Introduction to IGOR Pro Programming

IGOR Pro is a bona fide programming platform. It is software that can interpret long sequences of commands, compile them (translate them into lower level binary sequences that the computer can understand) and execute the programs. It has a simple development environment, complete with a user interface and a debugger. The system is designed primarily to allow you to build programs from IGOR's large repository of built-in functions. Hence, IGOR programming proceeds with greater speed than traditional programming because one need not write analytical or statistical algorithms from scratch. Many such functions are already built into IGOR.

IGOR programming focuses on automating tasks, but it is not limited to automation. It is also possible to write programs that do complex calculations, acquire data from instruments, and create elaborate user interfaces. The objective of this primer is to help you get started on the path to becoming an IGOR programmer.

This manual will also serve as a *very* brief survey of IGOR's built-in functions. It starts slowly, but the pace will quicken quite rapidly.

Before You Begin

Prerequisites

This tutorial requires IGOR Pro version 5.04 or later. If you do not have version 5.04 or later, please update to the latest IGOR Pro. In IGOR Pro 5 you can do this by choosing Updates from the IGOR Help menu.

There are some requirements to using this manual. Prior to proceeding with programming IGOR, it is *essential* to take the Guided Tour of IGOR Pro (in **Volume I** of the IGOR Pro manual set). You should be comfortable with IGOR waves, with IGOR experiments, with creating graphs and tables manually, and with running IGOR procedures written by others. If not, re-take the Guided Tour or refer to IGOR's manual and get acquainted with IGOR before you begin programming.

We also assume that you have some experience programming at any level in any language so that basic programming concepts like functions, parameters and loops are not entirely foreign to you.

Programming Essentials

IGOR programming is much easier than conventional programming because IGOR Pro already has innumerable built-in functions that perform most tasks. For example, you need not write your own curve fitting algorithm. The `CurveFit` command readily does that. Hence, IGOR programming focuses more on automating tasks, rather than implementing analyses from scratch. Of course, you are free to do that in IGOR if you so desire.

In addition, IGOR programming is easier because it is highly interactive. You can enter a function in IGOR's procedure window and compile and execute it right away from the command line.

Companion Experiment

Much of the material presented here works on data in the IntroToIGORProgramming.pxp experiment file. Open it now in IGOR so that you can follow along. The waves that you need for the examples are provided in the root data folder of this experiment. (From the menu choose Data and then Data Browser.)

The Structure of the Tutorial

This tutorial begins with some quick drills to get you acquainted with the the process of entering code into the Procedure window, compiling the code and executing the code. The most basic elements are introduced first, and the subsequent examples build on the simpler ones. New elements are introduced in **bold** text. The explanations are offered mainly in the text that follows each example, and partly in the comments that accompany each function. As you enter code, examine it line by line and read the comments.

Although you can cut and paste the functions into the Procedure window, we recommend that you **manually type** the functions into the Procedure window or use the built-in templates (Figure 1). This will help you get used to the mechanics of IGOR programming.

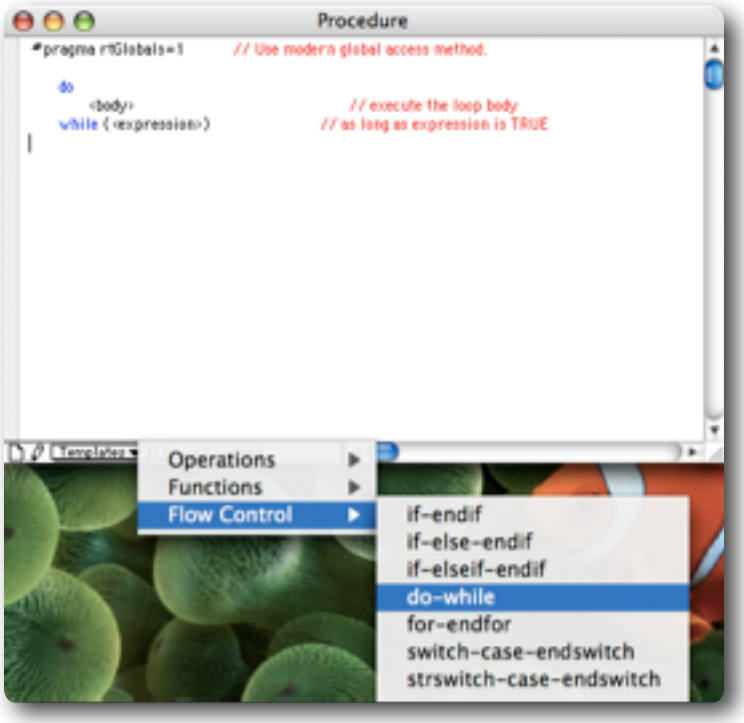

Figure 1 Using the templates menu to insert a programming element into the procedure window. In this case, it is a **`do-while`** loop.

Fundamentals

The Procedure Window

In IGOR Pro, program code is entered into procedure files. Procedure files can be standalone files or they can be part of an experiment package. Each experiment includes one special, built-in procedure file which you view through the Procedure Window. You can use the Windows menu to open the procedure window or, better yet, use a keyboard shortcut – cmd-M (*Macintosh*) or Ctrl-M (*Windows*).

Open the procedure window now using the keyboard shortcut. Now close it using cmd-W or Ctrl-W.

Programs entered into the built-in procedure window become part of that particular experiment. As such they are accessible from that experiment only. The built-in procedure window is a good place to experiment with programming, and that is how we will use it.

Once your program is working, you can save it as a separate, standalone procedure file that can be accessed from any experiment. You should save general procedures that you want to use over and over as standalone procedure files. For now, we stay with the built-in Procedure window.

Two features are important when you are a beginning programmer. First, text entered into the Procedure window is color coded as it is typed. The colors make it easy to avoid conflicts between user-selected object names and IGOR commands and statements. If the color does not look right, you need to change the text. Second, you can insert a generic template for any IGOR command or statement from the "template" menu on the lower left of the Procedure window. Figure 1 shows how a `do-while` loop template is inserted. These templates are useful in the early stages when one

is inclined to forget a comma, a parenthesis, or a bracket. As you can see, these templates also come with some standard comments (the text appearing in red).

The Function

The basic IGOR programming unit is the function, and, in its simplest form it has the following structure:

```
Function Function_Name()
    IGOR commands
End
```

The first line is the declaration of the function itself, and the assignment of its name (letters, numbers and underscore character only), which is `Function_Name` in this case. It will become clear soon why the parentheses are **required**. After the function declaration come the commands that will be carried out when the function is *called*. These commands constitute the body of the function. The last line is the `End` statement which marks the end of the function.

The function is executed whenever it is called. You can call it from the command line or from another function. IGOR also has many *built-in* functions. The ones we write are *user-defined* functions (**IV** *Ch. 3*; this is how future references to the IGOR manual set will appear).

Writing a Function

Let's start with the obligatory "Hello World" example.

Open the Procedure Window (Command-M on Mac, Ctrl-M on Windows, or from the Procedure Window menu item under the Windows menu) and type the following:

```
Function HelloWorld()
    Print "Hello World!"
End
```

When you start typing, the "Compile" button becomes active. You can press it to start compiling your function. You can also switch to another window, an act that will trigger a compile. If you have any typographical or syntactical errors, IGOR will report them to you at this stage.

If the function compiles without any errors, you are ready to execute it (or call it). If the compiler discovers any errors, you can choose Edit Procedure or Quit Compile from the error dialog. Edit Procedure will take you to the position where the error was encountered in the Procedure window. Quit Compile will halt compiling and return you to IGOR Pro to perform other tasks.

Errors noted by the compiler are compile time errors. They are usually typographical or syntactical errors. Errors that are encountered when functions are executed are run time errors. Run time errors can be difficult to discover and correct and may require debugging. For example, telling IGOR to kill a wave that does not exist is a runtime error. More on this topic later.

Calling a Function

Once written, a function must be *called*. When a function is called, it is executed in its entirety. The function is most commonly called from the command line, from a menu, from another function or from an assignment statement.

Call the function you wrote above from the Command window by typing

```
HelloWorld()
```

into the command line area of the command window and pressing the Enter or Return key on your keyboard. You will see `Hello world!` printed in the history area of the Command window.

Let's call this function from an IGOR menu (**IV** *Ch. 5*). Type the following menu definition into the Procedure window.

```
Menu "Macros"
    "Hello World", HelloWorld()
End
```

After compiling, you see "Hello World" listed as an item under the Macros menu. Choosing it yields the same result as when you called `HelloWorld` from the command line.

So, now you know what the `Print` command does. It is a useful command in debugging run time errors.

The Return Value of A Function

Historically and generally, functions strictly calculated certain values and returned the values to the source from which they were called. This value can be a string (i.e., text), a real number, or a complex number. The function is no longer required to return a value to its caller, but if we require a value from a function, the `return` statement does this job.

