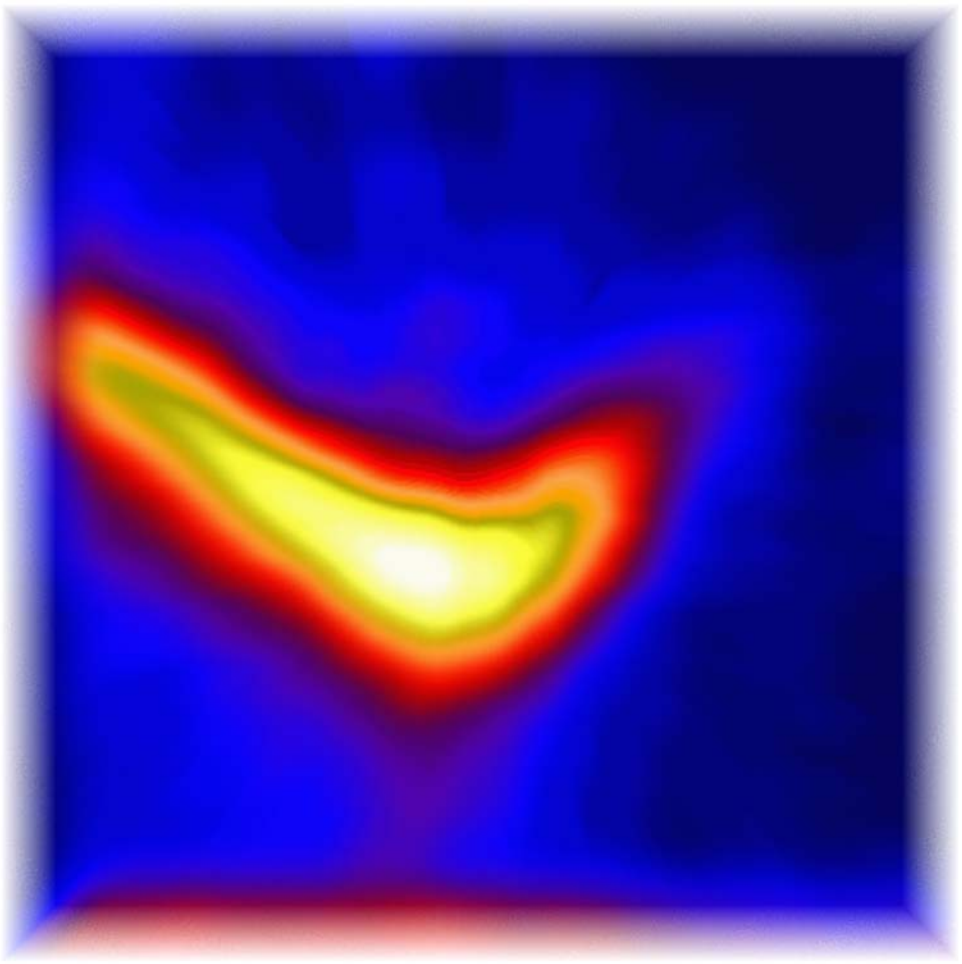

Application Development in IDL I

R.M. Dimeo
NIST Center for Neutron Research

NIST Center for Neutron Research
DAVE
Data Analysis and Visualization Environment

Contents

Acknowledgments

A number of sources made contributions to this manual. In particular John Copley provided source code and some text in the discussion of data rebinning. Richard Azuah provided most of the text in Chapter 10 on how to add modules to DAVE and the discussion of the DAVE internal data format. Richard Azuah, Larry Kneller, and John Copley all provided valuable feedback on various topics discussed throughout this manual.

This work is based upon activities supported by the National Science Foundation under Agreement no. DMR-0086210.

Chapter 1 Introduction

This course manual grew out of a set of notes and programs that I wrote for teaching a course on introductory application development in IDL for neutron scattering scientists at the NIST Center for Neutron Research (NCNR). Because this manual is based on a set of notes it should not be considered a formal text for learning IDL. There are a number of excellent texts on this subject (such as *IDL Programming Techniques* by David Fanning, *Practical IDL Programming* by Liam Gumley and others listed below). Rather it is a reference and brief summary of material discussed in the NCNR's introductory course.

The course is targeted to scientists interested in using IDL for the purpose of manipulating, reducing, visualizing, and analyzing neutron scattering data. Different readers will have different end goals. Some will want to write their own data manipulation programs for their own research. Research scientists who need to get scientific results out quickly fall into this first category. Others will want to write applications complete with a graphical-user-interface for use by non-experts. Instrument scientists who wish to spend as little time as is necessary helping novice users through the data reduction and analysis phase of their experiments fall into this second category. Still other developers will want to find some intermediate ground in which the end-user of an application can script their own high-level commands while offering the simplicity of a graphical user interface. This programming language offers all of these capabilities. In this course we will focus on the first two points discussed above, though the last point is addressed to some extent in the second course, *Application Development in IDL II*.

This course will serve as an introduction to programming in IDL from writing scripts of simple commands in an interactive batch mode to creating end-user applications with a graphical user-interface (GUI). Participants in this course will learn the fundamentals of IDL programming, how to use IDL to analyze data, how to write an application with a GUI, how to distribute an application using the free IDL Virtual Machine, how to write a relatively sophisticated data visualization application with interactive features, and how to add a standalone GUI application into the DAVE (Data Analysis and Visualization Environment) software system. Neither object graphics nor object-oriented programming are covered in this course. Object-oriented programming in IDL is the topic of the next course.

Although IDL is an appropriate platform for developing applications as stated above, there are a substantial number of routines written by software developers that you might find extremely useful in your own applications. Some such routines will be discussed here. Some of these third party programs will be needed for this course. In addition to these third-party programs you can download the programs for this course from a link at the following web address:

`http://www.ncnr.nist.gov/staff/dimeo/IDLAppI.html`

Most of what I learned (and continue to learn) about IDL comes from the great resources available, including a huge IDL programming community. The list below contains a few that you might find useful.

Many excellent texts exist on IDL programming:

- *IDL Programming Techniques*, David Fanning
- *Application Development in IDL*, Ronn Kling
- *Practical IDL Programming*, Liam Gumley
- *Power Graphics with IDL*, Ronn Kling

Furthermore there are numerous on-line resources available:

- **`www.dfanning.com`**
  David Fanning's site has lots of very nice widget (GUI) programs that are written in a very clear and pedagogical manner. Great for learning GUI programming!

- **`fermi.jhuapl.edu/s1r/idl/s1rlib/local_idl.html`**
  The Johns Hopkins Applied Physics Lab has lots of plotting and printing tools as well as some general utilities. The categories of IDL code found here are too numerous to mention.

- **`cow.physics.wisc.edu/~craigm/idl/idl.html`**
  Craig Markwardt's site contains many math routines, including arguably the best curve fitting routines for IDL. His MPFIT routine and its supporting procedures are the basis for PAN (the curve fitting utility in DAVE).

- **`www.kilvarock.com`**
  Ronn Kling has written some nice widget and object graphics books listed above. His web site contains freeware and code you can buy written in IDL.

- **`comp.lang.idl-pvwave`**
  The IDL newsgroup is a very friendly bulleting board with many useful tips and a searchable archive. If you have a question, it will likely get a response same day. I check for the latest postings frequently.

- **`www.metvis.com.au/idl/`**
  This has a terrific and fairly complete list of IDL resources available on the web.

Finally I also have a number of IDL programs available for download from my web site: **`www.ncnr.nist.gov/staff/dimeo/idl_programs.html`**.

Chapter 2 Data types and variables

In this first chapter we discuss (very briefly) some of the commonly used data types in IDL.

Scalars

There are many data types available for use in IDL and we will discuss here how to use some of the most common. We will perform most of our work in this chapter at the IDL command line so the output should look familiar. First let's see how IDL views the following variables (note that the ampersand symbol & allows multiple statements on one line):

```
IDL> t = 6LL & u = 0B & v = 1L & w = 1 & x = 0.0 & y = 1D & z = 'A'
IDL> help,t
T               LONG64    =                      6
IDL> help,u
U               BYTE      =    0
IDL> help,v
V               LONG      =            1
IDL> help,w
W               INT       =        1
IDL> help,x
X               FLOAT     =      0.000000
IDL> help,y
Y               DOUBLE    =        1.0000000
IDL> help,z
Z               STRING    = 'A'
```

As a word of explanation, the procedure named HELP is called with a comma prior to the variable name about which the user is inquiring. Procedures can be called with the following syntax:

```
IDL>  foo, arg1,arg2,arg3
```

Where `foo` is the name of the procedure and `arg1`, `arg2`, and `arg3` can be input or output parameters. Of course there is no limit to the number of parameters that can be passed in to or out of a procedure. Procedures and functions are discussed in more detail in Chapter 3.

If a variable has not been defined then help tells us it is undefined. For instance consider the following example.

```
IDL> help,xx
XX              UNDEFINED = <Undefined>
```

Since we had assigned any value to the variable name xx it is undefined. If we try to print this variable then we will get an error.

```
IDL> print,xx
% PRINT: Variable is undefined: XX.
% Execution halted at: $MAIN$
```

In addition to the common data types listed above, structures, pointer variables, and objects are also supported in IDL. We will not be discussing object variables in this course.

Structure variables, which are a convenient way to a group members of different data types, come in two types: named and anonymous. The main difference between the two types is that you cannot modify the contents ("tags") of a named structure after it has been created (except by resetting your IDL session). As an example of an anonymous structure, consider the following:

```
IDL> steven = {age:25,weight:165.0,height:152.0}
```

In this structure, whose variable name is steven, there are three *tags*. The tag names, *age*, *weight*, and *height*, are of type integer, float, and float. The names can be obtained via the *tag_names* function and the number of tags can be obtained via the *n_tags* function.

```
IDL> print,tag_names(steven),n_tags(steven)
AGE WEIGHT HEIGHT
            3
```

> TAG_NAMES and N_TAGS are functions and the syntax, discussed more fully in chapter 3, is shown below:
>
> ```
> IDL>  ret = foo(arg1,arg2,arg3)
> ```
>
> Here foo is the name of the function and arg1, arg2, and arg3 can be input or output parameters. As in procedures there is no limit to the number of parameters that can be passed in to or out of a function.

The natural way to retrieve the values of each of the structure tags is using the following convention in which the tag name is appended with a period after the structure variable name:

```
IDL> print,steven.age,steven.weight,steven.height
      25     165.000      152.000
```

Alternatively you can use the lesser-used method involving the tag subscripts known as *field subscripting*. Note that this only works with parentheses (not square brackets)!

```
IDL> print,steven.(0),steven.(1),steven.(2)
```

```
      25     165.000      152.000
```

You can set the structure keyword of `tag_names` function to return the name of the structure. Since this structure is anonymous, no specific name is returned when `help` is invoked as shown below:

```
IDL> help,steven
STEVEN          STRUCT    = -> <Anonymous> Array[1]
```

Note that this tells us that it is an anonymous structure array with a single member. We will not go into the details here but an array with one member is not the same thing as a scalar. Suffice it to say at this point, creation of a structure in the manner shown above actually creates a structure array.

An example of a named structure is given by the following:

```
IDL> str = {neutron_scatterer,name:'William Gates',salary:865114,height:1.78}
```

In this named structure, the variable name `str` is assigned to a structure with three fields: name, salary, and height. Since this is a named structure, `help` yields the following:

```
IDL> help,str
STR             STRUCT    = -> NEUTRON_SCATTERER Array[1]
```

Note that we have all of the same structure syntax as with the anonymous structure in the previous example. Therefore we can extract information from the structure as follows:

```
IDL> print,str.salary
      865114
```

which is equivalent to

```
IDL> print,str.(1)
      865114
```

With named structures we can run into problems by attempting to redefine the size and variable type of the individual fields. For instance, let's say that we want to change our definition after we have already defined it previously. For instance in the following statement we modify the salary to be a floating point scalar:

```
IDL> str = $
      {neutron_scatterer,name:'William Gates',salary:865124.5,height:1.78}
```

Now if we examine the variable type of `str.salary` using `help` we find

```
IDL> help,str.salary
<Expression>    LONG      =       865124
```

which indicates that the type cannot be changed.

We can attempt the same with the anonymous structure by changing the height field, previously a floating point number, to an integer,

```
IDL> steven = {age:25,weight:165.0,height:152}
```

which yields the following variable type for str.height using `help`

```
IDL> help,steven.height
<Expression>    INT       =      152
```

Therefore it is possible to change the variable type of a field in an anonymous structure (if we are not changing the size) but even *this* is forbidden in a named structure.

It is forbidden <u>in either case</u> to change the size of the variable in any of the fields. For instance, if we attempt to resize the `height` field of the anonymous structure `steven` as follows then we get an error message telling us that we are not allowed to resize the field.

```
IDL> steven.height = findgen(5)
% Expression must be a scalar in this context: <INT       Array[5]>.
% Execution halted at: $MAIN$
```

Note that `findgen` is a function that returns an array. Arrays are discussed in the next section.

Another difference between anonymous and named structures is that you cannot redefine the number nor identity of fields in a named structure. This is permitted in anonymous structures as the following example shows. Here we will redefine the structure by removing the age field.

```
IDL> steven = {weight:165.0,height:152}
IDL> help,steven
STEVEN          STRUCT    = -> <Anonymous> Array[1]
```

This is acceptable because now the structure is composed of two fields. However we get an error when we try to do this with a named structure. We try to redefine the structure without the salary or height fields.

```
IDL> str = {neutron_scatterer,name:'William Gates'}
% Wrong number of tags defined for structure: NEUTRON_SCATTERER.
% Execution halted at: $MAIN$
```

One programming technique that can be quite useful when it is necessary to create a number of similar structures is automatic structure definition. We will make use of this in the chapter on GUI programming, especially as it pertains to plotting data in windows so we will discuss automatic structure definition briefly here.

In an automatic structure definition we can create a named structure with each of the fields filled with known variable types. As an example let's define a structure for a *used car* using the automatic structure definition. We want four fields in the structure: year (integer), model (string), make (string), and mileage (double). The procedure below provides the definition.

```
      pro used_car__define
      void =   {     used_car,      $
                     year:0,        $
                     model:'',      $
                     make:'',       $
                     mileage:0D     }
      end
```

Type this code into a new editor window and save it as `used_car__define.pro`.

There are a number of points to note about this procedure. First the name of the procedure includes the name *used_car* but there is a double-underscore after it and before *define*. This syntax is required for any automatic structure definition. The other thing to notice is that we have used place-holders on the right hand side of each colon which tells IDL the variable type that is to be associated with each of the fields. Finally we use the name "void" in the definition because the variable name within the definition statement is not used, thus it is not important.

We can create a structure now using this automatic structure definition. At the IDL command line type the following sequence of commands:

```
IDL> car1 = {used_car}
IDL> car1.year = 1996
IDL> car1.make = 'Honda'
IDL> car1.model = 'Accord'
IDL> car1.mileage = 135122.5
```

The first statement associates an instance of the named structure with the variable name `car1`. The remaining statements identify particular values with each of the fields. If we had not filled in these values as shown above then the values would be identical to those place-holders given to the fields in the definition procedure, `used_car__define`.

