coordinates and/or switching the order of the atoms. Each SCF calculation will populate the global string atomic_orbital_list with a list of the atomic orbitals in the order generated by the atom list. If a string array of these assignments is desired, use a statement like atomic_orb() =string_split(**atomic_orbital_list**, ", ") , where atomic_orb() is a previously declared string array.

The CNDO/2 and INDO calculations available via the **molecule_run_scf**, **molecule_opt_scf** and **molecule_opt_scf_fast** statements provide INDO calculations for the first row elements and CNDO/2 calculations for the first two rows of atoms. The parameterization is based on that proposed by Pople and coworkers [J. Chem. Phys. 43, S129-S151 (1965); 44, 3289-3296 (1966); 45, 2026-2033 (1967)]. These methods include d orbitals for the second row elements. Averaged Field Approximate MO

methods are also available to calculate the properties of the first excited singlet state via variational methods [J. Am. Chem. Soc. 95, 8241-8249 (1973)].

H, (1s orbital)
Li, Be, B, C, N, O , F (2s, 2px, 2py, 2pz)
Na, Mg, Al, Si, P, S, Cl (3s, 3px, 3py, 3pz, $3dz^2$, 3dxz, 3dyz, 3dx^y, 3dxy)

Because there are various options available for constructing d orbitals, the figure below is provided showing the full orbital basis set for chlorine. Note that while virtually all semiempirical calculations use the same px, py and pz orbitals, there are various options for d orbitals. The one used in MathScriptor differs from the original basis sets proposed by Pople and coworkers [J. Chem. Phys. **43**, S129-S151 (1965); **44**, 3289-3296 (1966)].



Scriptor allows CNDO/2 and INDO calculations to be carried out using altered parameterization by setting the Q_use_external_parameters to true and placing the revised parameters in the appropriate location in the spreadsheet following the format shown in the figure below:

atom	Z	Slater	gamma	-beta0	(Is+As)/2	(Ip+Ap)/2	(Id+Ad)/2	Uss	Upp	Udd	Iss	Ipp	Idd	Ass	App	Add
H	1	1.200	20.408547	9.00000	7.17610			-17.38037			17.38037			-3.02817		
He	2	1.700	28.912108													
Li	1	0.650	6.425503	9.00000	3.10550	1.25800		-6.31825	-4.47075		6.31825	4.47075		-0.10725	-1.95475	
Be	2	0.975	9.638255	13.00000	5.94557	2.56300		-20.40295	-17.02038		10.76470	7.38213		1.12644	-2.25613	
B	3	1.300	12.851007	17.00000	9.59407	4.00100		-41.72159	-36.12852		16.01957	10.42650		3.16857	-2.42450	
C	4	1.625	16.063759	21.00000	14.05100	5.57200		-70.27416	-61.79516		22.08288	13.60388		6.01912	-2.45988	
N	5	1.950	19.276510	25.00000	19.31637	7.27500		-106.06067	-94.01930		28.95463	16.91326		9.67811	-2.36326	
O	6	2.275	22.489262	31.00000	25.39017	9.11100		-149.08111	-132.80194		36.63480	20.35563		14.14554	-2.13363	
F	7	2.600	25.702014	39.00000	32.27240	11.08000		-199.33549	-178.14309		45.12341	23.93101		19.42139	-1.77101	
Ne	8	2.925	28.914766													
Na	1	0.733	5.151150	7.72030	2.80400	1.30200	0.15000	-5.37958	-3.87758	-2.72558	5.37958	3.87758	2.72558	0.22842	-1.27358	-2.42558
Mg	2	0.950	6.673081	9.44710	5.12540	2.05160	0.16195	-15.13502	-12.06122	-10.17157	8.46194	5.38814	3.49849	1.78886	-1.28494	-3.17459
Al	3	1.067	7.492582	11.30110	7.77060	2.99510	0.22425	-26.50206	-21.72656	-18.95571	11.51689	6.74139	3.97054	4.02431	-0.75119	-3.52204
Si	4	1.383	9.716943	13.06500	10.03270	4.13250	0.33700	-44.04200	-38.14180	-34.34630	14.89117	8.99097	5.19547	5.17423	-0.72597	-4.52147
P	5	1.600	11.238873	15.07000	14.03270	5.46380	0.50000	-64.60763	-56.03873	-51.07493	19.65214	11.08324	6.11944	8.41326	-0.15564	-5.11944
S	6	1.817	12.760804	18.15000	17.64960	6.98900	0.71325	-87.83402	-77.17342	-70.89767	24.03000	13.36940	7.09365	11.26920	0.60860	-5.66715
Cl	7	2.033	14.282735	22.33000	21.59060	8.70810	0.97695	-114.42838	-101.54588	-93.81473	28.73197	15.84947	8.11832	14.44923	1.56673	-6.16442
Ar	8	2.250	15.804666													

These values are invariant and should not be changed

Data in this block are read by Scriptor and replace the internal CNDO/2 or INDO parameterization when the user sets Q_external_parameter to True prior to running a CNDO or INDO calculation. If any of the one-center repulsion (gamma) integrals are replaced, all two-center repulsion integrals will be calculated using the Ohno repulsion formula (all values in a.u.):

$$\gamma_{AB} = \left[R_{AB}^2 + \frac{1}{4}\left(\frac{1}{\gamma_A} + \frac{1}{\gamma_B} \right)^2 \right]^{-1/2}$$

Data in this block are calculated from the integrals in the block at left by using the following formulas. Changing data in this block has no impact on the CNDO or INDO calculations, nor are these results updated when changes are made in the data in the block at left. These values are for reference and reflect the standard CNDO/2 parameterization.

$$U_{\mu\mu} = -\frac{1}{2}\left(I_\mu + A_\mu\right) - \left(Z_A - 1/2\right)$$

$$I_\mu = -U_{\mu\mu} - \left(Z_A - 1\right)\gamma_{AA}$$

$$A_\mu = -U_{\mu\mu} - Z_A\gamma_{AA}$$

The user should only manipulate values in columns 3-8 (but do not change the headers). One can start by loading the default CNDO/2 parameters into the spreadsheet by executing call atom_properties_cndo(1, "S", 1, s1), which generates the above spreadsheet. Modify the variables either by hand or by using a program, making sure that the changes are made in the correct location in the table. Save the spreadsheet for use in future calculations if desired. Note that if any of the one center repulsion integrals are changed, all two-center repulsion integrals will be calculated by using the Ohno repulsion formula rather than the standard method, as the standard method is based on s-orbital repulsion integrals. If the one-center values are changed, the use of the s-orbital two-center repulsion values no longer makes sense.

Molecule_scf_dipole_moment(icomponent) **as double** returns the SCF dipole moment in Debyes from the most recent SCF calculation. This result differs from the results returned by molecule_dipole_moment(...) because the scf dipole moment includes the sp and sd polarization terms which are only available from an SCF calculation. The value of icomponent determines the component returned [0 or 4 = total, 1=mu(x), 2=mu(y), 3=mu(z)] .

Molecule_scf_eigenvectors(vectors(,), eigenvalues(), pop(), atm(), iatom_number(), orbital_type(), norbs, kspin) returns the eigenvectors in vectors(1..norbs, 1..nmos), the eigenvalues in eigenvalues(1..nmos), the MO population in pop(1..nmos), the atom number contributing the atomic orbital in iatom_number(1..norbs), the atomic orbital type in orbital_type(1..norbs), the number of orbitals in norbs. All of these values are returned byreference with all arrays redimensioned accordingly. The only input parameter is kspin which if 0 requests the closed shell value, 1 requests the alpha open shell values and 2 requests the beta open shell values. The appropriate SCF calculation must be run prior to calling this method.

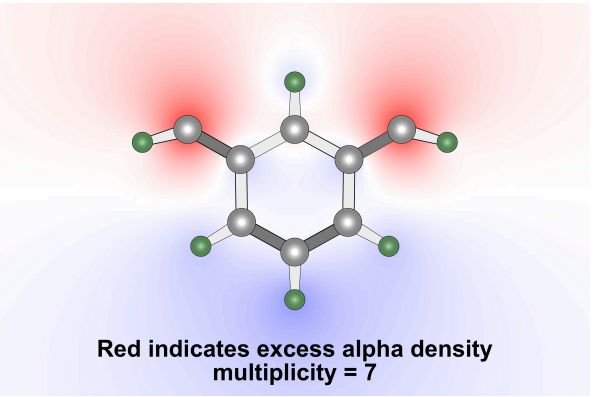
Molecule_set_electron_mobility(pi_mobility, sigma_mobility) This method allows adjustment of pi electron mobility and sigma electron mobility. The standard values for both are 1.0. Del Bene recommends values of 0.585 (pi) and 1.000 (sigma) to improve the calculation of excitation energies. Setting both parameters to 0.0 or 1.0 turns off electron mobility adjustment.

Molecule_set_repulsion_method = 0, 1, 2 or 3 If set to 0, standard CNDO/2 and INDO repulsion methods are used unless the user has set Q_use_external_parameters to true, which then forces the use of the Ohno formula. If molecule_set_repulsion_method > 0, then this parameter selects Ohno (=1), Mataga (=2) or Weiss (=3) integrals. The Weiss integrals should be used when simulating Zindo methods.

Molecule_set_scf_method = 0, 1 or 2 If set to 1, SCF uses INDO methods. If set to 2, SCF uses CNDO/2 methods. A default value of 0 uses INDO when possible but switches to CNDO/2 if second row atoms are included.

Molecule_spin_densities as array This function returns the atomic spin densities from the last SCF CNDO/2 or INDO open shell calculation. Use the statement Qspin()=molecule_spin_densities to place the atomic spin densities (alpha spin density minus beta spin density) into the double array Qspin(1..natoms). These results can be plotted out by using the spin densities in place of the charges in molecule_plot_efield(…) as shown in the example below:

```
molecule_plot_efield(x(), y(), z(), Qspin(), natoms, 300, 100, 1, 2)
call graphics_font("Arial", 120, false, true)
graphics_forecolor(rgb(0, 0, 0))
draw_string("Red indicates excess alpha density", 1500, 1800)
draw_string("multiplicity = "+str(molecule_multiplicity), 1500, 1920)
```



Molecule_XYZ_to_compact(atom(), x(), y(), z(), natoms, ndigits) **as string** convert the Cartesian coordinates in x(), y(), z() to a compact notation that uses array statements. These statements not only assign the coordinates but redimension the arrays. This statement is an excellent way to generate coordinates in a compact form that can be inserted into a program.

Molecule_XYZ_to_internal(atom(), x(), y(), z(), natoms, r43(), a432(), t4321(), n4(), n3(), n2(), n1()) **as string** convert the cartesian coordinates in x(), y(), z() into internal coordinates involving bond lengths (r43), bond angles (a432), dihedral angles (t4321). If n4(), n3(), n2(), n1() are filled with the atoms numbers these values are used to determine the internal coordinates. If n4(1)=n4(2)=n4(3)=0, then these are filled with the atom numbers associated with the *abcd* assignments. The string that is returned is a formatted list of the internal coordinates, which is convenient for printing using a non-proportional font.

Morse_eigenvalue(deev, wavnum, re, mu, v, byref ewav, byref vmax, byref r1, byref r2) **as double** returns the energy of the vth vibrational level of a Morse oscillator with dissociation energy, deev, in eV, fundamental frequency, wavnum, in wavenumbers, equilibrium bond length, re, in Angstroms and reduced mass mu in amu. Function returns the additional parameters ewav (level energy in wavenumbers), vmax (maximum value of v, higher levels occupy the continuum), r1 (the minimum classical bond length (Å) in level v) and r2 (the maximum classical bond length (Å) in level v).

Morse_eigenvector(xr(), psi(), emorse(), eshift, r1, r2, v, npoints) returns the eigenvectors of the quantum mechanical Morse oscillator based on the properties assigned via a previous call to morse_eigenvalue(...). The wavefunction [psi(1..npoints)] in units of 1/Sqrt[Å] (assuming eshift=0.0) and classical potential function [emorse(1..npoints)] in eV are returned as a function of the r values in xr(1..npoints). If eshift is non-zero, the wavefunction is returned shifted in energy space by eshift so that the wavefunction can be displayed on top of the morse potential. Once shifted, the wavefunction can no longer be used to derive matrix elements of the Morse oscillator.

Mouse_down_x returns the x coordinate when the mouse button was first pressed inside the graphics canvas in the Graphics Panel.

Mouse_down_y returns the y coordinate as above.

Mouse_position(ix, iy, ix0, iy0, ix1, iy1) returns the global pixel position of the mouse in ix, iy (computer screen upper left is 0, 0). The program window upper left edge is at ix0, iy0 and lower right edge is at ix1, iy1. All variables are integer pixel values. The mouse button is ignored and data are always available.

Mouse_up_x returns the x coordinate when the mouse button was released inside the graphics canvas in the Graphics Panel

Mouse_up_y returns the y coordinate as above. Note that the mouse_down and mouse_up data are only valid when Q_mouse_data_available is true.

|MS| **Mult**(s1, s2) **as string** returns s1*s2 using arbitrary precision string arithmetic

New keyword used to instantiate a class variable (see class).

Next part of For...Next statement. This statement increments the loop variable and exits the loop if the looping is complete.

Nil an internal variable representing an undefined or invalid value.

NthField(s0, delimiter, ith_field) **as string** returns the ith field within the string s0 delimited by the string delimiter specified. Use countfields to figure out how many fields are available.

Numerical_best_spacing(x(), y(), npoints) **as integer** returns the best value for nwidth to be used in the numerical routines below. The value is based on an analysis of the data in the arrays x(1..npoints) and y(1..npoints. The value is based on finding the best compromise value of one that is large enough to minimize truncation error and noise yet small enough to preserve the fine structure within the derivatives. The value returned is an estimate and trial and error coupled with visual inspection of the derivatives may be necessary.

Numerical_complexity(x) **as integer** returns the numerical complexity of a double. This function is used to determine the best choice for axis increments, data set increments, or other lists or data sets where one wants to avoid using unnatural increments. For example, a data set that has x values 200.0, 200.2, 200.4, … is much easier for a human to work with than 200.0, 200.19765, 200.3953, … . The sign of the value is ignored. All integers have complexity 0 (if even) or 1 (if odd). All real numbers have complexities that range from 1 to 19 based on the number and nature of digits to the right of the decimal points.

Numerical_derivative(x(), y(), yd(), npoints, nwidth, nderiv) generate the nderiv derivative of the data set x(1..npoints), y(1..npoints) and put the derivative into yd(1..npoints). This routine calculates the 0th, 1st, 2nd, 3rd and 4th derivative and does not assume that the x() data are equally spaced. The nwidth parameter allows you to spread out the window by skipping nwidth data points, which serves to smooth the data. Choose nwidth to be roughly double the number of points that you would need to average to get rid of the noise. It is recommended that numerical_best_spacing be run to assign an estimate for nwidth.

Numerical_double_data(x(), y(), npoints, [nwidth]) use optimized interpolation to double the data in the data set x(1..npoints), y(1..npoints). The first three parameters are byref and are changed by this routine so that upon exit npoints is roughly doubled and the new data replace the original data. This routine also selects the delta x increment to have a minimal numerical_complexity and unless you include the optional parameter, nwidth, this routine uses numerical_best_spacing to select the interpolation width. Accordingly, if the data set is very noisy or jagged the data are smoothed. If less smoothing is desired, enter a user value of nwidth closer to 1 (no smoothing at all).

Numerical_fit_to_gaussians(xkk(), y(), n, nbands, niter, xp(), yp(), x1, x2, np [, sharpen]) **as string** fit the spectrum in wavenumber space to gaussians. The spectrum to fit is in y(1..n) as a function of wavenumber in kiloKaysers (1000 cm^-1) in xkk(1..n). The initial number of gaussians is assigned via nband, but this variable may be reduced during the fit if fewer gaussians are needed. The total number of iterations is set by niter (500 is usually sufficient). The spectrum based on the sum of gaussians is returned in xp(1..np) and yp(1..np) from x1 to x2, which also must be in kK. If the optional parameter sharpen is included, the spectrum in xp(), yp() can be sharpened (>0.0 to 0.75) or broadened (<0.0 to -0.9).

Numerical_fraction(f1, ndigits) **as string** converts a real number to its equivalent fraction, or a fraction that reproduces the number to an accuracy of ndigits. The string that is returned includes the fraction as well as the expansion of the fraction to a precision of 32 digits. A maximum of 15 digits of precision is allowed.

Numerical_generate_expression(target, ndepth) **as string** return an expression, f(...), that provides the best fit to the target. This function not only assigns the function but the values of parameters of that function, and minimizes the error given by abs(f(...)-target)/abs(target). Expressions examined include (k1/k2)*exp((k3/k4)*pi^(k5/k6)), (k1/k2)*log((k3/k4)*pi^(k5/k6)), k1*sin(k2*pi/k3), k1*cos(k2*pi/k3), (k1/k2)*const_pi^(k3/k4), and (k1/k2)*(k3!/k4!) where k1..k6 are all integers which can be different for each expression. The value of ndepth controls the depth of the search and assigns the maximum value for the integers in the expressions. If ndepth is negative, the depth of search is equal to abs(ndepth) but the best expression for each type is listed for comparison. This function is useful whenever a calculation yields a value which you suspect is represented by one of the above expressions. Values of ndepth=15 provide a fairly rapid search and are the minimum ndepth used regardless of input. Ndepth values above 15 take progressively longer such that ndepth=100 will take many hours. The target should be in the range ±(0.01 to ndepth) for this method to explore all of the expressions listed above.

Numerical_interpolate(x0, x(), y(), npoints, nwidth) **as double** uses a five point interpolation function to calculate values of y as a function of x0. The interpolation is based on the same polynomial expansion as used in numerical_derivative. It is recommended that numerical_best_spacing be run to assign an estimate for nwidth.

numerical_interpolate_points(x(), y(), n, xp(), yp(), np, nlevel) transfer the sparse x(1..n), y(1..n) arrays into dense xp(1..np), yp(1..np) arrays by using 2, 3, 4 or 5 point Lagrangian interpolation, as set by nlevel. The value of np on entry is the desired size, but it may be changed slightly to accommodate the process, so it is a byref parameter.

Numerical_maxent_extend(pdata(), pdnew(), lpc(), ndata, ncoef, knew, iap) extends a periodic data set pdata(1..ndata) and places the result in pdnew(1..knew), where knew is the number of extended data points. This routine requires the linear prediction coefficients, lpc(1..ncoef), provided by numerical_maxent_lpc(..). The pdnew(1..knew) data can be apodized by selecting iap>0 (0=none, 1=linear, 2=convex, 3=concave). If iap<0, then this method returns the apodization in pdnew(1..knew).

Numerical_maxent_lpc(pdata(), lpc(), xms, npdata, ncoef) returns ncoef linear prediction coefficients in the array lpc(1..ncoef) based on an analysis of the periodic data in pdata(1..npdata). Also returns the xms, the mean-square-discrepancy between the lpc predicted versus observed periodic data.

Numerical_maxent_spectrum(fdt, lpc(), xms, ncoef) **as double** returns the intensity of the power spectrum at fdt, where fdt is the frequency times the sampling interval of the periodic data that was used to generate the linear prediction coefficients. fdt should lie within the Nyquist range from -0.5 to 0.5. The lpc(1..ncoef) and xms parameters are those returned from numerical_maxent_lpc(...).

Numerical_normalize(y(), npoints, Qinvert, Qnorm) invert and/or normalize the data in y(1..npoints. If Qinvert is true, then prior to normalization the data are all multiplied by -1 (i.e. inverted). If Qnorm is true, the data are normalized so that the minimum value is 0.0 and the maximum value is 1.0.

Numerical_normalize_integral(y(), n, dx, target) normalize the integral of y(1..n) where n is the number of elements, dx is the x spacing between the elements, and target is the value of the integral to be numerically achieved. A simple summation integration is carried out assuming equal spacing in x(1..n) where dx=x(2)-x(1)=x(n)-x(n-1).

Numerical_point_in_polygon(x1, y1, xp(), yp(), np) **as boolean** returns true if the coordinates x1, y1 are within the closed polygon defined by xp(1..np), yp(1..np). Np is an integer. All other parameters must be all doubles or all integers. The polygon can by highly complex in shape.

Numerical_smooth(y(), npoints, nsmooth) smooth the data in y(1..npoints) by a 3 to 101 point smoothing function set by nsmooth. If nsmooth is not odd, it is adjusted to the nearest odd value. Values of nsmooth of 3, 5, 7, 9, 11 and 13 use optimized methods that are very fast. Values of 15 and higher use slower methods. The method can also smooth a two-dimensional array, y(1..nx, 1..ny), by calling numerical_smooth(y(), nx, ny, ntimes), where ntimes is the number of smoothing iterations all carried out at nsmooth=3.

Numerical_spectral_enhance(x(), y(), ye(), n [, a2, a4, n24width, ns24, gaus0, gauswidth]) enhance the resolution of the spectrum x(1..n), y(1..n) by using derivative methods. The original spectrum is enhanced by subtracting the second derivative and adding the fourth derivative. The weights (a2 and a4), widths (n24width) of the numerical derivatives, and smoothing (ns24) are all varied by trial and error. In addition, a section of the spectrum can be selectively enhanced by providing values for gaus0 (center) and gauswidth (FWHM) in the same units as x(1..n). If only the first four parameters are included, a window is opened and the user can manipulate the various parameters in real time to see the effect.

Oct(i) **as string** returns the octal equivalent of the integer i in string format.

Open_all_user_pictures(filenames(), nx(), ny(), filename_filter, nxmax, nymax) **as integer** returns the number of picture files in the user_pictures folder that satisfy the filename_filter criterion (see below) and places each of the valid pictures into the pictures set (1..ntotal), where ntotal is the value returned by the function. For the ith picture the arrays return the string filename(i), the pixel width nx(i) and the pixel height ny(i). The user can limit the size by setting nxmax and nymax to the maximum dimensions allowed. If the picture is larger, it is scaled down to be within the maximum dimensions while maintaining aspect ratio. These variables should be set to zero if no size constraints are desired. The string filename_filter, if assigned a non-null string, will be used to select only those files with names including the filename_filter. For example, if one wants only jpeg files, then filename_filter = ".jpg". If one only wants files that include the name "fruit", then filename_filter = "fruit". The string only needs to be found somewhere within the filename. Because all of the filenames are returned in their entirety, more sophisticated searching can be done after loading the initial set of pictures.

Open_picture_conversion_window when this statement is executed the program opens up a modal window that allows the user to open a picture, manipulate the picture, and return the picture to the buffer (or if multiple buffers have been created, to buffer number 1).

Open_user_data_file(ifilenumber, filename) **as Boolean** this routine goes to the folder data_files, counts the number of files inside, and then uses the ifilenumber to open the ith file. The filename is returned in the string variable filename, and provided the file is of the correct type (*.cet) the data are loaded into the spreadsheet within the data set panel. If ifilenumber is 0 and the filename="" then the function returns the number of files found in the folder in ifilenumber. If ifilenumber is 0 then the filename is used and that file is opened and loaded into the spreadsheet. If file_number=0 and filename="unknown" then a dialogue is opened and the user can select the file to open. If the process fails for any reason, the function returns false.

Open_user_picture_file(ifile_number, filename[, idestination]) **as Boolean** operates in a fashion identical to the open_user_text_files routine but goes to the user_pictures file and then loads the picture into the buffer. If multiple buffers have been created, then the picture is loaded into buffer 1. If the optional parameter idestination is included, it designates the picture slot into which you want to place the opened picture. The picture file must be created prior to calling this routine.

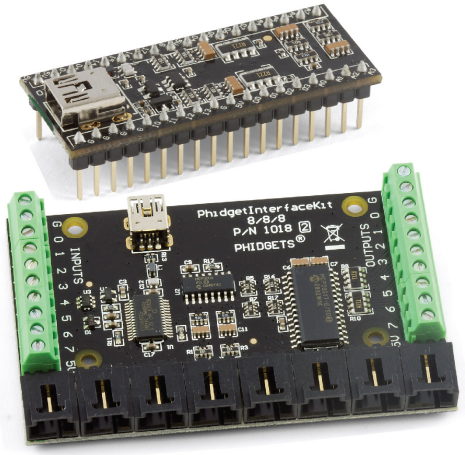
Open_user_text_file(ifilenumber, filename, filecontents) **as Boolean** this routine goes to the folder user_files, counts the number of files inside, and then uses the ifilenumber to open the nth file. The filename is returned in the string variable filename, the entire contents of the file is returned in filecontents. If file_number is 0 then the filename="" then the function returns the number of files found in the folder in ifilenumber. If ifilenumber is 0 then the filename is used and that file is opened. If ifilenumber=0 and filename="unknown" then a dialogue is opened and the user can select the file to open. If the process fails for any reason, the function returns false.

Oscillator_strength(mass, mass_units, dEmn, eunits, Rmn, runits, byref lifetime, iprint) **as double** returns the dimensionless oscillator strength for a system where mass is the mass of the particle or reduced mass of the oscillator undergoing the m->n transition. The units of mass are passed in the string munits (amu, g, kg, au). DEmn is the transition energy or wavelength determined by the units passed in eunits (eV, J, aJ, 1/cm, kK, GHz, THz, Hartree, au, kcal/mol, kJ/mol), Rmn is the transition length or transition dipole moment, with runits determining which is being passed (A, cm, m, eA, D, au). Note that for all units, case is ignored, and angstrom should be entered using A, not the symbol Å to avoid parsing problems. The intrinsic lifetime of the n-->m transition is returned in seconds. This number should be multiplied by the quantum yield of emission to estimate the observed lifetime. iprint=0 (no commentary), 1 (minimal comments), or 2 (full reporting)

Pause(ms as integer) pause program for a minimum of ms milliseconds and updates the current window so that graphics and printed output is displayed fully. If all you want to do is update the display, you can call Pause(0). This function also scrolls the text output cursor to the last character in the output windows.

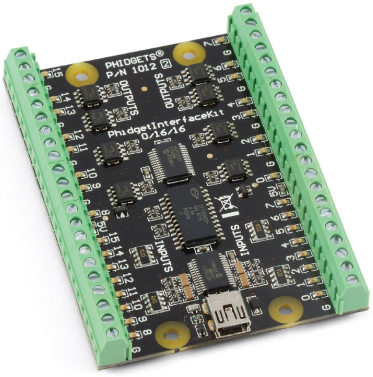
Phidgets are electronic boards that connect to the computer via usb. MathScriptor versions above 2.0 have the following statements which can be used to control and get information from selected phidget boards. For more information on the available phidget boards visit http://www.phidgets.com. To emphasize that these statements are only available in version 2.0 and above, a |Ph| symbol is added to the left of the keyword. A phidget board cannot be used until it is first opened by calling the appropriate method with icontrol=999 (if there is only one board) or icontrol=-serial_number (if there are multiple boards). The serial numbers of the boards can be deduced by printing out phidget_list (then remember to add a minus sign in front and assign it to icontrol). Properties of the objects to be controlled are assigned during open, where appropriate. All subsequent calls use icontrol to select ports or control other properties as described above. The board will stay attached until it is closed using icontrol=-1. Any relays or outputs that are activated will be turned off when the device is closed. Only one board of a given type can be attached at a time with the exception of the 8/8/8 interface board (see phidget_interface888_2nd(...) for details). If you are having trouble communicating with the phidget boards, make sure you have run the Phidget 21 installer (www.phidgets.com/drivers.php) on the current computer and verified that the phidget monitor, which is installed with the drivers, can find and manipulate the boards. It is often necessary to exercise the boards using the monitor to bless the communication process. Also verify that the phidget framework has been loaded by running Test Mathscriptor Environment under the Compiler menu and observing the statement "Phidget framework has been found and loaded."

Fast Output Pulse Option. Some interfaces support the fast output pulse option, and this fact will be indicated in the last sentence of the command description. To use this capability, set the icontrol parameter to 9xxmmmm where xx=the output port and mmm.m=milliseconds pulse duration. Use the code icontrol= 9000000+ 10000*kport +msx10, where kport is the output port and msx10 is the activation time in increments of 0.1 ms. Thus, to activate output 3 for 0.1 millisecond, set icontrol=9030001. If multiple ports need to be activated in fast pulse mode, set kport=99, and the Qout() array (or vout() for the analog board) will be used to set the states or values. If a value of 0 ms is selected (i.e. 90300000), the output is pulsed as fast as possible. An estimate of the actual output activation time is returned. Note for boards using mechanical relays, a mechanical latency of 30ms or more is common. Thus, if the relay is not closed for a time greater than the latency, the relay is not activated at all. If faster relays are required (<30 ms), the 0/16/16 board is recommended. The short pulse option also works for generating voltage pulses using the phidget_analog() method. When using kport=99 to fire multiple ports recognize that a modest latency of about 0.1 ms is introduced. It should be noted that while the computer interface may support this option, the board may have additional latency. Many phidget boards can only generate 1 ms or longer pulses. Examples of phidget boards that can be controlled via MathScriptor are shown on the next page.