Enter the following into the Procedure window.

```
Function ValueDemo()
    return pi*3^2
End
```

Now, execute the following command in the Command window.

```
Print “The area of a circle with a radius of 3 units is “, ValueDemo(),”
units squared.”
```

As you can see, the `return` statement defines the value that the function returns, the area of the circle in this instance.

By default, IGOR expects functions to return a real number. Let’s demonstrate the importance of value types by making ValueDemo a string function with the /S flag and compile it again.

```
Function/S ValueDemo()
    return pi*3^2
End
```

Compiling this function fails because IGOR reports a type mismatch. IGOR expects `ValueDemo` to return a string value, but the `return` statement returns a number. Invoking one IGOR built-in function solves the problem:

```
Function/S ValueDemo()
    return num2str(pi*3^2)
End
```

The `num2str` function converts the numerical value inside the parentheses that follow it into a string. There is a no longer a type conflict, and the function compiles and executes normally. Types

must be defined and observed for parameters as well because computer stores and processes numbers, text and complex numbers differently. And, yes, the reverse function, `str2num`, exists.

Of course, this is a fairly worthless function, for it is capable of evaluating the area of a circle with a fixed radius. We need to use *parameters* to make functions more general.

Parameters

When a function needs to operate on a particular object(wave, string or text variable, etc.), it must receive the information about the object as a *parameter* during the function call. The function must also contain instructions on how to treat each parameter. In other words, values, strings and waves are passed to functions as parameters.

The three parameters types are numeric (number, real and complex), string (i.e., text), and wave. Numeric parameters are declared with the `Variable` declaration. String parameters are declared with the `String` declaration. Waves are declared with the `Wave` declaration. Complex parameters are declared with the `/C` flag; i.e., `Variable/C`. Let's start a few drills with parameters by changing `ValueDemo` function to read:

```
Function ValueDemo(radius)
    Variable radius

    return pi*radius^2
End
```

Now you must supply a numeric parameter when calling `ValueDemo`.

Execute this line in the Command window:

```
Print "The area of a circle with radius = 2.75 in. is",ValueDemo(2.75)," sq.
in."
```

Now let's try a function with both a numeric and string parameter.

Enter this in the procedure window:

```
Function CircleArea(radius,units)
   Variable radius
   String units
   Print "If radius=",radius,units,", then area is ",pi*radius^2," square",
units
End
```

Now call the function from the command line with `CircleArea(8.5,"meters")` and examine the output in the history area. Values for the variables need to be entered in the order in which they appear in the function *declaration*. String values are marked by double quotes.

Save the experiment file. *Remember to save your files periodically.* At present, IGOR offers no backup solution. So, your only backup is to save your experiment and/or procedure files regularly.

Wave Parameters and Forming Wave Names

Wave parameters deserve special attention because mishandling waves is mishandling data, and you don't want to lose any data. Although waves can be accessed directly by a function, certain situations require that they be declared like string and numerical values before they can be used in a function. For example,

```
Function test1()
   Duplicate/O fluorescence fluortemp
End
```

test1 will execute successfully, and create a duplicate of the wave fluorescence. Modifying it this way will lead to failure:

```
Function test1()
   Duplicate/O fluorescence fluortemp
   fluortemp=fluorescence/3
End
```

The failure happens because the assignment statement (in bold) cannot act on an object that has not been declared, namely fluorescence. We must make a wave declaration to make the assignment successful. This modification will make the function work.

```
Function test1()
   Duplicate/O fluorescence, fluortemp

   Wave fluorescence

   fluortemp = fluorescence/3
End
```

Why do we not need to declare fluortemp also? Because fluortemp was created by test1 in the duplicate statement. As a result, test1 can be said to have sufficient awareness (an implicit one) of fluortemp. test1 lacks sufficient awareness of fluorescence for the assignment statement to work. Therefore, we provide test1 (explicitly) with the requisite awareness with the Wave declaration.

To make the function more general, we can rewrite it so that the data that we want to process is passed on to test1 as a wave parameter.

```
Function test1(wave2treat) //Function to get stats and print maximum

   Wave wave2treat   // Declares parameter
         //and makes the name wave2treat a local alias for original wave

   Duplicate/O wave2treat fluortemp
   fluortemp=wave2treat/3
End
```

Now, we must call test1 with the name of our wave: test1(fluorescence). The name of the wave is then stored in the wave parameter wave2treat. The wave declaration then makes the name wave2treat a local or temporary alias for fluorescence.

As you might have guessed, because actions applied to wave2treat are applied directly to fluorescence, you must exercise caution when invoking commands that alter the wave. For example, killwaves wave2treat will kill fluorescence. While you are testing a function, you may want to use the duplicate operation to create a copy of the wave, and carry out your transformations on the copy until your function is working properly. We all learn this lesson eventually after accidentally killing some original data.

Now, it would be nice to have a more descriptive name than fluortemp for the new wave. We can do this with the aid of a local string variable and one built-in IGOR function.

```
Function test1(wave2treat) //Function to get stats and print maximum
   wave wave2treat   //Wave reference receives wave at function call

//Store the name of the wave plus the string "_OneThird" into newWaveName
   string newWaveName=NameOfWave(wave2treat)+"_OneThird"

//Duplicate fluorescence as a wave named fluorescence_OneThird
   duplicate/O wave2treat $newWaveName

//Reference fluorescence_OneThird so that you can use it
   Wave newWave=$NewWaveName

//Use fluorescence_OneThird
   newWave=wave2treat/3
End
```

Execute the function as you did previously, and note the name of the new wave that was created. Here's what the commands do.

The string declaration creates a string consisting of the name of the wave to which the wave parameter `wave2treat` points (namely, `fluorescence`) and appends the text `_OneThird` to make the string value `fluorescence_OneThird`. The `Duplicate` command then creates a copy of `wave2treat` (i.e., `fluorescence`) with the name `fluorescence_OneThird`. Note that the `Duplicate` command must act on the content of the string variable. Therefore, we use the `$` to tell `Duplicate` that it should use the content of `newWaveName` to produce a wave named `fluorescence_OneThird`. If we omitted the `$`, `Duplicate` would create a wave literally named `newWaveName`.

The same principle is in action in the next `wave` declaration, which is required for the final assignment. Remember this distinction between a string parameter, `NewWavename`, and a literal string. Using the wrong object in a command will cause an error.

The bottom line is that a wave (also other variables and parameters) *must be declared before it can be used* in an assignment statement and certain operations. Absent an *implicit* declaration via operations like `Make` and `Duplicate`, you must make an *explicit* reference with the `Wave` declaration.

Of course, in the course of performing its duties, a function may require additional local variables. These can be declared in the normal fashion *after* declarations for parameters.

Formatting Output and Combining Commands

Now, let's make more readable output with the aid of the `Printf` operation. Execute the following in the Command window:

```
Printf "When r = 2.75, area is %+10.2f sq.in.\r", ValueDemo(2.75)
```

The `Printf` command can format the output. It requires an *escape character*, "`%`" in this instance. The escape character tells IGOR to start processing the following conversion specification. The conversion specification here is "`+10.2f`" and will format the the appearance of the values that follow the closing quotations. "`+`" tells IGOR to place the plus sign before the value, "`10`" tells IGOR to take 10 spaces to print the value, "`.2`" tells IGOR to report the value to the second decimal place, and "`f`" tells IGOR to use a floating point format.

After the closing quotations, IGOR expects a list of literal numbers, variables or function calls separated by commas.

Unlike `print`, `printf` does not go to the next line by default. Therefore, the carriage return command must be made explicit using the `\r` escape sequence.

Recap

So, functions have the following general structure:

```
Function Function_Name(numeric_parameter,string_parameter,wave_parameter)
    Variable parameter
    String parameter
    Wave parameter

    Variable local_variable
    String local_string

    Wave wave_name
    IGOR Commands and/or flow control statements

    return return_value
End
```

The parameter declarations **must** have the same name as the parameters named in the function declaration line. Variables declared later can have any name. Unless they are declared with the `/G` flag, all these variables are *local* in *scope*. This means that they you can refer to them in the body of the function within which they are declared, **only**, and they cease to exist when the function returns. Global variables (declared with `variable/G`) are *global in scope* and can be used by *any* function.