We can use some of the built-in IDL functions to extract the contents of the structure we just created as follows.

```
IDL> for i = 0,n_tags(car1)-1 do print,car1.(i)
1996
Accord
Honda
```

```
135122.50
```

We can easily create a new instance of a used_car structure with new values in the fields and print out the contents as follows.

```
IDL> car2 = {used_car,2002,'Toyota','Camry',45000.}
IDL> for i = 0,n_tags(car2)-1 do print,car2.(i)
2002
Toyota
Camry
45000.000
```

Note that when we define a structure this way, we simply need to fill in the values without writing in their names. Note that entering the values of the fields in the proper order is very important.

We will find in later chapters that GUI programming requires working knowledge of structure variables.

Arrays

Neutron scattering data can be represented conveniently as an array of numbers. If the data is collected in an histogram on the data acquisition system then one can store this in memory (in an IDL program) as an array of long integers. To start let's consider methods to create arrays.

A floating point array of length 10 can be created using the FLTARR function.

```
IDL> x = fltarr(10)
IDL> help,x
X               FLOAT     = Array[10]
```

This creates an array with 10 zeros in it. The individual elements in the array are extracted using subscripting with zero indexing and square brackets. For instance, the 5th element in the array is printed using the following command:

```
IDL> print,x[4]
     0.000000
```

Note that we use the index 4 to subscript the 5th element. If we wanted to subscript the 1st element then we would do so as follows:

```
IDL> print,x[0]
     0.000000
```

A simple 5-element integer array can be constructed using comma-separated values within brackets.

```
IDL> arr = [5,4,7,6,9]
IDL> help,arr
ARR             INT       = Array[5]
```

There are many array functions in IDL but we will not demonstrate all of them. However we can discuss a few as the need arises. For instance, we can determine the number of elements in an array using a built-in function named `N_ELEMENTS` as follows.

```
IDL> print,n_elements(arr)
           5
```

The asterisk (*) symbol can be used to reference all elements in the array such as

```
IDL> print,arr[*]
       5       4       7       6       9
```

which gives identical output as

```
IDL> print,arr
       5       4       7       6       9
```

Furthermore a portion of the array elements can be extracted using colon subscripting. The following example illustrates how you can extract the 2nd through the 4th elements.

```
IDL> print,arr[1:3]
       4       7       6
```

We can add elements to the array if we wish. For instance if we wanted to prepend the numbers [1,2,3] to the array then we can simply execute the following command.

```
IDL> arr = [1,2,3,arr]
IDL> print,arr
       1       2       3       5       4       7       6       9
```

Appending values can be done just as easily.

In IDL you can create an array with up to 8 dimensions. The arrays are stored in column-major form (i.e. columns stored first) and the first array dimension varies fastest. As a simple example we can create a (2-dimensional) 3×2 array of integers using nested brackets.

```
IDL> a = [[1,2,2],[5,4,3]]
IDL> help,a
A               INT       = Array[3, 2]
IDL> print,a
       1       2       2
       5       4       3
```

Ranges of elements can be extracted from an n-dimensional array (n>1) using a combination of colons, commas, and/or asterisks. As in illustration we show how to extract all of the elements in the second column.

```
IDL> print,a[1,*]
       2
       4
```

Similarly we can extract any individual elements using comma-separated subscripts in square brackets. For instance, the value in the 3rd column and second row is obtained via

```
IDL> print,a[2,1]
       3
```

As in the case of pre-pending and appending array elements, we can do similarly with multidimensional arrays. As an example we can append the row [6,7,8] to the array, a.

```
IDL> z = [[a],[6,7,8]]
IDL> print,z
       1       2       2
       5       4       3
       6       7       8
```

There are a number of useful built-in array generating functions. The function `FINDGEN` is a convenient function that creates an array of floating point numbers whose elements are made up of the floating point values of the subscripts. For instance, we can print 5 numbers in sequence beginning at 5 as follows:

```
IDL> print,5+findgen(5)
      5.00000      6.00000      7.00000      8.00000      9.00000
```

Similarly we can create a double-precision array using the `DINDGEN` function.

```
IDL> print,5+dindgen(5)
5.0000000       6.0000000       7.0000000       8.0000000       9.0000000
```

`FINDGEN` can be used to create a multidimensional array too. Since the arrays are stored in column-major format, the elements are stored sequentially. Therefore the elements in a multidimensional array created with findgen are sequential with column. That is, for (col,row) notation, the (0,0) element is 0.0, the (1,0) element is 1.0, the (1,2) element is (2.0), etc. As a concrete example we can create a 3×2 array using `findgen`.

```
IDL> z = findgen(3,2)
IDL> print,z
     0.000000      1.00000      2.00000
      3.00000      4.00000      5.00000
```

We can also create an array of structures using a function called `REPLICATE`. Replicate can be used to create a one-dimensional array of any variable type. For instance we can create an array of length 100 of the number 3.14 as follows.

```
IDL> pi_array = replicate(3.14,100)
```

In this function, the first argument is the number to be replicated and the second argument is the number of times to replicate it.

In order to demonstrate how to create an array of structures, let's consider 5 makes and models of used cars and define them as structures. Naturally we will use the automatic structure definition described in the last section. Individual arrays can be created by using comma-separated values within square brackets. We therefore can create an array of 5 makes as follows.

```
IDL> makes = ['Honda','Toyota','Ford','Dodge','Mazda']
IDL> models = ['Civic','Corolla','Mustang','Daytona','RX8']
IDL> nstructures = 5
IDL> car_struct_array = replicate({used_car},nstructures)
```

The last statement creates 5 structures in the array variable `car_struct_array`. We can now populate the make and model fields of each of the elements of the structure array as follows.

```
IDL> for i = 0,nstructures-1 do car_struct_array[i].make = makes[i]
```

Note that the location of the subscript on `car_struct_array` is before the reference to make, <u>not after</u> the reference to make. Also, we have introduced the for-do control statement here. The syntax for this statement should be self-explanatory for those with some basic programming experience. In particular the syntax is

```
for i = initial,final do something
```

where initial is the initial value for the incremental counter and final is the final value for the counter. *Something* represents any single command (function call, procedure call, or assignment. In general, a for-do loop can include a block of commands executed in sequence at each increment of the counter and the increment of the counter can be an integer (not necessarily 1). This requires begin and end statements as well as the increment as shown below.

```
for i = initial,final,increment do begin
      something1
      something2
endfor
```

Each element of `struct_array` can be accessed using the subscript notation as described earlier. For instance, we can access the first structure and print the used-car model as follows.

```
IDL> for i = 0,nstructures-1 do $
   for j = 0,n_tags(car_struct_array[i])-1 do print, car_struct_array[i].(j)
```

In the example above we used both array subscripting for the structure array and field subscripting for the individual fields. Note also the use of nested for-do control statements.

Pointers

A pointer is a reference to a variable and it is global in scope. Though not a true pointer in the sense of other programming languages such as C, an IDL pointer provides similar functionality.

If a pointer is just a reference to a variable, why bother using it and not the original reference? One reason is related to the size of a pointer reference. Since it is simply a reference, it is lightweight in the sense that it occupies only 4 bytes of memory. If you have large data structures for instance that must be passed into and out of procedures and/or functions then it is possible that performance might degrade. However you can create a pointer reference to that large data structure and pass it among the relevant procedures and functions. This is a good way to improve performance.

When writing GUI programs we will see that a standard way to pass information around from one module (subroutine) to another is to pass a pointer reference to a structure.

We can create two types of pointers: a null pointer and a pointer to a variable (even if the variable is undefined). A null pointer is created via the `PTR_NEW` function.

```
IDL> px = ptr_new()
IDL> help,px
PX              POINTER   = <NullPointer>
```

The null pointer can be useful when the variable (or its contents) to which it points is unknown at runtime. A pointer to a variable is created in one of two ways. The first way is to use `PTR_NEW` directly on the variable. For instance,

```
IDL> py = ptr_new(findgen(5))
```

will create a pointer to a floating point array of length 5 as created with `findgen`. The contents of `py` can be queried with `help`,

```
IDL> help,py
PY              POINTER   = <PtrHeapVar6>
```

which tells us that `py` is a pointer variable. We can extract the contents of the pointer, known as pointer *de-referencing*, by using an asterisk in front of the pointer,

```
IDL> print,*py
     0.000000      1.00000      2.00000      3.00000      4.00000
```

It is possible to extract an individual element from the variable to which the pointer is referencing. For instance, say we wish to extract the third element in the array that is pointed to by py. Then we can access this value using the following de-referencing syntax:

```
IDL> print,(*py)[2]
      2.00000
```

Pointers are also known as heap variables in IDL. When you are finished using the variables to which the pointer refers you should free up the memory.

The second way to create a pointer to a variable is to create a pointer to an undefined variable and then de-reference the desired data *into* it. This is done in the following way. First, create a pointer reference to an undefined variable using the `/allocate_heap` argument in the `ptr_new` function. (We will see in the next chapter that this syntax is used to set a keyword in IDL).

```
IDL> py = ptr_new(/allocate_heap)
```

Next create the "data" that we wish the pointer to reference.

```
IDL> y = findgen(5)
```

Finally de-reference the new variable `y` into the pointer using the asterisk syntax.

```
IDL> *py = y
```

Now you can use `py` in exactly the same way (i.e. accessing the data) as has already been discussed. As usual, when finished with the pointer variable, use `ptr_free` to free up the memory.

```
IDL> ptr_free,py
```

It should be mentioned also that if you use the `/allocate_heap` argument to ptr_new but never de-reference any data into that pointer, you still need to free up the memory with ptr_free. However if you create a null pointer at runtime but no defined variable is ever put into the pointer then there is no need to free any memory.

Now let's consider an example that is slightly more complicated in terms of de-referencing. Consider the used car structure defined in the previous section on scalar variables.

```
IDL> pstruct = ptr_new(car1)
```

A simple way to extract the fields from the pointer is to de-reference the pointer in the following way.

```
IDL> this_car = *pstruct
IDL> print,this_car.year
```

Alternatively we can remove the middle step of de-referencing the pointer into `this_car` and get out the year value directly.

```
IDL> print,(*pstruct).year
```

Finally, don't forget to free up the memory.

```
IDL> ptr_free,pstruct
```

Now we can get still more complicated and create a pointer to the used car structure array contained in the variable `car_struct_array`. We create the pointer as before and de-reference the make of the 4th structure as follows.

```
IDL> pastruct = ptr_new(car_struct_array)
IDL> print,(*pastruct)[3].make
```

As usual, free up the memory using ptr_free.

```
IDL> print,(*pastruct)[3].make
Dodge
```

We now consider one final complicated example involving pointers and structures. Let's create a data structure that has a number of fields including `x`,`y`, and `pz` where `x` is a floating point array of length 5, `y` is an integer, and `pz` is a pointer to a 256×256 byte array.

```
IDL> data_struct = {x:findgen(5),y:15,pz:ptr_new(bindgen(256,256))}
IDL> pdata = ptr_new(data_struct)
```

In the first statement we used the function called `BINDGEN` which creates a 256×256 byte array filling in the values in sequence much like `FINDGEN` and `DINDGEN`. We can extract x and y in the usual way.

```
IDL> print,(*pdata).x
IDL> print,(*pdata).y
```

Note however the order of parentheses and asterisks to de-reference the variable referenced by `pz`.

```
IDL> help,*(*pdata).pz
<PtrHeapVar56>  BYTE      = Array[256, 256]
```

Finally we can extract the value of the variable in pz, say the 126th column and 57th row, as follows.

```
IDL> help,(*(*pdata).pz)[125,56]
<Expression>    BYTE      =  125
```

Lastly, we can free the memory in pdata in one of two ways. The first (and arguably the simplest) is to use heap_free, which is a recursive function that descends all the way down to the bottom of the structure hierarchy, and frees pointers on its way back up.

```
IDL> heap_free, pdata
```

Alternatively you can explicitly free the memory referenced by `pz` first and then free the memory from pdata as follows.

```
IDL> ptr_free,(*pdata).pz
IDL> ptr_free, pdata
```

Hopefully it is apparent now that growing structures with embedded pointers, structures, and other variables can quickly lead to complicated data structures requiring some care in extracting information. Nevertheless you should gain a working knowledge of pointer de-referencing for programming GUIs.

System variables

IDL has a number of variables that are defined at the system level and are available in a global sense. That is they are available to all program modules on all operating levels. System variables are used to set the options for plotting, to set various internal modes, to return error status, etc. Note that it is also possible for a user to define a new system variable using the procedure `DEFSYSV`.

Some of these system variables are constants such as the number for π. Such variables are accessed using the `!` symbol (the exclamation point is also referred to as "bang"). For instance, π is represented in IDL as `!pi` ("bang pi").

```
IDL> print,!pi
      3.14159
```

There is a double-precision version of π and is obtained via

```
IDL> print,!dpi
      3.1415927
```

Other system variables of interest include the conversion factors `!dtor` and `!radeg` which are the conversion factors from degrees to radians and radians to degrees.

```
IDL> print,!dtor,!radeg
    0.0174533      57.2958
```

There are numerous system variables and we will introduce them throughout this course on an as-needed basis.

Chapter 3 Functions and procedures

Programs in IDL amount to a collection of functions and procedures. Functions are program elements that accept parameters and/or keywords and return some value. The return value can be any valid data type as discussed in the last chapter. You have already been using some of the built-in functions (`N_ELEMENTS`, `TAG_NAMES` and `N_TAGS`) and procedures (`HELP`) of IDL so you have seen the syntax for their use. For completeness, however, we will discuss the general syntax of IDL functions and procedures. In addition, we will write our own example functions and procedures. Note that the example presented in the last chapter, `used_car__define`, was written as a procedure.

Mathematics

Mathematical functions such as multiplication, division, addition, subtraction, and exponentiation have the following syntax:

```
y = a * b
y = a / b
y = a + b
y = a - b
y = a ^ b
```

IDL has numerous built-in mathematical functions and we will not discuss all of them. We will discuss those that are most common and those that arise in neutron scattering data treatment. The on-line help can be consulted for a complete coverage of all built-in functions and procedures.