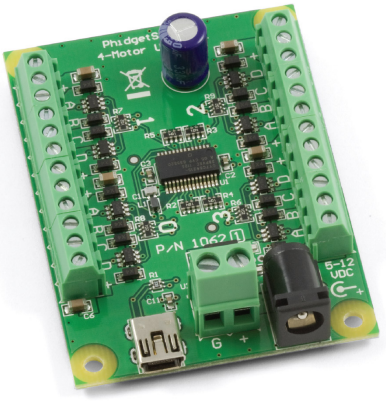


PhidgetInterfaceKit 8/8/8 Mini-Format (top) and below it PhidgetInterfaceKit 8/8/8 both controlled by using phidget_interface888()

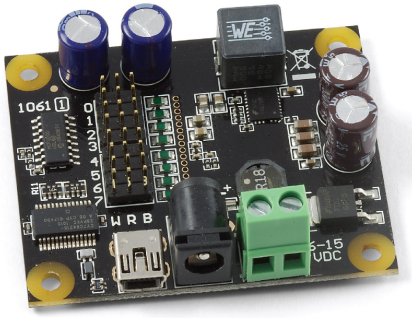
PhidgetTextLCD 20X2 : White and Integrated PhidgetInterfaceKit 8/8/8 controlled by using PhidgetTextLCD() and phidget_interface888()



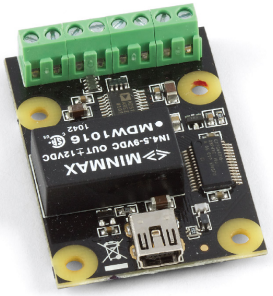
PhidgetInterfaceKit 0/16/16 controlled by using phidget_interface01616()



PhidgetStepper Unipolar 4-Motor board controlled by using phidget_stepper()



PhidgetAdvancedServo 8-Motor board controlled by using phidget_servo()



Phidget Analog Board controlled by using phidget_analog()

|Ph| **phidget_analog**(icontrol, vout()) Opens an attached phidget analog board (icontrol=999) and on subsequent calls, sets the state of the four output voltages (icontrol=1) where vout(i)=output voltage, where i=0, 1, 2 or 3. Outputs are limited to -10 to 10 VDC with 12 bit (4.8mV) resolution and 20mA max current (aim for 5mA for voltage accuracy). When done, call this method with icontrol=-1, to close the interface. This interface supports the fast output pulse option.

|Ph| **phidget_interface004**(icontrol, Qrelay()) Opens an attached phidget InterfaceKit 0/0/4 (icontrol=999) and on subsequent calls, sets the state of the four relays (icontrol=1) where Qrelay(i)=true to turn relay i on, Qrelay(i)=false to turn relay i off, where i=0, 1, 2 or 3. Note the first relay is 0, not 1. When done using the interface, call this method with icontrol=-1, to close the interface. This interface supports the fast output pulse option.

|Ph| **phidget_interface008**(icontrol, Qrelay()) Opens an attached phidget InterfaceKit 0/0/8 (icontrol=999) and on subsequent calls, sets the state of the eight relays (icontrol=1) where Qrelay(i)=true to turn relay i on, Qrelay(i)=false to turn relay i off, where i=0, 1, 2..7. Note the first relay is 0, not 1. When done using the interface, call this method with icontrol=-1, to close the interface. This interface supports the fast output pulse option.

|Ph| **phidget_interface01616**(icontrol, Qin16(), Qout16()) **as string** Opens an attached phidget InterfaceKit 0/16/16 (icontrol=999). Subsequent calls either set or read the state of the 32 digital I/O channels. To read the inputs set icontrol=0, and the digital inputs are returned in Qin16(0, 1, 2, 3, 4 ... 15) as booleans (where true means the a voltage between 4-30 VDC has been applied to the input). To set the digital outputs set icontrol=1, and pass the desired output states in Qout16(0, 1, 2, 3, 4, 5, 6 and 7) as booleans (where true means the output has been shorted to ground). Outputs can handle 2amps at 30VDC. When done using the interface, call this method with icontrol=-1, to close the interface board. This interface supports the fast output pulse option.

|Ph| **phidget_interface222**(icontrol, Qin2(), Qout2(), analog_in()) **as string** Opens an attached phidget InterfaceKit 222 (icontrol=999). Subsequent calls either set or read the state of the 6 I/O channels. To read the inputs set icontrol=0, and the digital inputs are returned in Qin2(0, 1) as booleans (where true means the open collector input has been set to ground), and the analog inputs are returned as integers in analog_in(0, 1). The analog inputs measure voltages from 0 to 5 VDC. The value returned is scaled to 0-1000 (Q_phidget_raw=false) or 0-4095 (Q_phidget_raw=true). The analog inputs will measure the absolute voltage (0-5VDC) as (0-1000) if Q_phidget_ratiometric=false, or (0-Vpwr) as (0-1000) if Q_phidget_ratiometric=true. To set the digital outputs set icontrol=1, and pass the desired output states in Qout2(0, 1) as booleans (where true means CMOS output is high). When done using the interface, call this method with icontrol=-1, to close the interface and release the asynchronous monitoring of the inputs.

|Ph| **phidget_interface888**(icontrol, Qin8(), Qout8(), analog_in()) **as string** Opens an attached phidget InterfaceKit 888 (icontrol=999). Subsequent calls either set or read the state of the 24 I/O channels. To read the inputs set icontrol=0, and the digital inputs are returned in Qin8(0, 1, 2, 3, 4, 5, 6 and 7) as booleans (where true means the open collector input has been set to ground), and the analog inputs are returned as integers in analog_in(0, 1, 2, 3, 4, 5, 6 and 7). The analog inputs measure voltages from 0 to 5 VDC. The value returned is scaled to 0-1000 (Q_phidget_raw=false) or 0-4095 (Q_phidget_raw=true). The analog inputs will measure the absolute voltage (0-5VDC) as (0-1000) if Q_phidget_ratiometric=false, or (0-Vpwr) as (0-1000) if Q_phidget_ratiometric=true. To set the digital outputs set icontrol=1, and pass the desired output states in Qout8(0, 1, 2, 3, 4, 5, 6 and 7) as booleans (where true means CMOS output is high). When done using the interface, call this method with icontrol=-1, to close the interface and release the asynchronous monitoring of the inputs. This interface supports the fast output pulse option.

|Ph| **phidget_interface888_2nd**(icontrol, Qin8(), Qout8(), analog_in()) **as string** Opens the second 8/8/8 interface when two 888 interface boards are attached. Identical in performance and calling procedures as phidget_interface888(...) but always manipulates the 2nd of two 8/8/8 boards. The second board must be opened first, as it must open the first board, lock it, and then open the second board subsequently closing the first. This statement can also be used to help open an 8/8/8 when other interface boards are attached and show up in the list above the 8/8/8 board.

|Ph| **phidget_list as string** returns a string listing all of the attached Phidgets and their serial numbers. If you have multiple boards of the same type, you must assign them in the order given by this list. Note that each board has a unique serial number.

|Ph| **phidget_servo**(icontrol, set_to_position, kservo_type() [, microsecond_min(), microsecon_max(), degree_range(), velocity_max()]) **as string** Opens the Phidget Advanced Servo(icontrol=999) and assigns the servo parameters either using the kservo_type(0...7) integer array or via the four optional arrays (values are used when kservo_type=99). Subsequent calls assign the position of the ith servo by setting icontrol to the servo port (0, 1, 2, 3..7) and assigning set_to_position to the target degrees (or extension for linear servos). The method will not return until the servo has reached the destination. When done using the servos, it is important to close the servo board by calling this method with icontrol=-1. If the optional parameters are included, then all subsequent calls must include these parameters, because static variables are used within the overloaded methods.

The array servotype(iport_number) assigns the servo type integer for each of the connected servos:

0 (No servo is connected to port)
1 (Futaba FP-S148, range= 0-220°, vmax=240°/s)
2 (raw microsecond mode)
3 (Hitec HS322HD, range= 0-180°, vmax=316°/s)
4 (Hitec HS5245MG, range= 0-145°, vmax=400°/s)
5 (Hitec 805BB, range= 0-180°, vmax=316°/s)
6 (Hitec HS422, range= 0-180°, vmax=286°/s)
7 (Towerpro MG90, range= 0-175°, vmax=545°/s)
8 (Hitec HSR1425CR, range= 0-100°, vmax=500°/s)
9 (Hitec HS785HB, range= 0-2880°, vmax=225°/s)
10 (Hitec HS485HB, range= 0-180°, vmax=272°/s)
11 (Hitec HS645MG, range= 0-180°, vmax=300°/s)
12 (Hitec 815BB, range= 0-180°, vmax=250°/s)
13 (Firgelli linear servo L12R, range= 0-30 mm, vmax=23 mm/s)
14 (Firgelli linear servo L12R, range= 0-50 mm, vmax=12 mm/s)
15 (Firgelli linear servo L12R, range= 0-50 mm, vmax=5 mm/s)
16 (Firgelli linear servo L12R, range= 0-100 mm, vmax=23 mm/s)
17 (Firgelli linear servo L12R, range= 0-100 mm, vmax=12 mm/s)
99 (servo properties passed via 4 double arrays during the open operation):
[microsecond_min(), microsecond_max(), degree_range(), velocity_max()])
where microsecond_min(i)= The minimum supported PCM in microseconds,
microsecond_max(i)= The maximum supported PCM in microseconds,
degree_range(i)= The degrees of rotation defined by the given PCM range, and
velocity_max(i)= The maximum velocity in degrees/second.

|Ph| **phidget_stepper**(icontrol, position(), maxcurrent(), maxvelocity()) **as string** Opens the Phidget Unipolar Stepper interface board (icontrol=999) and assigns the properties of up to four stepper motors attached. During open, the int64 position() array assigns the starting position of the 0, 1, 2 and 3 steppers. Maxcurrent() and maxvelocity() are double arrays which assign the corresponding maximum current in amps and the maximum velocities in steps/second for each of the stepper motors. If a port has no stepper attached or the stepper is to be kept inactive, set maxcurrent(i)=0.0. Subsequent calls allow the stepper to be positioned by assigning position(i) for each of the i steppers and then calling this method using either icontrol=stepper number or icontrol=99 to move all motors to the desired position() set. The board will continue to provide current to the stepper to hold the stepper in position until closed. To close the interface and release the steppers, execute this method after setting Icontrol = -1. The method reports on progress and positions in the returned string. (Application note= unlike servos, stepper motors draw current even when stopped. The power supply should provide the correct voltage for the motor, and be capable of providing the total current for all steppers attached. Current is only supplied to the level assigned by maxcurrent(), but if the power supply lacks the current capability, the stepper motor will skip steps. This interface is not smart, and will not know if a stepper has skipped a step. It is a good idea to use limit switches and a phidget_interface (8/8/8 or 0/16/16) to monitor position.

|Ph| **phidget_textLCD**(icontrol, row1, row2) **as string** Opens an attached phidget LCD Text display (icontrol=999). Subsequent calls set the two rows of text where row1 is a string containing the top 20 characters and row2 is a string containing the bottom 20 characters. The contrast is set by assigning icontrol to values from 1 - 255 (125 is nominal), and the brightness is set by assigning icontrol a value from 500 - 755 (650 is more than adequate). If the phidget is a combination LCDText and 8/8/8 interface board, this method only controls the LCD portion.

pictures_blend_to_buffer(ipicture1, fraction1, ipicture2, fraction2) **as boolean** blends two pictures and places the result into the buffer (or buffer 1 if multiple buffers exist). Thus, buffer(1) = fraction1*picture(ipicture1) + fraction2*picture(ipicture2), where fraction is a double between 0 and 1, and the pictures must be of the same size. A simple way to blend two pictures is to create the first one in the buffer, then use buffer_copy_to_picture(ipicture1). Create the second and use buffer_copy_to_picture(ipicture2). Note that if both fractions are set equal to 1.0, white space will dominate and in most cases, a blank white picture will result. Best results are obtained when fraction1+fraction2=1.

pictures_clear_all clears all the pictures from memory and allows new pictures to be loaded with sizes determined by the new properties. Pictures are static objects that persist after a program has been closed. Hence, it is important to use this method to make sure each session starts without any pictures remaining from previous runs unless the user plans to make use of this capability.

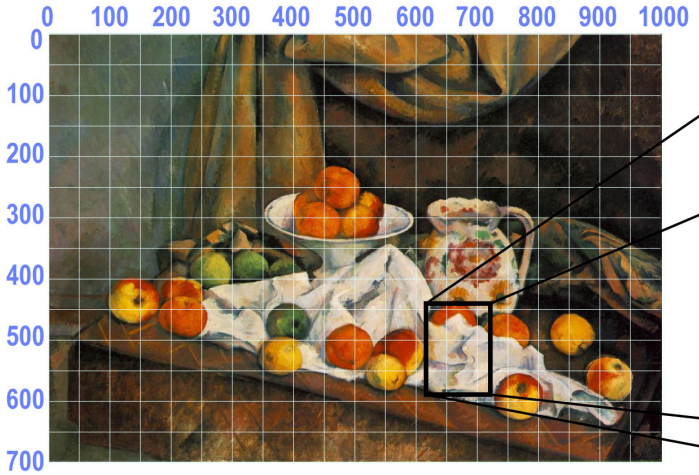
picture_copy_to_buffer(ipicture) copies picture number ipicture into the buffer on a 1-to-1 pixel ratio starting at the upper left.

picture_create(ipicture, nwidth, nheight, Qtransparent) creates a new picture file in the picture slot ipicture with size nwidth by nheight. If Qtransparent is true, pure white becomes transparent. Pure white is RGB(255, 255, 255). You can use the picture conversion window, available under the Edit menu, to manipulate the location and amount of transparency for any given picture. You can create as many pictures as memory allows, but it is important that the picture slots be created in order from low to high.

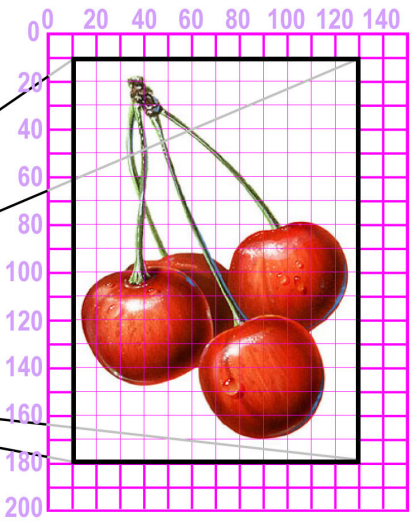
picture_height(ith_picture) **as integer** returns the pixel height of the ith_picture (if it exists) or returns -1, if the ith_picture does not exist.

picture_make_transparent(ith_picture) makes the ith_picture transparent. If the value of ith_picture is negative, the abs(ith_picture) is made non-transparent, which means that purewhite is now solid. This function does nothing if the ith_picture has not yet been created. An example is shown in the figure below.

picture_width(ith_picture) **as integer** returns the pixel width of the ith_picture (if it exists) or returns -1, if the ith_picture does not exist.



buffer 1 (itarget=0)

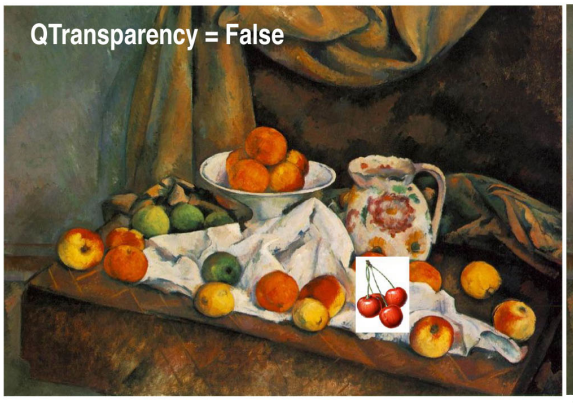


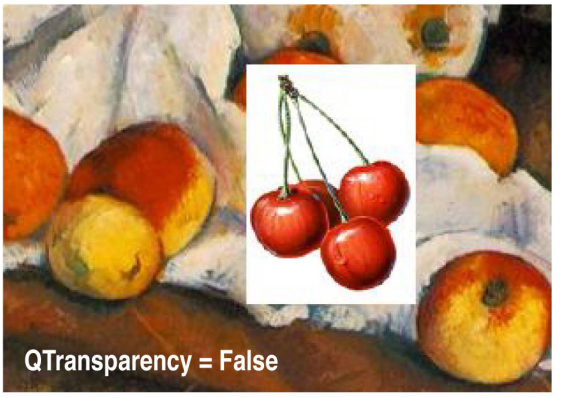
picture 1 (isource=1)

targetx = 630, targety = 450
destwidth = 120*0.8, destheight = 170*0.8

picture_write(itarget,isource,
targetx,targety,destwidth,destheight,
sourcex,sourcey,sourcewidth,sourceheight)

sourcex = 10, sourcey = 10
sourcewidth = 120
sourceheight = 170





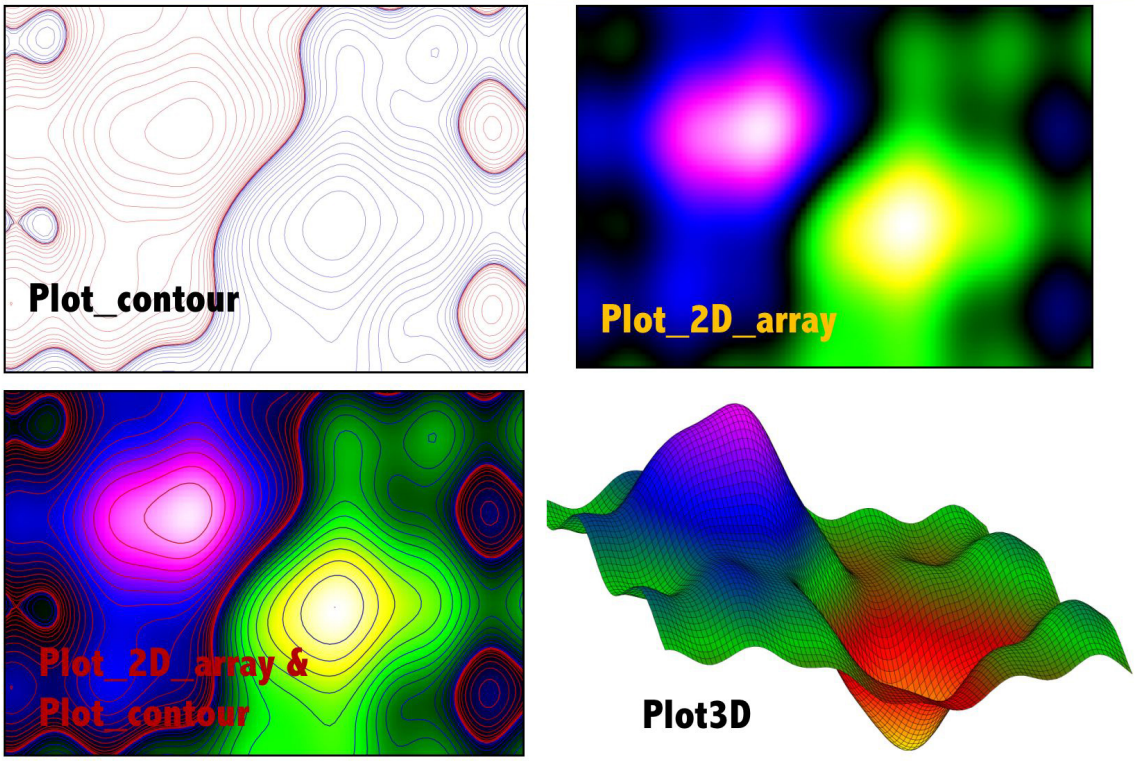
picture_write(isource, targetx, targety [, destwidth, destheight, sourcex, sourcey, sourcewidth, sourceheight]) draws picture number isource into the buffer. TargetX and TargetY are the upper left hand pixel coordinates. If no other parameters are included, then the entire picture is written into the target at targetx and targety. One can, if desired, include another six parameters to designate the size of the window and the portion of the picture you want to draw. Destwidth and destheight set the size of the window into which you want to write the picture. Sourcex, sourcey, sourcewidth and sourceheight set the upper left hand corner and size of the picture area you want to copy into the window previously defined. The sizes and aspect ratios need not be the same and thus your picture, or picture section, can be compressed or skewed to accommodate the target window. The picture on the previous page illustrates the concepts of both picture_write and transparency. You can duplicate this exercise by running template_picture_transparency.txt.

const_Pi as double internally defined constant =3.141592653589793. If you want more significant digits, use arprec_pi as string to return as many digits as was previously assigned using arprec_set_precision (see above).

plot3D(z(), theta, phi, xzoom, yzoom, [ioption] or [cz()]) plots a three dimensional representation of the 2D double array z(1..ubound1, 1..ubound2) from a view direction of theta and phi degrees (20, 20 usually works) with size options determined by xzoom and yzoom (start with 1, 1). The last parameter is either ioption or a 2D array of colors the same size as z(,). Ioption selects transparent mesh (0), wire frame (1), gray scale (2) or color (3).

plot3d_xshift assigns as integer assigns the x axis offset shift of the 3D plot. A positive value shifts the plot to the right.

plot3d_yshift assigns as integer assigns the y axis offset shift of the 3D plot. A positive value shifts the plot up.



Comparison of four methods of plotting data in a two-dimensional array.

Plot_2d_array(a2(), n1, n2, ioption, imod, Qzero) plot the values of the two-dimensional array a2(1..n1, 1..n2) or if Qzero is true, a2(0..n1-1, 0..n2-1) where ioption provides for the option of apodization (ioption=0, 1=none, 2=triangular (linear) ww = 1 - (rxy/rxymax), rxymax = (nxbasis-center) = center, 3=lorentzian (quadratic) ww = 1 - rxy^2/rxymax^2 = 1 - rxy^2/center^2, >3=gaussian (exponential) with fwhm = 1/ioption (1/4, 1/5, etc.), if negative, the apodization is based on abs(ioption) but the absolute value is plotted. The parameter imod selects the modulus of the display (min=1, max=4). Add 100 to ioption if you want gray scale.

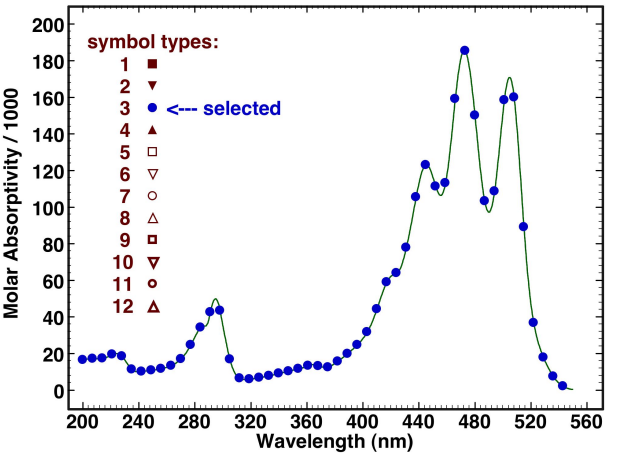
Plot_contour(a2(), mxz, nyz, ncontours, linewidth, ioption) Generate a contour plot into the buffer of the data in a2(1..mxz, 1..nyz) where ncontours = number of contours, linewidth = linewidth of the contour lines and ioption = 0 (plot on top of white), 1 (plot on top of background color), 2 (plot on top of whatever was already there). Use ioption=2 if you want to plot a contour on top of a plot_2d_array display.

plot_dashed_data(x(), y(), npoints, line_thickness, line_color, L1, S1, L2, S2) identical in function to plot_more_data except the line that is plotted is dashed. The dash lengths are L1 and L2 and the intervening spaces are S1 and S2. For a simple dash, L1=S1=L2=S2. All values are in buffer pixels. If L1, S1, L2, S2 are replaced with two parameters: [L1, nresolution], a simple dashed line is generated with line lengths L1 and equal separating spaces of L1. Nresolution gives the number of points used in the total path expansion, and should be about 2000 or more, depending upon the sharpness of the peaks. The sharper the peaks, the higher the nresolution required.

plot_dashed_line(x1, y1, x2, y2, line_thickness, line_color, L1, S1, L2, S2) plot a dashed line from x1, y1 to x2, y2 of thickness line_thickness and color line_color. The dash lengths are L1 and L2 and the intervening spaces are S1 and S2. For a simple dash, L1=S1=L2=S2. All values are in buffer pixels.

Plot_data(x1() , y1(), npoints, x1, x2, y1, y2, xlabel, ylabel, gridlevel) plot data in x1(1:n) and y1(1:n) from x1, x2 and y1, y2. The axes labels are given via strings xlabel, ylabel and a background grid can be added using 0<= gridlevel<= 1.0

Plot_data_point(x, y, s0, ilocation, symbol_type, symbol_size, symbol_color) plot an individual data point using the axes generated from a previous call to plot_data. Each data point is plotted using one of 12 symbol_types as shown in the figure at right. The size in pixels is controlled using symbol_size and the color is set using symbol_color. You can label each data point using the string s0 and the ilocation integer symbol 1(upper left), 2(above), 3 (upper right)4 (at left), 5 on top, 6 (at right), 7 (lower left), 8 below, 9 (lower right). The font name and font size as specified by graphics_font. The figure was created using demo_plot_data_points.txt.



Plot_dashed_data(x(), y(), npoints, line_thickness, line_color, L1, S1, L2, S2) identical in function to plot_more_data except the line that is plotted is dashed. The dash lengths are L1 and L2 and the intervening spaces are S1 and S2. For a simple dash, L1=S1=L2=S2. All values are in buffer pixels. If L1, S1, L2, S2 are replaced with two parameters, L1, nresolution, a simple dashed line is generated with line lengths L1 and equal separating spaces of L1. Nresolution gives the number of points used in the total path expansion, and should be about 2000 or more, depending upon the sharpness of the peaks. The sharper the peaks, the higher the nresolution required.

Plot_data_points(x1(), y1(), n, symbol_type, symbol_size, symbol_color) plot data in x1(1:n) and y1(1:n) as individual symbols following the parameter methods previously described for plot_data_point. In the present case, however, one cannot label the individual data points.

plot_data_points_with_errors(x(), y(), yerror(), npoints, symbol_type, symbol_size, symbol_color, error_bar_type) plot data points including a vertical error bar. All parameters defined as in plot_data_points except yerror(1..npoints) gives the total length of the errorbar in units of y, and error_bar_type specifies the type of error bar (0=single line, >0=width of horizontal lines at top and bottom of error bar in pixels).

plot_data_with_xstrings(x(), y(), npoints, x1, x2, y1, y2, xlabel, ylabel, gridlevel, sx(), xshift) identical to plot_data() but with two added parameters at the end. sx(1..n) as string is an array containing the x axis max tick labels where sx(1) is at x=1, x2(2) is at x=2, etc. The xshift parameter allows the labels to be shifted to the left (xshift<0) or to the right (xshift>0) relative to the major ticks. The user must execute plot_set_ticks(1, 1, ysmall_tick, ymajor_tick) first.

Plot_fontname as string set the fontname for plot_data. This function should only be used when the user is sure that the fontname assigned is present as no error checking is done during assignment. The plot font can also be adjusted by using the statement graphics_font(font_name, isize, Qitalics, Qbold), which returned true if the font_name is available on the current computer.

Plot_fontsize as integer set the fontsize for plot_data. The easiest way to adjust the size of the plot text when the font does not need to be altered.

Plot_histogram(harray(), icolor(), nh, dfwhm, barwidth, xp1, xp2) generate and plot a histogram of the data in harray(1..np) using the integer array icolor(1..nh) to designate the color to be assigned to each individual point in harray(). The maximum number of different values is 32, but numbers less than 12 work best for clarity. The sampling width is given by dfwhm, and a barwidth of dfwhm/3 is nominal. The plot is from xp1 to xp2. Set both to zero if you wish the program to select the range.

plot_line(x1, y1, x2, y2, linewidth, line_color) draw a line from x1, y1 to x2, y2 in the plot coordinate system with the linewidth and line_color designated. Plot_data() must be executed first.

Plot_more_data(x1(), y2(), n, line_thickness, line_color) plot data in x1(1:n) and y1(1:n) on top of the previous axes using a line_width of line_thickness and a line color of line_color.

Plot_rectangle(x1, y1, x2, y2, linewidth, line_color, [fill_color]) draw a rectangle with plot coordinates with vertices at x1, y1 and x2, y2. The rectangle's outer boundary is drawn with a penwidth of linewidth and color line_color. If the optional parameter, fill_color, is included at the end, the rectangle is filled with that color. Otherwise, the rectangle has no fill (transparent).

Plot_set_options(nx_major_ticks, nx_minor_ticks, nx_axis_shift, ny_major_ticks, ny_minor_ticks, ny_axis_shift) Takes control of plot options and uses integers to control the number of major ticks (positive major_ticks increase, negative decreases), the number of minor ticks (positive minor_ticks increases, negative decreases), and allows shifting the axes (positive axis_shift shifts x up and y to the right). Be careful manipulating minor ticks as they must fall on top of major ticks to show the major ticks. Set all values to 0 to return control of the plot options to Scriptor.

Plot_set_ticks(xsmall, xbig, ysmall, ybig) manually sets the major (xbig, ybig) and minor (xsmall, ysmall) tick separations. If the values are all integers, and the subsequent plot ranges are integers, then ticks are assigned using modular arithmetic. If the values are real, the ticks are separated from x1 and y1 must be assigned on a major tick mark for aesthetic reasons. If the user wants to return to automatic tick assignment, execute plot_set_ticks(0, 0, 0, 0) and the plot statements will auto assign ticks as best they can. Must be executed prior to calling any plot statements, and if used in conjunction with plot_set_options, it overrides the tick controls.