Modularity

As you might have deduced already, since a function can be called from another function, it is possible to encode the tasks that need to be performed into different functions so that each calculation or transformation can be performed when it is needed. To demonstrate this, let's write a series of functions that calculate all the values associated with a circle.

```
Function CirclePerimeter(radius)
    Variable radius
    return 2*pi*radius
End

Function CircleArea(radius)
    Variable radius
    return pi*radius^2
End

Function SphereVolume(radius)
    Variable radius
    return 4/3*pi*radius^3
End


Function PrintProperties(radius)
   variable radius

   Variable perimeter = CirclePerimeter(radius)
   Variable area = CircleArea(radius)
   Variable volume = SphereVolume(radius)
```

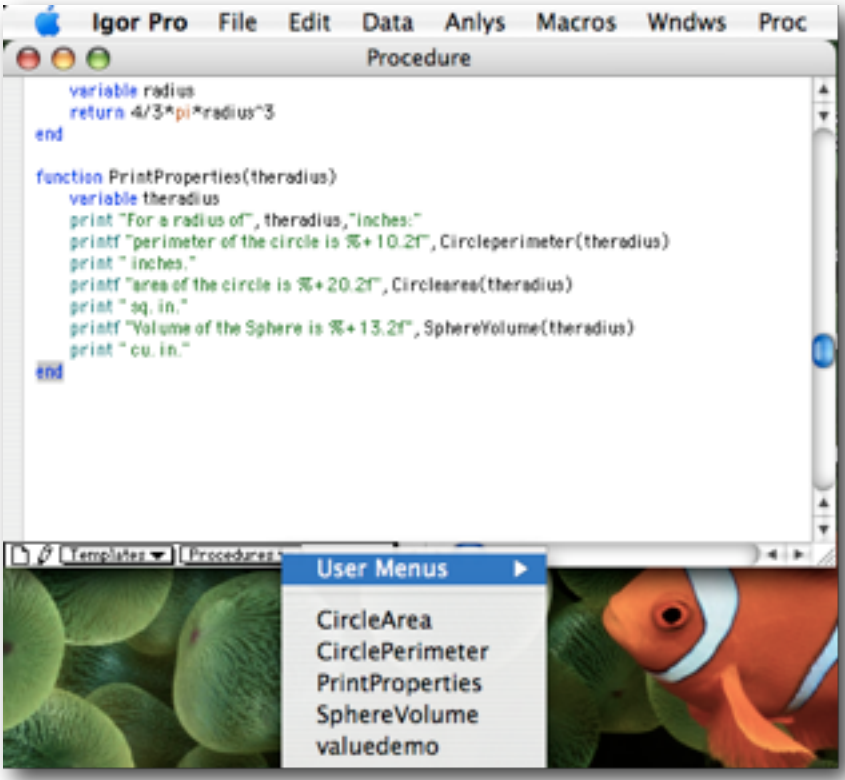

Figure 2 Using the Procedures menu to navigate to the function that needs attention.

```
  Printf "For a radius
of %.2f inches: ", ra-
dius

  String format =
"Perimeter=%+.2f in.,
area=%+.2f sq. in.,
volume=%+.2f cu. in.\r"
  Printf format, perime-
ter, area, volume
End
```

Note that the function `CircleArea` has been modified from the earlier version above.

Call the function with `PrintProperties(6.37)` , or any other number you want to choose as the radius. You can adjust the formatting parameters in the `printf` commands to get the desired number of decimal points.

Because each task is carried out by a separate function, each property of the circle can be accessed whenever it is needed. This example is trivial, of course, but, as you already surmise, we will employ modularity to separate complicated tasks.

`PrintProperties` can be said to be driving the other functions. As the more advanced examples will show, employing this strategy makes writing complicated programs much more manageable.

If you have many of functions in the procedure window, navigating to the function that you want to correct can become a hassle. Fortunately, you can quickly jump to the desired function by choosing it from the "Procedures" drag-down menu in the Procedures window (Figure 2). In that vein, choosing a descriptive name for the functions is helpful.

Simple Functions

The purpose of the previous sections was to acquaint you with IGOR Pro's programming environment and to make you comfortable with the process of writing functions, compiling them and executing them. Now, let's continue this process by writing short, useful functions.

Open IntroToIGORProgramming.pxp, and do all work within that file. Be sure to keep any and all waves upon which the following functions operate in the root folder. Do not create any data folders (in the Data Browser).

A Simple Tag

If you share the writer's obsession for marking peaks in your two-dimensional graphs with a tag, then you will love this function. Go ahead and execute (in the Command window)

```
Display fluorescence vs fluorwavelength
```

Bring up the cursors with the `showinfo` command, and choose a point near the peak by dragging the round cursor to the peak. Point number 99 is close enough. Now, enter this function in the procedure window.

```
Function TagPoint(traceName,tagpoint)  //tag trace at specified point
   String traceName  //receives name of trace to tag
   Variable tagpoint //receives point to tag

   Tag/F=0/L=1 $traceName,tagpoint,"\\OX" //tags graph with x-value
End
```

It is worthwhile to review the use of `$` here. The `Tag` operation is looking for the name of a trace in the graph. We have the name of the trace – it is in the parameter `traceName`. If we used "`traceName`" without the `$`, the Tag operation would look for a trace whose name is *literally* "`traceName`". Finding no such trace, it would return an error. We need a way to tell the `Tag` operation that the parameter we are supplying is not *literally* the name of the trace but rather is a string variable *containing* the name of the trace. That is what `$` does. It tells the operation to obtain the name by extracting the contents of the string variable that follows.

Mark point 99 by executing:

```
TagPoint("fluorescence",99)
```

Change the values of `/F` and `/L` to see what happens. Let's make this function easier to use before we discuss it. Let's re-write it so that it gets all of the necessary information from cursor A.

```
Function TagCursorA() //tag trace at specified point
   String traceName = CsrWave(A)
   Variable xValue = xcsr(A)

   Tag/F=0/L=1 $traceName, xValue, "\Z12\\OX"
End
```

This function does not require any values to be passed on to it when it is called. Therefore, it can be added to the menu of your choice.

Choose another point on `fluorescence` with cursor A (the circular one), and execute

```
TagCursorA()
```

Let's parse this function. First, let's start with IGOR's documentation of the `Tag` command. Read the IGOR documentation for `Tag`. The fastest way is to right-click (or Ctrl-click if you have a one-button mouse) the word `Tag` and to choose "Help for Tag" from the context menu. The next fastest way is to search for `Tag` alphabetically in the Command Help tab of the IGOR Help Browser (under the Help menu). The slowest but clearest way would be to look up `Tag` in **Volume V** of the manual set. Use your favorite method to bring up the synopsis for `Tag`.

We are using the `/F` and the `/L` flags, which address the frame and the line type, respectively. The last three values are the name of the trace, the x value of the point at which the tag is to be attached, and the text that will be attached to this point. Normally, you would specify the name of the trace that you want to tag. To make this function a bit more general, we use the `Csrwave` function, which returns the name of the wave on which the A cursor rests.

The expression `xcsr(A)` returns the point of attachment for cursor A, and the string expression inside the quotes uses the `\0X` escape sequence to place the x-value into the tag.

This particular `Tag` command could very well be executed from the command line. Calling it from a function makes it merely convenient. In this instance, we use one of IGOR's built-in functions to get the name of the wave that we want to label. In the next example, we will look at the most basic way in which the wave name and other parameters are supplied by you.

Again, aearch the IGOR documentation for `Tag`, `xcsrwave`, `xcsr`, and `tagval` in the online help, and make sure that you are clear on how the structure of the command yields this particular result. This understanding is crucial.

Simple Normalization

Normalizing traces to the peak maximum is a common task. Let's write a simple function that does this. We will start commenting the code, and we will gradually expand it to make it a more general algorithm. Read the comments because they do provide some explanation. New elements appear in bold.

```
Function CreateNormalizedCopy(input)
   Wave input

   // Get the stats on input wave
   WaveStats/Q input
   Variable maxValue = V_max

   // Generate name of output wave
   String outputName = NameOfWave(input) + "_norm"

   // Create output wave
   Duplicate/O input, $outputName

   // Create a wave reference for output wave
   Wave output = $outputName

   // Do the normalization
   output = output / maxValue
End
```

Now execute

```
CreateNormalizedCopy(fluorescence)
```

A new wave called `fluorescence_norm` is created.

Graph the new wave by executing:

```
Display fluorescence_norm vs FluorWavelength
```

The new graph shows that the trace peak is now 1.0.

Let's discuss this function line by line, in sequence. You will find it useful to bring up the documentation for each function via the help resources whenever it is discussed.

The wave to be processed, `fluorescence`, is passed to the function as a (wave) parameter. The first command, `wavestats`, generates the statistics for `input` (i.e., the local alias for `fluorescence`). (The `/Q` flag, designates "quiet mode", and prevents the printing of results to the history area of the Command window with nearly all IGOR operations.) The string variable `outputName` is used to construct the name of the normalized wave, which is the output of the function. We then duplicate `input` (i.e., `fluorescence`) as a wave with this new name. We create a wave reference named `output` so that we can use the `output` wave in the assignment statement that follows. In the final line, we use the wave `output` (i.e., `fluorescence_norm`) to generate the normalized spectrum.

Assignments of this sort are entirely legal and normal. `output = output + 1` would increase every value of output by exactly 1. The c++ grammar allows for the following shorthand for assignments of this sort: `output += 1`.