Examples of some simple mathematical operations in IDL are shown below:

```
IDL> print,(2^6)^(1/6)+1
       2
IDL> print,(4.0*(5.0+6.0-8.0)^3)/2.
      54.0000
IDL> print,(cos(!pi/4))^2
     0.500000
```

Binary mathematical operations, i.e. those that operate on two quantities at a time such as addition, work on both scalar quantities and arrays provided that the arrays have identical dimensions. For simplicity we can demonstrate that the binary operations are array operations for the case of addition. In particular we consider two examples: one in which we add a scalar value to every element in an array and one in which we add one array to another array. The following example illustrates the first case:

```
IDL> x = [1.0,3.0,-10.0]
IDL> print,x+5.0
      6.00000      8.00000     -5.00000
```

The second case is given by the following (using the array, x, in the previous example):

```
IDL> y = [2.0,4.0,8.0]
IDL> print,x+y
      3.00000      7.00000     -2.00000
```

Logical operations can also be performed on scalars and arrays. Some of these operations include EQ, GT, LT, GE, LE for "equals", "greater than" , "less than", "greater than or equal to", and "less than or equal to", respectively. These logical operators return, in general, a byte array of the same size as the first argument composed of ones and zeros indicating which elements satisfy (or violate) the condition. For example we can find which elements in x (given in the last example) are positive as follows.

```
IDL> print,x gt 0
   1   1   0
```

The first two elements are true (a byte value of 1) because 1.0 and 3.0 are greater than zero and the last element is false (a byte value of 0). This byte array can be used as a mask so that only those elements satisfying the condition are returned and all other elements are zero. This can be done as follows:

```
IDL> print,mask*x
      1.00000      3.00000     0.000000
```

This can also be done using >, the greater than symbol as follows:

```
IDL> print,x > 0.0
      1.00000      3.00000     0.000000
```

The > symbol is one of a number of symbols that can be used to return values of the same type as the array rather than logical values. The symbols that can be used in this manner are >,<, and =.

Syntax

The syntax for calling an IDL function is:

```
                  ret_val = func(p1,p2,p3,…k1=k1,k2=k2,…)
```

where `ret_val` is the return value of the function, p_i are parameters and the k_j are keywords. If parameters or keywords are passed in as variables, they are said to be passed in *by reference*. Thus their scope is not simply limited to the function itself.

The syntax for calling an IDL procedure is:

```
proc p1,p2,p3,…k1=k1,k2=k2,…
```

where p_i are parameters and the k_j are keywords. If parameters or keywords are passed in as variables, they are said to be passed in *by reference*. As in functions, their scope is not limited to the procedure.

Let's first consider a simple example of a function with one input parameter and one input keyword. Type this function into a new editor, save it, and compile it.

Example:

Write a simple function that scales a number by a factor with an interface with a single parameter (the number) and a single keyword (the scale factor).

```
function scale_it,number,factor = factor
if n_elements(factor) eq 0 then factor = 1.0
result = number*factor
return,result
end
```

We can test the this function by executing the following statements at the IDL command line.

```
IDL> print,scale_it(10.0)
      10.0000
```

Since no scale factor was passed in (via the factor keyword), the code uses a default factor value of 1.0. The second line in the function makes this assignment. In this function we use the function `N_ELEMENTS` to determine if the function was called with the `FACTOR` keyword assigned. We can call the function with the factor keyword assigned as follows.

```
IDL> print,scale_it(20.0,factor = 2.0)
      40.0000
```

Let's consider another example of a function that has a number of optional input keyword parameters.

Example:

Write a function that creates a regularly-spaced vector of values between user-defined limits and whose number is defined by the user.

```
function makepts, npts = nnpts,  $
                  xlo = xxlo,    $
                  xhi = xxhi
; This function computes a vector,x, of equally-spaced values between
; xlo and xhi.
; Test for presence of the input keywords and fill them in
; with defaults if not present.
if n_elements(nnpts) eq 0 then nnpts = 10
if n_elements(xxlo) eq 0 then xxlo = 0.0
if n_elements(xxhi) eq 0 then xxhi = 1.0
dx = (xxhi-xxlo)/(nnpts-1.0)
return,xxlo+dx*findgen(nnpts)
end
```

We can test this function by compiling it and running it from the IDL command line as follows.

```
IDL>  .compile makepts
IDL> x = makepts(npts = 10,xlo = -5.0,xhi = 5.0)
IDL> print,x
     -5.00000     -3.88889     -2.77778     -1.66667    -0.555555
0.555556      1.66667      2.77778      3.88889      5.00000
```

During the course of this training we will need access to some "data" for both visualization and analysis. The remainder of this chapter is devoted to creating some fake scattering data using functional implementations of Gaussians, Lorentzians, and random numbers to generate noise.

In the previous example we saw how to incorporate keywords that can be assigned some value. IDL also allows users to use a special mechanism in which Boolean keywords can be "set" using a forward slash (/) character. For instance, we have already used the HELP procedure to determine the nature of some variables that we defined. There are numerous keywords associated with HELP. One such keyword is MEMORY which, when set, provides information on the current graphics device that is set for your local computer system. On my PC this procedure with the keyword set yields the following output:

```
IDL> help,/device
Available Graphics Devices: CGM HP METAFILE NULL PCL PRINTER PS WIN Z
Current graphics device: WIN
    Screen Resolution: 1152x864
    Simultaneously displayable colors: 16777216
    Number of allowed color values: 16777216
    System colors reserved by Windows: 0
    IDL Color Table Entries: 256
    NOTE: this is a TrueColor device
    NOT using Decomposed color
    Graphics Function: 3 (copy)
    Current Font: System,     Current TrueType Font: <default>
    Default Backing Store: None.
```

The following syntax is equivalent:

```
IDL> help,/device
IDL> help,device = 1
```

The next function that we will need for our work is a Gaussian distribution. The requirements for the function are as follows:

1. Independent variable, `x`, is an optional input/output parameter. If `x` is undefined but used as an argument in the function then the function will create some sensible default values for `x` which will be passed out.
2. The functional parameters, integrated `AREA`, `CENTER`, and full-width at half-maximum (`FWHM`) will be specified as optional input/output keywords.
3. A Boolean keyword named `PLOTIT` will be available so that, when set, a plot of the Gaussian distribution will appear in a new window.
4. Additional keywords can be passed into the plot procedure via keyword inheritance. Keyword inheritance is explained in more detail below.

Keyword inheritance is a mechanism by which keywords can be passed into a function (procedure) where those keywords are not defined in that function (procedure) but are passed into another routine contained therein. This is particularly useful in cases where you wish to write a "wrapper" routine. A wrapper routine is a variation of an existing routine (either provided by RSI or it is a user-written routine) where only a small part of the functionality is changed.

In the next example we construct a function that returns a Gaussian distribution based on a number of inputs. The independent variable, `x`, at which the function is to be evaluated is an optional input parameter. The distribution parameters such as the integrated area, center, and full-width at half-maximum are passed in as optional keywords. A `plotit` keyword is also available to be set by the user in which case a new window with the distribution will be plotted using the IDL procedure named `plot`.

The plot command has many keywords available for use and we would like to make those all available to the user of our `GAUSSIAN` function. The way that we do this is using keyword inheritance. In particular we add the `_EXTRA = extra` statement in the first line of the definition statement. Thus one can call the `GAUSSIAN` function with keywords that are not defined explicitly for it but rather are passed into the `plot` procedure call. The function definition is defined as stated in the next example.

Example:

Implement a Gaussian function which returns the distribution evaluated at one or more points defined by an optional input parameter,x, and allow optional keywords for the integrated area, center, and full-width at half-maximum. Finally allow the possibility to plot the function in a new window by setting a keyword.

```
function gaussian,   x,                $
                     area = area,      $
                     center = center,  $
                     fwhm = fwhm,      $
                     plotit = plotit,  $
                     _Extra = extra

; Test for the presence of the input parameter x.
if (n_params() eq 0) or (n_elements(x) eq 0) then $
   x = makepts(xlo = -10.0,xhi = 10.0,npts = 100)
; Test for the presence of the keywords
if n_elements(area) eq 0 then area = 1.0
if n_elements(center) eq 0 then center = 0.0
if n_elements(fwhm) eq 0 then fwhm = 1.0

; Convert the full-width at half-maximum to the
; standard deviation of the Gaussian distribution
sig = fwhm/sqrt(8.0*alog(2.0))
yg = (area/sqrt(2.0*!dpi*sig^2))*exp(-0.5*((x-center)/sig)^2)

; If the keyword PLOTIT has been set then plot the function
; in a new window.
if keyword_set(plotit) then begin
   window,/free,xsize = 300,ysize = 300
   plot,x,yg,psym = -4,xtitle = 'x',ytitle = 'y(x)', $
      title = 'Gaussian Distribution',_Extra = extra
endif
return,yg
end
```

The mathematical definition for the Gaussian is

$$y_{\mathrm{Gauss}}(x)=\frac{A}{\sqrt{2\pi}\sigma}\exp\left(-\frac{1}{2}\left(\frac{x-x_o}{\sigma}\right)^2\right), \qquad (3.1)$$

where A is the integrated area of the distribution, x_o is the center, and σ is related to the full-width at half maximum of the distribution via $\sigma = \mathrm{FWHM}/\sqrt{8\ln(2)}$.

Let's try this function out at the command line.

```
IDL> x = makepts(xlo = -5.0,xhi = 5.0,npts = 5)
IDL> print,gaussian(x)
7.4108380e-031  2.7997398e-008      0.93943728  2.7997398e-008  7.4108380e-031
```

Now we can plot the function with a similar call as that just specified.

```
IDL> x = makepts(xlo = -5.0,xhi = 5.0,npts = 50)
IDL> yg = gaussian(x,/plotit,psym = 0,thick = 2.0,xtitle = 'x', $
     ytitle = 'y(x)',title = 'Gaussian Distribution')
```

You should see a plot of the Gaussian function appear on your display similar to that shown in Figure 1. In the Gaussian function call we set the `plotit` keyword to invoke the `plot` procedure. Furthermore we set the `PSYM` keyword to zero to tell the `plot` procedure to plot with a line but no symbols. Other keywords passed in such as `thick`, `XTITLE`, `YTITLE`, and `TITLE` specify the line thickness and the axis titles. Thus the `PSYM`, `THICK`, `XTITLE`, `YTITLE`, and `TITLE` keywords are sent to the `plot` procedure via the keyword inheritance mechanism. We will discuss the `plot` procedure in more detail in chapter 5.

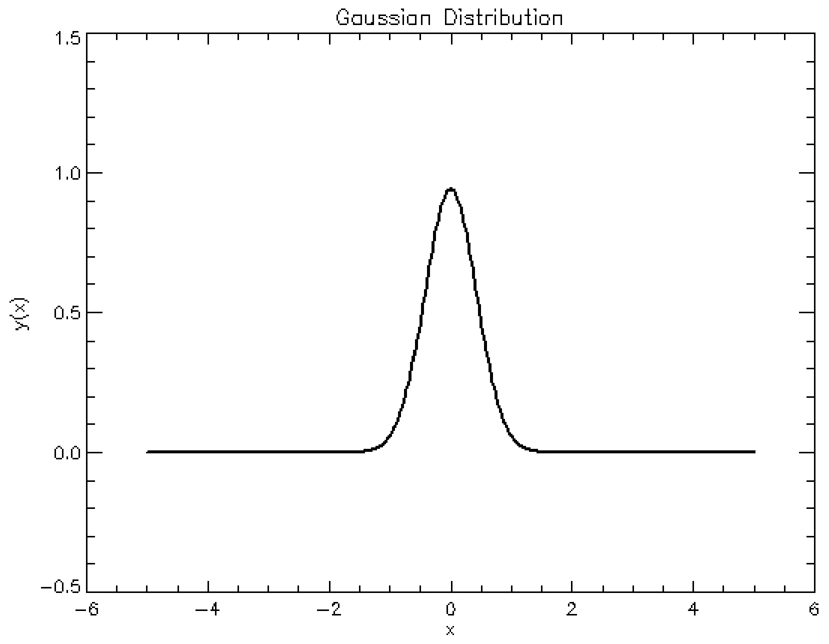

Figure 1 Gaussian distribution using the default values (area = 1.0, center = 0.0, fwhm = 1.0).

The code for a Cauchy distribution (also known as a Lorentzian lineshape) is presented below. It is similar in implementation as the Gaussian. The mathematical definition for this distribution is

$$y_{\mathrm{Lorentz}}(x) = \frac{A\Gamma}{\pi}\frac{1}{(x-x_o)^2+\Gamma^2}, \tag{3.2}$$

where A is the integrated area of the distribution, x_o is the center, and Γ is related to the full-width at half maximum of the distribution via $\Gamma = \mathrm{FWHM}/2$.

The implementation of this function is listed below.

Example:

Implement a Lorentzian distribution function with a functional interface similar to that used in `GAUSSIAN` but with a definition consistent with equation 3.2.

```
function lorentzian, x,                $
                     area = area,      $
                     center = center,  $
                     fwhm = fwhm,      $
                     plotit = plotit,  $
                     _Extra = extra

; Test for the presence of the input parameter x.
if (n_params() eq 0) or (n_elements(x) eq 0) then $
   x = makepts(xlo = -10.0,xhi = 10.0,npts = 100)
; Test for the presence of the keywords
if n_elements(area) eq 0 then area = 1.0
if n_elements(center) eq 0 then center = 0.0
if n_elements(fwhm) eq 0 then fwhm = 1.0

gam = 0.5*fwhm
yl = (area*gam/!pi)/((x-center)^2+gam^2)

; If the keyword PLOTIT has been set then plot the function
; in a new window.
if keyword_set(plotit) then begin
   window,/free,xsize = 300,ysize = 300
   plot,x,yl,psym = -4,xtitle = 'x',ytitle = 'y(x)', $
      title = 'Lorentzian Distribution',_Extra = extra
endif
return,yl
end
```

As in the example with `GAUSSIAN` you can test the implementation of this function using the following command:

```
IDL> yl = lorentzian(x,/plotit,psym = 0,thick = 2.0,xtitle = 'x', $
     ytitle = 'y(x)',title = 'Lorentzian Distribution')
```

In this course we will need to have access to some neutron scattering data. We could just load some in from a raw data file but it is more illustrative to generate some fake data by using the functions we have written so far and add some Poisson noise to simulate "real" data.

IDL has a number of built-in random number generators which we will use in our measurement simulation. The function `RANDOMN` allows calculation of `BINOMIAL`, `GAMMA`, `NORMAL`, and `POISSON` random deviates. Samples from each distribution can be drawn by setting the appropriate keywords to RANDOMN. To see an example of uniform deviates between 0 and 1, we can consider the following code:

```
IDL> x = randomn(s,100,/uniform)
IDL> plot,x
```

The parameter s is an output parameter that is the seed for the random number generator. Users of this function should use an undefined variable for s at first but then use the same variable for subsequent calls to the random number generator.

A plot showing data that varies between 0 and 1 should appear on your screen. If we wish to see how close this distribution is to uniform, we can use the IDL function HISTOGRAM as follows:

```
IDL> blo = -1.0 & bhi = 2.0 & nbins = 20
IDL> b = makepts(xlo = blo,xhi = bhi,npts = nbins)
IDL> xh = histogram(x,min = blo,max = bhi,nbins = nbins)
IDL> plot,b+0.5*db,xh/total(xh),psym = 10
```

The result of the histogram is shown in Figure 2.