Plot_string(s0, x_center, y_center, string_color) plots the single line string s0 centered at x_center, y_center with font color string_color. If multiline, x_center, y_center designates the upper left hand corner of the left-justified text. The difference between this command and draw_string is that x_center and y_center are in the plot_data coordinate system, not the canvas coordinate system. This is the statement of choice if the goal is to label features inside a data plot.

|MS| **Plus**(s1 as string, s2 as string) **as string** arprec statement that adds the string numbers s1 and s2, which can be real or complex (indicated by a comma between the real and imaginary components).

Pow(a, b) **as double** returns a to the power of b (=a^b). This function is identical to using the ^ symbol (a^b).

Pragma(string_directive) Mediates the way in which Scriptor compiles the users program, or communicates with the user during a run. The following directives are available:

comments_minimize: turns on the preference Minimize text commentary generated by Scriptor.

comments_allow: turns off the preference Minimize text commentary generated by Scriptor.

arprec_validate: turns on arprec argument validation.

arprec_force_real: as above and forces numbers to be real.

arprec_validate_off: turns off arprec argument validation.

auto_clear_buffer: turns on automatic clearing of the buffer preference.

do_not_auto_clear_buffer: turns off automatic clearing of the buffer preference.

simple_mode: activate simple mode (no external objects).

external_objects: allow program access to external objects.

optimize_speed: turns on compiler optimizations and turns off a variety of run-time checks that slow up execution. Not recommended during program development

normal_speed: turns off compiler optimizations and turns on the standard run-time checks that are important for program debugging

stop_via_mouse_click: turns on the menu item Activate debug_stops via mouse click. However, this prohibits working on other programs during execution

stop_via_stop_button: turns off the menu item Activate debug_stops via mouse click. Now the user must press the stop button in Main to activate a debug_stop

quickdraw: sets the preference Use QuickDraw instead of Quartz graphics engine (Mac OSX only, ignored on other platforms).

quartz: uses the Quartz graphics engine on Mac OSX (ignored on other platforms).

Note that only one string_directive can be handled per statement, because after a valid pragma is found and handled, a return is executed.

#Pragma directive [boolean] Alternative and more flexible

BackgroundTasks	Enables or disables auto-yield to background threads. In addition to the pragma directive, specify True or False. Setting this directive to False is the same as using DisableBackgroundTasks.
BoundsChecking	Enables or disables bounds checking. In addition to the pragma directive, specify True or False. Specifying False is the same as using DisableBoundsChecking.
DisableAutoWaitCursor	Used to disable the automatic display of the wait cursor (or watch cursor). The scope of this pragma is local. The wait cursor will be disabled in the method that calls the pragma until the method ends. For example, if DisableAutoWaitCursor is called in a Pushbutton that runs a loop, the wait cursor will be disabled only until the loop runs and the Action event has completed.
DisableBackgroundTasks	Used to turn off automatic background task handling for code after the #pragma. It prevents Scriptor from yielding time back to the operating system, other applications, and threads. It can speed up very processor-intensive operations but prevents display of the Watch cursor, may halt normal background updating of interface elements, and prevents other threads from executing.
DisableBoundsChecking	Used to turn off array bounds checking on array index values in code after the #pragma. Not recommended unless the program has been fully debugged. Particularly useful for speeding up matrix operations.
NilObjectChecking	Controls whether to automatically check objects for Nil before accessing properties and calling methods. In addition to the pragma directive, specify True or False.
StackOverflowChecking	Controls whether to check for stack overflows. In addition to the pragma directive, specify True or False.

The use of pragmas to speed up program execution should never be used until the program is fully debugged and tested. Three of these pragmas, BoundsChecking, NilObjectChecking and StackOverflowChecking when disabled, remove important safeguards. While Scriptor and MathScriptor operate in an encapsulated environment, and one need not worry that a program will damage anything permanently, any of these three events can produce run-time errors that are very hard to diagnose. Also, turning off background tasks can make the user interface completely unresponsive, and the program nearly impossible to stop without implementing a force quit.

|MS| **Prime**(n as int64) **as int64** returns the nth prime number where n<3, 200, 000, 000.

|MS| **PrimeQ**(ix as int64, ntrials as integer) **as boolean** returns true if the number ix passes the Miller-Rabin test for prime numbers based on a series of ntrials. If the test returns false, the number is definitely not prime. If it returns true, the probability of an incorrect positive is 0.25^ntrials.

Print(s0) prints the string s0 to both text output fields. The print operation automatically adds an end-of-line character at the end. If you want to print individual characters, use print_with_style.

print_destination as integer sets the destination of print statements (0=both, 1=editfield in Main, 2=editfield in Text)

Print_with_style(s0, fontname, isize, color, Qitalic, Qbold) prints the string s0 using the font, size, color, and style requested. This option is more flexible than using set_text_style and print because this statement does not add an end-of-line- thus you can do each letter in a string using a different style.

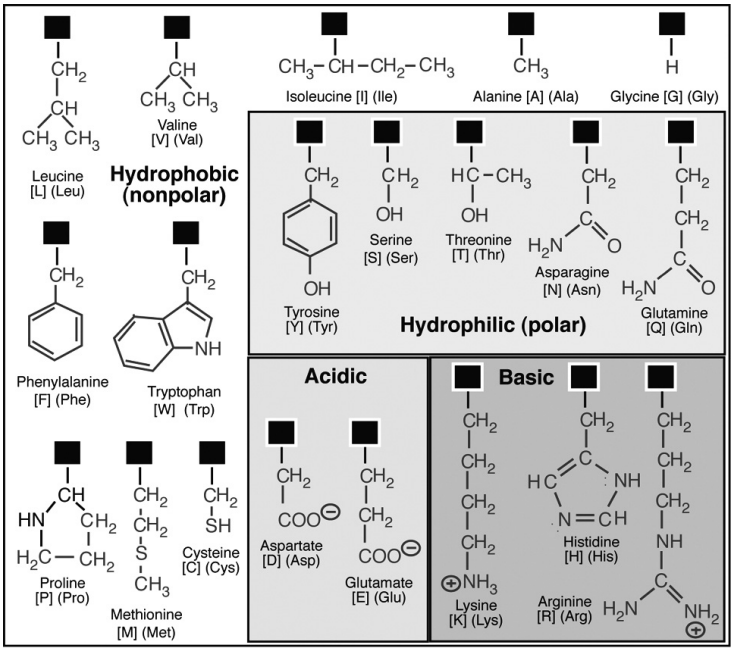
Private indicates that the properties or method within the module are only available to code within the module. Properties and methods of modules default to public if no designation is provided, so properties and methods must be so labeled if they are to be private.

Bioinformatics of Proteins and DNA

Protein bioinformatics functions are available for the analysis of primary sequences, evolutionary distances between sequence pairs, and generate pairwise and multiple alignments. In the following functions the parameters plinear() and paligned() are used to represent unaligned (no gaps) and aligned sequences, respectively. One can work with complete sequences or shorter segments as desired. The following example demonstrates input for two protein segments. One can mix upper and lower case characters to help mark features. In this case, the capital K is the lysine residue to which the retinal chromophore is attached.

```
// 7th transmembrane segments
plinear(1)="pifmtipaffaKtsavynpviyim"
pname(1)="rho: " // bovine rhodopsin pigment
plinear(2)="nietllfmvldvsaKvgfglillrs"
pname(2)="BR: " // bacteriorhodopsin
```

Each residue is represented by its single letter code, not the more common three letter codes. In addition, there are four letters used to represent uncertain assignments. The structures and codes are listed below:



Alanine	Ala	(A)
Arginine	Arg	(R)
Asparagine	Asn	(N)
Aspartic Acid	Asp	(D)
Cysteine	Cys	(C)
Glutamate	Glu	(E)
Glutamine	Gln	(Q)
Glycine	Gly	(G)
Histidine	His	(H)
Isoleucine	Ile	(I)
Leucine	Leu	(L)
Lysine	Lys	(K)
Methionine	Met	(M)
Phenylalanine	Phe	(F)
Proline	Pro	(P)
Serine	Ser	(S)
Threonine	Thr	(T)
Tryptophan	Trp	(W)
Tyrosine	Tyr	(Y)
Valine	Val	(V)

Asparagine or aspartic acid	Asx	(B)
Glutamine or glutamic acid	Glx	(Z)
Leucine or Isoleucine	Xle	(J)
Unknown	Xaa	(X)

A majority of the protein functions require the prior selection of a homology scoring method. The scoring (or substitution) matrices associated with each method provide a score for each possible pair of amino acids. This score is positive if the two amino acids are similar and negative if the two are different. One seeks to align proteins so that the similar or identical residues are lined up. This approach is appropriate both for structure-function analysis as well as phylogenetic (evolutionary) analysis. The function **protein_select_homology_method**(ioption) is used to select the method from one of five options: 0=Blosum40, 1=Pam250, 2=Pam250(TM) (TM = transmembrane proteins), 3=Blosum50, 4=Blosum62. The BlosumL (L=40, 50, 62) matrices are optimal for aligning proteins where structure and function are of primary interest. If the two proteins are significantly different and alignments involve multiple gaps, use the lower L values (e.g. Blosum40). In contrast, the Blosum62 matrix is more appropriate for aligning ungapped regions of highly similar proteins. The Pam250 matrices are the standard choice when phylogenetic alignment is important. This matrix is based on long term extrapolation of evolutionary relationships, and is optimal for directing the alignment of proteins which have distant but clear evolutionary relationships. The Pam250(TM) method has been optimized for trans-membrane proteins. In general, if the proteins you are aligning are very different, use ioption=0 or 1. If very similar, use ioption=3 or 4. Trial and error is often the best way to choose the scoring matrix.

Virtually all protein alignments require the introduction of gaps. The gaps are created so as to align smaller subsequences with high similarity, and one has to make a decision about how to score the gaps. All methods agree that gaps are bad, and hence there is a gap penalty assigned which is subtracted from the homology score when a gap is first created. This penalty is automatically assigned when **protein_select_homology_method** is executed, but the user can change it by assigning a new (positive) value to the variable **protein_gap_penalty**. The larger the number, the larger the penalty and the fewer the gaps in the alignment. Most investigators believe that the penalty for extending a gap should be smaller than the penalty for introducing a gap, and hence the use of affine gaps. The word affine, from the latin affinis or "connected with", is used to indicate that if two or more gaps are connected to each other, the scoring should be different. The method used here is that the gap penalty is calculated by the formula:

$$\text{Penalty}(N_{gaps}) = \textbf{protein\_gap\_penalty} + \textbf{protein\_gap\_extend\_penalty} * (N_{gaps} - 1)$$

Typical assignments for these variables are **protein_gap_penalty** = 6 and **protein_gap_extend_penalty** = 1, which means that there is a 6-fold greater penalty for opening a new gap as there is to extend the gap by one. There is strong evolutionary evidence to suggest that affine gaps are more realistic, but there is little evidence to

suggest what values should be assigned to the above two variables.

The following demonstrates the code used to generate the alignment of the 7th transmembrane segments of rhodopsin and bacteriorhodopsin. These two proteins have little if any evolutionary connection, nor do they share common functionality. However, both of these alpha helical segments bind retinal to the lysine residue marked with a capital K.

```
mthd=protein_select_homology_method(1)
protein_gap_penalty=4
k=protein_align(plinear(), paligned(), 0)
print(" linear gap "+mthd+" alignment score = "+str(k))
protein_print(paligned(), pname(), 2, 1, "Courier", 18)
print("----------------------------------------")

protein_gap_extend_penalty=1
k=protein_align(plinear(), paligned(), 1)
print(" affine gap "+mthd+" alignment score = "+str(k))
protein_print(paligned(), pname(), 2, 1, "Courier", 18)
```

Following is the output generated by the above code:

```
linear gap pam250 alignment score = 8
               10         20
 {1-28}         |          |
 rho: pifmtipaffa-Ktsa-v-ynpviyim-
 BR:  ni-etl-lfmvldvsaKvgfg-lillrs
-------------------------------------
affine gap pam250 alignment score = 16
               10         20         30
 {1-31}         |          |          |
 rho: pi----fmt--ipaffaKtsavynpviyim-
 BR:  nietllfmvldvsa---Kvg--fg-lillrs
```

The second alignment based on affine gap scoring is significantly better because the lysine residues to which the retinal chromophore are covalently bound are aligned. However, one should not assume that a higher score means a better alignment because affine gap-based alignment scores are almost always higher because of the increased degrees of freedom inherent in cheap extra gaps. Affine gaps are even more important when doing multiple alignments. The following is an example of the code and output for a short segment multiple alignment for six retinal proteins (two visual pigments, two archaeal pigments and two bacterial pigments).

```
protein_gap_penalty=4
protein_gap_extend_penalty=1
prof=protein_align_multiprofile(plinear(), paligned(), 6, 1)
k=protein_homology_score(paligned(), 6, 1)
print(" affine gap "+mthd+" alignment score = "+str(k))
paligned(7)=protein_gen_consensus(paligned(), 6)
pname(7)="consensus: "
protein_print(paligned(), pname(), 7, 1, "Courier", 18)
```

```
affine gap pam250 alignment score = 589
                    10        20        30
 {1-31}              |         |         |
 rho:        p-------ifmtipaffaKtsavynpviyim
 canary:     pltaa-------lpaffaKsatiynpiiyvf
 BR:         niet---llfmvl-dvsaKvg--fglillrs
 SR2:        ptvdvalivy--l-dlvtKvg--fgfialda
 GPR:        alnln--liyn-ladfvnKi--lfgliiw-n
 BPR:        nln----liyn-ladfvnKi--lfgliiw-n
 consensus:          ++  + +  +K    ++ +
```

The following 16 variables and functions are available in version 1.8.4. Additional bioinformatics functions are being added for version 1.8.5.

protein_align(plinear(), paligned(), ioption) **as double** returns the homology score between the two proteins after sequence alignment based on the scoring method selected using protein_select_homology_method. The original sequences without spaces are in plinear(1) & plinear(2) and the aligned proteins are returned in paligned(1) & paligned(2). Residues using single letter codes can be upper or lower case. The variable ioption determines how the gaps are handled. If ioption=0, a linear gap penalty is assigned based on protein_gap_penalty. If ioption=1, an affine gap penalty is assigned based on the formula: penalty(ngaps) = -protein_gap_penalty - (ngaps - 1)*protein_gap_extend_penalty.

protein_align_2to1(paligned()) **as double** aligns paligned(2) to paligned(1) without altering paligned(1). Returns a positive number if the alignment was improved in the process. Must verify that result is an improvement by checking if the score improved.

protein_align_by_feature(plinear(), paligned(), nproteins, ioption) **as boolean** aligns two or more proteins making sure that a designated residue in each protein (indicated by using a capital letter) is aligned. All other residues must be lowercase. Ioption assigns the gap penalty (0=linear gap penalty, 1=affine gap penalty)

protein_align_multiple(plinear(), paligned(), nproteins, ioption) carries out a multiple protein alignment of nproteins with primary sequences in plinear(1..nproteins). The aligned proteins are returned in paligned(1..nproteins). Gap method is selected by ioption (0=linear gap penalty, 1=affine gap penalty)

protein_align_multiprofile(plinear(), paligned(), nproteins, ioption) **as string** returns the profile used and creates the alignments for the linear sequences provided in plinear(). Alignments returned in paligned() for the selected option (0=linear gap penalty, 1=affine gap penalty) .

protein_best_score(plinear(), ioption) **as double** returns a reasonable estimate of the best homology score between two protein sequences that are not aligned. This routine scores but does not align. Ioption determines how gaps are handled: 0=normal, 1=affine. Call protein_select_homology_method(ioption) first, and assign protein_gap_penalty and protein_gap_extend_penalty (if selecting ioption=1).

protein_clean_alignments(paligned(), nproteins) cleans a set of aligned proteins by removing gaps that fill an entire column. This function is intended for those writing their own multiple protein alignment routines.

protein_distance(plinear(), ioption) **as double** returns the evolutionary distance between two proteins. Three methods are available and selected by ioption=0, 1 or >7. Ioption=0 selects Jukes-Cantor model (d = -(3/4)*log(1 - 4*f/3) where f = fraction of sites with different nucleotides. Ioption=1 selects Kimura model (d = -0.5*log(1-2P-Q)-0.25*log(1-2Q), where Q = fraction of transversions and P = fraction of transitions. Transversion refers to the substitution of a purine for a pyrimidine or vice versa and is more chemically significant and evolutionarily uncommon, than transitions. Ioption>7 selects Feng and Doolittle [d = -log(Seff) = -log[(Sobs - Srandave)/(Smax - Srandave)] where Sobs = the observed homology score for the two proteins, Smax = average of the scores with both proteins against themselves, Srandave = average score for multiple random proteins of same lengths as those under analysis. The accuracy is proportional to the number of random proteins, and this number is determined by Ioption. It is recommended that Ioption=16 to 32 but values as large as 1024 are possible. Designed for pam250, but will work with any homology method. Time for Feng and Doolittle method is proportional to Ioption^2. .

protein_draw_phylogenetic_tree(plin(), np, pnames() , px(), py(), pclr(), ioption) **as string** returns a string summary and draws an approximate phylogenetic tree based on a global Kimura distance optimization. The protein primary sequences are in plin(1..np), where np is the number of proteins. Short names of the proteins are in pnames(1..np), and if one is a profile, the profile should be called "prof" or "profile". After the relative positions of the proteins are determined, the positions are marked with a solid dot, and the names are printed at lower right. The positions of the names can be moved by setting the offsets into px(1..np) and py(1..np) where it is noted that 0, 0 is at upper left, so to move the y position down, a positive py(i) is required. The pclr(1..np) array sets the color that is used to mark the protein and name the protein. ioption ranges from 00 to 29 where the left most (tens) digit sets the separation error: 0[sqrt(delta)], 1[linear in delta], and 2[delta^2]. Use trial and error to select. The right most digit (ones) sets the minimum Kimura distance to be twice that value divided by 100. Thus, ioption=12 uses a positional error function that is linear in Kimura distance with a closest contact target of 0.04.

protein_gap_extend_penalty as double a positive value that represents the penalty associated with a gap extension by one unit. This value is initialized by protein_select_homology_method, but can be reassigned in the user program to observe the effect. This variable is only relevant to affine gap scoring where the gap penalty is calculated as:

penalty(ngaps) = -protein_gap_penalty - (ngaps - 1)*protein_gap_extend_penalty.

protein_gap_penalty as double a positive value that represents the penalty associated with the creation of a gap in an alignment. Although automatically assigned by the selection of a scoring matrix, it is often necessary to manipulate this value to optimize alignments.

protein_gen_consensus(paligned(), nproteins) **as string** returns the consensus analysis for two or more proteins. If all the residues in a column are identical, the consensus shows that residue. If all residues are the same and there is a single gap, a consensus is indicated nonetheless. If all the residues are similar, a + is shown. Otherwise the column is blank.

protein_gen_profile(paligned(), [nproteins]) **as string** returns a profile that provides a best fit to the aligned protein sequences in paligned(1...2) [or if nproteins is present, paligned(1..nproteins)] based on the previously assigned protein_select_homology_method(imethod).

protein_homology_score(paligned(), [nproteins], ioption) **as double** returns the homology score between two aligned proteins [or nproteins proteins] where gaps are indicated by - or ~. Residues can be upper or lower case, it matters not. ioption determines how gaps are handled: 0=normal, 1=affine, 2=normal+ignore end gaps, 3=affine+ignore end gaps

protein_print(paligned(), pnames(), nproteins, ioption, fontname, ifontsize) prints aligned protein sequences directly to the Main and Text editfields where paligned(1..nproteins) are the protein sequences. Short names for each protein are assigned to pnames(1..nproteins). Ioption=0 (no color) or 1 (color) simply prints the proteins. Ioptions=2 (no color) or 3 (color) prints two proteins with homology analysis in between.

protein_random(ilength, ioption) **as string** returns a random protein in which each amino acid residue has an equal but random probability of being present. If ioption=1 then probability of each residue is weighted by observed frequency.

protein_residue_codon_shift(s1, s2, np, nq) **as integer** returns the number of nucleotide changes associated with changing residue s1 to residue s2 (or vice versa). The number of transitions is returned in np, the number of transversions is returned in nq. The standard genetic code is used and codons are selected to minimize the number of bases switched.

protein_residue_composition(plin) **as string** returns an analysis of the protein composition in a list array in the order GAIVLFWPMCYSTNQDEKRHBZJX. The sequence of numbers gives the population for each of the 20 amino acids followed by the three indeterminate pairs (B, asparagine or aspartate; Z, glutamine or glutamate; J, Leucine or isoleucine) and X (unknown). If there are gaps, the gaps are randomly distributed as well, but the gaps generate an error statement because the number of slots is different.

protein_residue_pair_score(s1, s2) **as double** returns the homology score for the pair of residues in strings s1 and s2. Execute protein_select_homology_method(imethod) first.

protein_select_homology_method(ioption) **as string** selects the homology matrix based on the value of ioption:0=Blosum40, 1=Pam250, 2=Pam250(TM) (TM = transmembrane proteins), 3=Blosum50, 4=Blosum62. This function returns the name of the method for confirmation. This method must be called prior to doing any protein sequence analyses.

protein_shuffle(plin) **as string** returns a protein sequence of the same length and residue composition but with the residues randomized in terms of location. This only works for single letter code amino acids.

Private indicates that the property or method is only available to code within the same object.

Public indicates that the properties or method within the module are available to code outside of the module. Properties and methods of modules default to public if no designation is provided. Hence, you do not need to use this keyword to identify variables or functions that are global. However, if variables declared at the top of a class are involved, such public variables can only be accessed via using the dot extension from the instantiated class variable.

Q_afaos_excited_singlet_state as boolean Set this Boolean to true to calculate the properties of the lowest-excited singlet state using AFAOS (Averaged Field Approximate Open Shell) theory. This Boolean needs to be assigned prior to calling a CNDO or INDO calculation.

Q_damp_scf as boolean Set this boolean to true to average the Fock matrix from the previous iteration with the current Fock matrix. This helps to prevent oscillation of the electron density, and for certain molecules, improves convergence. Normally the user does not need to bother with this variable because the program will automatically turn this Boolean to true if oscillations are observed. However, there are molecules for which no convergence occurs in the absence of damping from the very first SCF iteration.

Q_double_buffer_graphics Set this boolean to true to double buffer all graphics. This can cause havoc on Windows computers and often fails in non-cocoa Mac systems. This option can also be set in preferences.

Q_force_unit_eigenvectors Set this boolean to true to seek unit eigenvectors so that each vector points to one and only one atomic orbital. This often works for atoms, and rarely works for molecules. It is included as a teaching tool, not a research option.

|MS| **Q_greater_than**(s1, s2) **as Boolean** returns true if s1>s2, where s1 and s2 are arbitrary precision string floats or integers.

|MS| **Q_less_than**(s1, s2) **as Boolean** returns true if s1<s2, where s1 and s2 are arbitrary precision string floats or integers.

Q_histogram_use_external_colors set this boolean to true if external colors are to be read in to color the histogram regions. See plot_histogram for more information.

Q_monitor_keyboard as Boolean When set to true all key presses are monitored and are available by calling keyboard_monitor_activity().

Q_mouse_data_available set to true by system when mouse data are available.

Q_phidget_raw set to true to collect voltage data from phidget cards using the raw voltage (0-4095) rather than the scaled (0-5) voltage.

Q_plot_angular_mode set to true by user to indicate that the x axis is in degrees. The plot routine will select ticks to conform with degrees.

Q_plot_fill set to true by user to indicate that the plot should be filled to zero, or to the value, **Yfill_reference_value**, if that variable is set in the program prior to the plot.

Q_plot_log_X set to true to use a log scale for the x axis.

Q_plot_log_Y set to true to use a log scale for the y axis.

Q_plot_zero_line as boolean set this variable to true if the Y=0 line is to be indicated on the subsequent plot. This variable is linked to plot_data, and should be assigned prior to calling this method.

Q_plot_reverse_x_axis as boolean If set to true, the x-axis is reversed with x2 on left and x1 on right. This variable is linked to plot_data, and should be assigned prior to calling this method.

Q_show_mouse_rectangle when set to true, a red rectangle shows the mouse drag region in the canvas of the graphics panel.

Q_use_external_parameters when set to true read the CNDO parameterization from the spreadsheet. To load the standard parameters into the spreadsheet, execute atom_property_cndo(0, ""S"", 0, S1) and then modify by hand. One can then store the modified parameters as a data set for future use. The program only reads the parameters in the first 7 columns (see discussion under molecule_run_CNDO). These new parameters will also be used in the INDO calculations, although it is not possible to change the one-center correction intergrals as any attempt to do so might damage invariance.

ran2 as double returns a random number between 0 and 1 (exclusive of end values) based on the shuffle methods of L'Ecuyer and Bays-Durham. This generator has excellent statistics and a very long period (>2E18).

ran2_seed assigns as integer reseeds the ran2 random number generator of L'Ecuyer and Bays-Durham. Use a large integer.

random_gaussian as double returns a random number generated from a Gaussian distribution with a mean of zero and a standard deviation of 1.0 There are three independent methods of generating random numbers. Both rnd and random_number generate random numbers between 0.0 and 1.0, but random_number can have a seed value assigned using random_seed so that the random numbers can be repeated as identical sets in subsequent runs. This is important for some Monte Carlo calculations. Arpec_random is the third method, and it can also be provided with a seed value.

random_integer(ilow, ihigh) **as integer** returns a random integer from ilow to ihigh inclusive. There are three independent methods of generating random numbers. Both rnd and random_number generate random numbers between 0.0 and 1.0, but random_number can have a seed value assigned using random_seed so that the random numbers can be repeated as identical sets in subsequent runs. This is important for some Monte Carlo calculations. Arpec_random is the third method, and it can also be provided with a seed value.

random_integer_sequence(nseq(), nlow, nhigh) generate a random integer sequence from nlow to nhigh so that each value appears once but only once. The result is returned in nseq(0..ns) which is redimensioned to exactly the right size. Note that the list starts at nseq(0) and will stop at nseq(nhigh-nlow).

random_number as double returns a random number between 0.0 and 1.0. There are three independent methods of generating random numbers. Both rnd and random_number generate random numbers between 0.0 and 1.0, but random_number can have a seed value assigned using random_seed so that the random numbers can be repeated as identical sets in subsequent runs. This is important for some Monte Carlo calculations. Arpec_random is the third method, and it can also be provided with a seed value.

random_seed assigns as double or as double sets or returns the current seed of the random generator. If setting the seed, real numbers between 10^7 and 10^8 are recommended.

read_binary_file(filepath, [filecontents] or [a1(), n1] or [a2(), n1, n2]) **as integer** read the string or doubles in the binary file designated by filepath. The filepath represents a path within the same folder as Scriptor. For example, filepath="data_sets:bf12" read the data in the file "bf12" inside the folder "data_sets". This routine can read a string, or a double in the form of a one-dimensional array a1(0..n1) or a two-dimensional array a2(0..n1, 0..n2). Returns the file length in bytes (0=failure).

real(s1) **as string** returns the real part of an arbitrary precision complex string number.

Redim redimensions a previously declared array to the specified size. One cannot use this statement to change the type or create a new variable. To redim the array x(2) to size n, use Redim x(n).

Redim_multiple(n, a(), b(), [c(), [d(), [e()]]]) redimension between 2 and 5 double arrays where c(), d() and e() are optional. All are dimensioned to size (n).

Rem everything that follows is a comment. When Scriptor is used for teaching, the REM statement is reserved for use by the instructor to add comments to a student's program. The Rem statements can be removed by the student using the Debug menu, but hopefully after reading them.

Replace(source, substring, replacement) **as string** replace the first occurrence of the substring with replacement in the source.

ReplaceB(source, substring, replacement) **as string** as above for a byte character string but now the replacement is case sensitive.

ReplaceAll(source, substring, replacement) **as string** replace all occurrences of the substring with replacement in the source.

ReplaceAllB(source, substring, replacement) **as string** as above for a byte character string but now the replacement is case sensitive.

Return when encountered in a function, returns the value that follows and exits. When encountered in a subroutine, causes an immediate exit from the subroutine.

RGB(ired, igreen, iblue) **as color** set the color using the RGB method (params 0…255).

Right(s0, n) **as string** returns the n characters from the right of s0.

RightB(s0, n) **as string** returns the n bytes from the right of s0.

rnd as double random double from 0.0 to 1.0. This function is an internal function that is provided by the compiler, and is very fast, but not as competent as the other random number generators provided by MathScriptor.

Round(x) **as double** returns the rounded integer equivalent of x (1.49→1, 1.51→2)

Round_to_precision(x, ndigits) **as double** rounds the number x to the number of significant digits specified by ndigits.

Round_to_precision(s1, nsd) **as string** rounds the arprec string number s1 to the number of significant digits specified by ndigits, with the result returned in an arprec string.

RTrim(s0) **as string** returns the string s0 with trailing blank spaces removed

save_binary_file(filepath, [filecontents] or [a1(), n1] or [a2(), n1, n2]) **as Boolean** is no longer supported. Use binary_file_write, which does the identical task with greater flexibility.

save_spreadsheet(file_name) **as boolean** save the current spreadsheet using the file name file_name in the data_sets folder. You can create a folder inside data_sets and place the spreadsheet inside that folder by specifying folder_name\file_name (or on a Mac, folder_name:file_name). If using a folder, it must be created first. This statement cannot create a new folder. Returns true is successful, otherwise false.