A (Slightly) More General Normalization Function

It is possible that you do not want the maximum intensity of the normalized spectrum to be unity. You want it to be any arbitrary unit that you choose. Doing so is trivial. Simply introduce a new variable that will receive this arbitrary maximum from you and modify the normalization command. (Changes are in bold.)

```
Function CreateNormalizedCopy(input,peakvalue)
   //Receives name of the wave
   Wave input
   //Receives the normalized peak value
   Variable peakvalue

   // Get the stats on input wave
   WaveStats/Q input
   Variable maxValue = V_max

   // Generate name of output wave
   String outputName = NameOfWave(input) + "_norm"

   // Create output wave
   Duplicate/O input, $outputName

   // Create a wave reference for output wave
   Wave output = $outputName

   // Do the normalization
   output = output / maxValue * peakvalue
End
```

Now, call `CreateNormalizedCopy` with the following, and note the peak value of `Fluorescence_norm`.

```
CreateNormalizedCopy(fluorescence,4)
```

A General and Useful Normalization Function

Perhaps you have a noisy spectrum, or you have a spectrum with multiple peaks (like `masspec`), and you want to normalize the spectrum to a particular peak which may not be the most intense. Let's generalize the above function just slightly more and introduce the first flow control element, the `if-endif` statement. Again, the changes are in bold.

```
Function CreatNormalizedCopy(input,peakvalue,leftmark,rightmark,rangetype)
```

```
   Wave input
   Variable leftmark,rightmark,peakvalue,rangetype
   Variable maxValue

   //_norm will be added to name of original wave to make name
   //of normalized wave

   String outputName=NameofWave(input)+"_norm"

   //Make sure that the input for rangetype is valid.
   //Otherwise, exit function.
   if (rangeType<0 || rangeType>2)
            return -1
   endif

   //Duplicate the original wave as a wave with the new name
   Duplicate input $outputName
   Wave output = $outputName

   //If rangetype is 0, then normalize to absolute max.
   if (rangetype==0)
            WaveStats/Q output
   endif

   //If rangetype is 1, then normalize to peak over
   //the specified range of POINTS (denoted by brackets)
   if (rangetype==1)
            WaveStats/Q/R=[leftmark,rightmark] output
   endif

   //If rangetype is 2, then normalize to peak
   //over the specified X RANGE (denoted by parentheses).
   if (rangetype==2)
            WaveStats/Q/R=(leftmark,rightmark) output
   endif

   output = output / V_max * peakvalue //Normalize the wave to specified peak
   return 0
End
```

The new parameter `rangetype` will be used by the function to determine the conditions with which `wavestats` will be executed. The first `if-endif` statement limits `rangetype` to a value between 0 and 2. If the input is negative (<0) or larger than 2, `return -1` is executed. The `return` statement ends execution of the function. Although it is not necessary to return a value of -1 when we exit the function, there is no harm done if we return this value. As you might be guessing, when we integrate this function into a collection of functions, we can use this `return` value to determine if the function executed successfully or where it failed. `return 0` statement at the end would tell other functions that `CreatNormalizedCopy` executed successfully.

If the input of `rangetype` is correct, we then employ the next three `if-endif` statements to determine if the absolute maximum will be used to normalize the wave, or if the maximum over a given **x range** (noted with parentheses) will be used or the maximum over a given **point range** (noted with square brackets) will be used. Be aware that the x range over which `Wavestats` will operate is determined by the wave's X scaling.

To make this function less general but similar in utility to the tag function that we wrote above, we can write a function that gets the information that `CreateNormalizedCopy` needs from the cursors and calls `CreateNormalizedCopy`.

```
Function GetNormalizationInfo(newpeakvalue)
    Variable newpeakvalue
    Variable Apoint,Bpoint
    Wave  ToNormalize = $csrwave(A)
    Apoint = pcsr(A)
    Bpoint = pcsr(B)
    CreatNormalizedCopy(ToNormalize,newpeakvalue,Apoint,Bpoint,1)
End
```

Now, display the mass spectrum from the Command window with `Display masspec vs mass2charg`. Display the cursors with `ShowInfo` from the Command window. Now, place the cursors about a peak, and call `GetNormalizationInfo` with `GetNormalizationInfo(4)`. Make sure that the graph is the frontmost window before you make the function call. `Display masspec_norm vs mass2charg`. Look at the maximum value of the peak you identified with the cursors.

`GetNormalizationInfo` algorithm gets the name of the wave from the location of cursors, and normalizes the trace to the peak maximum bordered by the cursors in the frontmost window. It passes 1 as the value for `rangetype` because using the cursors makes this parameter useless. This parameter is useful to other functions that might receive a wave to normalize and an x range over which to carry out the normalization from the user.

The pair of `GetNormalizationInfo` and `CreateNormalizedCopy` are clearly more practical and easier to call from the Command window. We created a practical way of calling `CreateNormalizedCopy` without diminishing its potential to be called from other functions.

Next we will deal with the other two important elements of flow control: the `for-endfor` loop and the `do-while` loop.

Extracting Rows or Columns from a Matrix

The data set from this point forth, `pe311`, comes from a CCD camera. Each row of the matrix contains a full luminescence spectrum (intensity vs wavelength in nm) of europium (the red color in some televisions). Each row was collected at a different time. It is convenient to load such large data sets as matrices in order to reduce clutter.

Display `pe311` in a table now by executing:

```
Edit pe311.ld
```

The first row (indexed 0) was collected at t = 0, the second (indexed 1) at t = 100 microseconds, the third (indexed 2) at t = 200 microseconds, and so on.

When we imported the data into IGOR, we set the column dimension labels of `pe311` to indicate the wavelength corresponding to the data. As you move down and right in the table, you move to later times and longer wavelengths.

Naturally, we need to use individual rows or columns of the matrix for display or analysis. Let's write a function that extracts all of the rows into separate 1D waves. This process is easily achieved with the `for` loop. Pay special attention to the systematic naming scheme introduced here.

Here is a function that extracts the row data into 1D waves.

```
Function RowsToWaves(theMatrix)     //Function declaration
   Wave theMatrix

   Variable rowIndex    //the loop variable

   //Store the number of rows and columns into local variables
   Variable numRows = dimsize(theMatrix,0)
   Variable numCols = dimsize(theMatrix,1)

   //Declare a string variable to store new wave names
   String name

   //Make a wave to hold each row temporarily
   Make/O/N=(numCols) tempRow

   //Use a loop to extract rows one by one
   for (rowIndex = 0; rowIndex < numRows; rowIndex += 1)
            //Make a name that contains the row number.
            name=NameofWave(theMatrix)+"_Row_"+num2str(rowIndex)

            //Extract the indexed row into tempRow
            tempRow=theMatrix[rowIndex][p]

            //Create 1D wave
            duplicate tempRow, $name

   //End of loop body. Commands between "for" and "endfor" are repeated
   endfor

   KillWaves tempRow    //cleanup
End
```

Let's cover the new elements in sequence. First, we use the `Dimsize` function to store the number of rows and columns of `matrix` in the variables `numRows` and `numCols`, respectively. Because we are extracting *rows*, the number of points in `each 1D wave` has to be equal to the number of *columns* in the input matrix. The number 1 tells `dimsize(matrix,1)` to return the number of columns in `matrix`.

Note that when you use an IGOR function or a variable to supply a numeric value in a flag parameter, you must enclose the flag parameter in parentheses; i.e., `make/N=50` vs `make/N=(numCols)`

We use the `for` loop to traverse the rows of the matrix by using the loop variable `rowIndex` as the row index. We will traverse the rows one-by-one by increasing `rowIndex` by one after each pass of the loop body with `rowIndex += 1` in the loop declaration.

The loop body starts by creating a new name that contains the matrix name and the row index. Next, it makes a new 1D wave, via `Make/O`, wave with the name that was formed in the first line of the loop body. This process is repeated for every value of `rowIndex`.

The values that `rowIndex` can take are determined by the loop parameters. The loop declaration tells IGOR, in sequence, to start `rowIndex` at the value 0, to repeat the body of the loop (the sequence of commands that follow the declaration but precede the `endfor` declaration) as long as `rowIndex` is less than the number of rows in `matrix`, and to increase the value of `rowIndex` by 1 after each iteration of the loop body. When the loop condition is no longer true—when `rowIndex` is equal to or larger than the number of rows in `matrix`—the loop ends. (Also, note that the highest row index is one *less than* the number of rows because IGOR starts indexing at 0.)

Don't worry about the huge number of waves in the Data Browser. We will write an easy function to clean them up shortly.

As an exercise, duplicate this entire function in the procedure window (copy and paste), change the name of the copy to `Cols2Waves`, and edit it into a function that extracts columns. You should change the first line of the loop body to read

```
extractedwavename=NameofWave(theMatrix) + "_Col_"+num2str(ColIndex)
```

Also, the end condition for the loop needs to be changed to `ColIndex<dimsize(theMatrix,1)`.