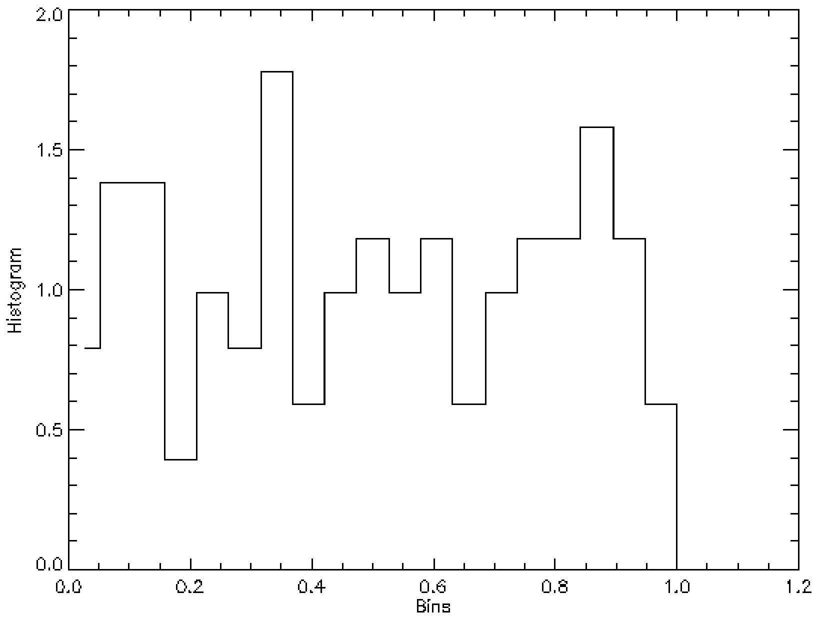

Figure 2 Histogram of the uniform random number generator in IDL for 100 samples.

It is useful to perform the same histogram for 10000 uniform random deviates and see how well the uniform distribution is approximated. The result of this is shown in Figure 3. It is clear that the histogram with more samples is a better approximation to the uniform distsribution.

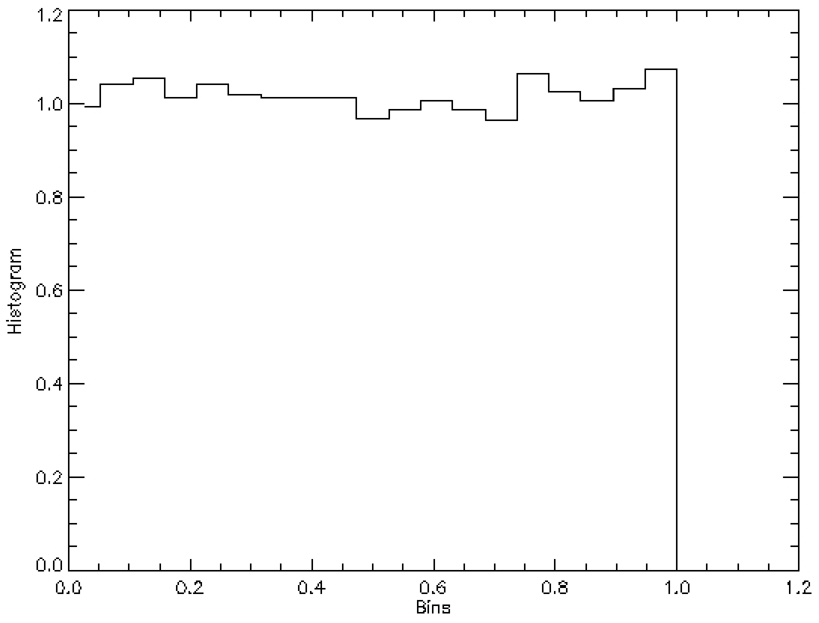

Figure 3 Same as figure 2 except for 10000 uniform random deviates.

Convolution

In a neutron scattering measurement the resolution of the instrument can distort the intrinsic lineshape given by the physical process under investigation. Such distortion is due to the non-ideal properties of the monochromator and/or analyzer materials, guide systems, and intrinsic timing errors in the data acquisition system. This distortion often manifests itself as a blurring process and, in many cases, the blurring can be quantified by a convolution integral,

$$\begin{aligned} S_{meas}(Q,\omega) &= S_{int}(Q,\omega) \otimes R(Q,\omega) \\ &= \int S_{int}(Q,\omega')R(Q,\omega-\omega')d\omega' \end{aligned} \tag{3.3}$$

where $R(Q,\omega)$ is the instrumental resolution function of the instrument used in the measurement process. The conditions under which the instrumental resolution can be factored into the measurement process in this manner are somewhat restricted. The main requirement is that the measurement system is linear shift-invariant. This means that the width (for instance) of the resolution function does not change with energy transfer, ω. This is not valid for measurements on a triple-axis spectrometer nor is it valid for inelastic measurements on direct (or indirect) geometry time-of-flight spectrometers. Nevertheless the convolution integral as specified in equation 3.3 can be appropriate for the analysis of quasielastic neutron scattering data on time-of-flight and backscattering spectrometers for instance.

As written in equation 3.3, it should be apparent how one can discretize the integral and formulate an approximation of the integral as a sum. IDL has a built-in function named `CONVOL.PRO` that purports to do this. However we will see that the function is not strictly a correct approximation to a convolution integral as originally written and requires a bit of modification. First we will review the result of a convolution integral in a simple case. For the rectangle function shown in Figure 4, the result of a self-convolution (i.e. a convolution of the function with itself) is a triangle function.

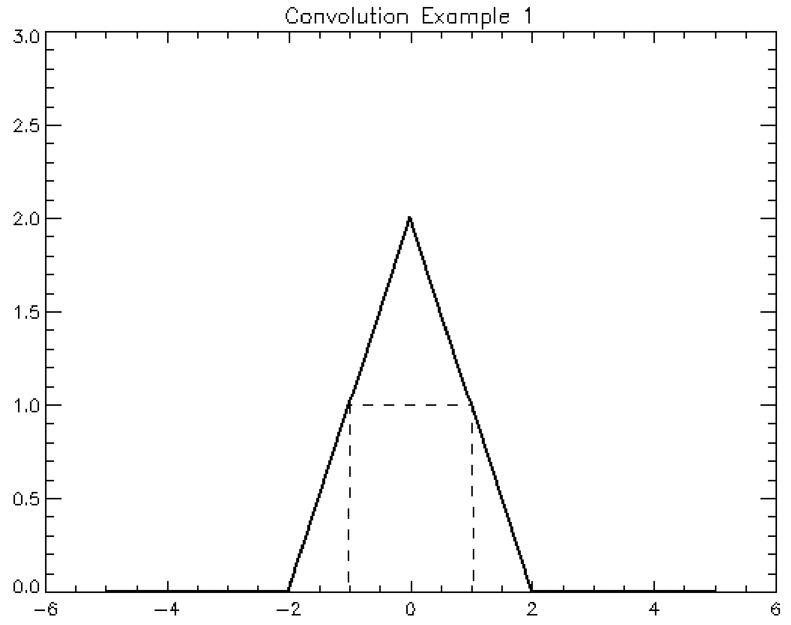

Figure 4 The thick solid line is the result of the self-convolution of the rectangle function.

Now if we consider the convolution of a narrow symmetric function with an asymmetric function as is shown in Figure 5 then the result is a blurred version of the asymmetric function.

In IDL convolution can be performed in a couple of ways. The first method is to use the Fast Fourier Transform (FFT) technique. In this method the FFT is calculated for the functions involved in the convolution integral and the result transformed via an inverse FFT. For large arrays this can be an expedient way to do the convolution.

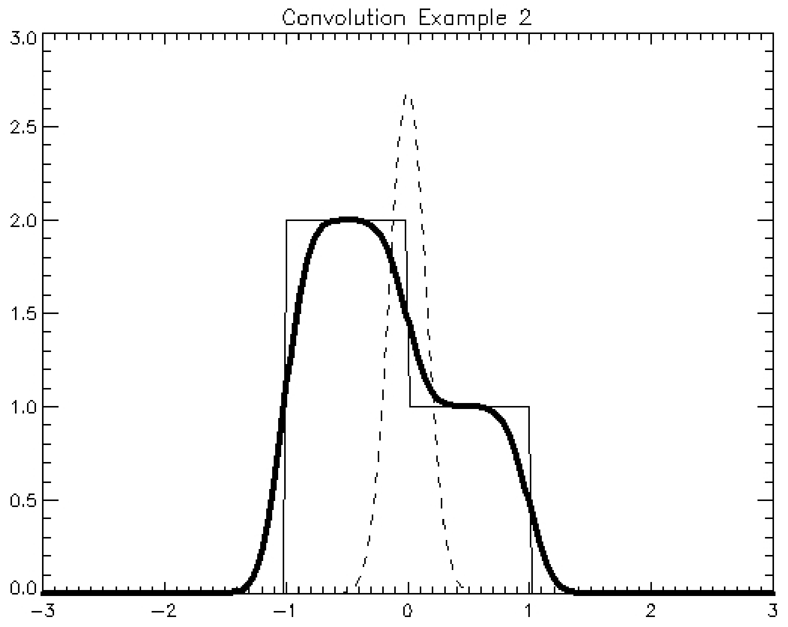

Figure 5 Result (thick solid line) of the convolution of an asymmetric function (thin solid line) with a narrow symmetric function (thin dashed line). The result is a blurred version of the asymmetric function.

The FFT has the following syntax as listed in the on-line documentation

```
Result = FFT( Array [, Direction] [, DIMENSION=vector] [, /DOUBLE]
[, /INVERSE] [, /OVERWRITE] )
```

where DIRECTION is an optional input parameter (positive number for inverse FFT and negative number for forward FFT), dimension is the dimension in the array for which to calculate the transform, DOUBLE forces calculation using double-precision arithmetic, INVERSE is the same as setting the DIRECTION equal to a positive number, and OVERWRITE results in the transform taking the place of the original array. Given two arrays defined over the same domain (composed of equally-spaced values), ARRAY1 and ARRAY2, the convolution integral can be approximated using the following sequence of commands:

```
dx = x[1] - x[0]
nx = n_elements(x)
n21 = nx/2+1
result = shift(fft(fft(array1,1)*fft(array2,1),-1),n21)*dx
```

In the last line above we used the SHIFT function which wraps an array by some specified amount. In the example the array is shifted by n21. This is necessary to compensate the FFT which returns a shifted result.

The second convolution method uses a variant on the CONVOL routine. The on-line documentation for CONVOL lists the following syntax.

```
Result = CONVOL( Array, Kernel [, Scale_Factor] [, /CENTER]
[, /EDGE_WRAP] [, /EDGE_TRUNCATE])
```

We have omitted two additional keywords in the interest of brevity: MISSING and NAN. The array is the original array to be convolved, kernel is the "resolution" function, scale factor is the number by which to multiply the result of the convolution, and the remainder are keywords that control how the convolution is performed. The kernel is expected to have fewer elements than the array. The default value for scale_factor is 1.0. Ignoring the CENTER keyword or setting it to zero will cause the kernel to be centered over each point in the array during the computation. Setting the EDGE_WRAP keyword will cause the elements at the edge of the array to be calculated by wrapping the subscripts of the array at the edge. Setting the EDGE_TRUNCATE keyword will cause the elements at the edge of the array to be calculated by repeating the subscripts of the array at the edge.

Given two arrays defined over the same domain (composed of equally-spaced values), ARRAY1 and ARRAY2, the convolution integral *should* be approximated using the following sequence of commands involving CONVOL:

```
dx = x[1] - x[0]
nx = n_elements(x)
nlo = 3 & nhi = nx-nlo-1
result = convol(array1, array2[nlo:nhi],/edge_truncate)*dx
```

The result of using the CONVOL routine as described in the code snippet above is shown in Figure 6.

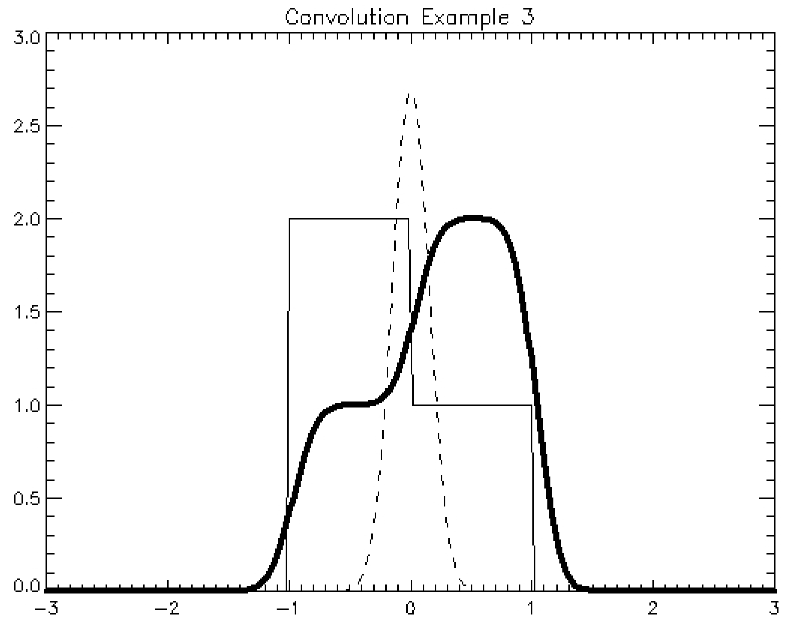

Figure 6 Result of using CONVOL routine as-is.

There is clearly a problem with how the kernel is treated in this function. In order to get the correct answer it is necessary to reverse the kernel using the REVERSE function. Therefore the correct result is given by

```
result = convol(array1, reverse(array2[nlo:nhi]),/edge_truncate)*dx
```

The correct approximation to the convolution integral as described by the code above is displayed in Figure 5. As a final example in this chapter, we present the complete code used in constructing Figure 5. Note the use of the HEAVISIDE function where we have made use of the logical operations GT, LT, and EQ.

Example:

An example illustrating the correct use of the CONVOL function is presented here. Two new functions are presented here: HEAVISIDE which is an implementation of the unit step function and RECT which is an implementation of a rectangle function.

```
; *********************************
function heaviside,x
y = (x gt 0)*1D + ((x lt 0))*0D + 0.5D*(x eq 0)
return,y
end
; *********************************
function rect,x,b,h
; b: base (width) of the rectangle
; h: height of the rectangle
r = h*(heaviside(x+0.5*b)-heaviside(x-0.5*b))
return,r
end
; *********************************
pro convolution_example
nx = 400
x = makepts(xlo = -5.0,xhi = 5.0,npts = nx)
dx = x[1] - x[0]
base = 2.0 & height = 1.0
kernel = rect(x+0.5,1.0,2.0)+rect(x-0.5,1.0,1.0)
array = gaussian(x,area = 1.0,center = 0.0,fwhm = 0.35)
nlo = 3 & nhi = nx-nlo-1
plot,x,array,yrange = [0.0,3.0],/ysty,linestyle = 2, $
   title = '!3Convolution Example 2',xrange = [-3.0,3.0],/xsty
oplot,x,kernel,linestyle = 0
ycon1 = convol(array,reverse(kernel[nlo:nhi]),/edge_truncate)*dx
oplot,x,ycon1,thick = 4.0,color = black
end
```

It should be noted that, for the purposes of these discussions, the data ranges are assumed to be on an equally-spaced grid. We will revisit the convolution operation in the next chapter to circumvent this limitation.