Save_user_text_file(filename, filecontents, Qreplace) **as Boolean** store the string in filecontents into a file called filename inside the user_files folder, and replace if Qreplace is true. Returns true if process worked.

Select Case the first line of a select case construct. The last line is end select.

Set_graphics_slider(ipercent_graphics) programmatically controls the slider on the Main panel such that ipercent_graphics controls the percentage of space allocated to the graphics window versus the output text field.

Set_text_style(fontname, ifontsize, fontcolor, Qitalic, Qbold) sets text output style for the output canvasses in both Main and Text panels. Further print statements will use this style until a new one is chosen either programmatically or from the font/size menus. This statement is the only way to generate different colors, bold or italic font output. If the font you request is not present, no error is generated, and nothing happens.

Set_to_data shifts visibility, mouse manipulation and keyboard input to the data (spreadsheet) panel.

Set_to_graphics shifts visibility, mouse manipulation and keyboard input to the graphics panel and updates active_canvas so that it equals 2.

set_window_size(byref nx, byref ny) The integers nx and ny are used to set the desired scriptor window width (nx) and window height (ny). The method prevents a size less than 800x600 nor larger than 95% of the computer screen. The method also returns the final size in the byref parameters. Setting the parameters to 0, 0 leaves the window unchanged and returns the current size in nx and ny.

SF1(real_number) converts a single or double precision variable to a string for printing. Unlike the str() function, which returns 6-7 significant digits, or convert_to_string() which returns all significant digits, this function returns roughly 12-13 digits with the digits separated into groups of three following the conventions recommended by the American Physical Society. Examples are presented below:

```
0.001 161 409 727 843
6.022 141 790 000 000 e+23
8987 551 787.399 9996
```

Note that if a single digit would be printed alone, it is joined with the preceding or subsequent triplet. The goal of this function is to present numbers in a scientific format that is easiest to read. The short name is to facilitate the use of the function inside a print statement.

SF2(arprec_string, ndigits) **as string** returns an arprec string in the form 1.23456...[nhidden]...123456 where ndigits assigns the number of digits shown at the extremes (example shows ndigits=6) and the variable nhidden displays the number of digits hidden from view.

Show_progress_bar(ipercent_progress) displays progress on the Main Panel or Music Panel progress bars using ipercent_progress (0..100) as the variable.

Show_progress_line(s0, [fontname, fontsize]) Displays the string s0 in the input line using the default fontname and fontsize, but the user can override the defaults by explicitly specifying both the fontname and fontsize.

Sin(x) **as double** returns the sine of x assuming x is in radians. If you want to input x in degrees, use sin(x*const_degree).

Sinh(x) **as double** returns the hyperbolic sine of x.

Spreadsheet_add_column(header, colwidth, ioption) add column at far right with width colwidth (0 uses the width from the column at left) and ioption sets alignment (0=default, 1=left, 2=center, 3=right, 322=same as column at left). Max=64.

Spreadsheet_add_row adds a new row at bottom of spreadsheet. The number of rows in a spreadsheet is limited only by the size of the computer memory.

Spreadsheet_cell(irow, icol) **as string**, or **assigns as string** read or assign the string data in spreadsheet cell irow, icol.

Spreadsheet_column_width(icol) **assigns as integer** sets the column width of column icol in pixels.

Spreadsheet_create(nrows, ncols, colheads(), colwidths() , ialign) create a new spreadsheet containing the number of rows (nrows), columns (ncols<=64), column headers (colheads(1..ncols)), column widths (colwidths(1..ncols)) and alignment based on ialign (0=default, 1=left, 2=center, 3=right)

Spreadsheet_delete_column(idel) delete column idel and shift any columns to the right of idel to the left.

Spreadsheet_delete_row(idel) delete row idel and shift any higher numbered rows up. Remember that the top row is number 1, not number 0, and that the headers are separate entities and cannot be deleted. The headers can be changed by using the Spreadsheet_header() function (see below).

Spreadsheet_eomccsd_open(filename) **as boolean** open the filename in user_files as an gaussian eomccsd output (.log) file and place results in the spreadsheet. Return true if the filename was valid. If filename="unknown" then open a user dialogue.

Spreadsheet_filename as string [or assigns as string] sets or reads the filename of the current spreadsheet. This string is located in the lowest right editfield in the data sets panel. This variable is useful for checking if a spreadsheet has been loaded into memory, although it is not foolproof in that this function can set the name of any spreadsheet to a given name.

Spreadsheet_flip_columns(irow, jrow) flip the two columns in the spreadsheet by moving all elements in both columns to the corresponding locations in the second column. No elements are lost in the process.

Spreadsheet_flip_rows(irow, jrow) flip the two rows in the spreadsheet in terms by moving all column elements to the corresponding locations in the second row. No elements are lost in the process.

Spreadsheet_gaussian_cis_open(fname) **as Boolean** open the gaussian CIS excited state output file, fname, inside of user_files and extract the excited state symmetries, transition energies and oscillator strengths and fill a new spreadsheet with the information. The method returns true if successful.

Spreadsheet_gaussian_cis_transitions_open(fname) **as boolean** opens the gaussian output file designated by the string fname inside the user_files folder and transfers the information to the spreadsheet. This method is designed to collect the CIS interstate transition energies generated by using the keyword AllTransitionDensities. As many as 64 states can be collected.

Spreadsheet_gaussian_open(filename) **as Boolean** open the filename in user_files as a gaussian output (.out, .log) file and place results in the spreadsheet. Return true if the filename was valid. Only the optimized coordinates and the Mulliken charges are collected. Although this command will also work on a SAC-CI calculation output, if the excited state information is desired, use Spreadsheet_sacci_open() instead.

Spreadsheet_gaussian_orbitals_open(fname) **as Boolean** open the gaussian output file, fname, inside of user_files and extract the orbital energies, symmetries and occupation and fill a new spreadsheet with the information. Col 1 is the molecular orbital number, Col 2 is the energy in Hartrees, Col 3 is the symmetry and Col 4 is the occupation (only reliable if a ground state calculation was done, revise as necessary for excited states).

spreadsheet_gaussian_TDDFT_open(fname) **as Boolean** open a gaussian TDDFT excited state output file, fname, inside of user_files and extract the excited state symmetries, transition energies and oscillator strengths and fill a new spreadsheet with the information. The method returns true if successful.

Spreadsheet_header(icol) **as string**, or **assigns as string** read or assign the header for column icol.

Spreadsheet_insert_column(column_at_right, header, icolwidth, ialign) insert a column to the left of the column_at_right with header, width (colwidth) and alignment.

Spreadsheet_lock Executing this command prevents a user from sorting the spreadsheet by clicking on column headers or changing individual cells by direct editing. This is a safety feature, but the programmer should understand that the user can override this command. Sorting can be allowed by selecting the menu item “make all columns header click sortable”. Direct editing of the cells can be activated by selecting the menu item “allow direct editing of cells”. Thus, the Spreadsheet_lock command only serves to prevent accidental manipulation. A determined user can still modify the spreadsheet.

Spreadsheet_maxcolumn as integer returns the current number of columns. Note that the number of columns that are visible in the spreadsheet will often be larger than the number of columns that are under programmatic control. Use this variable to find out how many columns are available for manipulation.

Spreadsheet_maxrow as integer returns the current number of rows. Note that the number of rows that are visible in the spreadsheet will often be larger than the number of rows that are under programmatic control. Use this variable to find out how many rows are available for data manipulation.

Spreadsheet_mndoci_open(filename) **as Boolean** open the filename in user_files as an mndoci output (.cio) file and place results in the spreadsheet. Return true if the filename was valid. This command also works on mndoci arc (.cia) files, but produces a smaller spreadsheet with fewer entries.

Spreadsheet_pdb_open(sfilename) **as boolean** open the filename in user_files as a protein data bank (pdb) file (.pdb) and place results in the spreadsheet. Return true if the filename was valid. If filename="unknown" then open a user dialogue.

Spreadsheet_redcolumn set to the column of the cell contents you want marked in red.

Spreadsheet_redrow set to the row of the cell contents you want marked in red.

Spreadsheet_row_height assigns as integer assign the pixel height of the rows.

Spreadsheet_sacci_open(filename) **as Boolean** open the filename in user_files as a sacci output (log) file and place results in the spreadsheet. Return true if the filename was valid.

Spreadsheet_set_row_colors(carray(), ncolors) assign the header and row colors of the spreadsheet where ncolors is the number of assigned colors and carray(0..ncolors) is the array of colors, where the zeroth element sets the header background color. To assign default colors, pass a valid array but filled with black. To return to pure white, call this method with ncolors=0.

spreadsheet_spike_open(filename [, t1, t2]) **as boolean** opens up a spike (neural signal analysis) file and places the contents into the spreadsheet. Returns true if successful. If t1 and t2 are included, only data with times in range t1<=time<=t2 are added to the spreadsheet.

Spreadsheet_to_buffer(rowmin, colmin, rowmax, colmax, linewidth, row_separation, col_separation, column_align, fontname, fontsize) draws the spreadsheet into the buffer from cell(rowmin, colmin) through cell(rowmax, colmax). Row_separation and Col_separation add whitespace between the rows and columns, respectively, and are essential for preparing tables that are readable. Column_align sets the alignment of the cell contents (1=left, 2=centered,

3=right). The cell contents and headers are drawn using the fontname and fontsize specified.

Spreadsheet_update_header(icol **as integer) assigns as string** replace the header of column icol with the assigned string.

sqrt(x) **as double** return the square root of x

Static declares and dimensions variables which are retained in memory after the method goes out of scope. Use in place of dim. Variables declared inside of methods remain private to the method when the method is reentered. Note that static variables only remain in memory during the Run session, and when the program stops, the values are lost. If the goal is to retain variables between runs, use common_blocks, disk files or the spreadsheet.

Step optional step size assignment in for..next loop as in for i=2 to 10 step 2. (Formally, step should always be positive. Backward looping is indicated by using downto with a positive step as in for i=10 downto 2 step 2. However, versions of Scriptor beyond 1.8.2 now accept negative step sizes in for-to loops [e.g. for i=10 to 2 step –2.])

Str(x) **as string** converts the integer or real variable x into a string for printing. If converting other objects to strings, or to convert a number while retaining full accuracy, use convert_to_string(any_varible).

StrComp(s1, s2, imode) **as Integer** compares two strings, s1 and s2, and returns –1 if s1<s2, 0 if s1=s2 and +1 if s1>s2. The comparison mode is binary (case-sensitive) if imode=0 and text (lexicographic) if imode=1.

String_block_convert(string_data, string_in_hex, string_in_binary, integer_array(), double_array()) Converts a plain string or a set of bytes in hexadecimal format into the other three representations based on each variables representation in memory. This function is useful for using or writing cypher codes. All five parameters are byref but the user must select one of the strings as the independent variable. The other variables will return the appropriate value based on how that variable is represented in memory. Arrays are redimmed to a size appropriate to hold the data and the number of elements can be obtained using ubound.

String_countfields_quoted(s0, sepstr) **as integer** countfields in string based on sepstr being the field separator. Ignores separators within quoted fields.

String_countfields_regex(s0, pattern) **as integer** count the number of occurrences of a RegEx pattern within a string.

String_decipher_blowfish(string_to_decipher, key_string) **as string** decipher the string_to_decipher using the key_string and the blowfish algorithm. The string_to_encipher will have spaces added at the end to pad the entire string so that it has a length that is a multiple of 8 bytes. These spaces will be present in the deciphered version. This fact must be kept in mind when comparing the original with the deciphered version.

String_decode_base64(string_to_decode) **as string** encodes an arbitrary string into a 64-character alphabet composed only of printable ascii characters. Four bytes of input are converted to three bytes of output. Spaces and linebreaks are ignored.

String_decode_case(s0) **as string** undoes the encoding of case done by string_encode_case, resulting in the original string.

String_editdistance(s1, s2) **as integer** return the Levenshtein distance, aka the edit distance, between the two strings. This number equals the smallest number of insertions, deletions or changes necessary to convert one string into the other.

String_encipher_blowfish(string_to_encipher, key_string) **as string** encipher the string_to_encipher using the key_string and the blowfish algorithm. The string_to_encipher will have spaces added at the end to pad the entire string so that it has a length that is a multiple of 8 bytes. These spaces will be present in the deciphered version. This fact must be kept in mind when comparing the original with the deciphered version.

String_encode_base64(string_to_encode, line_wrap_length) **as string** ncodes an arbitrary string into a 64-character alphabet composed only of printable ascii characters. Three bytes of input are thus converted to four bytes of output. The line_wrap_length controls the maximum number of characters per line and should be set to 76 to conform with MIME Encoding. 0 requests no linebreaks.

String_encode_case(s0) **as string** encodes the uppercase/lowercase state of each letter using ^ as the marker.

String_hexbyte(s0) **as string** converts each byte of the string into the corresponding pair of hexadecimal digits separated by spaces.

String_instr_quoted(kstart, source, sfind) **as integer** identical to InStr function, except that it ignores any occurrence of "find" within double quotes. The starting position, kstart, is required.

String_instr_reverse(kstart, source, sfind) **as integer** similar to InStr, but searches backwards from kstart (or if kstart = -1, then from the end of the string). If find can't be found, returns 0.

String_join_quoted(sarray(), sdelim) **as string** join the given strings with a delimiter sdelim, just like RB's intrinsic Join method, except that if any of the fields contains the delimiter, that item will be surrounded by quotes in the output. See string_split_quoted for the inverse function.

String_line_ending(s0) **as string** returns the end of line character used in the string s0. If none is found, const_eol is returned.

String_metaphone(s0, sprimary, salternate) compute the Double Metaphone of the source string, an algorithm that finds one or two approximate phonetic representations of a string, useful in searching for almost-matches.

String_nthfield_quoted(source, sdelim, i) **as string** equivalent to nthField() function, but respects quoted values.

String_pixel_height(s0, fontname, fontsize, Qbold, Qitalic) **as integer** returns the number of graphics pixels of height the text in s0 takes when drawn using draw_string(). The value includes whitespace above and below the characters. Keep this string short to avoid wrapping.

String_pixel_width(s0, fontname, fontsize, Qbold, Qitalic) **as integer** returns the number of graphics pixels of width the text in s0 takes when drawn using draw_string().

String_random(i, [schars]) **as string** return a string of random ascii characters made up of A-Z, 0-9, or if the optional parameter schars is included, then the characters will be selected randomly from the list.

String_random_quotation as string returns a random quotation with attribution each time it is called. There are roughly 270 quotations, from serious to humorous, stored in condensed form in the text file qbf.txt inside the help folder. If this file is missing, the string_random_quotation function will return random_quote failure: Could not find help_files\qbf.txt?

String_RegEx_options(option_name, option_integer) sets the following options for regular expression (regex) routines:

"case", 1 = search is case sensitive (0=case insensitive, default)

"doteol", 1 = the period matches every character include eol character(s) (0=dot ignores eol character, default)

"greedy", 1 = search finds everything from first to last delimiter (default) (0=localize search)

"eoltype", sets end of line type: 0=any, 1=const_eol, 2=mac, 3=win32, 4=unix (0=any=default)

"replaceall", 1 = replace all occurrences of match (0=only the first, default)

"ignoreEOLs", 1 = treat entire text as a single line (ignore line endings) (0=default)

String_RegEx_replace(source_string, search_pattern, replace_pattern, kstart, kmatch)
as string Uses regular expressions (Perl) to search for the match_pattern which, when found, is replaced by a string defined by the replace_pattern

pattern codes (to be used only in replacement string):

$` Replaced with the entire target string before match
$& The entire matched area (this is identical to \0 and $0)
$' Replaced with entire target string following the matched text
\xnn Replaced with the character represented by nn in Hex
\nnn Replaced with the character represented by nn in Octal
\cX Replaced with the character that is the control version of X

wildcards:

. Any single character except const_eol (end of line character(s))
^ beginning of a line unless used in a character class (below)
$ end of a line unless used in a character class (below)

character classes:

[aeiou] Any one of the characters a, e, i, o, u
[^aeiou] Any character except a, e, i, o, u
[a-e] Any character in the range a-e, inclusive
[a-zA-Z0-9] Any alphanumeric character
[[] Finds a [
[]] Finds a]
[a-e^] Finds a character in the range a-e or the caret character
[a-c-] Finds a character in the range a-c or the - sign

Special Character Matches:

\r Line break (return)
\n Newline (line feed)
\t Tab
\f Formfeed (page break)
\xNN Hex code NN
\s Any whitespace character (space, tab, return, linefeed, form feed)
\S Any non-whitespace character
\w Any “word” character (a-z, A-Z, 0-9, and _)
\W Any “non-word” character (All characters not included by \w)
\d Any digit [0-9]
\D Any non-digit character

Repetition Character Matches

* Zero or more characters
.* finds an entire line of text, up to but not including const_eol
+ One or more consecutive characters
[0-9]+ finds a string of one or more consecutive numbers, such as “16238”
? Zero or one characters

String_RegEx_search(source_string, search_pattern, kstart, kmatch) **as string** search for the pattern in the source string using regular expressions (sometimes called Perl). The following wildcards and codes are used in the search_pattern:

. Matches any character except newline
[a-z0-9] Matches any single character of set
[^a-z0-9] Matches any single character not in set
\d Matches a digit. Same as [0-9]
\D Matches a non-digit. Same as [^0-9]
\w Matches an alphanumeric (word) character — [a-zA-Z0-9_]
\W Matches a non-word character [^a-zA-Z0-9_]
\s Matches a whitespace character (space, tab, newline, etc.)
\S Matches a non-whitespace character
\n Matches a newline (line feed)
\r Matches a return
\t Matches a tab
\f Matches a formfeed
\b Matches a backspace
\0 Matches a null character
\000 Also matches a null character because of the following:
\nnn Matches an ASCII character of that octal value
\xnn Matches an ASCII character of that hexadecimal value
\cX Matches an ASCII control character
\metachar Matches the meta-character (e.g., \, ., |)
(abc) Used to create subexpressions.\1, \2, … Matches whatever first (second, and so on) of parens matched
x? Matches 0 or 1 x's, where x is any of above
x* Matches 0 or more x's
x+ Matches 1 or more x's
x{m, n} Matches at least m x's, but no more than n
abc Matches all of a, b, and c in order
a|b|c Matches one of a, b, or c
\b Matches a word boundary (outside [] only)
\B Matches a non-word boundary
^ Anchors match to the beginning of a line or string
$ Anchors match to the end of a line or string

String_repeat(s0, ntimes) **as string** return a string containing ntimes repeats of s0.

String_replace_lineendings(s0, sline_ending) returns s0 with all of the line endings replaced with sline_ending.

String_reverse(s0) **as string** return s0 with all of the characters in reversed order.

String_show_gremlins(s0) **as string** return s0 with all the control characters replaced with printable ascii control names in bra-kets (e.g. chr(9) is replaced with <tab>).

String_soundex(s0) **as string** return the sounded code for the string s0 (first letter followed by a three digit numeric code).

String_speak(talk_string, Qinterrupt) use the system voice to speak talk_string and interrupt prior speaking if Qinterrupt is true. Selecting the system voice is computer and operating system dependent. This function usually does not work on unix platforms.

String_split(source, sdelim) **as string** split source string up into sarray() based on delimiter sdelim as follows: sarray=string_split("1, 2, 3, 4, 5", ", ") Note that sarray is redimensioned to equal the size necessary and that the first element is put into sarray(1) (sarray(0) is set to a null string).

String_split_by_regex(s0, spattern) split string s0 into fields delimited by the regular expression, spattern.

String_split_quoted(s0, sdelim, Q) **as string**() split s0 into fields based on sdelim respecting the quotes, and if Q is true, remove the quotes.

String_time_and_date as string returns the time and date when the program was first run in format: 9:18:44 AM Friday, December 10, 2004
Update the date to the current date and time by calling update_time.

String_zap_gremlins(s0) **as string** remove all control characters in s0 except for the local end-of-line characters that are present.

String_zap_multiple_spaces(s0) **as string** replace all contiguous spaces (2 or more) into single spaces.

Sub sname(..) start of a subroutine where the name is sname. Following the name come the parameters in parentheses. Subroutines can only returned modified variables that are preceded by the keyword “byref”. However, arrays are automatically passed byref. Subroutines within a class definition with subnames `constructor` or `destructor` are executed upon instantiation of the class variable (constructor) or when the class variable goes out of scope (destructor). Note that `sub constructor` is not called by the system in the event that the class is instantiated with initialization parameters. If the first parameter is preceded by "extends" then the subroutine is called as an extension of a variable of the type specified following the extends. The subroutine name is then added following a period to the right of the variable upon which it operates, or extends.

Swap(v1, v2) or **Swap**(v1(), i1, i2) swaps the values in two double or integer variables or two array elements of a double or integer one-dimensional array. If swapping array elements, the entire array is passed and the variables i1 and i2 represent the elements to be swapped.

System_compiler_optimization_level as integer read only variable that returns the optimization level as set under the compiler menu.

System_compiler_version as string returns the version of the internal XojoScript compiler that is being used to compile the Scriptor/MathScriptor code.

System_compile_time as double read only variable that returns the most recent compile time in microseconds.

System_computer_information as string returns information on the computer that is being used to run Scriptor. Sample information for a Mac (left) and a Windows (right) computer is listed below:

Mac OS Version: Mac OS X 10.9.5
Gestalt: Mac OSX Mavericks 10.9.5
CPU: Intel(R) Core(TM) i7-3820QM CPU @ 2.70GHz

Total physical RAM: 16.000 GB
Unused (available) physical RAM: 0.148 GB
Mac Serial Number: C02J6453DKQ5
Mac Model: MacBookPro10, 1
Mac UUID: F0761E5C-C99E-5380-9CC1-7A2546CB7090
User name: Sabrina Colchester
Computer name: Sabrina's MacBook Retina
Machine ID:
215F87D0C35BDA5008BB5C775BA0BDF2
Mac Address: 20:C9:D0:43:AC:BB
Mac VRAM Size: 256 MegaBytes
Mac has Hardware accelerated CoreImage: true

Windows OS Version: Windows 7 Service Pack 1 (Build 7601)
Gestalt: Windows 7 Build 7601 Service Pack 1
Major version number of the operating system: 6Build Number: 7601
Service Pack Information:
Service Pack 1
Windows Product Key:
BBBBB-BBBBB-BBBBB-BBBBB-BBBBB

CPU: Intel(R) Core(TM) i7-3820QM CPU @ 2.70GHz

Total physical RAM: 4.000 GB
Unused (available) physical RAM: 2.104 GB
User name: sabrina
Computer name: SABRINACOLCHESTER55FD
Machine ID: 14EA53EED7342711381BD8710310D8A2

System_convert_filename(fname) **as string** returns the filename with separators converted to the local system environment.

System_fontname(ifontnumber) **as string** returns a valid fontname for font number ifontnumber. Use the system variable, system_number_of_fonts to discover the number of available fonts.

System_fontname_label as string returns a fontname that is appropriate for labeling plots.

System_fontname_mono as string returns a valid monospaced (non-proportional) font that is available on the current system. If there are none to be found, a null string is returned. Andale Mono, Courier, Courier New, Monaco, and Prestige are examples of mono-spaced fonts.

System_fontname_sans as string returns a valid sans serif font that is available on the current system. If there are none to be found, a null string is returned. A sans serif font is one which lacks the small features at the edges of the letter, called "serifs", which were added to make type more readable. Helvetica, Arial, Futura, Geneva, Gill Sans, Lucida Sans, Impact, Verdana are common sans serif fonts. The "sans" adjective is French for "without".

System_fontname_serif as string returns a valid serif font. If there are none to be found, a null string is returned. A serif font is one which has non-structural details on the ends of the character strokes that were originally designed to make the typefaces easier to read, particularly when the type was small. Times, Times New Roman, Palatino, New Century Schoolbook, Garamond, Bookman, Antiqua are examples of serif fonts.

System_font_available(fontname) **as Boolean** returns true is the font called fontname is available in the system's font folder. Fontname is a string and must appear in quotes.

System_number_of_fonts as integer set by the system at startup to the number of fonts installed on the computer. You can access the names of the fonts by calling system_fontname(ifontnumber), where ifontnumber<=this integer.

System_OS as string returns the operating system currently in use: Options are: Windows (win32), Windows (win64), Mac Classic/non-PPC, Mac Classic/0S9, Mac OSX (Carbon), Max OSX (Cocoa) or Linux. Set by the system at startup. Can be used in conditionals to optimize the performance of a program on different platforms. Current versions of Scriptor do not run on Mac Classic.

System_heap_memory as integer returns the amount of memory currently allocated to the MathScriptor memory heap. This number includes all of the memory used to allocate the program and its variables, but does not include memory assigned to large arrays. which is allocated by the operating system to shared memory space.

System_number_of_fonts as integer set by the system at startup to the number of fonts installed on the computer. You can access the fonts by using System_fontname(ifontnumber).

System_scriptor_version as string returns the version of Scriptor (MathScriptor) that you are using.

System_verbosity as integer User set variable that determines the amount of printed output to the Main text window generated by internal functions. There are five levels available: 0= no printing except for compiler errors
1= include run time errors
2= include run time warnings
3= include diagnostic messages (default)
4= include commentary

Tan(x) **as double** returns the tangent of x assuming x is in radians.

Tanh(x) **as double** returns the hyperbolic tangent of x.

Thread_compiletime_error global string that will contain the compile time error in the event the thread_launch process fails due to an error in the thread code.

Thread_evaluate_string_expression(s1, ioption) **as string** uses a thread to evaluate the equation in the string s1 and returns the result as a string number. If ioption=0, only the result is returned. If ioption=1, the script that was executed is also returned in the returned string. Any errors are reported in the main editfield.

Thread_launch(sthread or slines()) **as Boolean** launches the script in the string sthread or in the string array, slines(1..n). The thread must be self stopping, but the program can check for user_action() and/or check_for_stop_button. Print statements can generate output, which will be placed in thread_print_output. Errors that are found are returned in thread_compiletime_error or thread_runtime_error. The thread can also read input placed in thread_input_string by using the input(") statement. Any prompts in the quotes will be ignored.

Then part of the if…then statement.

Ticks as integer time in 60ths of a second since computer was turned on.

Titlecase(s0) **as string** returns the string s0 with the first letter of each word capitalized

To part of the For statement construct(e.g. For i=1 to 10)

Trim(s0) **as string** removes leading and trailing spaces (blanks) from string s0

True is the opposite of false, and is one of the two possible states of a Boolean expression or variable.

Type(variant extension) **as integer** this extension (usage = variant_variable.type) returns the integer ID that identifies the properties of the variant: ID = 0(nil), 2(integer), 3(Int64), 4(single), 5(double), 6(currency), 8(string), 11(boolean), 16(color), where the variable type or property is listed in parentheses.

Ubound(array) **as integer** the index of the last element in a one-dimensional array

Ubound(array, i) **as integer** the index of the last element of a multidimensional array for the ith dimension where i=1 for the first dimension, i=2 for the second, etc.

Until mediates do until and/or loop until statements.

Update_time updates the time_and_date string to the current time and date based on the system clock. The absolute accuracy is determined by the accuracy of the system clock on the computer. The relative accuracy is limited by the internal clock of the computer, but is typically a few microseconds/day.

Uppercase(s0) **as string** converts all letters in the string s0 to uppercase.

Val(s0) **as double** converts the leading numbers in a string to a numerical value. This command is limited and cannot handle scientific notation. For any complicated string, use Value() instead. It is much slower, but more flexible.

Value(s0) **as double** converts the leading numbers in a string to a numerical value, but has the additional capability of handling a scientific value (1.234e12) or a number with commas (1, 234, 567.89).

iValue(s0) **as integer** as above but after conversion, rounds to the nearest integer.

Variant_type(v0) **as integer** returns the integer ID that identifies the properties of the variant, v0; ID = 0(nil), 2(integer), 3(Int64), 4(single), 5(double), 6(currency), 8(string), 11(boolean), 16(color), where the variable type or property is listed in parentheses. This same information can be generated by using the extension .type.

variational_min(x(), y(), n, kopt) **as double** returns the value of x (xbest) for which y is a minimum for a data set x(1), y(1), x(2), y(2) ... x(n), y(n) where n=3 or 4. Best results are obtained for n=3 under a majority of cases. kopt is a byref integer, and returns 0 if the min was found within the data set x(1)..x(n), otherwise +1 if xbest is larger than all value, -1 if xbest is smaller than all values. There is an arprec version of this routine available (arprec_variational_min(...)).

Wend last statement in a **while…wend** loop (see below).

While testcondition first statement of a **while…wend** loop where the testcondition is evaluated at the start of each loop and the loop is exited after **wend** when testcondition is false.

Zeta(s1) **as complex string** returns zeta for complex argument s1 (=real, imaginary) for abs(s1)<400.

Zeta_critical_abs(x) **as double** evaluates the function abs(zeta(0.5+x*I)) for real values of x using an algorithm optimized for critical line calculations. Very large x is possible, with an evaluation time proportional to sqrt(x).

zeta_critical_root(n) **as double** returns the nth root of zeta along the critical line. The function returns the value of s where |zeta(0.5+s*I)|=0.0 for the nth zero crossing. Values of n<=2, 001, 040 are allowed. Values of n>2, 001, 040 return an estimate of the root that can be used to search for a more accurate value using zeta_critical_abs(). The largest available root is returned by the function zeta_critical_root(-1). By convention, zeta_critical_root(0) = 1.0, even though no such root exists.