Finding a Level in a Wave

To demonstrate the next important element of flow control, the `do-while` loop, let's write a duo of functions that determine if a particular intensity level is achieved, and at what point in a wave the intensity is reached.

```
Function GetPoint(inwave,threshval)
    Wave inwave
    Variable threshval   //receives threshold value from Function call

    //loop variable, the maximum index value of wave
    Variable point=0
    Variable maxindex = numpnts(inwave)-1 //Indexing starts at 0

    //Test the first data point
    if (inwave[0]>=threshval)
     return 0
    Endif

    do    //loop declaration
     point+=1  //proceed to next data point
     if (point>maxindex)   //if we run out of points
        return -1   //return the value
     Endif
    while(inwave[point]<threshval)  //repeat loop body while this is true
    return point //return the point at which the threshold is exceeded
End
```

Let's test this function on the `mass2charg` wave that exists in this experiment. Let's see at what point the mass to charge ratio exceeds the 9000 barrier. Let's call `GetPoint` from the command line with

```
print GetPoint(mass2charg,9000)
```

The answer is point index number 5283. How about when we pass the 100000 barrier? This answer is -1, which means never. The 1000 barrier? The answer is 0, which means before the first point in the wave. Let's start with the first `if-endif` statement to see how the function works.

The function receives a wave reference and a numeric variable as parameters. We test for the first point in the wave with the first `if-endif` statement. If we are already higher than the threshold value, then we return the value 0. If not, we start the loop.

First, we increase the value of `point`by 1. Then, we test to make sure that we have not exhausted the number of points in our wave. If we have, then we return the special value of -1. If we have not exhausted the wave, then we evaluate the loop condition and repeat the loop if this condition is true. At the start of the loop body, we move to the next point in the wave by increasing `point`by 1, and we repeat the process.

Once the value of `inwave[point]` equals or exceeds the threshold value, we exit the loop, and return the value of `point` as the value of the function. This value of `point` is the index of the point at which the threshold value is equaled or exceeded. Hence, we have written a function that carries out a process and returns a unique value for each possible outcome.

The difference between the `for` and the `do-while` loops should be obvious now. `for-endfor` is used when a defined endpoint is known before the loop is entered. In the above examples, the number of points or rows or columns is known before the loop is started. Therefore, we simply state one of these numbers as the loop condition. In this `do-while` example, we do not know when the loop condition will be false. The `do-while` loop, then, serves as a probe that discovers whether a particular condition is met at run time. It should be clear that by choosing the right loop condition, the `do-while` loop can be made to behave exactly like the `for-endfor` loop.

It should also be clear from this `do-while` example that the `if-endif` statement in the body of the loop is necessary in order to avoid causing an error in the algorithm. For example, if we supply too high a threshold, we will never meet the exit condition, and we will eventually increase `point`to a value larger than the number of points in `mass2charg`.

When writing loops, one must ensure either that the exit condition will be met, or that one provides an exit route because it is possible that the exit condition will never be meet. Failing to provide an exit will result in the famous infinite loop trap. Do not forget that such traps can be manually halted with Command-dot on the Macintosh platform and Ctrl-Break on the Windows platform.

We can complete this example by writing another function to drive `GetPoint` and to put out a nicely formatted result:

```
Function DriveGetPoint(inputwave,thethreshold)
   Wave inputwave
   Variable thethreshold

   Variable outcome=GetPoint(inputwave,thethreshold)
   if (outcome==0)
      Printf "Threshold value %g is reached prior to point 0 in %s.\r",
thethreshold, inputwave
   elseif (outcome==-1)
      Printf "Threshold value %g is never reached in %s.\r", thethreshold,
inputwave
   else
```

```
      Printf "Threshold value %g is reached at point %d in %s.\r", thethresh-
old, outcome, NameOfWave(inputwave)
   Endif
End
```

Now, by calling `DriveGetPoint(mass2charg,6000)`, we get nicely formatted output. The escape sequences in the print statements, `%g` and `%r`, tell IGOR to insert the values of the variables that appear after the the text into those particular spots. The variables must then be listed in the correct order.

Automated Cleanup

The last function we will write in this preliminary section is one that will receive a string as input and then use this string as a criterion to choose waves to kill.

We introduce the very useful IGOR tool of string lists. These are string variables that contain items separated by semicolons all as *one string value*. Special IGOR commands can then be invoked to address the items thus demarcated by semicolons, individually. Let's illustrate this with a simple, useful function.

```
Function CleanWaves(matchString)
   String matchString   //The criterion for elimination

   //Get list of wave names and create a temporary variables
   String wavenames=WaveList("*", ";", "")

   String name
   Variable i  //loop Variable

   //Examine items in list one by one.
   for (i = 0; i < ItemsInList(wavenames); i += 1)
      name = StringFromList(i,wavenames)

      //Compare the name of the wave with the elimination criterion
      if (StringMatch(name,matchString))
         KillWaves $name   //Kill the wave
      Endif

   endfor
End
```

Let's go through the new concepts one line at a time. The first new line is the rather strange string assignment:

```
String wavenames=WaveList("*", ";", "")
```

In this assignment we use the `WaveList` function which returns a list of wave names and concatenates them with the specified separator. IGOR deduces the instructions for constructing the list from the three arguments of `WaveList`, which are `"*"`, `";"` and `""`. `"*"` is a wildcard character, and it tells IGOR to concatenate all wave names. `";"` tells IGOR to use the semicolon as the separator between individual wave names as it concatenates them into one long string. `""` tells IGOR that no special options are to be used.

Hence, if our experiment has three waves named `wave0`, `wave1` and `newaverage`, the `WaveList` statement assigns the string value `"wave0;wave1;newaverage"` to `wavenames`. Built-in IGOR

functions like StringFromList and ItemsInlist then know how to parse this long string value into its individual components. If we want to exclude newaverage from the list, we can write String wavenames=wavelist("wave*",";",""). This WaveList command will construct a list of wave names that start with the letters "wave".

Once the list is constructed, we then set our loop variable to 0, and begin moving along our list with the for-endfor loop and use StringMatch to compare each wave name with the elimination criterion contained in the string variable matchString.

Note that strings can not be compared using the == operator. Here we rely on the Stringmatch function to do the comparison and to return a numeric value that the if statement understands. (The CmpStr function is more often used and tests for equality.) In each pass of the loop, we first extract the the ith in the list of wave names into name, and then we tell StringMatch to evaluate the string content of name against the elimination criterion. If the elimination criterion is met, we use the string value of name to kill the wave immediately. We then return to the start of the loop, increase the value of i by 1 (i+=1) and, thus, move to the next wave name.

It is worth mentioning that the if operator operates on a numeric value. 0 is considered false and any non-zero value is considered true. In this function, the numeric value on which if is operating is the value returned by the StringMatch function.

Now, you can use this function to clean up the mess that RowsToWaves made. First, open the Data Browser (Data-->Data Browser menu sequence) and notice that you have waves named pe311_Row_0, pe311_Row_1, and so on. Now execute this line:

```
CleanWaves("*_ROW_*")
```

Notice that pe311_Row_0, pe311_Row_1 and other such waves have been killed.

Summary

The purpose of the above examples was to introduce you, first, to the basic elements of IGOR programming.

You've now learned how to enter code in the Procedure window, to declare functions, to pass parameters, to create local variables, to create loops and conditionals, to reference waves, to write wave assignment statements, to call (or to execute) functions from the command line, and to call subroutines (i.e., functions that are called by other functions).

We are now ready to write more complicated algorithms, and the above outline should cement the approach that we need to take. First, we must break down the process into individual tasks that we can implement programmatically. Then, we conceive of the logical structure of the steps. Lastly, we choose the appropriate IGOR commands to realize the algorithm.

Intermediate Algorithms

In this section we will write slightly more complex and hopefully more useful programs. We will introduce simple Graphical User Interfaces (GUIs), and we will build programs composed of several functions. We will also start the habit of outlining the algorithm before committing it to bits.

Since this tutorial was composed primarily in Los Angeles, it is necessary to start with a movie.

A Simple Movie

We next turn to the simple `NewMovie` IGOR command to demonstrate how easy it is to make a movie with IGOR. The `NewMovie` command opens a movie file and appends frames to this movie. We will extract the individual frames from the same two-dimensional matrix, `pe311`. We also introduce several more built-in IGOR commands.

For this particular functionality, IGOR relies on Apple's Quicktime. Windows users need to install Quicktime. You can get it from:

http://www.quicktime.com/download/win.html

Unfortunately for Windows users, Apple tries very hard to bundle Quicktime with iTunes. Pay attention on the download site, and choose your options carefully. Should you get stuck with the bundle installer, you can uninstall iTunes afterward (using Add/Remove Programs in the Windows Control Panel). If you are queasy about installing things on your machine, which we can relate to, just read this part of the tutorial without doing it.

Let's break down the tasks that our function needs to perform.

1. Receive the name of the matrix as a parameter.

2. Create a temporary 1D wave that will store successive rows of the matrix.

3. Create a graph that will provide serve as each frame of the movie.

4. Scale the x- and y-axes of the graph so that the frames will appear correctly.

5. Start a loop which will replace the trace in the graph with the next row in the matrix and then append the updated graph to the movie as the next frame.