Synthesizing data

At this point we have created all of the functions necessary to create our "synthetic" neutron scattering data. The following example illustrates how we can create an elastic peak plus an excitation with a Lorentzian lineshape. The instrumental resolution will be assumed to be linear shift-invariant and approximated well by a Gaussian lineshape. The measured lineshape is described by the following expression,

$$S_{\mathrm{meas}}(\omega;\Gamma) = R(\omega) \otimes \left[\delta(\omega) + \frac{\Gamma}{\pi} \frac{1}{(\omega-\omega_o)^2 + \Gamma^2} \right], \tag{3.4}$$

where $R(\omega)$ is the instrumental resolution function, $\delta(\omega)$ is the elastic scattering, and the Lorentzian represents an inelastic excitation centered at ω_o with a full-width at half-maximum of 2Γ. If the resolution function is a Gaussian then we can re-write equation 3.4 as

$$S_{\mathrm{meas}}(\omega;\Gamma,\sigma) = G(\omega;\sigma) + G(\omega;\sigma) \otimes \left(\frac{\Gamma}{\pi} \frac{1}{(\omega-\omega_o)^2 + \Gamma^2} \right), \tag{3.5a}$$

where

$$G(\omega;\sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2} \left(\frac{\omega}{\sigma} \right)^2 \right). \tag{3.5b}$$

The convolution of the Gaussian and Lorentzian cannot be expressed in an analytic way so it must be calculated numerically. (There is a parametric way of representing the convolution of a Gaussian with a Lorentzian and it is known as a Voigt function, available in IDL).

In order to simulate a neutron scattering measurement we need to add some noise to the "data". In a neutron scattering measurement one detects an event (the detection of a neutron) in a finite interval (a channel). The statistics of this process are Poissonian. We can use the random number generator function RANDOMN to generate random deviates drawn from a Poisson distribution. For instance, if we wish to draw 100 samples from a Poisson distribution whose mean value is 3 then we can issue the following command:

```
IDL> rand_poisson = randomn(s,100,poisson = 3)
```

As we did with the uniform deviates we can histogram the Poisson deviates and see how well the histogram approximates the theoretical Poisson distribution. This is illustrated in **Figure 7** for 100000 samples.

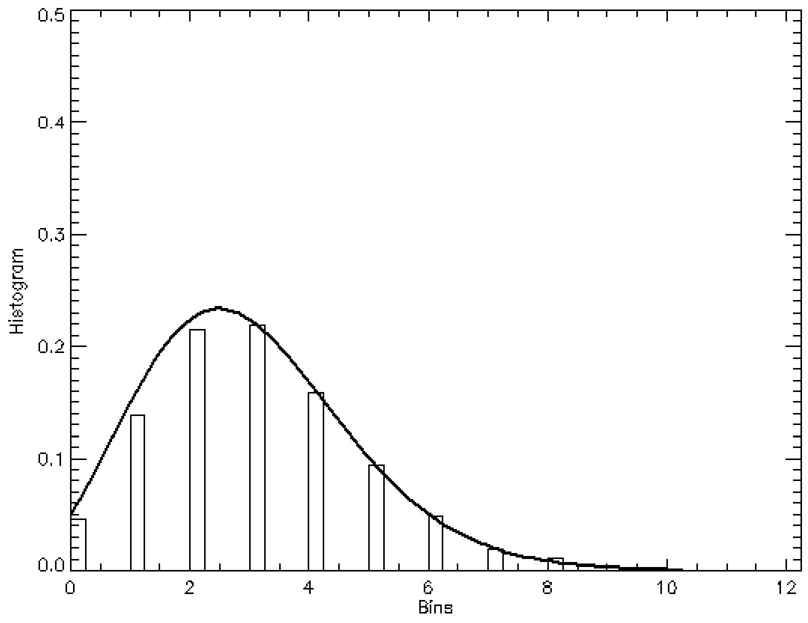

Figure 7 Histogram of the Poisson random number generator for 100000 samples with a mean value of 3. Solid thick line is the theoretical Poisson distribution for a mean value of 3.

The routine that generates data consistent with equations 3.4 and 3.5(a-b) is shown in the next example. In this example we have a couple of options available via keywords. The procedure can be called as follows:

```
IDL> make_fake_data,x,y,yerr,/draw,/write
```

The first three parameters, `x`, `y`, and `yerr` are output parameters and contain the simulated data. Setting the `DRAW` keyword results in the appearance of a plot of the simulated data. Setting the `WRITE` keyword results in writing a three column text file containing `x`, `y`, and `yerr`. The parameters and keywords are optional so that you can also call the procedure as follows

```
IDL> make_fake_data, /draw
```

to see a plot of the simulated data appear on the screen.

<u>Example:</u>

Write a procedure that returns simulated neutron scattering data in three output parameters, allows the user to plot the data to the screen with an optional input keyword, and allows the user to save the data in three column ascii format.

```
pro make_fake_data,x,y,yerr,draw = draw,write = write
!except = 0 ; suppress mathematical underflow messages
```

```
npts = 200

; Create the grid of values for the independent variable
x = makepts(xlo = -10.0, xhi = 10.0, npts = npts)

; Create a resolution function: a Gaussian with FWHM of 1.0
res = gaussian(x,area = 1.0,cen = 0.0,wid = 1.0)

; Create the elastic component: a Gaussian with area 600.0
y1 = 600.0*res
y2 = lorentzian(x,area = 300,cen = 5.0,wid = 1.0)

; Convolve the Lorentzian with the resolution function
dx = x[1]-x[0]
nlo = 3 & nhi = npts-nlo-1
y2con = convol(y2,reverse(res[nlo:nhi]),/edge_truncate)*dx

y = fltarr(npts)
yerr = fltarr(npts)
for i = 0,npts-1 do begin
  y[i] = randomn(s,1,poisson = y1[i]+y2con[i]+20.0)
endfor

yerr = sqrt(y)
if keyword_set(draw) then begin
  plot,x,y,psym = 4, xtitle = '!4x!3', $
         ytitle = '!3S(!4x!3)',        $
         title = 'Measured Intensity'
  oplot,x,y,psym = 4
  dx = x[1]-x[0]
  errplot,x,y-yerr,y+yerr,width = 0.0
endif

if keyword_set(write) then begin
   cd,current = dir
   filename = dir+path_sep()+'fake_data.txt'
   openw,lun,filename,/get_lun
   for i = 0,npts-1 do begin
      printf,lun,x[i],y[i],yerr[i],format = '(3f15.6)'
   endfor
   free_lun,lun,/force
endif

end
```

Calling this procedure as follows

```
IDL> make_fake_data, /draw
```

results in a plot like that shown in Figure 8.

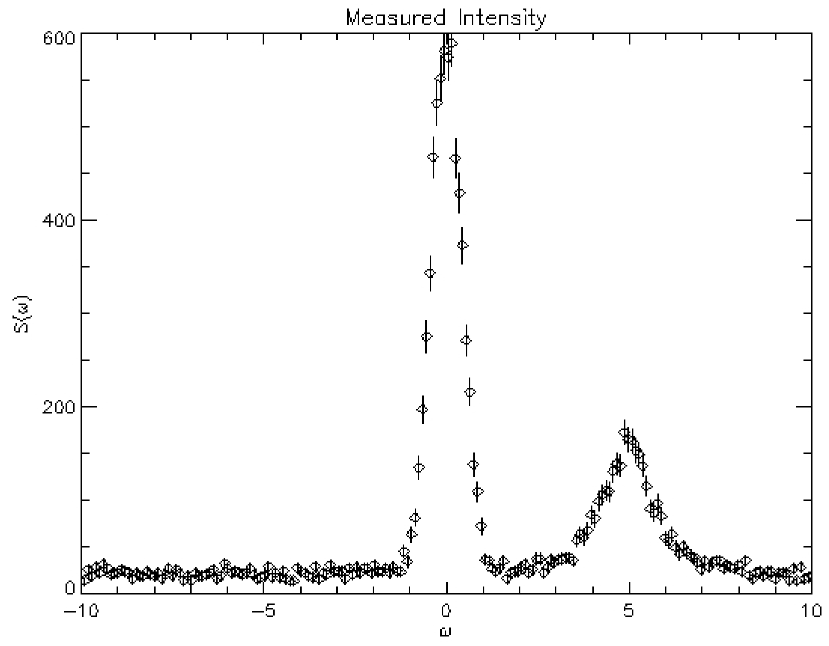

Figure 8 Synthetic neutron data created using MAKE_FAKE_DATA.PRO.

The only major part of the code shown in MAKE_FAKE_DATA.PRO that we have not discussed yet is that related to file output. We will examine this code briefly.

In IDL, as in other languages such as FORTRAN, you must open a file and associate a logical unit number with the opened file. This is done in the 3rd statement shown in the code snippet below. The PRINTF statement is used to print the variables to the file using the logical unit number as a critical part of the statement. The format statements are similar to those used in FORTRAN. When finished, the file is closed by freeing the logical unit number via FREE_LUN.

```
cd,current = dir
filename = dir+path_sep()+'fake_data.txt'
openw,lun,filename,/get_lun
for i = 0,npts-1 do begin
   printf,lun,x[i],y[i],yerr[i],format = '(3f15.6)'
endfor
free_lun,lun,/force
```

It is also worth noting the use of the CD procedure with the output keyword CURRENT. In the code snippet above the variable dir is the working IDL directory.

Chapter 4 Accelerating your IDL code

IDL has a large number of built-in array functions that allow run-time execution times similar to compiled C or FORTRAN. In this chapter we will discuss how you can use these functions to speed up your code.

Simple first examples

The novice IDL programmer often uses what he/she knows from other languages. For instance many of the programming constructs of FORTRAN are present in IDL. One particular example is a FOR loop. However the FOR loop should be used sparingly in IDL code since it will tend to slow code execution. Let's consider an example in IDL in which array operations can be used to accelerate code that you might ordinarily think of writing using loops.

Example:

Create an equally spaced vector of 500000 values between –10.0 and +10.0. The code to perform this calculation using loops first and then using a built-in IDL floating point generator function called FINDGEN is shown below.

```
pro create_vector_speed_test
num = 500000
; Calculate the vector using loops
nx = num
; Create the x-vector
xlo = -10.0
xhi = 10.0
dx = (xhi-xlo)/(nx-1.0)

tstart = systime(/seconds)
x = fltarr(nx)
for i = 0L,nx-1 do begin
      x[i] = xlo+dx*i
endfor
telapsed = systime(/seconds)-tstart
print,'Elapsed time:',telapsed

; Now calculate the vector using MAKEPTS which takes advantage of the
; array functionality.
tstart = systime(/seconds)
x = makepts(xlo = xlo,xhi = xhi,npts = nx)
telapsed = systime(/seconds)-tstart
print,'Elapsed time:',telapsed
end
```

On my modest desktop PC the first method took 250 ms and the second method took 16 ms.

Perhaps one of the more difficult things to do with IDL is rewrite a double FOR-loop using array operations. Let's take a look at an example in which we wish to calculate the sum of all of the elements of a two-dimensional array.

Example:

Create a two dimensional floating point array (using FINDGEN) and calculate the sum of all of the elements.

```
pro total_example
; Create an array of floating point numbers and calculate the sum
; of one of the dimensions.
n = 3000
array = findgen(n,n)
; Loop method
tstart = systime(/seconds)
sum = 0D
for i = 0L,n-1 do begin
  for j = 0L,n-1 do begin
      sum   = sum + array[i,j]
  endfor
endfor
print,sum
t_elapsed_loop = systime(/seconds)-tstart
print,'Elapsed time (loop):',t_elapsed_loop

; Method using TOTAL function
tstart = systime(/seconds)
sum = total(array)
print,sum
t_elapsed_total = systime(/seconds)-tstart
print,'Elapsed time (TOTAL function):',t_elapsed_total
print,t_elapsed_loop/t_elapsed_total
end
```

On my desktop PC the loop calculation executed in 6.24 seconds and the TOTAL calculation executed in 4.7 ms, a time savings of more than 130.

Exercise:

Simulate 5000 tosses of an unbiased coin using the random number generator RANDOMU, the WHERE function, and TOTAL. Then plot that the accumulated results (i.e. number of heads with respect to the total number of tosses) illustrating that they approach ½ as the number of coin tosses increases.

Matrix multiplication

Matrix multiplication can also be performed in a manner that eliminates the need for a loop. Before we see how that is done, let's first see how to perform matrix multiplication in IDL.

Matrices are written in column-major format meaning that the ith column and jth row of a matrix *m* are accessed using the notation *m*[i,j]. This may be a bit different from what you are used to so make a special note of this order. Let's say we want to multiply two matrices that are written as

$$m_1 = \begin{pmatrix} 1 & 3 \\ 4 & 2 \end{pmatrix}, m_2 = \begin{pmatrix} 6 & 1 \\ 2 & 4 \end{pmatrix}.$$

We should get the answer $m_1 m_2 = \begin{pmatrix} 12 & 13 \\ 28 & 12 \end{pmatrix}$.

In IDL notation we enter these two matrices as follows:

```
IDL>  m1 = [[1.,3.],[4.,2.]]
IDL>  m2 = [[6.,1.],[2.,4.]]
```

You can see the contents of these matrices via,

```
IDL> print,m1
      1.00000      3.00000
      4.00000      2.00000
IDL> print,m2
      6.00000      1.00000
      2.00000      4.00000
```

Now recall that matrix multiplication is performed where the row of the first matrix gets multiplied by the column of the second matrix. In IDL this operation is performed using the double-hash symbol ##. Therefore you can get the result of the proper matrix multiplication as follows:

```
IDL> print,m1##m2
      12.0000      13.0000
      28.0000      12.0000
```

Of course a matrix multiplication can be written as a double for loop so we have just saved a great deal of time by using the ## symbol to replace the matrix multiplication.