The following extensions are available. All sorting operations on arrays include the zeroth element.

Extension	Result
a1.append	adds an element to the array and assigns the value that follows
a1.pop	returns the last element in a1() and decreases the size of a1 by 1
a1.sort	sorts the a1 one-dimensional array in ascending order
a1.sortwith(b1)	sorts a1 as above and sorts b1 (same size as a1) according to a1
a1.shuffle	randomly shuffles all the elements in the one-dimensional array a1
variant.type	returns an integer representing the variant type
clr.red	returns or sets the level of red (0..255) in the color variable clr
clr.green	returns or sets the level of green (0..255) in the color variable clr
clr.blue	returns or sets the level of blue (0..255) in the color variable clr
clr.hue	returns or sets the hue (0.0-1.0) of the color variable clr
clr.saturation	returns or sets the saturation (0.0-1.0) of the color variable clr
clr.value	returns or sets the value (0.0-1.0) of the color variable clr
clr.cyan	returns or sets the level of cyan (0.0-1.0) in the color variable clr
clr.magenta	returns or sets the level of magenta (0.0-1.0) in the color variable clr
clr.yellow	returns or sets the level of yellow (0.0-1.0) in the color variable clr

COMPILER ERROR LIST

Error Number followed by description. If the error is common, a short clarification is added in italics. Note that the newest XS versions of Scriptor do not use error numbers but rather a sentence that explicitly states not only the error but the section of the line in which the error occurred.

01 Syntax does not make sense. This error is common and simply indicates the compiler is confused by the line of code and cannot even figure out what you are trying to do.
02 Type mismatch. You are using a variable of the wrong type in a function or subroutine that expects a different variable type.
03 Select Case does not support that type of expression. You are using an expression in the Select Case statement that is too complex. Simplify or use a variable defined by a previous expression.
04 not used (obsolete)
05 The parser's internal stack has overflowed. Simplify your code or break up your program into smaller segments.

06 Too many parameters for this function call.
07 Not enough parameters for this function call.
08 Wrong number of parameters for this function call.
09 Parameters are incompatible with this function. You are trying to use one or more parameters that is of the wrong type.
10 assignment of an incompatible data type. Your expression is mixing data types.

11 Undefined identifier. Very common error caused by using a variable that has not been defined in a Dim or Const statement.
12 Undefined operator. The operator is not recognized. Usually a spelling error.
13 Logic operations require Boolean operands.
14 Array bounds must be integers.
15 Can't call a non-function. You are probably using an array as if it were a function.

16 Can't get an element from something that isn't an array.
17 Not enough subscripts for this array's dimensions.
18 Too many subscripts for this array's dimensions.
19 Can't assign an entire array. Scriptor requires that individual array elements be assigned individually. You cannot, for example, use A = 0.0 to set an entire array to zero, or set A = B, where A and B are arrays.
20 Can't use an entire array in an expression. see above

21 Can't pass an expression as a ByRef parameter. Because a ByRef assignment is a pointer to a location in memory, you must pass the variable that defines the memory location, and not an expression.
22 Duplicate identifier. You are trying to dimension the same variable twice. You can redim as many times as you desire, but you must dim only once.
23 The backend code generator failed. Compilers are not perfect. Please report this error with a copy of the code that generated the error to rbirge@uconn.edu.
24 Ambiguous call to overloaded method. Method overloading must be defined so that there is no ambiguity in selecting which method to call. If the number of parameters is the same, the data types must be different.

25 Multiple inheritance is not allowed. You can only have one parent class.
26 Cannot create an instance of an interface. Not implemented in Scriptor.
27 Cannot implement a class as though it were an interface.
28 Cannot inherit from an object that is not a class.
29 This class does not fully implement the specified interface. When you specify a class interface, the class that implements this class interface must be a perfect match. The matching methods must use the same parameter types and return types as the corresponding method in the class interface. If you omit one or more of these methods, then you will receive this error.
30 Event handlers cannot live outside of a class. If you are defining an event handler, it must be inside a defined class.
31 It is not legal to ignore the result of a function call. If you call a function, that function by definition must return a result, and that result must be placed into the appropriate variable.
32 Can't use the keyword "Self" outside of a class.
33 Can't use the keyword "Me" outside of a class.
34 Can't return a value from a Sub. You probably should have set this method up as a function, not a subroutine. Otherwise, remove the return statement.
35 An exception object required here.

36-39 Obsolete. Please email rbirge@uconn.edu if one of these error numbers is generated and include a copy of the program, or program section, that generated the error.
40 Destructors can't have parameters.
41 Can't use "Super" keyword outside of a class.
42 Can't use "Super" keyword in a class that has no parent.
43 This #else does not have a matching #if preceding it.
44 This #endif does not have a matching #if preceding it.

45 Only Boolean constants can be used with #if.

46 Only Boolean constants can be used with #if.
47 The Next variable (%1) does not match the loop's counter variable (%2).
48 The size of an array must be a constant or number.
49 You can't pass an array to an external function.
50 External functions cannot use objects as parameters.
51 External functions cannot use ordinary strings as parameters. Use CString, PString, WString, or CFStringRef instead.
52 This kind of array can not be sorted.
53 This property is protected. It can only be used from within its class.
54 This method is protected. It can only be called from within its class.

55 This local variable or constant has the same name as a Declare in this method. You must resolve this conflict.
56 This global variable has the same name as a global function. You must resolve this conflict.
57 This property has the same name as a method. You must resolve this conflict.
58 This property has the same name as an event. You must resolve this conflict.
59 This global variable has the same name as a class. You must resolve this conflict.

60 This global variable has the same name as a module. You must change one of them.
61 This local variable or parameter has the same name as a constant. You must resolve this conflict.
62 (%1) is reserved and can't be used as a variable or property name.
63 There is no class with this name.
64 The library name must be a string constant.

65 This Declare Function statement is missing its return type.
66 You can't use the New operator with this class.
67 This method doesn't return a value.
68 End quote missing.
69 A class cannot be its own superclass.

70 Cannot assign a value to this property.
71 Cannot get this property's value.
72 The if statement is missing its condition.
73 The current function must return a value, but this Return statement does not specify any value.
74 Parameter options (%1) and (%2) are incompatible.

75 Parameter option (%1) was already specified.
76 A parameter passed by reference cannot have a default value.
77 A ParamArray cannot have a default value.
78 An Assigns parameter cannot have a default value.
79 An Extends parameter cannot have a default value.

80 Only the first parameter may use the Extends option.
81 Only the last parameter may use the Assigns option.
82 An ordinary parameter cannot follow a ParamArray.
83 Only one parameter may use the Assigns option.
84 Only one parameter may use the ParamArray option.

85 A ParamArray cannot have more than one dimension.
86 The keyword "Then" is expected after this if statement's condition.
87 Syntax error in #Pragma statement or other compiler directive
88 Constants must be defined with constant values.
89 Illegal use of the Call keyword.

90 No cases may follow the Else block.
91 (%1) is not a legal property accessor type.
92 This (%1) accessor must end with "End (%1)", not "End (%2)".
93 Duplicate method definition.
94 The library name for this declaration is blank.

95 This If statement is missing an End If statement.
96 This Select Case statement is missing an End Select statement.
97 This For loop is missing its Next statement.
98 This While loop is missing its Wend statement.
99 This Try statement is missing an End Try statement.

100 This Do loop is missing its Loop statement.
101 Too few parentheses.
102 Too many parentheses.
103 The Continue statement only works inside a loop.
104 There is no (%1) block to (%2) here.

105 Shared methods cannot access instance properties.
106 Shared methods cannot access instance methods.
107 Shared computed property accessors cannot access instance properties.
108 Shared computed property accessors cannot access instance methods.
109 The constructor of this class is protected, and can only be called from within this class.

110 This string field needs to specify its length.
111 Structures cannot contain (%1) fields.
112 Structures cannot contain multidimensional arrays.
113 Enumerated types can only contain integers.
114 An enumeration cannot be defined in terms of another enumeration.

115 This is a constant; its value can't be changed.
116 A string field must be at least 1 byte long.
117 The storage size specifier only applies to string fields.
118 A structure cannot contain itself.
119 (%1) is a structure, not a class, and cannot be instantiated with New.

120 (%1) is an enumeration, not a class, and cannot be instantiated with New.
121 This type is private, and can only be used within its module.
122 Class members cannot be global.
123 Module members must be public or private; they cannot be protected.
124 Members of inner modules cannot be global.

125 A Dim statement creates only one new object at a time.
126 A constant was expected here, but this is some other kind of expression.
127 This module is private, and can only be used within its containing module.
128 Duplicate property definition.
129 This datatype cannot be used as an array element.

130 Delegate parameters cannot be optional.
131 Delegates cannot use Extends, Assigns, or ParamArray.
132 The Declare statement is illegal in Scriptor or MathScriptor.
133 It is not legal to dereference a Ptr value in an Scriptor or MathScriptor.
134 Delegate creation from Ptr values is not allowed in Scriptor or MathScriptor.

135 Duplicate constant definition.
136 Ambiguous interface method implementation.
137 Illegal explicit interface method implementation. The class does not claim to implement this interface.
138 The interface does not declare this method.
139 This method contains a #if without a closing #endif statement.

140 This interface contains a cyclical interface aggregation.
141 The extends modifier cannot be used on a class method.
142 You cannot assign a non-value type to a value.
143 Duplicate attribute name.
144 Delegates cannot return structures.

145 You cannot create a delegate from this identifier.
146 You cannot use an Operator_Convert method to on an interface.
147 The ElseIf statement is missing its condition.
146 This type cannot be used as an explicit constant type.
149 Recursive constant declaration error.
150 Custom error created using #error.

Appendix 2. Glossary of Terms used in Programming

Here we define terms that are commonly used in the programming literature. Some of the terms are only appropriate to certain languages or programming environments such as Java, Fortran or C++. It is important to understand these terms if you are to make effective use of the literature. Where appropriate, we discuss the terms as they apply to MathScriptor. Note that only those terms that are commonly used in other languages are included here. For a glossary of terms and keywords unique to MathScriptor, see Appendix 1.

abstract class A class that contains one or more abstract methods, and therefore can never be instantiated. Abstract classes are defined so that other classes can extend them and make them concrete by implementing the abstract methods. Abstract classes do not exist in MathScriptor.

abstract method A method that has no implementation.

alpha value A value that indicates the opacity of a pixel.

API Application Programming Interface. The specification of how a programmer writing an application accesses the behavior and state of classes and objects. In MathScriptor, this is done using the TIDE as described in Chapter 1.

appliances Hardware that is normally available on the local intranet and to which a program can communicate or control by using TCP/IP protocols. A printer or a servo motor controller are examples of appliances.

applet A small program, often written in Java, that can run independently or within an application such as a web browser. The term does not refer to a small program that runs exclusively on an Apple computer.

argument A data item specified in a method call. An argument can be a literal value, a variable, or an expression.

array A variable that has two or more values which are addressed by using integer variable. For example, the array a(

ASCII American Standard Code for Information Interchange. A standard assignment of 7-bit numeric codes to represent characters. Some are visible (such as the letters of the alphabet) and some are control characters such as the tab, carriage return and line feed. The ASCII codes are listed in Appendix ASCII.

atomic An operation that is never interrupted or left in an incomplete state under any circumstance.

binary operator An operator that has two arguments or when operating on a variable, will return a value that is either 0 (or true) or 1 (or false).

bit The smallest unit of information in a digital computer, with a value of either 0 or 1.

bitwise operator An operator that manipulates two values comparing each bit of one value to the corresponding bit of the other value.

block A section of code that is contained within a set of delimiters (as in Java, C, C++, and Pascal) or selected by using conditionals (as in MathScriptor, Basic, C, C++ and FORTRAN). MathScriptor uses the reserved symbols { and } to delimit blocks of data within classes (see classes below).

boolean An expression or variable that can have only a true or false value.

bounding box A raster graphics object that is the smallest rectangle that completely encloses all the pixels that are not fully transparent.

break A statement that when encountered will cause the program to stop execution to allow user intervention or to providing debugging information.

byte A sequence of eight bits.

casting The operation of converting one data type into another. For example, converting an integer variable into a floating point variable involves a casting operation.

class An explicit set of instructions that define the function of an object and the variables and methods that are part of the object. Each object must be a member of a class, and each class can be a member of a superclass, from which it will inherit some of its properties, variables and methods. Classes are considered one of the highest level objects in object oriented programming because of their intrinsic power.

class method A method that is invoked without reference to a particular object. Class methods affect the class as a whole, not a particular instance of the class. Also called a static method. In MathScriptor, methods (subroutines or functions) can be defined without formal association with any class.

class variable A data item associated with a particular class as a whole--not with particular instances of the class. Class variables are defined in class definitions. Also called a static field. In MathScriptor, there are three types of class variables: public, private and local. Public variables are available to all objects that are instantiated. This includes the Main program that is controlling the flow. Private variables are only available to methods that are within the class. Local variables are only available within the method that defines them, and cannot be accessed by other methods either inside or external to the class.

client In the client/server model of communications, the client is a process that remotely accesses resources of a server. An email program that downloads email from the server is a common example of a client.

closed-form solution A set of equations or expressions provides a closed-form solution if it can be expressed in terms of a finite number of well-defined functions which when evaluated provide a numerical result. An infinite series is an example of a solution which is not closed-form. A closed-form solution is sometimes referred to as an explicit solution.

comment Explanatory text that is ignored by the compiler. In MathScriptor, comments are indicated by using the single quote ('), double slashes (//) or the REM statement. All text following these text elements is treated as a comment until an end-of-line character is encountered.

common block A block of memory assigned within Fortran to allow different subroutines or functions to share the same memory and/or pass values to one another. The availability of common blocks makes Fortran extremely efficient at using memory. At the same time, common blocks violate virtually all of the tenants of object-oriented structured programming design because common blocks allow free access to the memory and a programming error in one object can affect all other objects that use the same common block. Common blocks are no longer available in MathScriptor as the same functionality can be obtained in a more transparent way by using data files or the spreadsheet.

compiler A program to translate source code into the machine code that is executed by the computer. A compiler typically carries out its function in a separate operation prior to running the program, and in the case of MathScriptor, the program is compiled fully prior to running. This approach differs from an interpreter which steps through the code one line at a time and compiles each line separately. Compilers generate code that is much faster than interpreters. However, interpreters can start up almost instantly. The compiler within MathScriptor is a very fast compiler and operates with a speed that rivals most interpreters.

compositing The process of superimposing one image on another to create a single image.

constructor A pseudo-method that creates an object. Constructors are instance methods that run when a class is instantiated. In MathScriptor, the constructor is a subroutine within the class definition with the name constructor. Constructors are invoked using the new keyword.

constants A variable that is defined once and not allowed to change values during the and cannot be altered. A constant in MathScriptor is defined using the const name=value statement. If you try and redefine it within the program, the program refuses to run, but does not issue a run-time error statement.

crash When a program encounters a runtime error that has not been trapped by the programmer, it is not uncommon for the program to simply quit. This can happen when MathScriptor encounters are request for runtime memory that is beyond what is physically available. In modern operating systems like Windows XP and Mac OSX, the operating system then takes over and opens a small window that notifies the user that the program has quit unexpectedly. This is known as a soft crash because the computer is still running and the program is the only casualty. Earlier Windows or Mac operating systems often responded to a program crash with a subsequent crash of the operating system. This is known as a hard crash because you need to restart the computer. Programmers have benefited significantly from the development of modern operating systems which protect memory, monitor programs and allow a user to force quit a program that has become stuck in an infinite loop or is unresponsive for another reason.

critical section A segment of code in which a thread uses resources (such as certain instance variables) that can be used by other threads, but that must not be used by them at the same time.

debugging The act of removing coding errors in a program so that it functions the way the programmer intended. In the MathScriptor TIDE, there is a menu item called debugging that provides various options that help the programmer identify errors.

declaration A statement that establishes an identifier and associates attributes with it, without necessarily reserving its storage (for data) or providing the implementation (for methods). In MathScriptor, when you dimension a variable, memory space is allocated based on the size and type of variables. You can, however, redimension an array during runtime at which point additional memory can be allocated as necessary.

definition A declaration that reserves storage (for data) or provides implementation (for methods).

deprecation The act of specifying a class, interface, constructor, method or field that was once used in previous versions of the software as no longer recommended. A depreciated component will likely not be supported in subsequent versions.

destructor An optional subroutine within a class definition that is run automatically when the class goes out of scope. The routine can help reset memory or variables that were used by the class. It is optional in MathScriptor because the compiler is able to handle memory allocation and garbage collection automatically so that in most cases a destructor is not required.

distributed Code that is running in more than one address space on the same or on different hardware. A distributed computer is one which has more than one processor and normally has multiple memory regions which can be independently addressed by the local processor.

double A keyword used to define a floating point variable of type double. In MathScriptor this data type provides approximately 16 digits of precision, and can take on values from 2.2250738585072013 E-308 to 1.7976931348623157 E+308. In Fortran and other Extended Basic languages, this variable would be defines as Real*16 or Double Precision. A double variable uses 8 bytes (64 bits) of memory.

double precision A floating point number that holds 64 bits of data. Languages and compilers differ on how many bits are assigned to the exponent and how many are assigned to the mantissa (the number), but typically the number has 16 bits of precision and the exponent has 2-3 digits of precision. In MathScriptor, the double data type is a double precision number.

else A keyword used to execute a block of statements in the case that the test condition with the "if" keyword evaluates to false.

encapsulation The localization of knowledge within a module. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but as long as the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten. See also instance variable, instance method.

exception An event during program execution that prevents the program from continuing normally; generally, an error. See also exception handler.

exception handler A block of code that reacts to a specific type of exception. If the exception is for an error that the program can recover from, the program can resume executing after the exception handler has executed.

extends A term used to describe the use of a new class to add functionality to another class upon which it is based or replace the class. For example, if class XP extends class X, it does so by the addition of new methods.

floating point The designation of a variable or a math operation that includes the decimal portion of a number. MathScriptor supports two types of floating point numbers represented using the keywords single or double.

for An extended basic and MathScriptor programming language keyword used to declare a loop that reiterates statements. The programmer can specify the statements to be executed, exit conditions, and initialization variables for the loop.

FTP The basic Internet File Transfer Protocol. FTP, which is based on TCP/IP, enables the fetching and storing of files between hosts on the Internet. See also TCP/IP.

formal parameter list The parameters specified in the definition of a particular method.

function An object that is a method. A function must return a value and that value must be assigned to a variable of the correct type.

garbage collection The automatic detection and freeing of memory that is no longer in use. There are principally two types of garbage collection. The older type uses "mark and sweep" collection to monitor memory usage, and is fast but inefficient and can sometimes produce a memory leak where each access to a method uses up additional memory simply because the marking process is inaccurate. MathScriptor uses "reference-counting" to monitor memory usage. While slower, it avoids memory leaks.

goto This is a reserved MathScriptor programming language keyword. However, it should not be used unless absolutely necessary because it can destroy the structure of the program flow and make it more difficult to understand and maintain the code. Some languages no longer provide this keyword to force structure, but MathScriptor includes it for compatibility and to help users transfer code from Fortran and Basic, where goto statements are common.

GUI Graphical User Interface. Refers to the techniques involved in using graphics, along with a keyboard and a mouse, to provide an easy-to-use interface to some program. The MathScriptor TIDE is an example of a GUI.

hexadecimal The numbering system that uses 16 as its base. The marks 0-9 and a-f (or equivalently A-F) represent the digits 0 through 15. A hexadecimal representation of a number can be generated in MathScriptor by using the hex(value) command, which returns a string. See also octal.

hierarchy A classification of relationships in which each item except the top one (known as the root) is a specialized form of the item above it. Each item can have one or more items below it in the hierarchy.

host A server which allows other users or computers controlled access to information via the internet. A local host is normally inside the same building, university or company. An external host is normally outside your local internet domain.

HTML HyperText Markup Language. This is a file format, based on SGML, for hypertext documents on the Internet. It is very simple and allows for the embedding of images, sounds, video streams, form fields and simple text formatting. References to other objects are embedded using URLs.

HTTP HyperText Transfer Protocol. The Internet protocol, based on TCP/IP, used to fetch hypertext objects from remote hosts.

identifier A generic term that is used to represent a named property (variable) or object (method or subroutine). This term is most often encountered in MathScriptor in the form of the "undefined" or "duplicate" identifier errors which mean the compiler has found a name for which neither a function, subroutine or variable with the same name has been defined.

IDL Interface Definition Language. APIs normally written in the Java(TM) programming language that provide standards-based interoperability and connectivity with CORBA (Common Object Request Broker Architecture).

if A keyword used to conduct a conditional test and execute a block of statements if a test condition evaluates to true. Virtually all modern languages provide some form of if statement. The associated set of conditionals within MathScriptor include: if … then … elseif … else … end if.

immutable In object-oriented and functional programming, an immutable object is an object whose state cannot be modified after it is created. This is in contrast to a mutable object, which can be modified after it is created. An object can be either entirely immutable or some attributes in the object may be declared immutable; for example, defining selected variables within a subroutine to be of type const. In some cases, an

object is considered immutable even if some internally used attributes change but the object's state appears to be unchanging from an external point of view. The initial state of an immutable object is usually set at its inception, but can also be set before actual use of the object. For example, a variant is mutable when first created, but once assigned a value and a type, the variant become immutable except for its value.

Immutable objects are often useful because some costly operations for copying and comparing can be omitted, simplifying the program code and speeding execution. However, making an object immutable is usually inappropriate if the object contains a large amount of changeable data. Because of this, many languages allow for both immutable and mutable objects.

inheritance The concept of classes automatically containing the variables and methods defined in their supertypes. See also superclass, subclass.

instance An object of a particular class, or a class that has been instantiated.

instantiate The act of activating a class object so that it is available for use in the program. In MathScriptor, this is done using the New command.

integer A programming language keyword used to define a variable of type integer, which is a whole number that has no fractional component. For example, 3 is an integer, but 3.14159 is a real or floating point number. An integer in MathScriptor is a 32-bit whole number in the range ± 2, 147, 483, 648 ($\pm 2^{31}$). Most languages would call this a long integer, and their default integer is a 16-bit number in the range ± 32, 768 ($\pm 2^{15}$). Note that in both definition, one bit is assigned to represent the sign of the number.

int64 A MathScriptor keyword used to define a variable of type integer, but using a 64-bit representation, twice that of the default (long) integer. An int64 is an 18-digit whole number in the range ±9, 223, 372, 036, 854, 775, 807.

interface A hardware or software component that serves to connect two computers, a computer and a device. The MathScriptor Interface Panel of version 2.0 and beyond allows a program to communicate with and collect data from Vernier LabPro instruments.

Internet An enormous network consisting of literally millions of hosts from many organizations and countries around the world. It is physically put together from many smaller networks and data travels by a common set of protocols. The Internet evolved out of the older DARPA Net project - Defense Advance Research Projects Agency Network. The DARPA Net was used to connect computers at various military and

classified civilian sites around the United States together, using modems and the existing phone line grid.

IP Internet Protocol. The basic protocol of the Internet. It enables the unreliable delivery of individual packets from one host to another. It makes no guarantees about whether or not the packet will be delivered, how long it will take, or if multiple packets will arrive in the order they were sent. Protocols built on top of this add the notions of connection and reliability. See also TCP/IP.

interface A class definition that consists of collection of subroutines and functions with all of the parameters defined, but no code within. The interface defines the properties, but not the behavior, of any new class that implements the interface. If a class implements an interface, it must contain all of the elements defined by the interface, and if it does not, the compiler rejects it and throws an error message. The purpose of an interface is to establish a set of rules that must be followed by subsequent classes.
interpreter A module that alternately decodes and executes every statement in some body of code. See also compiler, runtime system.

Java(TM) Sun's trademark for a set of technologies for creating and safely running software programs in both stand-alone and networked environments.

JavaScript(TM) A Web scripting language that is used in both browsers and Web servers. Like all scripting languages, it is used primarily to tie other components together or to accept user input.

Just-in-time (JIT) compiler A compiler that operates on the code only when the code is executed. This approach allows the compiler to optimize the code for the machine environment that is available rather than for a generic computer with the same processor. The concept is that this approach will lead to faster and smaller programs.

JPEG or Joint Photographic Experts Group An image file compression standard established by this group. It achieves tremendous compression at the cost of introducing distortions into the image which are usually imperceptible to the naked eye. Files generated of this type usually end with .jpeg or .jpg extensions.

keyword Words reserved by the language and are therefore not available as names for variables or methods created by the user. In MathScriptor, keywords are marked in blue after a precompile.

lexical Pertaining to how the characters in source code are translated into tokens that the compiler can understand.

linker A module that builds an executable, complete program from component machine code modules. The MathScriptor compiler also carries out the function of linking all of the program elements that you have written to those internal to the MathScriptor context.

literal The basic representation of any integer, floating point, or character value. For example, 3.0 is a double-precision floating point literal, and "a" is a character literal.

local variable A data item known within a block, but inaccessible to code outside the block. For example, any variable defined within a method (function or subroutine) is a local variable and can't be used outside the method.

mantissa The portion of a number that carries the significant digits and is multiplied by the exponent to create the complete representation. The mantissa is also called the coefficient or the significand.

method A function defined in a class. See also instance method, class method. In MathScriptor, methods are either subroutines or functions.

multithreaded Describes a program that is designed to have parts of its code execute concurrently. See also thread.

mutable When an object is mutable, it can be modified after it has been defined or declared. Most variables are mutable which means you can assign them new values. If a variable is immutable, once declared, it cannot be changed. A constant is an example of an immutable object (or an immutable variable). (see also immutable)

nil A programming language keyword used to specify an undefined value for reference variables.

object The principal building blocks of object-oriented programs. Each object is a programming unit consisting of data (instance variables) and functionality (instance methods). See also class.

object-oriented design A software design method that models the characteristics of abstract or real objects using classes and objects. MathScriptor provides access to objects in the Object Panel or as part of the Main Program in the Main or Music Panels.

octal The numbering system using 8 as its base, using the numerals 0-7 as its digits. In MathScriptor, the function oct(value) returns a string representing the octal equivalent of value. See also hexadecimal.

overloading Using one identifier to refer to multiple items in the same scope. MathScriptor allows functions and subroutines to be overloaded, which allows two or more methods to be defined with the same name but different numbers or types of parameters. This capability allows the user to enhance the functionality of methods, both user and those predefined by MathScriptor.

peer In networking, any functional unit in the same layer as another entity.

pixel The smallest addressable picture element on a display screen or printed page.

POSIX Portable Operating System for UNIX(TM). A standard that defines the language interface between the UNIX operating system and application programs through a minimal set of supported functions.

private A programming language keyword used in a method or variable declaration. It signifies that the method or variable can only be accessed by other elements of its class. For example, in MathScriptor, you can declare variables within a class to be "private properties" or "public properties". Those variables declared under "private properties" can only be used by methods declared within this class.

properties A generic term to identify the assignment of variables to variable types.

public A keyword used to indicate variables that can be accessed by elements residing in the main program or in other classes.

raster A single line of pixels. A raster is normally a horizontal line of pixels in the electronics community, but horizontal and vertical rasters are common within the graphics community and in software engineering.

real number A floating point number that can take on fractional values. For example, the number 3.14159 is a floating point, or real number. Real numbers in MathScriptor can be of type Single or Double.

reference A data element whose value is an address. When the term ByRef is used in front of a variable passed to a function or subroutine, this indicates that any changes on this variable within the method will alter the variable that was passed because the change will be made to the memory location. In contrast, ByVal indicates that the value of the original variable is passed rather than a reference to the variable itself. If you make changes to a ByVal variable within a method, the changes are lost upon exit and the original variable is unaffected. In MathScriptor, all variables without a reference declaration default to ByVal except arrays which are always passed ByRef regardless of reference declaration.

return This keyword indicates that the variable that follows is to be returned by the function to the calling program and the function excited. Subroutines are not allowed to return variables using this statement, but can return results ByRef (see above).

runtime (run-time) The time during which the program is actually running. When a program is running, most of the characteristics of memory allocation and variable typing has been carried out during the compilation and further changes are not allowed. However, there are important exceptions allowed by MathScriptor. String variables are dynamically allocated in memory based on the length of the character string assigned to the variable. In addition, arrays can be redimension using the Redim statement which allows runtime changes in memory allocation. This flexibility comes as a price in that if there is not enough memory to handle the reallocation, the program will crash.

scope A characteristic of a variable or identifier that determines where the identifier can be used. Most identifiers have either class or local scope. Instance and class variables and methods have class scope and cannot be accessed outside of the class.

Secure Socket Layer (SSL) A protocol that allows communication between a Web browser and a server to be encrypted for privacy.

single precision A term used in Fortran, Java and other languages to indicate a real variable stored with 32-bits of precision. In MathScriptor, the term "single" is used by itself to indicate this variable type, which can take on values between –1.175494 e-38 and 3.402823 e+38.

SGML Standardized Generalized Markup Language. An ISO/ANSI/ECMA standard that specifies a way to annotate text documents with information about types of sections of a document.