6. Clean up.

Here is a first crack at this task:

```
Function MakeMovie(matrix)
    Wave matrix

    Variable i //loop variable

    //make a dummy wave to accept individual rows
    Make/O/N=(dimsize(matrix,1)) framewave

    // Create the graph with name FrameGraph
    Display/N=FrameGraph framewave

    WaveStats/Q matrix  //Get statistics of matrix
//Manally set axis  to prevent autoscaling
    SetAxis left V_min,1.1*V_max

    //Name the movie after the original wave
    String movieName = NameofWave(matrix)  + ".mov"

    //create a new movie with the original wave's name
    NewMovie /F=30/L/I/O as movieName

    //start loop to add frames to movie
```

```
    for (i = 0; i < dimsize(matrix,0);i += 1)
     framewave=matrix[i][p]Extract next row

     DoUpdate  //update the graph with the next trace in the sequence
     AddMovieFrame  //add a frame to the movie

    endfor
    CloseMovie //Close the movie file and save it to disk.

    KillWindow FrameGraph //clean up
    KillWaves framewave //clean up
End
```

Let's cover the new concepts in bold.

We create a graph named `FrameGraph` with the `Display` command, and set the Y axis range to span from the minimum value in the matrix to to 110% of the maximum intensity, `1.1*V_Max`. We then initiate the process of making a movie with the `NewMovie` command, using the original name of our matrix as the name of this Quicktime movie file. The `for` loop then starts, in sequence, the process of extracting each row into `framewave`, updating the trace in the graph to reflect the new data in `framewave` (`DoUpdate`), and appending the updated graph as the next frame of the movie (`AddMovieFrame`).

Now call `MakeMovie` from the command line with:

```
MakeMovie("pe311")
```

Save the movie file to your disk, and view it by double-clicking it.

Close the movie window, and return to IGOR Pro. Comment out (make it into a comment by preceding it with `//`) the `Setaxis` command and execute `MakeMovie` again to note what happens to the movie. Using the output of `Wavestats` with `SetAxis`, we frame the movie such that the decay process is visible.

If we desire to have fewer frames, we can advance the frames more quickly by changing `i+=1` to `i+=2`. When `i+=1`, it takes on the values 0, 1, 2, 3, 4... When the increment is 2, `i` takes on the values 0, 2, 4, 6, 8... Hence, we end up with half as many frames. Alternately, we can set the loop end condition to `dimsize(matrix,0)/2` to animate only the first half of this decay process.

Let's complete this example by refining it such that the x- and y-axes are correctly labeled, and the x-axis scaling reflects the experimental property that is being displayed; namely, emission wavelength in nm. Changes are in bold.

```
Function MakeMovie(matrix,xWave)
    Wave matrix,xWave
    variable i //loop variable

    //make a dummy wave to accept individual rows
    Make/O/N=(dimsize(matrix,1)) framewave

//create the first frame of the movie in a Graph windows called "FrameGraph"
    Display/N=FrameGraph framewave vs xWave
    Label/W=FrameGraph left "Intensity (a.u.)"
    Label/W=FrameGraph bottom "Wavelength (nm)"
```

```
    WaveStats/Q matrix  //Get statistics of matrix
    //set axis to a constant to prevent autoscaling
    SetAxis left V_min,1.1*V_max

    //Name the movie after the original wave
    String movieName = NameofWave(matrix)  + ".mov"

    //create a new movie with the original wave's name
    NewMovie /F=30/L/I/O as movieName

    //start loop to add frames to movie
    for (i = 0; i < dimsize(matrix,0);i += 1)
     framewave=matrix[i][p]  //advance to the next trace in the sequence

     DoUpdate  //update the graph with the next trace in the sequence
     AddMovieFrame  //add a frame to the movie

    endfor
    CloseMovie //Close the movie file and save it to disk.

    Killwindow FrameGraph //clean up
    Killwaves framewave //clean up
End
```

The x-axis wave is provided in the experiment as `Wavelength_pe311`. Call the function with:

```
MakeMovie(pe311,Wavelength_pe311)
```

Now, let's explain what we just animated before going on to the next section. As mentioned earlier, each row of this matrix is a full luminescence spectrum of the europium(+3) ion collected at monotonically increasing time delays from the initial trigger. The movie shows how the intensity of this luminescence—the intensity of the red color—decays with time. As good scientists, we want to quantify such decays by fitting them to exponential decay laws in order to gain insight into nature. Let's do that now.

GUI Driven Analysis

Now we will attempt an algorithm to process a lot of matrices like pe311. These matrices will be part of different experiments, and in different IGOR files. Therefore, we should write this algorithm in an independent, standalone procedure file. Open a new procedure file from this menu sequence: *Windows-->New-->Procedure*. Call it LuminescenceLifetime. Then, save it with the *File-->Save Procedure* menu sequence or *File-->Save Procedure As* menu sequence. Remember its name and location. Now let's quantify the phenomenon we saw in our movie.

The movie example shows that we are dealing with a decay process. The movie shows that there are three peaks that are decaying: one at approximately 592 nm, another at 616 nm and the third at 694 nm. We don't want to sum all peaks because the three peaks may have different decay lifetimes. Therefore, we want to follow a particular peak in time. We need a way for the user to indicate the peak to be followed. Then we will take the user's input and extract the decay curve for that peak and fit it to a double-exponential decay function.

This will be a challenging example, but as you will see, the only new element is the GUI. GUIs are not straightforward, so we will stick with the absolute most basic form straight out of *Chapter 6* of **Volume IV**. First, let's outline the tasks that each module needs to perform.

User interface

1. Get the names of the matrix containing data and the x-wave, and the left and right boundaries of a peak from the cursors positions in the frontmost graph.
2. Ask the user for additional information regarding the data.
3. Call data extraction function.
4. Call fitting function.

Data Extraction

1. Receive the choice of matrix and the cross section to be extracted from user interface.
2. Make a 1D wave with a number of points equal to the number of rows in the matrix to receive the integral of the peak marked by the user at each time point.
3. Extract rows from the matrix one at a time, and evaluates the integral of the peak marked by the user.
4. Store the integral of each frame into the corresponding point in the integral wave.
5. Calls curve fitting module.

Curve Fitting

1. Receive the wave to be fit.
2. Create a graph containing the wave.
3. Fit the wave with a double exponential decay function.
4. Annotate the graph with the results.

The above order is, of course, the order in which the algorithm will be executed once all components have been written. However, this need not be the order in which they are written. In fact, it is easier to write functioning data extraction and curve-fitting modules before writing the GUI. This way, one will have a better idea of what information the GUI needs to get from the user. In general, this approach works better when you start writing IGOR programs. Once you develop greater competence and comfort with IGOR programming, you will readily know how your functions will flow, and you can start by programming the GUI without risking having to re-write it later.

Also, it is good practice to write one function and test it before moving to the next function. This approach helps you to solve problems one at a time and to establish robust building blocks that you can use to create higher level functions.

With that in mind, let's start with the function that extracts the integral of the peak of our choice. This one does it.

```
Function/S ExtractIntegrals(theMatrix,xWave,startpoint,endpoint)
   //Get the name of the matrix and the name of the x axis.
   Wave theMatrix,xWave

   //receive the points on either side of peak
   Variable startpoint,endpoint

   //Strings for name formation
   String outputname,sp,ep
   sp = num2str(startpoint)
   ep = num2str(endpoint)
```

```
   //formation of the name of the output wave
   outputname = NameofWave(theMatrix) + "_int_pts_" + sp + "_" + ep

   Variable i  //loop Variable

   //create the output wave and reference it with output

   make/N=(dimsize(theMatrix,0))/D/O $outputname
   Wave output = $outputname

   //Scale the extracted wave as the time scaling of the matrix
   Copyscales/P theMatrix, output

   Make/D/O/N=(dimsize(theMatrix,1)) ExtractedRow

   for (i = 0;i < dimsize(theMatrix,0);i += 1)

      ExtractedRow = theMatrix[i][p]   //Extract a row

      //Integrate the row vs. wavelength wave and save
      //the integral as a wave called intwave

      integrate/T ExtractedRow/X=xWave/D=intwave

      //Assign area of peak to the corresponding time point in tempslice

      output[i]=intwave[endpoint]-intwave[startpoint]

   endfor

   killwaves intwave,ExtractedRow      //clean up

   return outputname
End
```

Let's see what the function does. From the command line, display the first row vs the wavelength with

```
Display pe311[0][] vs Wavelength_pe311
```

Now, call `ExtractIntegrals` with

```
Extractintegrals(pe311,wavelength_pe311,280,372)
```

Point numbers 280 and 372 define the big peak. The output of `ExtractIntegrals` shows the integral over the specified points as a function of time.

Next, display and examine the output wave by executing:

```
Display pe311_int_pts_280_372
```

It is an exponential decay as we expected.

We have written a string function that returns the name of the wave it creates. This is helpful because other functions can use this return value as a handle to reference and to manipulate this output wave.