There is a second matrix multiplication operator given by the single hash symbol #. Application of this symbol is performed where the column of the first matrix gets multiplied by the row of the second matrix. You can recover the correct answer for the proper matrix multiplication using the # symbol as follows:

```
IDL> print,m2#m1
      12.0000      13.0000
      28.0000      12.0000
```

Another useful feature of the matrix multiplication operators is that it they can be used to build up arrays containing copies of vectors. For instance, say we wanted 5 rows of the vector given by `x = [0.0,3.0,10.0]`. In IDL we execute the following simple command:

```
IDL> xcopy = (1+fltarr(5))##x
IDL> print,xcopy
     0.000000      3.00000      10.0000
     0.000000      3.00000      10.0000
     0.000000      3.00000      10.0000
     0.000000      3.00000      10.0000
     0.000000      3.00000      10.0000
```

The quantity that is pre-multiplied by `x` is just a vector of ones. Alternatively we can get the same result using the single-hash operator # as follows:

```
IDL> print,x#(1+fltarr(5))
     0.000000      3.00000      10.0000
     0.000000      3.00000      10.0000
     0.000000      3.00000      10.0000
     0.000000      3.00000      10.0000
     0.000000      3.00000      10.0000
```

$\Rightarrow$ In a nutshell, A#B = B##A

We can use the IDL function `REBIN` to perform the same tasks as those for which we just used matrix multiplication. For instance we can create the variable `xcopy` as follows:

```
IDL> xcopy = rebin(x,n_elements(x),5,/sample)
```

which gives the same output as the previous definition of `xcopy`. We will actually use this when we revisit the convolution operation at the end of this chapter. It turns out that `REBIN` can be used for many different things (not just rebinning data) and replicating vectors like shown above is one of the most useful. We will not go into all of the details but rather refer you to David Fanning's website where the merits of `REBIN` and `REFORM` are discussed: http://www.dfanning.com/tips/rebin_magic.html.

Eliminating a double loop

Let's consider a concrete example where we can eliminate another double FOR-loop using matrix multiplication.

Example:
Create an array D whose elements d_{ij} equal the difference between the elements, x_i-x_j, of a vector x.

```
pro speed_calc
num = 1600
; Calculate the difference array using loops
nx = num
; Create the x-vector
xlo = 0.0
xhi = 1.0
dx = (xhi-xlo)/(nx-1.0)
tstart = systime(/seconds)    ; start the timer
x = fltarr(nx)
for i = 0,nx-1 do begin
      x[i] = xlo+dx*i
endfor
; Create the difference array
diff = fltarr(nx,nx)
for i = 0,nx-1 do begin
      for j = 0,nx-1 do begin
            diff[i,j] = x[i] - x[j]
      endfor
endfor
t_loop = systime(/seconds)-tstart   ; calculate elapsed time
print,'Time elapsed (loop method):',t_loop

; Ok, now do it the optimized IDL way
tstart = systime(/seconds)    ; start the timer
x = xlo+dx*findgen(nx)
ux = 1+bytarr(nx)
diff_array = ux#x-x#ux
t_no_loops = systime(/seconds)-tstart     ; calculate elapsed time
print,'Time elapsed (no loops):',t_no_loops
print,'Speed factor improvement: ',t_loop/t_no_loops
end
```

On my desktop PC the first method took 1828 ms and the second method took 110 ms.

Functions built for speed

There are a few other functions that are built-in to IDL that can save a considerable amount of computational time. The MIN function calculates the minimum value of an array, the maximum value of an array (yes both!) as well as specify the index location for the minimum element as the following example shows.

Example:

Calculate the minimum value and maximum value of a large two-dimensional array using loops and also using the MIN function.

```
pro min_example
; Find the minimum and maximum value of an array
n = 5000
array = abs(0.5*n^2-findgen(n,n))

; Calculate using loops and if statement
tstart = systime(/seconds)
minval = 1e32
maxval = 1e-32
for i = 0L,n-1 do begin
  for j = 0L,n-1 do begin
    if array[i,j] lt minval then begin
            minval = array[i,j]
    endif
      if array[i,j] gt maxval then begin
            maxval = array[i,j]

      endif
  endfor
endfor
print,minval,maxval
t_elapsed_loop = systime(/seconds)-tstart
print,'Elapsed time (LOOP):',t_elapsed_loop

; Calculate using min
tstart = systime(/seconds)
array_min = min(array,max = array_max)
print,array_min,array_max
t_elapsed_min = systime(/seconds) - tstart
print,'Elapsed time (MIN): ',t_elapsed_min

print,t_elapsed_loop/t_elapsed_min
end
```

On my desktop PC the first method took 26.3 s and the second method took 180 ms, a savings in computation time of nearly 140.

The WHERE function is another routine that can be extremely useful and efficient when searching an array for elements that satisfy some criteria. For instance, say you want to know the elements in an array that are within some range. As an example, consider the simple example of an array that varies between -10.0 and 10.0 created using the uniform random number generator as follows:

```
IDL> x = -10.0+20.0*randomn(s,100,/uniform)
IDL> plot,indgen(100),x,psym = 0
IDL> indices = where(x le 5.0 and x ge 1.0,count)
IDL> oplot,indices,x[indices],psym = -4,symsize = 2,thick = 2.0
```

The `WHERE` function determine in which indices the criteria is satisfied (`x le 5.0 and x ge 1.0`). Note further that the number of elements is passed out as an output parameter named `COUNT`.

Rebinning data

A data operation that is frequently used in neutron scattering data reduction and analysis is rebinning. There are a couple of ways in which this can be done: with loops and without loops. It should come as no surprise at this point that the most efficient way to rebin data is using array operations rather than loops. In this section we will discuss data rebinning and a speedy implementation of it in IDL.

In a neutron scattering experiment data can be collected in one of two modes: *histogram* and *point*. In a measurement on a triple-axis spectrometer the data is collected in *point* mode which is a simple way of stating that the data is sampled from a scattering function. If the data are noisy then the experimentalist may wish to "bin channels" together in order to decrease the point-to-point fluctuations of the data. Time-of-flight measurements record neutron events in non-zero width time channels and these data are usually displayed in a *histogram*. Often the time bins into which the data are histogrammed are equal width. However subsequent conversions to physical units such as energy, angle, or wavevector transfer yield unequal resulting bins. Since the subsequent transformation of the data results in a histogram of unequal bins, it is often desirable to rebin the data into equal width bins.

Rebinning can be a useful step in visualizing data as long as the consequences of rebinning are kept in mind. For instance, binning data into bins that are too large can result in loss of important features in the data. Likewise it is usually not a good idea to rebin data into more channels than were in the original data set.

We will consider the rebinning operation for *point* and *histogram* data separately. It is most natural to think of rebinning in terms of non-zero width bins, thus let us examine *histogram* data case first.

Rebinning histogram data

We are given an input histogram with n channels numbered 0…n-1. Channel boundaries are X(i), i=0…n. Heights are Z(i), i=0…n-1; corresponding errors are dZ(i), i=0…n-1. We are also given a set of boundaries for n' output channels, X'(j), j=0…n'.

The left and right hand sides of input channel i and its width are L(i), R(i) and W(i) respectively:

$$\left.\begin{aligned} \mathrm{L(i)} &= \mathrm{X(i)}, \\ \mathrm{R(i)} &= \mathrm{X(i+1)}, \\ \mathrm{W(i)} &= \mathrm{R(i)} - \mathrm{L(i)} \end{aligned}\right\} \quad \mathrm{i} = 0...\mathrm{n}-1$$

Similarly the left and right hand sides of output channel j and its width are L'(j), R'(j) and W'(j) respectively:

$$\left.\begin{aligned} \mathrm{L'(j)} &= \mathrm{X'(j)}, \\ \mathrm{R'(j)} &= \mathrm{X'(j+1)}, \\ \mathrm{W'(j)} &= \mathrm{R'(j)} - \mathrm{L'(j)} \end{aligned}\right\} \quad \mathrm{j} = 0...\mathrm{n'}-1\,.$$

The integrated counts and associated errors for input channel i are T(i) and dT(i) respectively:

$$\begin{aligned} \mathrm{T(i)} &= \mathrm{W(i)} \cdot \mathrm{Z(i)}, \\ \mathrm{dT(i)} &= \mathrm{W(i)} \cdot \mathrm{dZ(i)}. \end{aligned}$$

The integrated counts for output channel j, T'(j), may be written as follows:

$$\mathrm{T'(j)} = \sum_{\mathrm{i}} \mathrm{T(i)} \cdot \mathrm{f(i,j)}$$

where f(i,j) is the fraction of input channel i that contributes to output channel j. Similarly the error on T'(j) is written as

$$\mathrm{dT'(j)} = \sqrt{\sum_{\mathrm{i}} \left[\mathrm{dT(i)}\right]^2 \cdot \mathrm{f(i,j)}}$$

since the contents of different input channels are independent. Hence the output histogram heights and their errors are as follows:

$$\begin{aligned} \mathrm{Z'(j)} &= \mathrm{T'(j)} / \mathrm{W'(j)}, \\ \mathrm{dZ'(j)} &= \mathrm{dT'(j)} / \mathrm{W'(j)}. \end{aligned}$$

The fractions f(i,j) may be written as ratios Δ(i,j)/W(i), with

$$\Delta(\mathrm{i,j}) = \max\left\{0, \min[\mathrm{R(i)}, \mathrm{R'(j)}] - \max[\mathrm{L(i)}, \mathrm{L'(j)}]\right\}.$$

Figure 9 illustrates this for *histogram* type data.

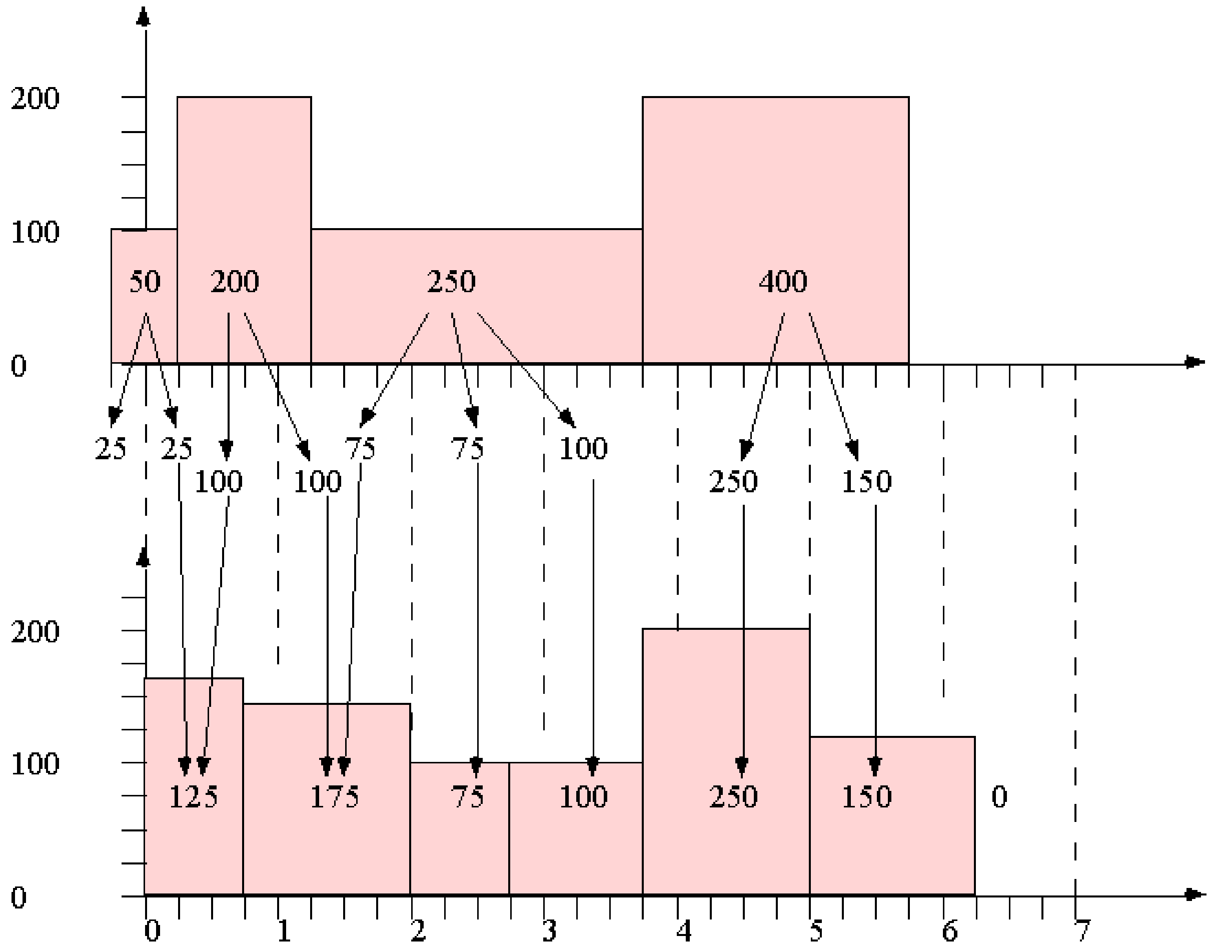

Figure 9 Illustration of how data is rebinned when input and output data are of the *histogram* type.

Rebinning point data

We are given an input dataset comprising n points [X(i), Z(i), dZ(i)], i=0...n-1. We are also given a set of boundaries for n' output channels, X'(j), j=0...n'.

The left and right hand sides of output channel j and its width are L'(j), R'(j) and W'(j) respectively:

$$\left.\begin{aligned} L'(j) &= X'(j), \\ R'(j) &= X'(j+1), \\ W'(j) &= R'(j) - L'(j) \end{aligned}\right\} \quad j = 0 \ldots n'-1 .$$

The integrated counts for output channel j, T'(j), may be written as follows:

$$T'(j) = \sum_i Z(i) \cdot f(i,j)$$

where

$$\begin{aligned} f(i,j) &= 0 \ \text{ if } \ X(i) < X'(j) \ \text{ or } \ X(i) > X'(j+1), \\ &= \tfrac{1}{2} \ \text{ if } \ X(i) = X'(j) \ \text{ or } \ X(i) = X'(j+1), \\ &= 1 \ \text{ if } \ X(i) > X'(j) \ \text{ and } \ X(i) < X'(j+1). \end{aligned}$$

Similarly the error on T'(j) is written as

$$dT'(j) = \sqrt{\sum_i \left[dZ(i)\right]^2 \cdot f(i,j)}$$

since the input data are independent. Hence the output histogram heights and their errors are as follows:

$$\begin{aligned} Z'(j) &= T'(j) / W'(j), \\ dZ'(j) &= dT'(j) / W'(j). \end{aligned}$$

Figure 10 illustrates this for the *point* data case.