SQL Structured Query Language is an ANSI/ISO standard used to create, modify, retrieve and manipulate data from relational database management systems. MathScriptor makes use of the SQL date and time format in selected statements, where the format is defined as: YYYY-MM-DD HH:MM:SS and as such can be parsed precisely by software because the position of each numerical value is fully defined in terms of absolute position.

static A term that indicates that a variable is kept in memory at all times. In most languages, the local variables within a method (function or subroutine) are removed from memory when the routine is complete and control has been passed back to the main program. In MathScriptor, you can designate a variable as static by using a Module or Class and declaring the variable as a member of the Public Properties.

static field Another name for a public class variable.

static method A common term used to indicate a Class Method.

subroutine An object that is a method. A subroutine does not return a value but can alter the variables that are passed to it by reference (ByRef). In MathScriptor, all arrays are automatically passed ByRef regardless of declaration.

TCP/IP Transmission Control Protocol based on IP. This is an Internet protocol that provides for the reliable delivery of streams of data from one host to another. See also IP.

thread The basic unit of program execution. A process can have several threads running concurrently, each performing a different job, such as waiting for events or performing a time-consuming job that the program doesn't need to complete before going on. When a thread has finished its job, the thread is suspended or destroyed. See also process.

Unicode A 16-bit character set defined by ISO 10646. See also ASCII.

URL Uniform Resource Locator. A standard for writing a text reference to an arbitrary piece of data in the WWW. A URL looks like "protocol://host/local_designator" where protocol specifies a protocol to use to fetch the object (like HTTP or FTP), host specifies the Internet name of the host on which to find it, and local_designator is a string (often a file name) passed to the protocol handler on the remote host.

variable An item of data named by an identifier. Each variable has a type, such as double, integer or string. See also class variable, instance variable, local variable.

virtual machine An abstract specification for a computing device that can be implemented in different ways, in software or hardware. You compile to the instruction set of a virtual machine much like you'd compile to the instruction set of a microprocessor.

void A C++ or Java programming language keyword used in method declarations to specify that the method does not return any value. "void" can also be used as a nonfunctional statement in these languages. In MathScriptor, a method that does not return a value is called a subroutine, although values can be returned by using the ByRef keyword with one or more of the variables passed in the subroutine argument list. In contrast, all MathScriptor functions must return a value and the value must be assigned to an appropriate variable in the calling routine. Although all functions must return a value, this value can be ignored by the calling routine by placing the keyword “call” in front of the function in place of the variable.

WANDA Working Application Not Doing Anything. A snippet of code or a collection of program elements that have correct syntax but when run do nothing, or nothing useful. A WANDA is often used to demonstrate a programming concept or turned in as part of a homework assignment. A term coined by the computer science department at Carnegie Mellon in the 1980s.

while A keyword used to declare a loop that iterates a block of statements. The loop's exit condition is specified as part of the while statement as in: while x>0 … wend

wrapper An object that encapsulates another object to alter its interface or behavior in some way.

WWW World Wide Web. The web of systems and the data in them that is the Internet. See also Internet.

Appendix 3.
ASCII Codes, and their character and control representations

Dec	Octal	Hex	Binary	Value	Special Meaning
000	000	000	00000000	NUL	(Null char.)
001	001	001	00000001	SOH	(Start of Header)
002	002	002	00000010	STX	(Start of Text)
003	003	003	00000011	ETX	(End of Text)
004	004	004	00000100	EOT	(End of Transmission)
005	005	005	00000101	ENQ	(Enquiry)
006	006	006	00000110	ACK	(Acknowledgment)
007	007	007	00000111	BEL	(Bell)
008	010	008	00001000	BS	(Backspace)
009	011	009	00001001	HT	(Horizontal Tab)
010	012	00A	00001010	LF	(Line Feed)
011	013	00B	00001011	VT	(Vertical Tab)
012	014	00C	00001100	FF	(Form Feed)
013	015	00D	00001101	CR	(Carriage Return)
014	016	00E	00001110	SO	(Shift Out)
015	017	00F	00001111	SI	(Shift In)
016	020	010	00010000	DLE	(Data Link Escape)
017	021	011	00010001	DC1	(XON) (Device Control 1)
018	022	012	00010010	DC2	(Device Control 2)
019	023	013	00010011	DC3	(XOFF)(Device Control 3)
020	024	014	00010100	DC4	(Device Control 4)
021	025	015	00010101	NAK	(Negative Acknowledgement)
022	026	016	00010110	SYN	(Synchronous Idle)
023	027	017	00010111	ETB	(End of Trans. Block)
024	030	018	00011000	CAN	(Cancel)
025	031	019	00011001	EM	(End of Medium)
026	032	01A	00011010	SUB	(Substitute)
027	033	01B	00011011	ESC	(Escape)
028	034	01C	00011100	FS	(File Separator)
029	035	01D	00011101	GS	(Group Separator)
030	036	01E	00011110	RS	(Request to Send)(Record Separator)
031	037	01F	00011111	US	(Unit Separator)
032	040	020	00100000	SP	(Space)
033	041	021	00100001	!	(exclamation mark)
034	042	022	00100010	"	(double quote)
035	043	023	00100011	#	(number sign)
036	044	024	00100100	$	(dollar sign)
037	045	025	00100101	%	(percent)
038	046	026	00100110	&	(ampersand)
039	047	027	00100111	'	(single quote)
040	050	028	00101000	(	(left/opening parenthesis)
041	051	029	00101001	)	(right/closing parenthesis)
042	052	02A	00101010	*	(asterisk)
043	053	02B	00101011	+	(plus)
044	054	02C	00101100	,	(comma)
045	055	02D	00101101	-	(minus or dash)
046	056	02E	00101110	.	(dot or period)
047	057	02F	00101111	/	(forward slash)

Appendix 3. Ascii Codes (continued)

Dec	Octal	Hex	Binary	Value	Special Meaning
048	060	030	00110000	0	
049	061	031	00110001	1	
050	062	032	00110010	2	
051	063	033	00110011	3	
052	064	034	00110100	4	
053	065	035	00110101	5	
054	066	036	00110110	6	
055	067	037	00110111	7	
056	070	038	00111000	8	
057	071	039	00111001	9	
058	072	03A	00111010	:	(colon)
059	073	03B	00111011	;	(semi-colon)
060	074	03C	00111100	<	(less than)
061	075	03D	00111101	=	(equal sign)
062	076	03E	00111110	>	(greater than)
063	077	03F	00111111	?	(question mark)
064	100	040	01000000	@	(AT symbol)
065	101	041	01000001	A	
066	102	042	01000010	B	
067	103	043	01000011	C	
068	104	044	01000100	D	
069	105	045	01000101	E	
070	106	046	01000110	F	
071	107	047	01000111	G	
072	110	048	01001000	H	
073	111	049	01001001	I	
074	112	04A	01001010	J	
075	113	04B	01001011	K	
076	114	04C	01001100	L	
077	115	04D	01001101	M	
078	116	04E	01001110	N	
079	117	04F	01001111	O	
080	120	050	01010000	P	
081	121	051	01010001	Q	
082	122	052	01010010	R	
083	123	053	01010011	S	
084	124	054	01010100	T	
085	125	055	01010101	U	
086	126	056	01010110	V	
087	127	057	01010111	W	
088	130	058	01011000	X	
089	131	059	01011001	Y	
090	132	05A	01011010	Z	

Appendix 3. Ascii Codes (continued)

Dec	Octal	Hex	Binary	Value	Special Meaning
091	133	05B	01011011	[	(left/opening bracket)
092	134	05C	01011100	\	(back slash)
093	135	05D	01011101	]	(right/closing bracket)
094	136	05E	01011110	^	(caret/cirumflex)
095	137	05F	01011111	_	(underscore)
096	140	060	01100000	`	
097	141	061	01100001	a	
098	142	062	01100010	b	
099	143	063	01100011	c	
100	144	064	01100100	d	
101	145	065	01100101	e	
102	146	066	01100110	f	
103	147	067	01100111	g	
104	150	068	01101000	h	
105	151	069	01101001	i	
106	152	06A	01101010	j	
107	153	06B	01101011	k	
108	154	06C	01101100	l	
109	155	06D	01101101	m	
110	156	06E	01101110	n	
111	157	06F	01101111	o	
112	160	070	01110000	p	
113	161	071	01110001	q	
114	162	072	01110010	r	
115	163	073	01110011	s	
116	164	074	01110100	t	
117	165	075	01110101	u	
118	166	076	01110110	v	
119	167	077	01110111	w	
120	170	078	01111000	x	
121	171	079	01111001	y	
122	172	07A	01111010	z	
123	173	07B	01111011	{	(left/opening brace)
124	174	07C	01111100	|	(vertical bar)
125	175	07D	01111101	}	(right/closing brace)
126	176	07E	01111110	~	(tilde)
127	177	07F	01111111	DEL	(delete)

Appendix 4.
Selected Mathematical Rules, Formulas and Definitions

The following set of mathematical rules and formulas were selected to facilitate solving of the problems in this book. When both a positive and negative sign appear, the top sign is used throughout or the bottom sign is used throughout. Thus if $f \pm g = r \mp s$, then $f + g = r - s$ or $f - g = r + s$. All trigonometric functions are in radians (1 rad =57.2958°).

1. Trigonometric and Geometric Relationships:

$$\sin\alpha \sin\beta = \tfrac{1}{2}\cos(\alpha-\beta) - \tfrac{1}{2}\cos(\alpha+\beta) \quad \text{(A4.1.1)}$$

$$\cos\alpha \cos\beta = \tfrac{1}{2}\cos(\alpha-\beta) + \tfrac{1}{2}\cos(\alpha+\beta) \quad \text{(A4.1.2)}$$

$$\sin\alpha \cos\beta = \tfrac{1}{2}\sin(\alpha+\beta) + \tfrac{1}{2}\sin(\alpha-\beta) \quad \text{(A4.1.3)}$$

$$\sin(\alpha \pm \beta) = \sin\alpha \cos\beta \pm \cos\alpha \sin\beta \quad \text{(A4.1.4)}$$

$$\cos(\alpha \pm \beta) = \cos\alpha \cos\beta \pm \sin\alpha \sin\beta \quad \text{(A4.1.5)}$$

$$\cos^2(\alpha) + \sin^2(\alpha) = 1 \quad \text{(A4.1.6)}$$

$$\tan\alpha = \sin\alpha / (\cos\alpha) \quad \text{(A4.1.7)}$$

right triangles:

$$\sin\theta = \text{opposite/hypotenuse}, \quad \text{(A4.1.8a)}$$

$$\cos\theta = \text{adjacent/hypotenuse}, \quad \text{(A4.1.8b)}$$

$$\tan\theta = \text{opposite/adjacent}, \quad \text{(A4.1.8c)}$$

equilateral triangle of height h and base b: area $= \frac{1}{2}(b\,h)$ (A4.1.9a)

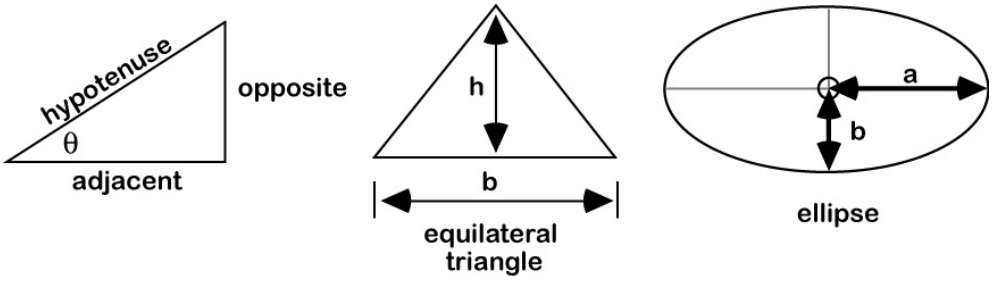
rectangle of length l and width b: area = b h (A4.1.9b)

circle of radius r: area $= \pi r^2$; circumference $= 2\pi r$ (A4.1.9c)

ellipse with radii a and b: area $= \pi a b$; perimeter $\pi(a + b)$ (A4.2.0a)

sphere of radius r: surface area $= 4\pi r^2$; volume $= \frac{4}{3}\pi r^3$; surface area $= 4\pi r^2$ (A4.2.0b)

cylinder of radius r and height h: volume $= \pi r^2 h$; surface area $= 2\pi r h + 2\pi r^2$ (A4.2.0c)



2. Complex Numbers

$\mathbf{N} = a + b\mathbf{i} = |\mathbf{N}|e^{\mathbf{i}\theta}$ (A4.2.1)

where a is a real number and represents the "real part",
b is a real number and represents the "imaginary part" and
$\mathbf{i} = \sqrt{-1}$ is the imaginary number (electrical engineers use **j**).
Note that $1/\mathbf{i} = 1/\mathbf{i} \times (\mathbf{i}/\mathbf{i}) = -\mathbf{i}$.

$|\mathbf{N}| = \sqrt{\mathbf{N}^* \mathbf{N}} = \sqrt{a^2 + b^2}$ (magnitude or modulus) (A4.2.2)

$\theta = \arctan(b/a)$ (A4.2.3)

$\left.\begin{array}{l} \mathbf{N}^* = a - b\mathbf{i} \text{ (if } \mathbf{N} = a + b\mathbf{i}) \\ \mathbf{N}^* = a + b\mathbf{i} \text{ (if } \mathbf{N} = a - b\mathbf{i}) \end{array}\right\}$ (complex conjugate) (A4.2.4)

$(a + b\mathbf{i}) + (c + d\mathbf{i}) = (a + c) + (b + d)\mathbf{i}$ (addition) (A4.2.5)
$(a + b\mathbf{i}) - (c + d\mathbf{i}) = (a - c) + (b - d)\mathbf{i}$ (subtraction) (A4.2.6)
$(a + b\mathbf{i})(c + d\mathbf{i}) = (ac - bd) + (bc + ad)\mathbf{i}$ (multiplication) (A4.2.7)

$$\frac{(a + b\mathbf{i})}{(c + d\mathbf{i})} = \frac{(ac + bd)}{(c^2 + d^2)} + \frac{(bc - ad)}{(c^2 + d^2)}\,\mathbf{i}$$ (division) (A4.2.8)

$e^{\pm\mathbf{i}\theta} = \cos\theta \pm \mathbf{i}\sin\theta$ (Euler's formula) (A4.2.9)

$$\cos\theta = \frac{e^{\mathbf{i}\theta} + e^{-\mathbf{i}\theta}}{2}$$ (A4.2.10)

$$\sin\theta = \frac{e^{\mathbf{i}\theta} - e^{-\mathbf{i}\theta}}{2\mathbf{i}}$$ (A4.2.11)

$ae^{\mathbf{i}b} = a\cos b + \mathbf{i}a\sin b$ (A4.2.12)

$\mathrm{Re}(ae^{\mathbf{i}b}) = a\cos b$ (real part) (A4.2.13)

$\mathrm{Im}(ae^{\mathbf{i}b}) = a\sin b$ (imaginary part) (A4.2.14)

$[ae^{\mathbf{i}b}]^* = ae^{-\mathbf{i}b}$ (conjugate) (A4.2.15)

$|ae^{\mathbf{i}b}| = a$ (magnitude) (A4.2.16)

3. Laws of Logarithms and Series Expansions

$$\log(ab) = \log(a) + \log(b) \qquad \text{(A4.3.1)}$$

$$\log(a/b) = \log(a) - \log(b) \qquad \text{(A4.3.2)}$$

$$\log(a^n) = n \log(a) \qquad \text{(A4.3.3)}$$

$$f(x) = f(a) + f'(a)(x - a) + \frac{1}{2!} f''(a)(x - a)^2 + \frac{1}{3!} f'''(a)(x - a)^3 + \cdots + \frac{1}{n!} f^{n}(a)(x-a)^n \qquad \text{(Taylor Series of n terms)}$$

$$e^{ax} = 1 + ax + \frac{a^2x^2}{2!} + \frac{a^3 x^3}{3!} + \frac{a^4x^4}{4!} + \cdots \qquad \text{(A4.3.4)}$$

$$e^{-ax} = 1 - ax + \frac{a^2x^2}{2!} - \frac{a^3 x^3}{3!} + \frac{a^4x^4}{4!} - \frac{a^5x^5}{5!} + \cdots \qquad \text{(A4.3.5)}$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \qquad \text{(A4.3.6)}$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \qquad \text{(A4.3.7)}$$

$$\frac{1}{1 - x} = 1 + x + x^2 + x^3 + x^4 + \cdots \qquad (x^2 < 1) \qquad \text{(A4.3.8)}$$

$$(1 \pm x)^n = 1 \pm nx + \frac{n(n - 1)}{2!} x^2 \pm \frac{n(n - 1)(n - 2)}{3!} x^3 + \cdots \qquad (x^2 < 1) \qquad \text{(A4.3.9)}$$

4. Selected Derivatives

The following formulas assume that a, b and c are scalar constants or formulas which are not a function of x, and that n is an integer. Note that Exp[ax] is an alternative notation for e^{ax}.

$$\frac{d}{dx}\, a\, f(x) = a\, \frac{d}{dx}\, f(x) \qquad \text{(constant rule)}$$

$$\frac{d}{dx}\left(f(x) + g(x)\right) = \frac{d}{dx} f(x) + \frac{d}{dx} g(x) \qquad \text{(sum rule)}$$

$$\frac{d}{dx}\left(f(x) \bullet g(x)\right) = \left(g(x) \frac{d}{dx} f(x)\right) \bullet \left(f(x) \frac{d}{dx} g(x)\right) \qquad \text{(product rule)}$$

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{g(x) \frac{d}{dx} f(x) - f(x) \frac{d}{dx} g(x)}{[\, g(x)\,]^2} \qquad \text{(division rule)}$$

$$\frac{d}{dx}\left(f\,[g(x)]\right) = \frac{d}{dg} f(x)\ \frac{d}{dx} g(x) \qquad \text{(chain rule)}$$

$$\frac{d}{dx}\, x^n = n x^{n-1} \qquad \frac{d}{dx}\, x^{a/b} = (a/b)\, x^{(a/b)-1} \qquad \text{(A4.4.1)}$$

$$\frac{d}{dx}\, e^x = e^x \qquad \frac{d}{dx}\, a e^{-bx} = -ab\, e^{-bx} \qquad \text{(A4.4.2)}$$

$$\frac{d}{dx}\, x e^x = x e^x + e^x \qquad \frac{d}{dx}\, x e^{-x} = e^{-x} \qquad \text{(A4.4.3)}$$

$$\frac{d}{dx}\, \sin(ax) = a \cos(ax) \qquad \frac{d}{dx}\, \cos(ax) = -a \sin(ax) \qquad \text{(A4.4.4)}$$

$$\frac{d}{dx}\, \tan(ax) = a \sec^2(ax) \qquad \text{(A4.4.5)}$$

$$\frac{d}{dx}\, a^x = \frac{d}{dx}\, (e^{x \ln(a)}) = a^x \ln(a) \qquad \text{(A4.4.6)}$$

$$\frac{d}{dx}\, \mathrm{Exp}\left(\frac{-a\,(b-x)^2}{c^2}\right) = \frac{2\,a\,(b-x)}{c^2}\ \mathrm{Exp}\left(\frac{-a\,(b-x)^2}{c^2}\right) \qquad \text{(A4.4.7)}$$

$$\frac{d}{dx}\, a e^{\mathbf{i}x} = a\mathbf{i}\, e^{\mathbf{i}x} \qquad \text{(A4.4.8)}$$

5. Selected Integrals

The following formulas assume that a, b and c are scalar constants or expressions which are not a function of x, that m and n are integers, and that δ_{mn} is the Kronicker delta which equals zero at all times, but unity when m=n [δ_{mn} = 1 (m=n); 0 (m≠n)].

$$\int_0^\infty x e^{-ax^2}\,dx = \frac{1}{2a} \tag{A4.5.1}$$

$$\int_0^\infty e^{-ax^2}\,dx = \left(\frac{\pi}{4a}\right)^{1/2} \tag{A4.5.2}$$

$$\int_0^\infty e^{ax^2}\,dx = -\frac{(-a\pi)^{1/2}}{2a} \tag{A4.5.3}$$

$$\int_{-\infty}^{\infty} \exp\left(\frac{(x-a)^2}{b^2}\right)dx = -ib\sqrt{\pi} \tag{A4.5.4}$$

$$\int_0^\infty x^n e^{-ax}\,dx = \frac{n!}{a^{n+1}} \qquad (n\ positive) \tag{A4.5.5}$$

$$\int_0^\infty x^{2n} e^{-ax^2}\,dx = \frac{1\cdot 3\cdot 5\cdots(2n-1)}{2^{n+1}a^n}\left(\frac{\pi}{a}\right)^{1/2} \qquad (n\ positive) \tag{A4.5.6}$$

$$\int_0^\infty x^{2n+1} e^{-ax^2}\,dx = \frac{n!}{2a^{n+1}} \quad (n\,\text{positive integer}) \tag{A4.5.7}$$

$$\int_0^a \sin\frac{n\pi x}{a}\,\sin\frac{m\pi x}{a}\,dx = \int_0^a \cos\frac{n\pi x}{a}\,\cos\frac{m\pi x}{a}\,dx = \frac{a}{2}\,\delta_{nm} \tag{A4.5.8}$$

$$\int_0^a \sin\frac{n\pi x}{a}\;x\,\sin\frac{n\pi x}{a}\,dx = \frac{a^2}{4} \tag{A4.5.9}$$

$$\int_0^a \sin\frac{n\pi x}{a}\;x^2\,\sin\frac{n\pi x}{a}\,dx = \frac{a^3}{6} - \frac{a^3}{4\,n^2\,\pi^2} \tag{A4.5.10}$$

$$\int_0^a \sin\frac{n\pi x}{a}\ \frac{d}{dx}\left(\sin\frac{n\pi x}{a}\right)dx = \frac{n\pi}{a}\int_0^a \sin\frac{n\pi x}{a}\ \cos\frac{n\pi x}{a}\ dx = 0 \qquad \text{(A4.5.11)}$$

$$\int_0^a \sin\frac{n\pi x}{a}\ \frac{d^2}{dx^2}\left(\sin\frac{n\pi x}{a}\right)dx = \frac{-n^2\pi^2}{2a} \qquad \text{(A4.5.12)}$$

$$\int_0^a \sin\frac{n\pi x}{a}\ x\sin\frac{m\pi x}{a}\ dx = \frac{-2a^2 m n}{(m^2-n^2)^2\pi^2}$$

$$+\frac{a^2\cos[(m-n)\pi]}{2(m-n)^2\pi^2} - \frac{a^2\cos[(m+n)\pi]}{2(m+n)^2\pi^2} \qquad \text{(A4.5.13)}$$

$$\int_0^{a/2} \sin\frac{n\pi x}{a}\ (b/\mathbf{i})\frac{d}{dx}\left(\sin\frac{n\pi x}{a}\right)dx = -\frac{\mathbf{i}\,b}{2}\ \sin^2\left(\frac{n\pi}{2}\right) \qquad \text{(A4.5.14a)}$$

$$\int_{a/2}^{a} \sin\frac{n\pi x}{a}\ (b/\mathbf{i})\frac{d}{dx}\left(\sin\frac{n\pi x}{a}\right)dx = -\frac{ib}{2}\left(1+2\cos(n\pi)\right)\sin^2\left(\frac{n\pi}{2}\right) \qquad \text{(A4.5.14b)}$$

$$\int_a^b x^n dx = \frac{b^{n+1}}{n+1} - \frac{a^{n+1}}{n+1} \qquad \text{(n positive)} \qquad \text{(A4.5.15)}$$

$$\int_a^b \frac{1}{x}\,dx = \ln(b) - \ln(a) \qquad \text{(A4.5.16)}$$

$$\int_a^b e^{-cx}\,dx = \frac{1}{c\,e^{ac}} - \frac{1}{c\,e^{bc}} \qquad \text{(A4.5.17)}$$

$$\int_a^b e^{-x^2}\,dx = b\,e^{-x^2} - a\,e^{-x^2} \qquad \text{(A4.5.18)}$$

6. Vectors and Tensors

We limit our examples to three-dimensional cartesian vectors and tensors. A function is said to be scalar if it has a single value at a given location in space {x, y, z}. In contrast, a vector function has both a magnitude and a direction, and thus requires three numbers. We normally use boldface to differentiate between a vector, **F**(x, y, z) from a scalar, f(x, y, z):

$$\mathbf{F}(x,y,z) = f_x(x,y,z)\hat{i} + f_y(x,y,z)\hat{j} + f_z(x,y,z)\hat{k} \qquad \text{(A4.6.1)}$$

where $\hat{i}$, $\hat{j}$, and $\hat{k}$ are unit vectors in the x, y and z directions, respectively. A unit vector is a unitless vector that points in a particular direction with a length equal to unity.

The dot product of two vectors, **A** and **B**, produces a scalar which is equal to the product of the magnitudes of the two vectors times the cosine of the angle, θ, between the two vectors:

$$\mathbf{A}\cdot\mathbf{B} = |\mathbf{A}||\mathbf{B}|\cos\theta = a_x b_x + a_y b_y + a_z b_z \qquad \text{(A4.6.2)}$$

where we have adopted the abbreviation, a_x, to represent $a_x(x, y, z)$, the component of the vector **A** along the x axis. The magnitude of a vector is obtained by taking the square root of the sum of the squares,

$$|\mathbf{A}| = \sqrt{a_x^2 + a_y^2 + a_z^2} \qquad \text{(A4.6.3)}$$

The cross product of two vectors produces another vector which is determined by the evaluation of the following determinant:

$$\mathbf{A}\times\mathbf{B} = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

$$= \left(a_y b_z - a_z b_y\right)\hat{i} + \left(a_z b_x - a_x b_z\right)\hat{j} + \left(a_x b_y - a_y b_x\right)\hat{k} \qquad \text{(A4.6.4)}$$

There are also a number of operations on vectors and scalars which evaluate the partial derivatives of these elements. A majority are based on the "del" or "grad" operator:

$$\nabla = \hat{i}\frac{\partial}{\partial x} + \hat{j}\frac{\partial}{\partial y} + \hat{k}\frac{\partial}{\partial z} \tag{A4.6.5}$$

When this operator is applied to a scalar function, it returns a vector which has a magnitude equal to the slope of the function in the direction of the greatest rate of change of the function, and which points in the same direction. This function is of great utility in both classical and quantum mechanical calculations. This operator can also be applied to a vector, and this operation results in the formation of a second rank tensor of $3\times3 = 9$ components.

More common are the two operations that involve the combination of the "del" operator with the dot or cross product operations to produce the divergence or curl of the function. The divergence of a vector produces a scalar quantity given by the following formula:

$$\nabla \cdot \mathbf{A} = \frac{\partial a_x}{\partial x} + \frac{\partial a_y}{\partial y} + \frac{\partial a_z}{\partial z} \tag{A4.6.6}$$

The curl of the same vector, **A**, is represented as curlA, or $\nabla \times \mathbf{A}$, and is generated via evaluation of the following determinant:

$$\nabla \times \mathbf{A} = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ a_x & a_y & a_z \end{vmatrix} \tag{A4.6.7}$$

Finally, we note that an important operator, $\nabla^2 = \nabla \cdot \nabla$, known as the Laplacian is defined by the relationship:

$$\nabla^2 \mathbf{A} = \frac{\partial^2 a_x}{\partial x^2} + \frac{\partial^2 a_y}{\partial y^2} + \frac{\partial^2 a_z}{\partial z^2} \tag{A4.6.8}$$

7. Matrices

A majority of calculations on the electronic and vibrational properties of molecules are based on the use of matrix mechanics, which allows the representation of very complex systems in a systematic and conveniently programmable form. A square matrix is one which has the same number of rows as columns, and in the case of a 3×3 matrix, can be represented as follows:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{A4.7.1}$$

where we refer to the rows of a matrix via the first subscript and the columns by the second subscript. A matrix is called "square" if it has the same number of rows and columns. A matrix is called "diagonal" if only the elements along the "diagonal", a_{ii}, are nonzero. A unit matrix is a special case of a diagonal matrix, and is generated by the equation $a_{ij} = \delta_{ij}$, where δ_{ij} is the delta function, which equals zero when $i \neq j$ and 1 when i = j. If we use the same convention to represent a matrix **B**, then the sum of two matrices, **A** + **B**, produces a new matrix, **C**, generated by the summation of the corresponding elements:

$$c_{ij} = a_{ij} + b_{ij} \tag{A4.7.2}$$

Thus, two matrices can only be added if they have the same number of rows and columns. Matrix multiplication is more complicated than the simple multiplication of corresponding elements. Rather, each new element is generated by a summation of individual products:

$$\mathbf{C}_{[n\times m]} = \mathbf{A}_{[n\times \ell]} \bullet \mathbf{B}_{[\ell\times m]} \tag{A4.7.3}$$

$$c_{ij} = \sum_{k=1}^{\ell} a_{ik} b_{kj} \tag{A4.7.4}$$

Thus, two matrices can only be multiplied if the first matrix has the same number of columns as the second matrix has rows. If **A** is a n× l matrix and **B** is a l× m matrix, then the resultant matrix **C** has dimensions n × m. The result of matrix multiplication is normally not commutative: $\mathbf{A} \bullet \mathbf{B} \neq \mathbf{B} \bullet \mathbf{A}$.

8. Matrix Operators

A matrix operator is a set of instructions, defined for some vector space, for changing one vector into a second vector belonging to the same space. Operators and transformation matrices are essentially equivalent. Usually, the term "operator" is reserved for the analysis or manipulation of physical quantities while "transformation" is applied to the manipulation of coordinates or coordinate systems.