`ExtractIntegrals` resembles the `RowsToWaves` function which we wrote earlier in order to extract the rows and store them as individual waves. The additional task is that we integrate each row over the area specified by the user, and we store the resulting integral in the corresponding point in the output wave. The only reason we go to this trouble is that integration is a good means of reducing noise without tampering with the data. To see this noise reduction, display the decay of the most intense peak at 616 nm (point 312) with

```
Display pe311[][312]
```

The integral wave is not as noisy. This should improve the statistics of the curve fitting operation, to which we turn next.

```
//Gets the wave, fits it to a double exponential, and graphs the results
Function DoFitting(wave2fit)
   Wave wave2fit

   //Build a name for the graph window
   String GraphName=Nameofwave(wave2fit)+"_Fit"

   String tunits

   //Will receive the two exponential decay constants
   Variable life1,life2

   //Get the time units from the wave for labeling (later)
   tunits = " " + waveunits(wave2fit,0)

   //Display wave2fit in a graph with a known name
   Display/N=$graphname wave2fit

   //Fit wave2fit to a double exponential decay, attach residuals
   Curvefit/N/W=0 dblexp wave2fit /D/R

End
```

In `DoFitting`, little new is taking place. The function gets the wave to fit, displays it in a window which we name with the local string variable `GraphName`, and carries out the double exponential curve fit. We call `Curvefit` with the `/W=0` flag because we do not wish to halt execution for user input.

The `CurveFit` operation creates a wave named `W_coef` which contains the coefficients found by the curve fitting process. The equation for the double exponential is:

y = y0 + A1 * exp((x-x0) / t1) + A2 * exp((x-x0) / t2)

The two values we want are t1 and t2 which are stored in `W_coef[2]` and `W_coef[4]` respectively. The lifetimes that we are after are the inverses of t1 and t2.

It would be nice to label this graph with this information, so lets write a simple function to annotate the graph with the fitting results.

```
Function LabelGraph(Gname,timeunits)
   String Gname,timeunits
   Variable l1,l2
```

```
   //Use this string to compose the graph annotation
   String Summary

   Wave FitResults=W_coef  //Reference Curvefit's output wave

   //Note that the lifetime is the inverse of the time constant
   l1 = 1/FitResults[2]
   l2 = 1/FitResults[4]
   summary="Lifetime 1 is "+num2str(l1)+timeunits+"\rLifetime 2 is "+num2s-
tr(l2)+timeunits
   TextBox/A=MC/C/W=$Gname/N=text0 summary
End
```

`LabelGraph` receives the name of the graph it needs to annotate along with the other string that it needs for the annotation, `timeunits`. `Fitresults` references `W_coef` wave (automatically generated by `Curvefit`), which contains the results of the fitting. The lifetime results are extracted from `FitResults` and assigned to `l1` and `l2`. (Note that `Curvefit` returns the inverse lifetime values.) These two numbers are then converted to strings. All these string variables are then combined with some standard text into the variable `summary`. The `Textbox` command then attaches our formatted summary to the correct window (addressed with `/W=$Gphname`), and the function ends. (In the `Textbox` documentation, introducing a carriage return with the `/r` escape sequence is explained.)

Add the following line to `DoFitting` to automatically do the annotation. (This must be the line immdiately before `End`.)

```
LabelGraph(Graphname,tunits)
```

Go ahead and test this function with the last output of `ExtractIntegrals`, namely `DoFitting(pe311_int_pts_280_372)` (or the output wave that corresponds to the points that you chose).

We now have two of the three components ready. We have two functions that perform the tasks we desire. They stand on their own, but maybe we must process 50 of such matrices (a common number for this particular experiment). Perhaps, then, it would be nice to have a simple GUI that drives these two processes and, thus, relieves us of the task of typing function calls with such long (but informative) wave names. This is a relatively easy task, and it is the subject of the last concept introduced in this manual: building a graphical user interface in IGOR.

User interfaces are covered in *Chapter 6* of **Volume III.** This topic can be difficult to understand conceptually. Therefore, we will keep this interface as simple as possible. We will use only the topics covered in the first section, "The Simple Input Dialog".

As usual, let's build this function slowly, in stages. Let's make the GUI as independent and as general as possible so that it can lend itself to calling other functions that we write later. So, let's start with the basic function that drives `ExtractIntegrals` and `DoFitting`.

```
Function DriveAnalysis(Matrix,Wavelength,LeftPoint,RightPoint)
   Wave Matrix,Wavelength
   Variable LeftPoint,RightPoint

   //Reference the output of ExtractIntegrals
```

```
   String WaveToFit =
ExtractIntegrals(Matrix,Wavelength,Leftpoint,Rightpoint)
   Wave WaveToPass = $WaveToFit

   DoFitting(WaveToPass)

End
```

Let's test it with `DriveAnalysis(pe311,wavelength_pe311,270,350)`. It works. Now we know the four parameters that our GUI needs to obtain and pass on to `DriveAnalysis`. Since the name of the wavelength wave can be obtained systematically, all we have to do is get the name of the matrix. Let's write this basic element of the GUI. Following the recipes in "The Simple Input Dialog", we write

```
Function ProcessMatrix()   //User Interface
   //Variables to receive the name of the matrix
   String chosenmatrix

   //Begin formatting the three prompts to be presented to user
   //First prompt presents the list of 2D waves as a popup menu. The chosen
   //value will be assigned to chosenmatrix

   Prompt chosenmatrix,"Name of Matrix: ", popup, wavelist("*",";","DIMS:2")

   Doprompt "Choose the matrix:",chosenmatrix

   //This if statement checks if you pressed cancel, and exits with
   //with a warning message if you have
   if (V_Flag)
      Abort "You chose to cancel."
   Endif

   Print chosenmatrix

End
```

Go ahead and call the function from the command line.

The Prompt command tells IGOR that the value of `chosenmatrix` will be obtained from the user via a simple prompt. The `popup` addition tells IGOR to present the user with a popup menu consisting of the items in the list that follows `popup`, a list of waves in this case. We get a list consisting of exactly one wave because the third parameter in the `wavelist` operation tells IGOR to limit the list to waves that have two dimensions. Once the content of the prompt has been defined, the `DoPrompt` presents the prompt to the user according to this definition. The user can be prompted for additional values simply by listing the variables after `DoPrompt`. We will do this in a moment.

The `if` statement aborts execution if cancel is pressed in the prompt. The Print command then shows that the correct wave name is obtained. So, now lets get the left and right points of the peak of interest from the user.

```
Function ProcessMatrix()   //User Interface
   //Variables to receive the name of the matrix
   String chosenmatrix

   //The two points defining peak to extract
```

```
   Variable Lpoint=0,Rpoint=0,temp

   Prompt chosenmatrix,"Name of Matrix: ", popup, wavelist("*",";","DIMS:2")
   Prompt Lpoint, "Point to the left of the peak: "
   Prompt Rpoint, "Point to the right of the peak: "

   Doprompt "Choose the matrix:",chosenmatrix,Lpoint,Rpoint

   //This if statement checks if you pressed cancel, and exits with
   //with a warning message if you have
   if (V_Flag)
      Abort "You chose to cancel."
   Endif

   //Validate the choice of points
   if (Lpoint == Rpoint)   //Abort if they are the same
      Abort "The points are the same."
      elseif (Lpoint > Rpoint)   //Order them if they are out of order
            temp = Lpoint
            Lpoint = Rpoint
            Rpoint = temp
      endif


   Print chosenmatrix,Lpoint,Rpoint

End
```

Go ahead and execute this function again. Choose equal values of `Lpoint` and `Rpoint`. Make `Lpoint`'s value greater than `Rpoint`'s. See what happens in the history area of the Command window. That last `if-elsif` statement checks that the values of the points are valid. The variable `temp` is needed to reorder the points. We may have neglected to reject point numbers that are larger than the number of points. We will take care of this soon by getting these values from cursors A and B. For now, let's complete the GUI by prompting for two additional important parameters: the time increment between points (which makes decay lifetimes meaningful) and the units for the time increment. These additions to the function satisfy this requirement.

```
Function ProcessMatrix()   //User Interface
   //Variables to receive the name of the matrix
   String chosenmatrix,timeunits=""

   //The two points defining peak to extract
   Variable Lpoint=0,Rpoint=0,temp,timeincrement=0

   Prompt chosenmatrix,"Name of Matrix: ", popup, wavelist("*",";","DIMS:2")
   Prompt Lpoint, "Point to the left of the peak: "
   Prompt Rpoint, "Point to the right of the peak: "

   Prompt timeincrement, "Time Increment: "

   Prompt timeunits, "Time units: ",popup "nanosec;microsec;millisec;sec"

   doprompt "Choose the matrix and enter the time
increment",chosenmatrix,Lpoint,Rpoint,timeincrement,timeunits

   //This if statement checks if you pressed cancel, and exits with
```

```
   //with a warning message if you have
   if (V_Flag)
      Abort "You chose to cancel."
   Endif

   //Make sure that choice of points is valid
   if (Lpoint == Rpoint)
      Abort "The points are the same."
      elseif (Lpoint > Rpoint)
            temp = Lpoint
            Lpoint = Rpoint
            Rpoint = temp
      endif

   Wave theMatrix = $chosenmatrix

   //Let's assume (correctly) that the xwave has been systematically
   //named for us. Build a string value corresponding to the name
   //and reference the wave
   String xWavename="Wavelength_"+NameofWave(thematrix)
   Wave xWave = $xWavename

   //If scaling information was entered, adjust matrix x scaling
   if (timeincrement != 0)
      setscale/p x,0,timeincrement,timeunits,thematrix
   endif

   DriveAnalysis(theMatrix,xWave,Lpoint,Rpoint)

End
```

Go ahead and test `ProcessMatrix`. It should work. The `Print` statement is gone because we know that we are obtaining the correct values. We have added a few statements to use the entered wave name to reference the actual wave, to build a systematic name for the wavelength wave and to reference it and to set the scaling for the matrix if the user has changed the default value of 0. With everything ready, we call `DriveAnalysis` with the information that it needs.