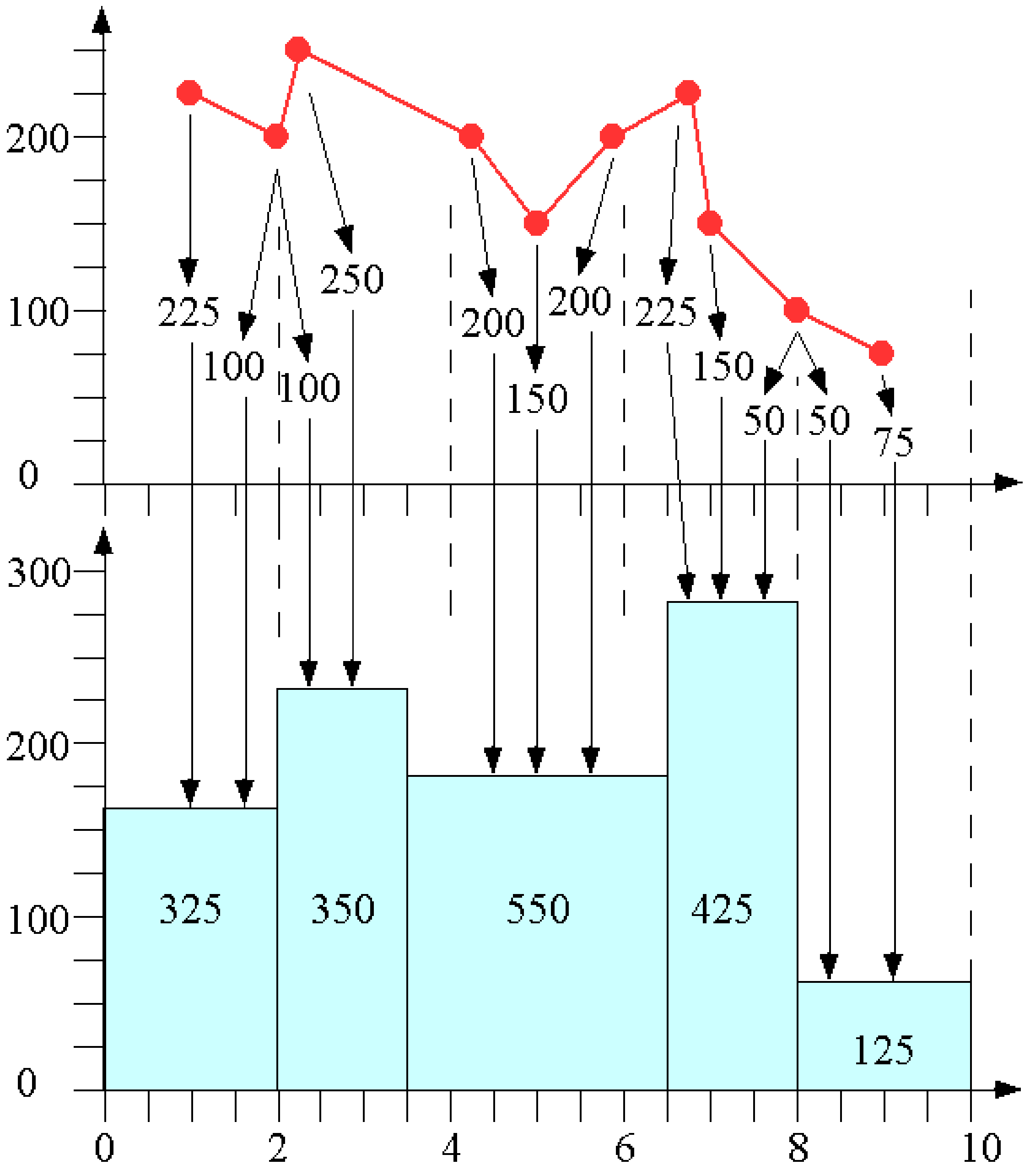

Figure 10 Illustration of how data is rebinned when it is point-type.

IDL Algorithms that rebin *histogram* and *point* data have been written in a manner that use no loops in order to run quickly. As a comparison in speed, the same algorithms have been written using loops. The execution times for each case are considered in the demonstration that follows.

The routine named `DREBIN_HISTO.PRO` is a procedure that uses the rebinning algorithm applied to *histogram* type data. The code that follows has numerous comments throughout and makes heavy use of array operations.

```
;****************************************************
pro drebin_histo,x_in,z_in,dz_in,x_out,z_out,dz_out
;****************************************************
; This procedure rebins 1-dimensional data. The data to
; be rebinned are described by the arrays x_in, which is
; dimensioned n_in+1, and by z_in and dz_in, both of which
; are dimensioned [n_in,ng], where n_in is the number of
; channels and ng is the number of groups, i.e. sets of
; data. The grid onto which the data are to be rebinned is
; defined by the array x_out and the results are placed in
; z_out and dz_out; x_out is dimensioned n_out+1 and z_out
; and dz_out are dimensioned [n_out,ng] where n_out is the
; number of output channels. It is up to the user of this
; procedure to ensure that the input arrays are properly
; dimensioned.
;
compile_opt strictarr
;
; Generate number, left limit, right limit and width
; in x of input channels.
n_in=n_elements(x_in)-1
l_in=x_in[0:n_in-1]
r_in=x_in[1:n_in]
w_in=r_in-l_in
;
; Generate number, left limit, right limit and width
; in x of output channels.
n_out=n_elements(x_out)-1
l_out=x_out[0:n_out-1]
r_out=x_out[1:n_out]
w_out=r_out-l_out
;
; Determine the number of sets of input data.
ng=n_elements(z_in)/n_in
;
; Create unit vectors for later use.
uvec_nout=intarr(n_out)+1
uvec_nin=intarr(n_in)+1
uvec_ng=intarr(ng)+1
;
; Calculate integrated counts and associated errors for
; input channels.
t_in=z_in*(w_in#uvec_ng)
dt_in=dz_in*(w_in#uvec_ng)
;
; The array element (*f_limPtr)[i,j] gives the fraction
; of input channel j that contributes to output channel i.
r_limPtr=ptr_new((r_out#uvec_nin) < (uvec_nout#r_in),/no_copy)
l_limPtr=ptr_new((l_out#uvec_nin) > (uvec_nout#l_in),/no_copy)
d_limPtr=ptr_new((*r_limPtr-*l_limPtr) > 0,/no_copy)
f_limPtr=ptr_new(*d_limPtr/(uvec_nout#w_in),/no_copy)
;
; Do the actual rebinning.
; Integrated counts and associated errors are computed
; for all output channels.
t_out=(*f_limPtr)#t_in
```

```
dt_out=sqrt((*f_limPtr)#dt_in^2)
;
; Hence count rate and associated error for output channels.
z_out=t_out/(w_out#uvec_ng)
dz_out=dt_out/(w_out#uvec_ng)
;
; Free some pointers.
ptr_free,r_limPtr
ptr_free,l_limPtr
ptr_free,d_limPtr
ptr_free,f_limPtr
;
end
```

The routine named `DREBIN_PTS.PRO` is a procedure that rebins *point* type data and the result is *histogram* type data.

```
;****************************************************
pro drebin_pts,x_in,z_in,dz_in,x_out,z_out,dz_out
;****************************************************
; This procedure rebins 1-dimensional ***POINTS*** data.
; The data are described by the arrays x_in, which is
; dimensioned n_in, and by z_in and dz_in, both of which
; are dimensioned [n_in,ng], where n_in is the number of
; channels and ng is the number of groups, i.e. sets of
; data. The grid onto which the data are to be rebinned
; is defined by the array x_out and the results are placed
; in z_out and dz_out; x_out is dimensioned n_out+1 and z_out
; and dz_out are dimensioned [n_out,ng] where n_out is the
; number of output channels. It is up to the user of this
; procedure to ensure that the input arrays are properly dimensioned.
;
compile_opt strictarr
;
; Generate number of input channels.
n_in=n_elements(x_in)
;
; Generate number, left limit, right limit and width in x of output
; channels.
n_out=n_elements(x_out)-1
l_out=x_out[0:n_out-1]
r_out=x_out[1:n_out]
w_out=r_out-l_out
;
; Determine the number of sets of input data.
ng=n_elements(z_in)/n_in
;
; Create unit vectors for later use.
uvec_nout=intarr(n_out)+1
uvec_nin=intarr(n_in)+1
uvec_ng=intarr(ng)+1
;
; The array element (*f_limPtr)[i,j] is 1 if input value j
; contributes to output channel i, otherwise 0, except that
; if input value j lines up with the boundary
; between output channels i and i+1,
; (*f_limPtr)[i,j]=0.5 and (*f_limPtr)[i+1,j]=0.5.
lPtr=ptr_new(l_out#uvec_nin,/no_copy)
rPtr=ptr_new(r_out#uvec_nin,/no_copy)
xPtr=ptr_new(uvec_nout#x_in,/no_copy)
;
l1_Ptr=ptr_new(*lPtr lt *xPtr,/no_copy)
r1_Ptr=ptr_new(*xPtr lt *rPtr,/no_copy)
f1_Ptr=ptr_new(*l1_Ptr and *r1_Ptr,/no_copy)
;
l2_Ptr=ptr_new(*lPtr eq *xPtr,/no_copy)
r2_Ptr=ptr_new(*xPtr eq *rPtr,/no_copy)
f2_Ptr=ptr_new(*l2_Ptr + *r2_Ptr,/no_copy)
;
ff_Ptr=ptr_new(*f1_Ptr + 0.5 * *f2_Ptr,/no_copy)
;
; Free some pointers.
ptr_free,lPtr
ptr_free,rPtr
ptr_free,xPtr
;
ptr_free,l1_Ptr
ptr_free,r1_Ptr
ptr_free,f1_Ptr
;
ptr_free,l2_Ptr
ptr_free,r2_Ptr
ptr_free,f2_Ptr
;
; Do the actual rebinning.
; Integrated counts and associated errors are computed for all output
; channels.
t_out=(*ff_Ptr)#z_in
dt_out=sqrt((*ff_Ptr)#dz_in^2)
;
; Hence count rate and associated error for output channels.
z_out=t_out/(w_out#uvec_ng)
dz_out=dt_out/(w_out#uvec_ng)
;
; Free the remaining pointer.
ptr_free,ff_Ptr
;
end
```

The same rebinning algorithm can be used without array operations and implemented using loops. This is listed in the following code box.

```
pro loop_rebin,x_in,z_in,dz_in,x_out,z_out,dz_out
zsize = size(z_in)
if (zsize[0] eq 2) then begin
   ng = zsize[2]
endif else begin
   ng = 1
endelse
;
nx_in = n_elements(x_in)
nx_out = n_elements(x_out)
```

```
nb_in=nx_in-1
nb_out=nx_out-1
;
z_out = fltarr(nb_out,ng)
dz_out = fltarr(nb_out,ng)
;
for k = 0L,ng-1 do begin
  for i = 0L,nb_out-1 do begin
   c=x_out[i]
   d=x_out[i+1]
   for j = 0L,nb_in-1 do begin
      a=x_in[j]
      b=x_in[j+1]
      if (d gt a and c lt b) then begin
        znorm=z_in[j,k]/(d-c)
        if (d ge b and c lt a) then begin
         z_out[i,k] = z_out[i,k] + znorm*(b-a)
        endif
        if (d ge b and c ge a) then begin
         z_out[i,k] = z_out[i,k] + znorm*(b-c)
        endif
        if (d lt b and c lt a) then begin
         z_out[i,k] = z_out[i,k] + znorm*(d-a)
        endif
        if (d lt b and c ge a) then begin
         z_out[i,k] = z_out[i,k] + znorm*(d-c)
        endif
      endif
   endfor
  endfor
endfor
end
```

In order to test the speed of the array operations as compared with looping, some two-dimensional data with a size of 452 x 7500 was rebinned. The loop algorithm took 299 seconds and the DREBIN algorithm took 3.14 seconds…a savings of nearly two orders of magnitude! The advantage of using an array-based rebinning algorithm is clear.

Convolution redux

In the previous chapter we discussed the convolution operation and the various methods that one can use to estimate the convolution integral. One particular case that we did not consider was the case where the data ranges for the input array and resolution function do not overlap nor are they on an equally-spaced grid. In the cases presented in the previous chapter, the algorithms to calculate the convolution product each assumed that the data was equally spaced and the ranges overlapped. In this section we will illustrate how one can use array operations to calculate the convolution integral. It should be noted that the technique presented here is not nearly as fast as the `FFT` or `CONVOL` techniques for data that is equally-spaced. However this is the only method you can use when this condition is not satisfied.

In order to compute an approximation to the integral in equation (3.3) when the bin sizes vary we consider a representation of the continous integral function

$$
\begin{aligned}
I(x) &= R(x) \otimes y(x) \\
&= \int_{-\infty}^{\infty} d\tilde{x}\, R(\tilde{x})\, y(x-\tilde{x})
\end{aligned}
\tag{4.1}
$$

by the following discrete approximation

$$
\begin{aligned}
I(x_i) &= \sum_j \Delta_j R(x_j) y(x_{i-j}) \\
&= \sum_j \Delta_j R_j y_{i-j}
\end{aligned}
\quad . \tag{4.2}
$$

In 4.2 the transformation from `R(x`$_j$`)` to `R`$_j$ should be obvious. `R`$_j$ is the data point located at `x`$_j$. However evaluating `y(x`$_{i-j}$`)` to get `y`$_{i-j}$ might not be obvious. In fact `y` might not be even evaluated at `x`$_{i-j}$ (or included in the "data"). But if we can calculate `x`$_{i-j}$ then we should be able to use interpolation to find `y(x`$_{i-j}$`)`. In the section title Eliminating a double loop we saw how to create a matrix of differences using matrix multiplication if `x`$_i$ and `x`$_j$ are sampled from the same vector, `x`. Please take another look at that example to refresh your memory if necessary. In the present case the two independent variables are not necessarily sampled in the same way. Therefore we will label them as `x`$_r$ and `x`. It is possible to expand a vector, `x`, that is `n`$_x$ in length to one that is `n`$_x$ by `n`$_r$ in dimension by executing the following command:

```
IDL> xmat = x#(1+bytarr(nr))
```

Alternatively we can use the built-in IDL `REBIN` function as follows to get the same result:

```
IDL> xmat = rebin(x,nx,nr,/sample)
```

If we wish to create an `n`$_x$ by `n`$_r$ representation of `x`$_r$ then we can execute either of the following equivalent commands:

```
IDL> xmat = (1+bytarr(nx))#xr
```

or

```
IDL> xmat = rebin(transpose(xr),nx,nr,/sample)
```

After the difference matrix, , has been calculated then we interpolate the values onto the new grid. This is done with the IDL `INTERPOL` function. Finally the TOTAL function and DERIV function are used to obtain the final approximation to the convolution integral. The source code implementation of this is shown in the box below.

```
function irreg_convol,xr,r,x,y
nxres = n_elements(xr) & nx = n_elements(x)
xmat = rebin(x,nx,nxres,/sample)-rebin(transpose(xr),nx,nxres,/sample)
mat = interpol(y,x,temporary(xmat))
return,total((rebin(transpose(r*deriv(xr)),nx,nxres,/sample))* $
   ((temporary(mat))),2)
end
```

We can see how this works with the following example, `irreg_convol_example.pro`.

```
pro irreg_convol_example
; Create an irregular grid of values for the independent
; variable for the "resolution function"
xrlo = -3.0 & xrhi = 1.0 & nr = 50
xres = xrlo+(xrhi-xrlo)*randomu(s,nr)
; Sort the values since adding the random component
; might shift the order of the points.
xres = xres[sort(xres)]

; Create the irregular grid of values for the independent
; variable for the "intrinsic function"
nx = 100 & xlo = -3.0 & xhi = 15.0
x = xlo+(xhi-xlo)*randomu(s,nx)
; Sort these ones for the same reason that we
; sorted the resolution points.
x = x[sort(x)]

cen = 2.0 & cenr = 0.0
fwhm = 2.0 & fwhmr = 1.0
area = 2.0 & arear = 1.0

; Create the resolution function
res = gaussian(xr,area = arear,cen = cenr,fwhm = fwhmr)
; Create the intrinsic function
y = gaussian(x,area = area,cen = cen,fwhm = fwhm)

; Use the irreg_convol function to perform the
; convolution approximation.
con = irreg_convol(xr,res,x,y)

plot,x,con,psym = 4,xrange = [-5,7],/xstyle

; Calculate and plot the theoretical convolution result
xth = makepts(xlo = -5.0,xhi = 5.0,npts = 200)
yth = gaussian(xth,  area = area*arear,center = cen+cenr,   $
                     fwhm = sqrt(fwhm^2+fwhmr^2))
oplot,xth,yth,psym = 0,thick = 2.0
print,'Calculated integrated intensity: ',int_tabulated(x,con)
print,'Area (theory): ',area*arear
end
```

The graphical output from this program is shown in **Figure 11** and the textual output is

```
Calculated integrated intensity:        1.9951632
Area (theory):       2.00000
```

It is clear that this is a fairly good approximation in terms of the integrated area and you can see that the differences with the theoretical curve are minor in **Figure 11**.