A linear operator obeys the following relationships:

$$\mathbf{A}(c\alpha) = c\mathbf{A}\alpha, \text{ where c is a real or complex constant} \qquad \text{(A4.8.1)}$$

$$\mathbf{A}(\alpha + \beta) = \mathbf{A}\alpha + \mathbf{A}\beta, \text{ where } \alpha \text{ and } \beta \text{ are vectors} \qquad \text{(A4.8.2)}$$

A linear operator is represented by a square matrix. A Hermitian operator is a linear operator with real diagonal elements and real or complex off-diagonal elements. It is self adjoint, which means that $a_{ij} = a_{ij}^*$, where the "*" superscript indicates the complex conjugate (see E.2.15 above). Thus, if all elements are real, a Hermitian operator is a square symmetric matrix. A Hermitian operator in an n-dimensional vector space has n distinct eigenvectors and n real eigenvalues. One or more of the eigenvalues may be identical, indicating a degeneracy due to symmetry or accident. If **H** is the Hermitian operator, then we can write:

$$\begin{aligned} \mathbf{H}\varphi^1 &= h_1\varphi^1 \\ \mathbf{H}\varphi^2 &= h_2\varphi^2 \\ &\vdots \\ \mathbf{H}\varphi^n &= h_n\varphi^n \end{aligned} \qquad \text{(A4.8.3)}$$

where φ^i represents the ith eigenvector and corresponds to h_i, the ith eigenvalue. We can write the above set of n equations in a compact form as follows:

$$\mathbf{HC} = \mathbf{CE} \qquad \text{(A4.8.4)}$$

where **E** is now a n×n diagonal matrix with the eigenvalues along the diagonal ($h_i = \mathbf{e_{ii}}$), and **C** is a n×n square matrix with the eigenvectors in columns. If we take the inverse of **C** to produce $\mathbf{C}^{-1}$ (the inverse is defined such that $\mathbf{C}^{-1}\bullet\mathbf{C} = 1$), and multiply both sides by $\mathbf{C}^{-1}$, we get:

$$\mathbf{C}^{-1}\mathbf{HC} = \mathbf{E} \qquad \text{(A4.8.5)}$$

which represents a fundamental mathematical operation in quantum mechanics. Viewed in this form, the eigenvector matrix **C** is said to diagonalize the matrix **H** to generate the diagonal matrix **E** containing the eigenvalues. The operation $\mathbf{C}^{-1}\mathbf{HC}$ represents a unitary transformation of **H**. Modern molecular orbital theory relies on matrix diagonalization to find the energies and coefficients of the molecular orbitals that describe the wavefunctions of the electrons.

Appendix 5

Values of the Fundamental Constants and Derived Relationships in the SI and cgs Unit Systems[a]

Quantity	Symbol	SI System	CGS System
speed of light in vacuum	c	2.99792458×10^{8} m/s	2.99792458×10^{10} cm/s
permeability of vacuum	μ_o	$4\pi\times10^{-7}$ N m^{-2}	1 (dimensionless in emu)
permittivity of vacuum $(\mu_o c^2)^{-1}$	ε_o	$8.854187817\times10^{-12}$ F/m	1 (dimensionless in esu)
elementary charge	e	1.60217733×10^{-19} C	4.8032068×10^{-10} cm$^{3/2}$ g$^{1/2}$/s
gravitational constant	G	6.67428×10^{-11} m^3/(kg s^2)	6.67428×10^{-8} cm^3/(g s^2)
standard gravity	g_o	9.80665 m/s^2	980.665 cm/s^2
planck constant	h	6.6260689×10^{-34} J s	6.6260689×10^{-27} erg s
planck reduced (h/2π)	$\hbar$	$1.054571628\times10^{-34}$ J s	$1.054571628\times10^{-27}$ erg s
planck mass $[(hc/2\pi G)^{1/2}]$	m_P	2.17644×10^{-8} kg	2.17644×10^{-5} g
planck constant (molar)	$N_A h$	$3.990312682\times10^{-10}$ J s/mol	3.990312682×10^{-3} erg s/mol
	$N_A hc$	0.1196265647 J m/mol	1.1962658647×10^{8} erg cm/mol
standard atmosphere	atm	101325 Pa (exact)	1.01325×10^{6} dyn/cm^2 (exact)
permeabilty (dry air)	μ_{air}	$4\pi\times10^{-7}$ N/m^2 (exact)	1 (dimensionless in emu)
permittivity (dry air at STP)	ε_{air}	1.000590 F/m	1.000590×10^{-2} F/cm
spec. heat (dry air const. P)	Cp_{air}	1.00468×10^{3} J/(kg K)	1.00468×10^{7} erg/(g K)
spec. heat (dry air const V)	Cv_{air}	7.17625×10^{2} J/(kg K)	7.17625×10^{6} erg/(g K)
app. mol. wt. (dry air)	Mw_{air}	2.89652×10^{-2} kg/mol	28.9652 g/mol
mol. gas const. (1kg dry air)	R_{air}	287.05 J/kg K	2.8705×10^{6} erg/(g K)
Avogadro constant	N_A	6.02214179×10^{23} mol^{-1}	6.02214179×10^{23} mol^{-1}
atomic mass unit	u	1.66053878×10^{-27} kg	1.66053878×10^{-24} g
electron volt	eV	$1.602176487\times10^{-19}$ J	$1.602176487\times10^{-12}$ erg
faraday constant	F	9.6485342×10^{4} C/mol	2.892557×10^{14} cm$^{3/2}$ g$^{1/2}$/(s mol)
Boltzmann constant (R/N_A)	k	1.3806504×10^{-23} J/K	1.3806504×10^{-16} erg/K
	k/h	2.083674×10^{10} Hz/K	2.083674×10^{10} Hz/K
stefan-boltzmann constant	σ	5.67051×10^{-8} W/(m^2 K^4)	5.67051×10^{-3} erg/(cm^2 s K^4)
first radiation constant ($2\pi hc^2$)	c_1	3.7417749×10^{-16} W m^2	3.7417749×10^{-3} erg cm^2/s
second radiation constant (hc/k)	c_2	0.01438769 m K	1.438769 cm K
wien displacement law constant	b	2.897768×10^{-3} m K	0.2897768 cm K
ideal gas molar volume at STP	V_m	22.41400 L/mol	22414.00 cm^3/mol
molar gas constant	R_g	8.314472 J/mol K	8.314472×10^{7} erg/(mol K)
		8.314472 Pa m^3/mol K	8.314472×10^{7} dyn cm/mol K
		0.082057459 L atm/mol K	82.057459 cm^3 atm/mol K

Appendix 5 continued

Quantity	Symbol	SI System	CGS System
fine-structure constant ($\mu_o ce^2$ 2h)	α	$7.2973525376\times10^{-3}$	$7.2973525376\times10^{-3}$
rydberg constant ($m_e c\alpha^2/2h$)	R_∞	1.0973731568×10^{7} m^{-1}	1.0973731568×10^{5} cm^{-1}
	$R_\infty hc$	2.17987197×10^{-18} J	2.17987197×10^{-11} erg
bohr radius ($\alpha/4\pi R_\infty$)	a_o	$5.291772086\times10^{-11}$ m	5.291772086×10^{-9} cm
quantum of circulation	$h/2m_e$	3.63694752×10^{-4} m^2/s	3.63694752 cm^2/s
elementary charge	e	$1.602176487\times10^{-19}$ C	4.80320680×10^{-10} $cm^{3/2}\,g^{1/2}/s$
	e/h	2.41798945×10^{14} A/J	8.0655410×10^{-3} $1/(s\,cm^{1/2}\,g^{1/2})$
magnetic flux quantum (h/2e)	Φ_o	2.06783367×10^{-15} Wb	2.06783367×10^{-7} Mx
josephson frequency-voltage quo.	2e/h	4.8359789×10^{-14} Hz/V	4.8359789×10^{22} $s/(cm^{3/2}\,g^{1/2})$
quantized hall resistance	R_H	25812.807 Ω	25812.807 Ω
bohr magneton ($eh/4\pi m_e$)	μ_B	9.27400915×10^{-24} J/T	9.27400915×10^{-21} erg/Gs
		5.788381756×10^{-5} eV/T	4.2543812×10^{-10} Ry cm^2/Mx
nuclear magneton ($eh/4\pi m_p$)	μ_N	5.05078324×10^{-27} J/T	5.05078324×10^{-24} erg/Gs
		3.152451233×10^{-8} eV/T	1.44616226×10^{6} Ry cm^2/Mx
electron mass	m_e	9.10938215×10^{-31} kg	9.10938215×10^{-28} g
compton wavelength ($h/m_e c$)	λ_c	$2.426310218\times10^{-12}$ m	$2.426310218\times10^{-10}$ cm
classical electron radius ($\alpha^2 a_o$)	r_e	$2.817940289\times10^{-15}$ m	$2.817940289\times10^{-13}$ cm
thomson cross section [$(8\pi/3)r_e^2$]	σ_e	6.65245856×10^{-29} m^2	6.65245856×10^{-25} cm^2
electron magnetic moment	μ_e	-9.2847638×10^{-24} J/T	-9.2847638×10^{-21} erg/Gs
proton mass	m_p	1.67262164×10^{-27} kg	1.67262164×10^{-24} g
proton compton wavelength ($h/m_p c$)	λ_{cp}	1.32141002×10^{-15} m	1.32141002×10^{-13} cm
proton magnetic moment	μ_p	1.41060761×10^{-26} J/T	1.41060761×10^{-23} erg/Gs
neutron mass	m_n	1.6749286×10^{-27} kg	1.6749286×10^{-24} g
neutron magnetic moment	μ_n	9.6623707×10^{-27} J/T	9.663707×10^{-24} erg/Gs

(a) From Mohr, Taylor and Newell, Rev. Mod. Phys. **80**, 633-730 (2008).

Appendix 6
Selected Conversion Factors

Property	from	to	multiply by
length	cgs	SI	10^{-2} m / cm
	cgs	(Å)	10^{8} Å / cm
	SI	(Å)	10^{10} Å / m
	au	SI	$5.29177249 \times 10^{-11}$ m
	au	(Å)	0.529177249 Å
	(inches)	(cm)	2.54 cm / in
	(foot)	(m)	0.3048 m / foot
time	au	SI; cgs	$2.41888434 \times 10^{-17}$ s
velocity	au	SI; cgs	2.18769141×10^{6} m / s
energy	cgs	SI	10^{-7} J / erg
	cgs	(eV)	$6.24150637 \times 10^{11}$ eV / erg
	SI	(eV)	$6.24150637 \times 10^{18}$ eV / J
	au	SI	$4.35974819 \times 10^{-18}$ J
	au	(eV)	27.2113961 eV
force	cgs	SI	10^{-5} N / dyne
	SI	au	1.21377939×10^{7} / N
	au	cgs	0.00823872945 dyne
	au	SI	$8.23872945 \times 10^{-8}$ N
linear momentum	cgs	SI	10^{-5} N / dyne
	au	cgs	$1.99285336 \times 10^{-19}$ dyne s
	au	SI	$1.99285336 \times 10^{-24}$ N s
current	cgs	SI	$3.335640952 \times 10^{-10}$ A s/esu
	SI	cgs	2.99792458×10^{9} esu / (A s)
	au	cgs	1.98571165×10^{7} esu / s
	au	SI	0.00662362109 A
magnetic flux density	cgs	SI	10^{-4} T / gauss
	SI	cgs	10^{4} gauss / T
	au	cgs	2.35051808×10^{9} gauss
	au	cgs	0.0784048438 esu s / cm^3
	au	SI	235051.808 T
magnetic dipole moment	cgs	SI	10^{-3} gauss J / (erg T)
	SI	cgs	10^{3} erg T / (gauss J)
	au	cgs	$1.85480308 \times 10^{-20}$ erg/gauss
	au	SI	$1.85480308 \times 10^{-23}$ J / T
	au	SI	$1.85480308 \times 10^{-23}$ A m^2

(a) Symbols for the atomic units are not included for clarity and convenience. The number of decimal digits meet or exceed the number justified based on the accuracy of the fundamental constants. The 3.335640952 factor has an exact value of 10/2.99792458.

Appendix 6. Selected Conversion Factors[(a)]

Property	from	to	multiply by
dipole moment	cgs	SI	$3.335640952 \times 10^{-12}$ C m / (cm esu)
	SI	cgs	$2.99792458 \times 10^{11}$ cm esu / (C m)
	cgs	(debye)	10^{18} debye / (cm esu)
	SI	(debye)	$2.99792458 \times 10^{29}$ debye / (C m)
	eÅ	(debye)	4.8032068 debye/ (eÅ)
	au	cgs	$2.54174776 \times 10^{-18}$ cm esu
	au	SI	$8.47835793 \times 10^{-30}$ C m
	au	(debye)	2.54174776 debye
electric field	cgs	SI	29979.2458 V cm^2 / (esu m)
	SI	cgs	$3.335640952 \times 10^{-5}$ esu m / (V cm^2)
	au	cgs	1.71525604×10^{7} esu / cm^2
	au	SI	$5.14220823 \times 10^{11}$ V / m
Polarizabilities:			
first order	cgs	SI	$1.11265006 \times 10^{-16}$ F m^2 / cm^3
	SI	cgs	$8.98755178 \times 10^{15}$ cm^3 / (F m^2)
	cgs	(Å)	10^{24} $Å^3$ / cm^3
	au	cgs	$1.48184744 \times 10^{-25}$ cm^3
	au	SI	$1.64877764 \times 10^{-41}$ F m^2
second order	cgs	SI	$3.71140110 \times 10^{-21}$ esu F^2 m^3 / (C cm^5)
	SI	cgs	$2.69440024 \times 10^{20}$ C cm^5 / (esu F^2 m^3)
	esu	cgs	1 cm^5 / esu
	esu	SI	$3.71140110 \times 10^{-21}$ F^2 m^3 / C
	au	cgs	$8.63922007 \times 10^{-33}$ cm^5 / esu
	au	SI	$3.20636109 \times 10^{-53}$ F^2 m^3 / C
third order	cgs	SI	$1.23799015 \times 10^{-25}$ esu^2 F^3 m^4 / (C^2 cm^7)
	SI	cgs	$8.07760869 \times 10^{24}$ C^2 cm^7 / (esu^2 F^3 m^4)
	esu	cgs	1 cm^7 / esu^2
	esu	SI	$1.23799015 \times 10^{-25}$ F^3 m^4 / C^2
	au	cgs	$5.03669416 \times 10^{-40}$ cm^7 / esu^2
	au	SI	$6.23537777 \times 10^{-65}$ F^3 m^4 / C^2

(a) Symbols for the atomic units are not included for clarity and convenience. The number of decimal digits meet or exceed the number justified based on the accuracy of the fundamental constants. The 3.335640952 factor has an exact value of 10/2.99792458.

Appendix 6B. Energy Conversion Factors

$E = h\nu = hc\tilde{\nu} = kT$; $E_m = N_A E$

	wavenumber $\tilde{\nu}$	frequency ν	energy E			molar energy E_m		temperature T
	cm^{-1}	MHz	aJ	eV	E_h	kJ/mol	kcal/mol	K
$\tilde{\nu}$: 1 cm^{-1}	1	2.997925×10^{4}	1.986447×10^{-5}	1.239842×10^{-4}	4.556445×10^{-6}	11.96266×10^{-3}	2.85914×10^{-3}	1.438769
ν: 1 MHz	3.33564×10^{-3}	1	6.626076×10^{-10}	4.135669×10^{-9}	1.519830×10^{-10}	3.990313×10^{-7}	9.53708×10^{-8}	4.79922×10^{-5}
1 aJ	50341.1	1.509189×10^{9}	1	6.241506	0.2293710	602.2137	143.9325	7.24292×10^{4}
E: 1 eV	8065.54	2.417988×10^{8}	0.1602177	1	3.674931×10^{-2}	96.4853	23.0605	1.16045×10^{4}
1 E_h	**219474.63**	**6.579684×10^{9}**	**4.359748**	**27.2114**	**1**	**2625.500**	**627.510**	**3.15773×10^{5}**
1 kj/mol	83.5953	2.506069×10^{6}	1.660540×10^{-3}	1.036427×10^{-2}	3.808798×10^{-4}	1	0.239006	120.272
E_m 1 kcal/mol	349.755	1.048539×10^{7}	6.947700×10^{-3}	4.336411×10^{-2}	1.593601×10^{-3}	4.184	1	503.217
T: 1 K	0.695039	2.08367×10^{4}	1.380658×10^{-5}	8.61738×10^{-5}	3.16683×10^{-6}	8.31451×10^{-3}	1.98722×10^{-3}	1

Examples of the use of this table: 1 aJ is equivalent to 50341 cm^{-1} ; 1 eV is equivalent to 96.4853 kJ mol^{-1} ; 1 Hartree (= E_h) is equivalent to 27.2114 eV

Adapted from I. Mills, T. Cvitas, K. Homann, N. Kallay, & K. Kuchitsu, Quantities, Units and Symbols in Physical Chemistry, Blackwell Scientific Publications, 1988.

Appendix 7A. Definition and Numerical Assignments of the Base Atomic Units

Physical quantity	*Name of unit*	*Symbol for unit*	*Definition and value of unit in SI*	*Definition and value of unit in cgs*
mass	electron rest mass	m_e	$m_e \approx 9.109\ 382\ 15 \times 10^{-31}$ kg	$m_e \approx 9.109\ 382\ 15 \times 10^{-28}$ g
charge	elementary charge	e	$e \approx 1.602\ 177\ 33 \times 10^{-19}$ C	$e \approx 4.803\ 206\ 80 \times 10^{-10}$ esu
action	Planck constant/2π	$\hbar$	$\hbar = h/2\pi \approx 1.054\ 571\ 63 \times 10^{-34}$ J s	$\hbar \approx 1.054\ 571\ 63 \times 10^{-27}$ erg s
length	bohr	a_o	$4\pi\varepsilon_0\hbar^2/m_e e^2 \approx 5.291\ 772\ 11 \times 10^{-11}$ m	$\hbar^2/(e^2 m_e) \approx 5.291\ 772\ 11 \times 10^{-9}$ cm
energy	hartree	E_h	$\hbar^2/m_e a_o^2 \approx 4.359\ 7442 \times 10^{-18}$ J	$e^4 m_e/\hbar^2 \approx 4.359\ 7442 \times 10^{-11}$ erg

Appendix 7B. Definition and Numerical Assignments of Selected Derived Atomic Units

Physical quantity	*Relationship*	*Definition and value of unit in SI*	*Definition and value of unit in cgs*
time (Dirac)	$\hbar/E_h$	$2\varepsilon_0^2 h^3/(e^4 m_e \pi) \approx 2.418\ 884\ 326\ 51 \times 10^{-17}$ s	$\hbar^3/(e^4 m_e) \approx 2.418\ 884\ 326\ 51 \times 10^{-17}$ s
time (Schrödinger)	h/E_h	$4\varepsilon_0^2 h^3/(e^4 m_e) \approx 1.519\ 829\ 846\ 00 \times 10^{-16}$ s	$2\pi\hbar^3/(e^4 m_e) \approx 1.519\ 829\ 846\ 00 \times 10^{-16}$ s
velocity	$a_o E_h/\hbar$	$e^2/(2\varepsilon_0 h) \approx 2.187\ 691\ 263 \times 10^6$ m s^{-1}	$e^2/\hbar \approx 2.187\ 691\ 263 \times 10^8$ cm s^{-1}
frequency	E_h/h	$e^2 m_e/(4\varepsilon_0^2 h^3) \approx 6.581\ 134\ 437 \times 10^{15}$ s^{-1}	$e^2 m_e/h^3 \approx 6.581\ 134\ 437 \times 10^{15}$ s^{-1}
force	E_h/a_o	$e^6 m_e^2 \pi/(4\varepsilon_0^3 h^4) \approx 8.238\ 7225 \times 10^{-8}$ N	$e^6 m_e^2/\hbar^4 \approx 0.00823\ 872\ 25$ dyne
momentum, linear	$\hbar/a_o$	$e^2 m_e/(2\varepsilon_0 h) \approx 1.992\ 851\ 556 \times 10^{-24}$ N s	$e^2 m_e/\hbar \approx 1.992\ 851\ 556 \times 10^{-19}$ dyne s
electric current	$eE_h/\hbar$	$e^5 m_e \pi/(2\varepsilon_0^2 h^3) \approx 0.006\ 623\ 617\ 82$ A	$e^5 m_e/\hbar^3 \approx 1.985\ 711\ 65 \times 10^7$ esu/s
resistance	$h/2e^2$	$h/2e^2 \approx 12\ 906.403\ 7787\ \Omega$	$h/2e^2 \approx 12\ 906.403\ 7787\ \Omega$
magnetic flux density (B_0)	$\hbar/(e\ a_o^2)$	$e^3 m_e^2 \pi/(2\varepsilon_0^2 h^3) \approx 235\ 051.736$ T	$e^3 m_e^2/\hbar^3 \approx 2.350\ 517\ 36 \times 10^9$ gauss
magnetic dipole moment	$e\hbar/m_e$	$eh/(2m_e\pi) \approx 1.854\ 801\ 827 \times 10^{-23}$ J T^{-1}	$e\hbar/m_e \approx 1.854\ 801\ 827 \times 10^{-20}$ erg/gauss
electric field (strength) (E_0)	$E_h/(e\ a_o)$	$e^5 m_e^2 \pi/(4\varepsilon_0^3 h^4) \approx 5.142\ 206\ 57 \times 10^{11}$ V m^{-1}	$e^5 m_e^2 \hbar^{-4} \approx 1.715\ 256\ 04 \times 10^7$ esu/cm^2
magnetic field strength (H_0)	$\hbar/(\mu_0\ e\ a_o^2)$	$e^3 m_e^2 \pi/(2\mu_0\varepsilon_0^2 h^3) \approx 1.870\ 482\ 28 \times 10^{11}$ A m^{-1}	$e^3 m_e^2/\hbar^3 \approx 2.350\ 518\ 08 \times 10^9$ gauss
electric dipole moment	ea_o	$\varepsilon_0 h^2/(e m_e \pi) \approx 8.478\ 352\ 846 \times 10^{-30}$ C m	$\hbar^2/(e\ m_e) \approx 2.541\ 747\ 76 \times 10^{-18}$ esu cm
polarizability	$4\pi\varepsilon_0\ a_o^3$	$4\varepsilon_0^4 h^6/(e^6 m_e^3 \pi^2) \approx 1.648\ 777\ 274 \times 10^{-41}$ F m^2	$\hbar^6/(e^6 m_e^3) \approx 1.48184744 \times 10^{-25}$ cm^3
second order polarizability	$a_o^3\ e^3\ E_h^{-2}$	$16\varepsilon_0^7 h^{10}/(e^{11} m_e^5 \pi^3) \approx 3.206\ 361 \times 10^{-53}$ F^2 m^3 C^{-1}	$\hbar^{10}/(e^{11} m_e^5) \approx 8.639\ 22 \times 10^{-33}$ cm^5/esu
third order polarizability	$a_o^4\ e^4\ E_h^{-3}$	$64\varepsilon_0^{10} h^{14}/(e^{16} m_e^7 \pi^4) \approx 6.235\ 380 \times 10^{-65}$ F^3 m^4 C^{-2}	$\hbar^{14}/(e^{16} m_e^7) \approx 5.036\ 69 \times 10^{-40}$ cm^7/esu^2

Appendix 8
SI, cgs, esu, emu, Gaussian and Atomic Units

Scientific programmers rarely deal with units within their programs, having stripped these units from the equations prior to entering the math operations. Nevertheless, keeping track of the units is essential or the math will be meaningless. The purpose of this appendix is to help students deal with the various types of units that are in common use. The majority of this book uses the "Systeme International d'Unites" or International System of Units which has been recommended for use since 1960. Nevertheless, other unit systems are used as convenient and common. Many researchers working in photonics have adopted esu units, while a majority of theoretical papers present both formulas and computational results using atomic units (au). Chemists and biophysicists have been relatively slow to convert to SI units, and to read the literature one must be conversant with both cgs and SI nomenclature. In a majority of instances, conversion is a trivial exercise. However, in the case of electrostatics, magnetostatics and nonlinear optics, conversion is neither trivial nor transparent. The confusion arises primarily due to the presence of factors such as 4π, ε_o (the permittivity of vacuum), μ_o (the permeability of vacuum) and c (the speed of light in vacuum) in equations based on SI units which are absent from the comparable equation based on cgs or esu systems. Thus, in this appendix we also discuss the origin and rational for introducing these quantities, which should help a student navigate among the various unit systems.

Historically, scientists used a combination of systems to define units. The basic system was known as the centimeter-gram-second (cgs) system, within which magnetic quantities were measured by using electromagnetic units (emu) and electrostatic quantities were measured in electrostatic units (esu). This approach is called a "mixed" system of units and because geometric factors are often excluded from the basic definitions, "irrational". There are significant advantages in adopting the SI system of units which is both internally coherent and geometrically rational. First, the ampere is now treated as a fundamental unit in addition to the meter, kilogram and second. By introducing the ampere as a fundamental unit, electrostatic and electromagnetic interactions can be handled in a more direct and rationale way and square roots no longer appear in dimensional expressions. Second, all electrostatic and electromagnetic quantities can now be expressed in units coherent with the four fundamental units. The frequent appearance of c in the mixed unit systems is associated with the fact that there is no separate dimension for electromagnetic properties and thus the generation of derivative units is required. In the SI system, whenever the speed of light does appear in an equation it is invariably associated with special relativity. Third, the SI units are rationalized so that factors of 4π only appear when they are justified based on geometry. Fourth, the permeability and permittivity of free space are now included in equations and are assigned specific values. Within the mixed system of units, both of these constants are dimensionless and equal to unity. Thus, they are frequently omitted from

cgs-based equations, which obscures their fundamental relevance. In addition, the magnetic field strength (represented by the symbol **H**) is often used in place of the magnetic flux density (represented by the symbol **B**) because both are identical under vacuum conditions within the mixed system. All observable physical properties, however, depend solely on **B**. Any confusion is avoided within the SI system because **B** and **H** differ by a factor of the permeability (μ_o) forcing the use of **B** in the appropriate equations.

Despite the above mentioned advantages of the SI system, there are some disadvantages that have precluded uniform adoption in many realms of science and engineering. This problem is most evident in chemistry and where the SI units of hyperpolarizabilities are complex and appear to lack any intuitive origin. Many researchers have therefore retained the use of electrostatic units in both experimental and theoretical publications. Furthermore, there is a vast amount of early literature that is based on the cgs system. Thus, one is forced to understand both the SI and the mixed system of units. Following our discussion of the SI system, we will discuss the cgs, electrostatic and atomic unit systems in sufficient detail to facilitate use and interconversion.

A8.1. SI units

The International System of Units (SI) is also known as the (rationalized) MKSA system because four of the base units employed are the meter, kilogram, second and ampere. There are three types of SI units: base, supplementary and derived. The seven base units are regarded as being dimensionally independent and are listed in Table A8.1

Table A8.1. The Seven Base Units of the International System of Units.

Physical quantity	SI Name	SI Symbol	Dimension
length	meter	m	L
mass	kilogram	kg	M
time	second	s	T
electric current	ampere	A	A
thermodynamic temperature	kelvin	K	
luminous intensity	candela	cd	
amount of substance	mole	mol	

The meter is defined as of 1983 as the path length traveled by light in a vacuum during a time interval of 1/(299 792 458) of a second. By making this definition of

length, the speed of light is now an exact quantity (c = 299 792 458 m/s). This definition of length has significant advantages over that which prevailed in the middle ages when the foot was used to measure distance and the value depended upon the ruling monarch's anatomy. The kilogram is defined based on the 1901 international prototype of the kilogram. The second is defined as of 1967 to be equal to the duration of 9 192 631 779 periods of the radiation corresponding to the transition between the tow hyperfine levels of the ground state cesium-133 atom. Thus, time marked by using a cesium clock represents the best possible method of monitoring both absolute and relative time in the laboratory. The ampere is defined as of 1948 as the current necessary to produce a force of 2×10^{-7} newton per meter of length along two infinitely long parallel conductors of negligible cross-section separated by one meter in vacuum. While such a definition is important historically, one is better off thinking of the ampere as representing the current equivalent of one coulomb of charge per second flowing through a medium. The kelvin is defined as of 1967 in terms of the triple-point of water. The zero of the Celsius scale is defined to equal 273.15 K exactly. The mole is defined as of 1971 to equal the amount of substance of a system containing as many elementary entities as there are atoms in 0.012 kilograms of carbon-12. The use of a mole unit requires the explicit specification of the elementary entities, which may be fundamental particles, ions, atoms or molecules or even specified groups of such entities. The fundamental constant that defines this unit is Avogadro's number or constant which equals approximately 6.022137×10^{23} entities per mole. The candela is defined as of 1979 to be the luminous intensity of a source that emits monochromatic radiation at 540×10^{12} hertz with a radiant intensity of 1/683 watt per steradian. The origin of the term candela derives from the historical term "candle" or "candlepower". Although the candela is one of the seven fundamental units, it is rarely used in practice. Light intensity or flux is normally defined in terms of energy (in joules), or in terms of power or radiant flux by using the derived unit of watts (watt = J s^{-1}).