To make things easier for the user, let's write a very simple function to drive `ProcessMatrix`. This one will get the values from cursors that the user has placed on the frontmost graph.

```
Function GetMatrixInfo()

   //Get the matrix name from the cursor wave
   String Matrixname = csrwave(A)

   //Variables for points and temporary variable for reordering
   Variable LeftPoint,RightPoint,temp

   //Get the point values
   LeftPoint = pcsr(A)
   RightPoint = pcsr(B)

   //Abort if cursors are not on the graph; reorder if necessary
   If (Leftpoint == Rightpoint || Numtype(LeftPoint) == 2 || Numtype(Right-
point) == 2)
            Abort "Choice of points is not adequate."
```

```
            elseif (Leftpoint > RightPoint)
                  temp = Leftpoint
                  LeftPoint = RightPoint
                  Rightpoint = temp
   endif

   ProcessMatrix(Matrixname,LeftPoint,RightPoint)
End


Function ProcessMatrix(chosenmatrix,Lpoint,Rpoint)    //User Interface
   //Variables to receive the name of the matrix
   String chosenmatrix

   //The two points defining peak to extract
   Variable Lpoint,Rpoint

   String timeunits=""
   Variable temp,timeincrement=0


   Prompt chosenmatrix,"Name of Matrix: ", popup, wavelist("*",";","DIMS:2")
   Prompt Lpoint, "Point to the left of the peak: "
   Prompt Rpoint, "Point to the right of the peak: "
   ...
```

Note the changes that need to be made to `ProcessMatrix`. It is now receiving some values as parameters. Therefore, the variables that are not acting as parameters need to be separated and declared later. Furthermore, `Lpoint` and `Rpoint` are now initialized to the values they receive from `GetMatrixInfo`.

Let's give it a test by first displaying the first slice of the matrix with (from the command line)

```
Display pe311[0][]
```

Then, let's get the cursors on there with

```
showinfo
```

Don't do anything with the cursors just yet. Execute `GetMatrixInfo()`.

It fails because the cursors are empty. If the cursors are not on the graph, the `pcsr` function returns `NaN` (not a number). The `if` statement uses the logical “or” operator `||` to abort if either one of the cursors is empty. The `numtype` operation returns a value of 2 if the type is NaN. (To see a list of the logical operators, go to the Command Help section of the IGOR Help Browser and make sure that the checkbox next to "Programming" is checked. Symbolic operators are listed first.)

Place the cursors on the graph, and execute `GetMatrixInfo()`. Now you see that the values from the cursors appear as the initial values for the matrix name, `Rpoint` and `Lpoint`. We could have prevented this by commenting out the lines prompting for these values, but it does make sense to keep the prompts because the user may want to change them slightly.

Furthermore, prompting for the matrix again is useful because it allows the user to choose points on a 1-d wave, and extract them from the 2-d wave of his/her choice. Thus, the algorithm offers some flexibility.

Mark the large peak near 616 nm, run `GetMatrixInfo()`, and enter 300 microseconds for the time increments. Do so again after marking a different peak. The lifetime results are different, in complete agreement with what is known about the fundamental nature of europium luminescence.

You can certainly think of many points of refinement. Perhaps, you would prefer to construct the extracted integrals from the individual waves that were extracted from the matrix in the `Rows2Waves` exercise because that is how you like to load your data. You now know enough about systematic name formation to change only a few lines to implement this change. Welcome to IGOR Pro.

Concluding Remarks

The typical IGOR user is a scientist or an engineer with specific data processing needs. Although IGOR ships with a huge number of built-in capabilities, it cannot fulfill all of one's needs. In fact, a single application that fits everyone's needs simply does not exist. IGOR's programmability fills this gap completely.

The goal of this tutorial was to acquaint you with IGOR's programing environment, to make you comfortable with writing and compiling IGOR functions, to make you accustomed to searching IGOR's help resources to find the right IGOR operation or function for the task, and to give you some (hopefully enough) exercise in constructing IGOR commands and functions. The basic elements of programming (the value of a function, loops, etc.) introduced here are by no means a complete survey of the methods that can be implemented in IGOR. In fact, a complete survey may be impossible.

Fortunately, in this quest to write good and valid IGOR Pro code, you will never be on you own. In addition to IGOR Pro's superb documentation, you can also draw upon the expertise of the global community of IGOR Pro users and IGOR Pro's developers by joining the IGOR Pro support email list. You can find details at

http://www.wavemetrics.com/support/mlist.htm

Naturally, an experienced programmer will make the most of IGOR's programmability, but *anyone* can make use of this feature. The three steps are 1) to conceive of a series of simple steps to your outcome, 2) to find the IGOR operations and commands that can perform the individual steps, and 3) to type them into the Procedure window.

Addenda

A Note on Debugging

Computers are inherently stupid machines. The only thing they can do is to follow the instructions that programmers give them *exactly*. Therefore, errors encountered by the program were necessarily introduced by the programmer, including the bugs introduced by Wavemetrics developers. The process of finding one's errors is debugging. When the error consists of a forgotten parenthesis, a missing comma, or the use of a parameter where a literal string is needed (i.e., using `theString` where `$theString` is required), IGOR will usually guide you to the position where the error occurs. These errors are usually noted by the compiler (i.e., at *compile time*).

Sometimes, the syntax is perfectly fine, but an error occurs when the function is being executed. These *run time* errors frequently happen because you made a mistake in constructing the name of

a wave. As a result, IGOR tries to kill a wave that does not exist, and it reports this error. It can not always tell you where the error appears in the source. To find these errors, you can either use the debugger or place strategic `Print` statements in your code.

To use the debugger, activate the Procedure window, and choose "Enable Debugger" from the "Procedure" menu. You can then use the debugger to monitor the value of every function and parameter as each line of code is executed.

During debugging you will want to enable the Debug on Error and NVAR, SVAR, Wave Checking features via the Procedure menu. The Debug on Error feature breaks into the debugger immediately when an error occurs and shows you the line, or at least the neighborhood of the line where the error occurred. The NVAR, SVAR, Wave Checking feature breaks into the debugger if your code tries to access a global numeric variable, global string variable, or wave that does not exist.

The debugger is the subject of *Chapter 8* of **Volume IV**.

A decidedly more old fashioned strategy is to add `print` statements to the code every time a variable's value is changed. The `Print` statement can be descriptive. For example, it can read `Print "The value of matrixname at function call is",matrixname`. If the value makes no sense, the you need to examine the lines in which the value of `matrixname` is formed and assigned. Sometimes `Print` statements find the problem more quickly than using the debugger, but you will find the debugger more effective once you have learned to use it.

How `Wavelength_pe311` was Obtained

The matrix pe311 was exported as an ASCII file and imported into IGOR Pro as a matrix. The wavelength information was imported as the column *label* for this matrix. To see this information, run `edit pe311.ld` in the Command window. The wavelength information is not linear. It can't be expressed as wave scaling. Therefore, it must be extracted as a wave against which other waves are plotted and integrated (e.g., `ExtractIntegrals`). This function was used to accomplish this task.

```
Function ColLabel2Wave(TheMatrix) //make a new wave from matrix column labels
   Wave TheMatrix
   //String variable for name formation
   String wl = "Wavelength_" + NameofWave(TheMatrix)

   //Loop variable used as column index
   Variable i

   //Make a temporary wave to store data
   Make/D/N=(dimsize(TheMatrix,1)) tempwl

   //Traverse columns one by one
   for(i=0;i<=dimsize(TheMatrix,1)-1;i+=1)

   //Convert label to number and assign it to the correct position in tempwl
   tempwl[i]=str2num(getdimlabel(TheMatrix,1,i))

   endfor

   Duplicate/O tempwl, $wl
   Killwaves tempwl
End
```

Every element of every wave is addressable programmatically in IGOR Pro. The same can be said of graph windows, graph properties, and so on. The key is finding the right IGOR operation or function in IGOR Pro's help resources!