All of the SI units, both fundamental and derived (see below), can appear with prefixes to represent decimal multiples and submultiples. The prefix symbols are listed in Table A.1.2 and should always appear in roman (i.e. not italic) type with no space between the prefix and the unit symbol. For example, one femtosecond is represented by the compound symbol fs and equals 10^{-15} s. Multiple prefixes are not correct, and thus one should use mg and not μkg to represent 10^{-3} grams and pF instead of μμF to represent 10^{-12} farads.

Table A8.2. SI Prefixes

Fraction	Prefix	Symbol	Multiple	Prefix	Symbol
10^{-1}	deci	d	10	deka	da
10^{-2}	centi	c	10^{2}	hecto	h
10^{-3}	milli	m	10^{3}	kilo	k
10^{-6}	micro	μ	10^{6}	mega	M
10^{-9}	nano	n	10^{9}	giga	G
10^{-12}	pico	p	10^{12}	tera	T
10^{-15}	femto	f	10^{15}	peta	P
10^{-18}	atto	a	10^{18}	exa	E

Although all quantities can be defined in terms of the seven base units, there are many derived units which are in active use and have assigned symbols. The derived units are given in Table A8.3. In addition, there are a number of units that are not part of the SI system but which remain in active use for historical reasons, human inertia or simple convenience. These supplementary units are presented in Table A8.4. Because supplementary units are not formally defined, it is common practice to use the full name of the unit rather than the symbol when first introducing the unit in scientific publications.

Table A8.3. Names and Symbols for Selected SI Derived Units

Physical quantity	SI unit	Abbreviation	Definition in base SI units
frequency	hertz	Hz	s^{-1} (cycle per second)
Celsius temperature	Celsius	°C	T(°C) = T(K) – 273.15
force	newton	N	m kg s^{-2}
energy	joule	J	N m = m^2 kg s^{-2}
pressure	pascal	Pa	N m^{-2} = m^{-1} kg s^{-2}
power	watt	W	J s^{-1} = m^2 kg s^{-3}
electric charge	coulomb	C	A s
electric potential	volt	V	J C^{-1} = m^2 kg s^{-3} A^{-1} = K A^{-1} s^{-1}
electric resistance	ohm	Ω	V A^{-1} = m^2 kg s^{-3} A^{-2}
electric conductance	siemens	S	$Ω^{-1}$ = m^{-2} kg^{-1} s^3 A^2
electric capacitance	farad	F	C V^{-1} = m^{-2} kg^{-1} s^4 A^2
magnetic flux	weber	Wb	V s = m^2 kg s^{-2} A^{-1}
magnetic flux density	tesla	T	V s m^{-2} = kg s^{-2} A^{-1}
inductance	henry	H	V A^{-1} s = m^2 kg s^{-2} A^{-2}
plane angle	radian	rad	1 = m/m^{-1}
solid angle	steradian	sr	1 = m^2/m^{-2}

Table A8.4. Names and Symbols for Selected Other Units in Active Use

Physical quantity	Name	Symbol	Value or defining relationship
time	minute	min	60 s
	hour	h	3600 s
	day	d	86 400 s
plane angle	degree	°	$(\pi/180)$ rad (57.295779513082 rad/degree)
binary data entities	bit	b	1 (quantum of binary information)
	byte	B	8 bits
	word	w	16 bits unless specified otherwise
length	ångstrom	Å	10^{-10} m = 10^{-8} cm
	micron	μ[(a)]	μm =10^{-6} m
	inch	in	0.0254 m
volume	liter	l, L	$dm^3 = 10^{-3}\ m^3 = 1000\ cm^3$
energy	electronvolt	eV	$e \times V \approx 1.602177328 \times 10^{-19}$ J
	calorie	cal	4.184 J
	Hartree	E_h	$\approx 4.35974819 \times 10^{-18}$ J$\approx$27.2114 eV
	wavenumber	cm^{-1}	$\approx 1.98644746 \times 10^{-23}$ J
	kilokaiser	kK	1000 cm^{-1}
mass	amu[(b)]	u	$= m_a(^{12}C)/12 \approx 1.6605402 \times 10^{-19}$ kg
	dalton	Da	$= u \approx 1.6605402 \times 10^{-19}$ kg
	electron volt	eV[(c)]	$eV/c^2 \approx 1.78266270 \times 10^{-36}$ kg
magnetic flux density	gauss	G	10^{-4} T = 10^{-4} V s m^{-1}
dipole moment	debye	D	$\approx 3.335640952 \times 10^{-30}$ C m

(a) The use of μ instead of μm is a common though incorrect practice.

(b) The amu or unified atomic mass unit is equal to the mass of carbon–12 divided by 12. This unit is called a dalton in biology, although this name and the symbol Da, is not formally approved by international convention.

(c) The use of eV, MeV ($\approx 1.783 \times 10^{-30}$ kg) or GeV ($\approx 1.783 \times 10^{-27}$ kg) without the c^2 denominator is a common simplification, particularly in particle physics.

A8.2. The cgs, esu, emu and Gaussian Unit Systems

There are four separate unit systems that have been in use extensively prior to the adoption of the international system of units. These are the centimeter-gram-second unit system (cgs), the electromagnetic unit system (emu) and the electrostatic unit system (esu). When combined in a single coherent system, these three systems are called the Gaussian unit system or simply the "cgs" system. While the latter designation is not strictly correct, it is very common and derives from the fact that the three basic units are the centimeter (length), gram (mass) and second (time). All other units are derived from these three, which is a basic limitation of the system. As such, equations involving electrostatic or magnetic quantities require the ad hoc introduction of the speed of light.

A majority of equations and experimental results can be converted from the cgs to the SI unit system by simply noting that the fundamental equations involving kilogram, -meter-second units are now converted into centimeter-gram-second units. This conversion is for the most part trivial. The difficulty arises when working with electrostatic or electromagnetic relationships. We will confine our discussion to the electrostatic relationships which are particularly relevant to this book.

The electrostatic force between charges Q_1 and Q_2 separated by a distance x within a dielectric medium of relative permittivity (dielectric constant) ε_r is written in SI units in the following form:

$$E_{SI} = \frac{Q_1 Q_2}{4\pi \varepsilon_0 \varepsilon_r x} \qquad \text{(joules)} \qquad \text{(A8.1)}$$

where the charges are in coulombs, the distance is in meters, the permittivity of free space is in faradays/meter, and the relative permittivity is dimensionless. To convert this equation into the cgs or Gaussian system, the $4\pi\mu_o$ is simply removed, and the charges are expressed in electrostatic units.

$$E_{cgs} = \frac{Q_1 Q_2}{\varepsilon_r x} \qquad \text{(ergs)} \qquad \text{(A8.2)}$$

Two things are happening simultaneously. First, the 4π is removed from the expression because we are moving from a rational system to an irrational system, which basically does not take the geometric origins of the basic equations properly into account. Second, we are combining the permittivity of the vacuum directly into the expressed charge on the electron. Thus, in a real sense, we are expressing the charge of

the electron in terms of its effective electrostatic force when observed in a vacuum. Perhaps the best way to look at this transformation is in terms of the charge on an electron. This charge when expressed within the cgs or esu system is equal to the charge on an electron based on the SI unit system divided by $(4\pi\varepsilon_o)^{1/2}$. The following sequence takes the charge from the SI unit system to the cgs and esu system.

$$e = 1.60217733\times10^{-19}\ \text{C} \qquad \text{(SI units)}$$

$$\mu_0 = 4\pi\times10^{-7}\ \text{N m}^{-2} \qquad \text{(SI units)}$$

$$\varepsilon_o = (\mu_o c^2)^{-1} = 8.854187817\times10^{-12}\ \text{F/m} \qquad \text{(SI units)}$$

$$4\pi\varepsilon_0 = 1.11265006\ \times10^{-10}\ \text{A}^2\ \text{s}^4\ \text{kg}^{-1}\ \text{m}^{-3} \qquad \text{(SI units)}$$

$$\frac{e}{4\pi\varepsilon_0} = 1.51890736\times10^{-14}\ \text{m}^{3/2}\ \text{kg}^{1/2}\ \text{s}^{-1} \qquad \text{(mks units)}$$

$$= 4.80320680\times10^{-10}\ \text{cm}^{3/2}\ \text{g}^{1/2}\ \text{s}^{-1} \qquad \text{(cgs units)}$$

$$= 4.80320680\times10^{-10}\ \text{esu} \qquad \text{(esu units)}$$

We note that an esu is equal to one $\sqrt{\text{erg cm}}$. Thus, we see that the electron charge in cgs or esu units is not really a charge at all, but the square root of the vacuum electrostatic energy times the square root of the length unit used to define the energy.

All of the other quantity equations and quantities expressed within the mixed unit system can be derived by carrying out the following set of substitutions, where the superscript "(ir)" simply indicates the use of an irrational quantity.

$\varepsilon^{(ir)} = 4\pi\varepsilon$	permittivity	(A8.3)
$\mu^{(ir)} = \mu/4\pi$	permeability	(A8.4)
$D^{(ir)} = 4\pi D$	electric displacement	(A8.5)
$H^{(ir)} = 4\pi H$	magnetic field	(A8.6)
$\chi_e^{(ir)} = \chi_e/4\pi$	electric susceptibility	(A8.7)
$\chi^{(ir)} = \chi/4\pi$	magnetic susceptibility	(A8.8)

Thus, for example, the electromagnetic force between current elements $I_1 d\mathbf{l}_1$ and $I_2 d\mathbf{l}_2$ in vacuum is written in the SI system as follows:

$$\mathbf{F} = (\mu_0/4\pi)\, I_1 d\mathbf{l}_1 \times (I_2 d\mathbf{l}_2 \times \mathbf{r})/r^3 \qquad \text{(A8.9)}$$

and in the Gaussian system as:

$$\mathbf{F} = \mu^{(ir)}\, I_1 d\mathbf{l}_1 \times (I_2 d\mathbf{l}_2 \times \mathbf{r})/r^3 \qquad \text{(A8.10)}$$

where $\mu^{(ir)} = 1$ (emu system).

To facilitate conversion from the mixed systems into the SI unit system, Appendix C is provided which explicitly presents some of the key conversion factors relevant to the material presented in this book. In addition, the fundamental constants which are presented in Appendix B are given in terms of both the SI as well as the mixed unit "cgs" system.

A8.3 The Atomic Unit System

The key atomic units are presented in Appendix 7. Quantum mechanical calculations of the electronic properties of atoms, molecules and other nanoscale systems are often carried out by using atomic units (abbreviated: a.u.). These units form a coherent set that has three principal advantages in theoretical chemistry. First, theoretical expressions are simplified significantly because the fundamental constants e, $\hbar$ and m_e, which often appear in such expressions, are now (by definition) equal to unity. Thus, these constants can be removed from the expressions, which leads to simplification (though sometimes at the expense of clarity). Selected examples will be provided below. Second, calculations based on ab initio theoretical methods are typically carried out in atomic units because no parameterization is adopted and all calculations are carried out in reference to specific relationships among the fundamental constants. By expressing all calculated values in terms of atomic units, explicit assignment of the values of the fundamental constants are avoided. Theoretical calculations can often be carried out at a level of accuracy that exceeds that of the experimental assignments of the fundamental constants. Thus, by expressing calculated values in atomic units, the results are invariant to subsequent revisions of the numerical values of the fundamental constants. The third advantage derives from the fact that the atomic units have been assigned based on the characteristics of atoms, and thus the values of atomic and molecular properties when expressed in atomic units are typically within one order of magnitude of the atomic unit which represents that property. For example, a typical interatomic distance is 1.5×10^{-10} m but in atomic units it is about 2.83 atomic units (or as we will see below, 2.83 a_0, where a_0 is the symbol used to represent the atomic unit of length). A typical dipole moment of a polar molecule is 1.2×10^{-29} C m, but in atomic units it is about 1.4 ea_0, where ea_0 is the derived atomic unit for dipole moment. Thus atomic units are the natural units of atoms, molecules and quantized nanoscale systems.

There are five base atomic units as defined in Appendix 7A. The atomic unit of energy, E_h, is called the Hartree and is approximately equal to twice the ionization energy of the ground state hydrogen atom. The atomic unit of length, a_o, is called the

bohr and is approximately equal to the distance of maximum radial density from the nucleus in the hydrogen 1s orbital. The atomic unit of charge is equal to the elementary proton charge, e, and the atomic unit for mass is assigned as the electron rest mass. The atomic unit of action is assigned as the reduced Planck constant, $h/2\pi$, or $\hbar$. It should be noted, however, that only four of the five base units (m_e, e, $\hbar$ and a_0) are actually independent. However, by convention, E_h, is also included as a base unit because of its importance in theoretical calculations. The interrelation among these five units is conveniently summarized in the following set of equalities:

$$E_h = \hbar^2/(m_e a_0^2) = e^2/(4\pi\varepsilon_0 a_0) = m_e e^4/(4\pi\varepsilon_0\hbar)^2 \qquad \text{(A8.11)}$$

The five base units are all given or have previously been assigned unique symbols as shown in Appendix 7A. When specifying these units, it is best to use these symbols. For example, an energy of 22.7 Hartrees should be written 22.7 E_h rather than 22.7 a.u. A length of 1.512 bohr should be written 1.512 a_0 rather than 1.512 a.u. However, the use of the symbol a.u. (or au) is common, even with the base units, and one must be prepared for both the preferred and the common practice when reading the literature. All other physical quantities can be defined in terms of these five basic atomic units and selected examples are presented in Appendix 7B. Again, it is preferable to use a combination of the base symbols to represent the derived units. For example, a dipole moment of 0.62 in atomic units should be written as 0.62 ea_0.

We noted above that atomic units greatly simplify quantum mechanical equations by setting $e = \hbar = m_e = 1$. A salient example is the Schrödinger equation for the hydrogen atom:

$$-\frac{\hbar}{2\,m_e}\,\nabla^2_{\mathbf{r}}\,\psi - \frac{e^2}{4\pi\varepsilon_0\mathbf{r}}\,\psi = E\psi \qquad \text{(A8.12)}$$

where $\nabla^2_{\mathbf{r}}$ indicates second derivatives with respect to the spatial coordinate (or vector) **r**. Dividing both sides by the various representations of E_h given in Eq. A.3.1, we get

$$-\tfrac{1}{2}\,a_0^2\,\nabla^2_{\mathbf{r}}\,\psi - \frac{a_0}{\mathbf{r}}\,\psi = \frac{E}{E_h}\,\psi \qquad \text{(A8.13)}$$

The above equation, while simplified, is still expressed in SI units. Transformation to atomic units is carried out by expressing the coordinate variable as **r'** where $\mathbf{r}' = \mathbf{r}/a_0$ and the eigenvalues (energies) in Hartrees, i.e. $E' = E/E_h$. Note that **r'** and E' are now dimensionless variables that express both dimension and energy in atomic units. Eq. A.3.3 can now be written as

$$-\tfrac{1}{2}\,\nabla^2_{\mathbf{r'}}\,\psi - \frac{1}{\mathbf{r'}}\,\psi = E'\,\psi \qquad \text{(A8.14)}$$

The above equation is normally described as 'being expressed in atomic units', and represents the principal approach adopted by theoreticians because of the notational simplicity. Nevertheless, this approach has the distinct disadvantage of creating a formalism that precludes dimensional analysis and therefore can lead to confusion and error. To compound the problem, many publications and some textbooks switch back and forth without explicit notification to the reader and without expressly changing the variable notation.

The advantage of writing computer programs using atomic units is significant. A simple example is the electrostatic energy associated with two charges, Q_1 and Q_2, separated by a distance r. In SI units, this is given by Eq. A8.15.

$$E_{SI}(J) = \frac{Q_1(C)\,Q_2(C)}{4\pi\,\varepsilon_0(F/m)\,r(m)} \qquad \text{(A8.15)}$$

where the units for each variable are given in parentheses. The minimum number of floating point operations is five, and because computer languages such as Fortran, C and C^{++} do not handle symbolic math, the units must be dropped and the program properly commented to keep track of the unit conventions and conversions that are taking place implicitly. Furthermore, because joules are inconvenient units for expressing molecular energies, conversion of the results to more conventional units (such as eV) invariably takes place prior to printing out the results. In contrast, the same equation in atomic units is as follows:

$$E_{au} = \frac{Q_1\,Q_2}{r} \qquad \text{(A8.16)}$$

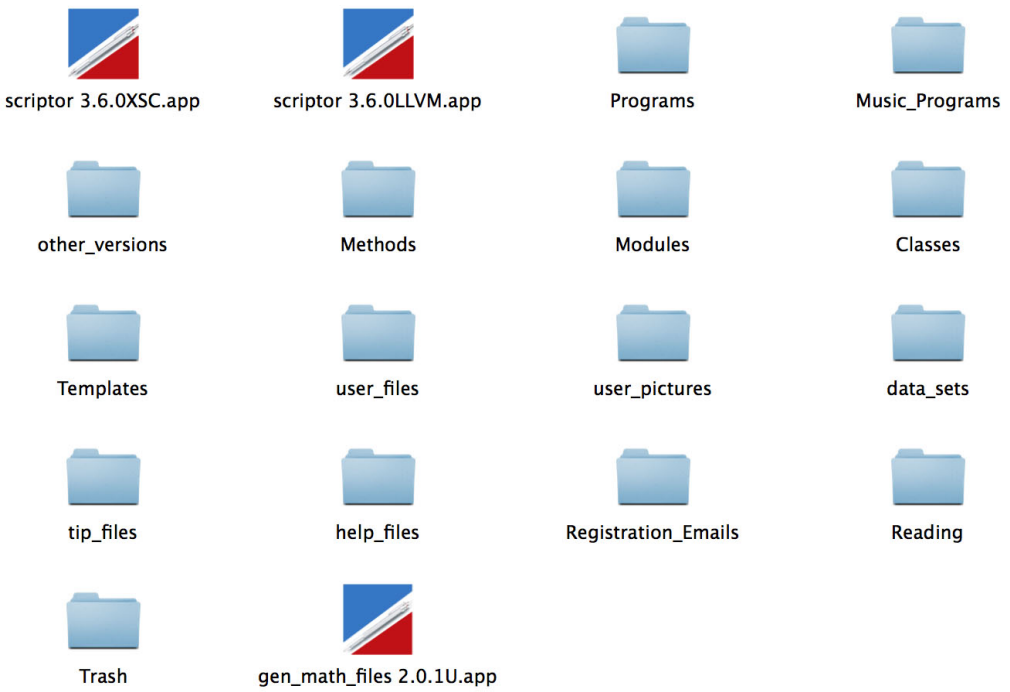
where all variables are assumed to be in atomic units. Only two floating point operations are required and the issue of unit conversions is obviated. A single comment can be added to the top of the computer program stating that all variables are expressed in atomic units unless noted otherwise. Finally, because the Hartree is a convenient output variable for energy, most programs express the eigenvalues and the total energies in Hartrees in the final output. The probability of a programming error is minimized and the computer code is significantly easier to read and debug. While the above example is trivial, when one is programming a two electron integral involving four-fold summations, the use of atomic units represents a significant advantage.

In closing, we note that our atomic unit convention is not unique, and other possibilities can and have been adopted in the literature. One alternative definition that is still in use is the use of h rather than $\hbar$ to represent the atomic unit of action. This

approach is often invisible to the reader, however, because the atomic unit of energy is still expressed in Hartrees. The early literature, however, is not consistent with respect to the atomic unit of energy and occasionally the adopted atomic unit is $e^2/(2a_0)$ (cgs) = $e^2/(8\pi\varepsilon_0 a_0)$ (SI) which is one-half of a Hartree (~13.6057 eV). Furthermore, some publications adopt the Rydberg constant for infinite mass ($R_\infty = E_h/2hc$ = 109737.315 cm^{-1}) as the atomic unit of energy. In a majority of such papers, however, the energies are explicitly listed in "rydbergs", so there is no ambiguity. Note that a rydberg is also equivalent in energy to one-half of a Hartree (~13.6057 eV). [Dividing an energy by hc converts from joules to wavenumbers: E(joules) = $hc\tilde{\nu}$ (cm^{-1}) . A wavenumber is a common unit of energy, particularly in spectroscopy, and is often expressed in kiloKaysers (kK = 1000 cm^{-1}).]

Appendix 9
Installation of Scriptor and MathScriptor

Scriptor and Mathscriptor reside within the same program but must be run within the Scriptor Environment to function properly. The Environment folder is called ScriptorWin or ScriptorOSX depending upon the target operating system (Windows or Mac OSX). The ScriptorWin environment has also been tested on Window 7 and 8 and function without problems, although Windows 8.1 can cause problems. These environments are available from www.mathscriptor.org. Versions are also available for Linux or Mac Classic by contacting R.R. Birge (rbirge@uconn.edu). The Environment contains a collection of folders which provide resources, either for the program to function or for the user (documentation, help files, tips, etc.). The Mac OSX environment is shown below.



Some of these folders are empty and are present to help organize the users files and program elements. The Windows environment looks comparable but the program names have "exe" extensions and each .exe file will have a library file (Lib) associated with it. MathScriptor uses many resources inside the data_sets folder. These resource files include routines and tables that are used for numerical integration, prime numbers and interpolation data for various functions. These files are only loaded when needed which allows the program to be many times smaller than would be necessary if all the data were included inside the program. During startup, MathScriptor checks to make

sure these resources are available and notifies the user if any are missing or corrupted. If any problems are found, a separate program is included inside the "other_versions" folder called gen_math_files that can be run to restore the necessary files. This same folder also holds versions of both scriptor and gen_math_files for other selected operating systems. However, it is still preferable to download the entire environment for the operating system of interest to make sure all of the files are optimized for the operating system.

The Environment can be placed at any desired location on the computer that is convenient for the user. One can even run the program from an environment placed on an external usb drive, a convenience when using the program as part of a class laboratory. However, a usb drive is much slower than a hard drive and the performance of the program is often proportional to the random access speed of the parent drive. If the programs are small and do not involve heavy resource usage, the use of a usb drive is a viable option.

As you choose the location of your program, keep in mind the following issues:

1. If the computer you are using is a public computer, but you have an account on the computer, it is best to put the entire environment inside your document folder. Other users should not have access to this folder and thus your work is protected. If the computer is public and you have no private area on the hard disk, a usb drive that you own may be the optimal choice.

2. If the computer belongs to you and you are the only user, select any location that is convenient for you. However, avoid using the desktop if it is cluttered because such placement increases the probability that you might accidentally throw the entire folder away. However, placing an alias of the program on the desktop is safe because it is trivial to recreate one should it be lost.

3. On a Windows system, it is common to place the environment inside your Programs Folder and place an alias of the application on the desktop. On a Mac OSX system, it is common to place the environment inside your user folder and place an alias of the application in the dock.

4. Make sure to backup the entire environment on a regular basis.

Appendix 10
Troubleshooting and Frequently Asked Questions

This appendix provides solutions to the most common problems experienced by Scriptor users, and answers the more common questions. These are listed in no particular order.

1. The use of arbitrary precision arithmetic, matrix operations and compilation options are not available even though I registered the program and should have access to MathScriptor.

Ans. Although registering the program is a necessary condition for access to MathScriptor, the program needs to be switched into MathScriptor mode by the user. There are two ways to switch into MathScriptor mode. First, select "Switch to MathScriptor" under the Compiler menu. Second, go to the Preferences tab and check "Startup program in MathScriptor". The only time one should avoid using MathScriptor is when one is working on a computer with less than 500 MB of memory available. MathScriptor can run on less than that amount, but it will be making heavy use of the disk for paging, and the result will be sluggish performance.

2. The Run and/or Stop buttons have disappeared from Main. How do I get them back?

Ans. These buttons can become invisible if a run-time error was encountered in a program, or the user has flipped back and forth between panels quickly. These buttons can be restored easily by pressing the Full Screen button twice to toggle into and back from full-screen mode

3. My program had errors in it, which I fixed. How do I now get rid of the line numbers that were added to help identify the error so I can try running the program again.

Ans. The **Stop** button turns into the **Reset** button when the compiler adds line numbers to the program to help the user locate the errors. Pressing the **Reset** button will remove the line numbers. Line numbers can also be toggled on and off by selecting the "Add or Remove line numbers" under the Debug menu, or using the keyboard shortcut ctrl-L (command-L on Mac OSX).

4. There are many versions of Scriptor which have the same version number, but different letters at the end. What do these letters mean?

Ans. The following table provides a summary based on version 3.6.0.

Windows Version*	Mac Version	Target OS and Comments
3_6_0.exe	3.6.0.app	Compiled for legacy operating systems such as XS or OSX 10.6
3_6_0LLVM.exe	3.6.0LLVM.app	Compiler allows for optimization. Most stable implementation.
3_6_0XS.exe	3_6_0XS.app	Advanced compiler reports both errors and warnings.
----	3_6_0XSC.app	Compiled for most recent Mac operating systems based on Cocoa**

*The LLVM and XS versions on windows require two files for each installation. One file is the program file as named above, and the second file is a library folder of the same name but ending in "Libs" instead of ".exe". Both the program and the library folder must be placed at the top of the environment folder. In general, the LLVM and XS versions are more stable on Windows 7 and Windows 8 whereas the legacy version may be more stable on earlier operating systems.

**Cocoa is the newest Apple programming interface (API) and provides full access to the various capabilities of MAC OSX. Carbon is an older version of the API that has been available since 2000 and is less powerful, but because it is no longer being modified, is also more stable. Eventually, Apple will fully deprecate Carbon in favor of Cocoa, and developers have been asked to develop new programs using Cocoa if they expect these programs to run under upcoming versions of Mac OSX. Because we are in a transition period, both carbon and cocoa versions are provided.

5. I upgraded Scriptor to a new version, and now I am getting all sorts of strange errors when I start it up. What did I do wrong?

Ans. The most common cause of this problem is program placement outside of the environment folder. Unless the program is moved into the top of the ScriptorWin (or ScriptorOSX) folder, it cannot locate the necessary resources which generates errors when the verification process is carried out. On Windows, this problem can be caused by a missing library ("Libs") folder. A third possibility is that the environment is damaged. Copy the current environment to a backup folder, and then select the menu item "Clean Environment" under the Help menu. Then follow any instructions that are presented at the end of the process. If MathScriptor continues to generate checksum errors upon startup, it may be necessary to restore the math library files by running the gen_math_files program which is present inside the environment.

6. My program is in an infinite loop. How can I stop it without losing my program, as I forgot to save the program before I ran it?

Ans. Infinite loops are common, particularly during the early stages of writing a new program. The following operations will not only force quit *Scriptor* but restore your program. You can force quit the program by using the *control-alt-delete* key combination on Windows or *command-option-escape* key combination on Mac OSX. These key combinations open the *Task Manager* on Windows or the *Force Quit Applications* window on the Mac which allows you to force quit *Scriptor*. After restarting *Scriptor,* it now remains to restore the program that was running. If the preference, Open most recent work when program starts up, was checked, your old program will be loaded into the Main program field. If not, go to the backups folder inside Programs, and open up the most recent file. The backup folder contains copies of all the programs that have been run, and the file with the most recent create date is the one that was running most recently. It would be a good idea to fix the loop before running the program again.

7. The trigonometric functions are not returning the correct answers.

Ans. All the internal trigonometric functions that accept an angle as an argument, expect the argument to be in radians. Similarly, all internal trigonometric functions that return angles, return values in radians. One radian equals 180°/π = 57.2958°. The internal constant **const_degree** when multiplied by degrees will generate radians at full precision. Hence, to calculate the sin of 45° use sin(45*const_degree).

const_degree = π /180= 0.017453292519943…

8. Is there a function that automatically prints all of the significant digits from my calculation on double precision variables.

Ans. Yes. The function **convert_to_string**(x) will convert a number to a string and retain all the significant digits. It will use exponential format when necessary, however.

9. Can my program read from an Excel file or write to an Excel file?

Ans. No. But there is a close option. The spreadsheet that is present in the data sets panel can import tab delimited (*.txt) or comma delimited (*.csv) files exported from excel. Once the data are loaded into the Scriptor spreadsheet, all the data can be manipulated or altered by the program. The resulting spreadsheet can then be exported to Excel if desired in tab delimited (*.txt) or comma delimited (*.csv) format.

10. My backup folder inside the programs folder is getting really big? Can I trim it without causing problems.

Ans. Although the text files containing the programs that you run are very small, they can add up if you are doing a lot of programming. It is not unusual for a student to generate a 200 MB size folder after a semester of programming. It is recommended you backup your entire Scriptor folder occasionally to a CD. After doing that, all the contents of the backup folder older than a recent date can be trashed if you are running low on disk space.

11. Somehow *Scriptor* is no longer registered and I do not have access to *MathScriptor* anymore. How do I register my program again.

Ans. Registered programs can become un-registered for a variety of reasons, most often because the environment was moved to a new location or a new computer. You can register (re-register) your program as many times as necessary. If you followed instructions, you stored your registration email, or the attached text file that was sent with your registration email, in the Registration_Emails folder inside your environment. Go to the preferences panel and press the **Open Registration Email** button. Direct the dialogue to your registration email file. Your program will be automatically registered. If you prefer, you can enter the information from your email by hand. You can register your program on as many computers as you wish. But every time you copy your environment to a new computer, you will have to register the program for use on the new computer.

d

e

f

g

h

i

j

k

l

m

n

o

p

t

u

v

w

y

z