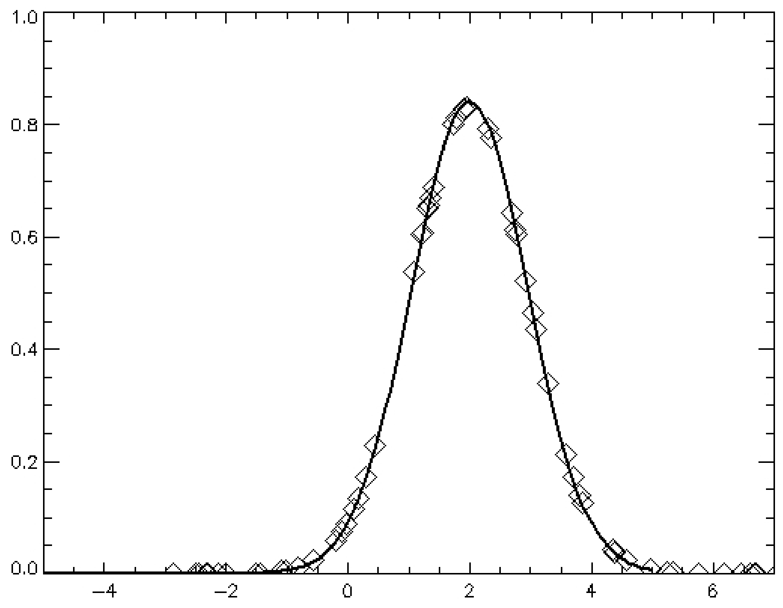

Figure 11 Plot of the numerical convolution shown with the diamond symbols and the theoretical convolution shown as a solid line. Plot created using `irreg_convol_example.pro`.

Exercise: Write a procedure that writes out (i.e. saves to a file) synthetic 2-dimensional scattering data that has the following functional form:

$$S_{meas}(Q,\omega) = R(\omega) \otimes \left[A_o(Q;r)\delta(\omega) + (1 - A_o(Q;r)) \frac{\Gamma}{\pi} \frac{1}{(\omega^2 + \Gamma^2)} \right] \tag{3.6}$$

where $A_o(Q;r) \equiv \frac{1}{3}\left[1 + j_o\left(Qr\sqrt{3}\right)\right]$, the Lorentzian has the same FWHM as in the `make_fake_data` program from the last chapter, the resolution function is Gaussian with FWHM = 1, and r = 1.1 (radius of quasi-elastic hopping). This expression (3.6) is the functional form for a quasi-elastic neutron scattering lineshape for a molecular rotor which hops between three equivalent sites on a circle. Note that IDL has the cylindrical Bessel functions built in. The Bessel function in A_o is a spherical Bessel function. The relationship between the cylindrical and spherical Bessel functions is given by the following IDL function:

```
     function sph_bessel,x,n
     return,(sqrt(0.5*!dpi/x))*beselj(x,n+0.5,/double)
     end
```

Select an ω range with 150 points that spans -10 to 10 and a Q range with 25 points that spans 0.1 to 3.5.

This data file will be essential when we write our final application.

Chapter 5-Data visualization

In neutron scattering, as with many other fields, one often wants to view the scattering data in some meaningful way. Depending on the nature of the measurement and the needs of the experimentalist, it is often desirable to view the data in many different ways. For instance, some like to view quasielastic neutron scattering data on a logarithmic intensity scale in order to gain an appreciation for the "wings" of the quasielastic component and compare these wings to the extents of the elastic component. A strictly inelastic excitation or a Bragg peak is often viewed on a linear scale. Additionally experimentalists often like to look at overplots of multiple data sets. Moreover when the data is parametrized in terms of two independent variables it is often convenient to view the data as a surface or in an image representation. In this chapter we will discuss how to view data in these different ways.

There are two graphics systems with which you can display plots: direct graphics and object graphics. In this course we will be dealing exclusively with the direct graphics system.

Simple plots

In the last chapter we used the `PLOT` procedure on a number of occasions without providing details on its syntax or capabilities. In this section we discuss how to use this procedure with successively more complicated requirements.

The main challenge with the `PLOT` command is that there are a great many optional keywords associated with it. However IDL does a pretty good job at formatting data so that its default appearance is nice. We will not investigate all of the optional keywords to `PLOT`. The reader is referred to the on-line documentation for a complete description. To begin let's consider a plot of the zeroth order cylindrical Bessel function, $j_0(10x)$. This is a function built-in to IDL called `BESELJ`.

```
IDL> x = makepts(xlo = 0.0,xhi = 1.0,npts = 100)
IDL> plot,x,beselj(10.0*x,0),psym = 0,thick = 2.0
```

This result is shown in the plot in Figure 12. The `PSYM` keyword states which symbol is to be used for plotting. `PSYM = 0` tells IDL to use a line for plotting. The `THICK = 2.0` keyword tells IDL to make the line thicker than the default (1.0).

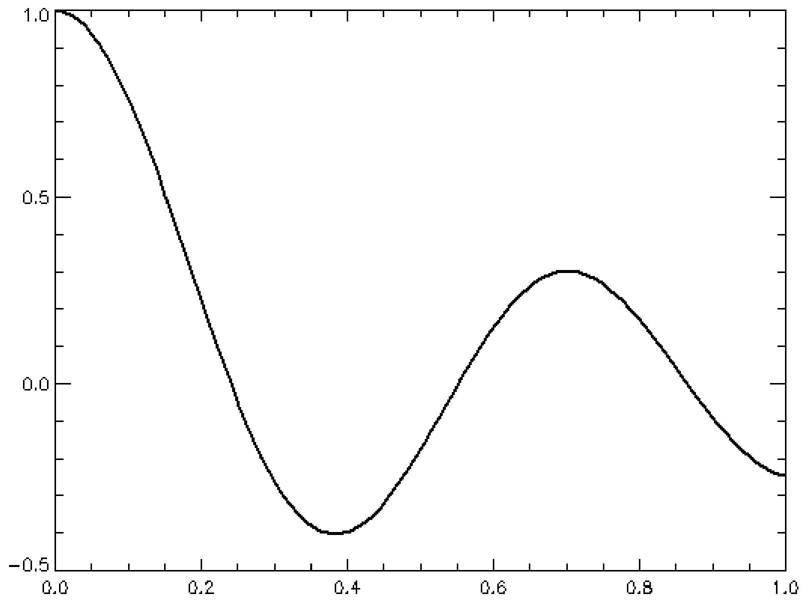

Figure 12 Plot of the zeroth order cylindrical Bessel function.

We can add a title, a label for the x-axis, a label for the y-axis using the keywords `title`, `xtitle`, and `ytitle` respectively as follows:

```
IDL>  plot,x,beselj(10.0*x,0),psym = 0,thick = 2.0,   $
      xtitle = 'x',ytitle = 'j!d0!n(10x)',            $
      title = 'Cylindrical Bessel Functions'
```

In this command we have used some text formatting characters in the ytitle keyword. The formatting character `!d` tells IDL to make the subsequent characters subscripts. The formatting character `!n` tells IDL to make the subsequent characters normal font level. Therefore in the ytitle keyword, 0 should be a subscript and everything else should be a the normal level. Note that `!e` (not used here) tells IDL to make the subsequent characters superscripts. The result of issuing the PLOT command above with all of those keywords is shown in Figure 13 below.

We can overplot different data sets using different line styles to differentiate between the data. As an example, we can type the following `OPLOT` commands to overplot on the existing plot.

```
IDL> oplot,x,beselj(10.0*x,1),linestyle = 1,thick = 2.0
IDL> oplot,x,beselj(10.0*x,2),linestyle = 2,thick = 2.0
IDL> oplot,x,beselj(10.0*x,3),linestyle = 3,thick = 2.0
```

The result is shown in Figure 14. The linestyle keyword yields the following results for different values: 0-solid, 1-dotted, 2-dashed, 3-dash dot, 4-dash dot dot, and 5-long dashes.

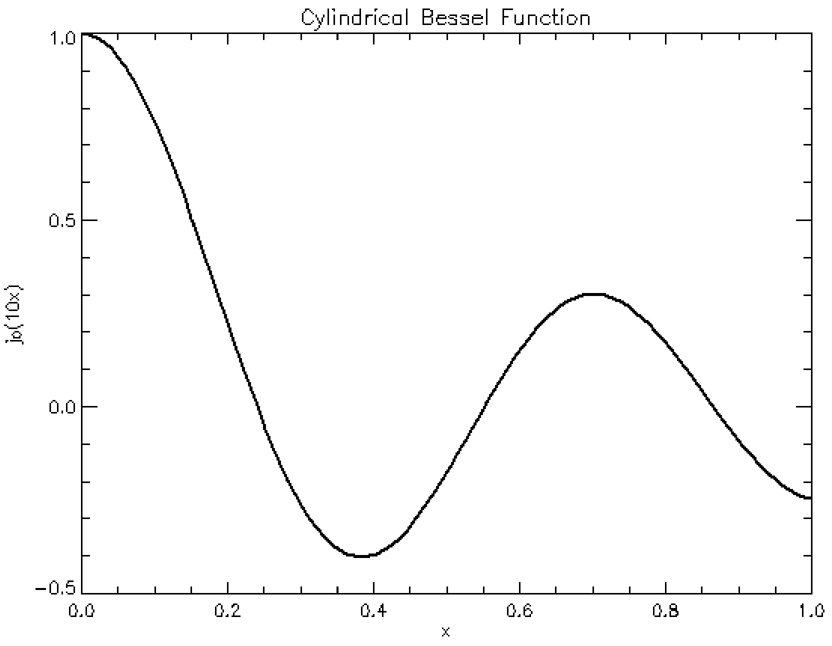

Figure 13 Result of the issuing the PLOT command with the axis annotation keywords set.

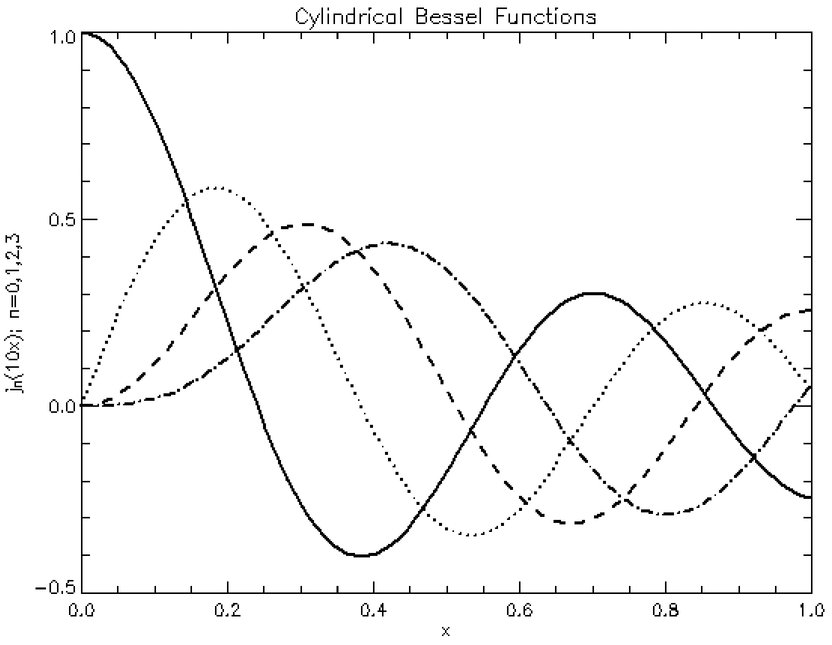

Figure 14 Same as in **Figure 13** except three OPLOT commands are issued to overplot three more data sets.

We can also create an inset within an existing figure. This requires a little bit more effort regarding the keywords. For example, let's say we wish to show a magnified view of a particular range in an inset for Figure 14. We first need to make sure that there is some space available within the axes that does not overlap any of the data that is displayed. We can expand the region above the data shown in Figure 14 using the YRANGE and YSTYLE keywords as follows.

```
IDL>  plot,x,beselj(10.0*x,0),psym = 0,thick = 2.0,      $
      xtitle = 'x',ytitle = 'j!d0!n(10x)',               $
      title = 'Cylindrical Bessel Function with Inset',  $
      yrange = [-0.5,2.0],/ystyle
```

The YRANGE keyword tells the PLOT command to plot only over the range from -0.5 up to 2.0. Setting the YSTYLE keyword forces the PLOT command to use the range specified in YRANGE. For completeness we repeat the next three OPLOT commands.

```
IDL> oplot,x,beselj(10.0*x,1),linestyle = 1,thick = 2.0
IDL> oplot,x,beselj(10.0*x,2),linestyle = 2,thick = 2.0
IDL> oplot,x,beselj(10.0*x,3),linestyle = 3,thick = 2.0
```

Next we must determine where the inset will go. There is a POSITION keyword in PLOT that specifies where in the plot window the plot will be drawn. This keyword is specified in normalized units in the following way: position = [xo,yo,x1,y1] where (xo,yo) define the coordinates for the lower left hand corner and (x1,y1) defines the upper right hand corner. We specify the position for this plot as follows:

```
IDL> pos = [0.5,0.5,0.9,0.9]
```

Finally we issue the PLOT command for the inset with the NOERASE and POSITION keywords set:

```
IDL>  plot,x,beselj(10.0*x,0),psym = 0,/noerase,position = pos,   $
      xrange = [0.5,0.75],/xstyle,thick = 4.0,                    $
      xtitle = 'x',ytitle = 'j!d0!n(10x)'
```

Setting the NOERASE keyword tells IDL not to erase anything that is already displayed on the current graphics device (the window in this case). The result is shown in Figure 15.

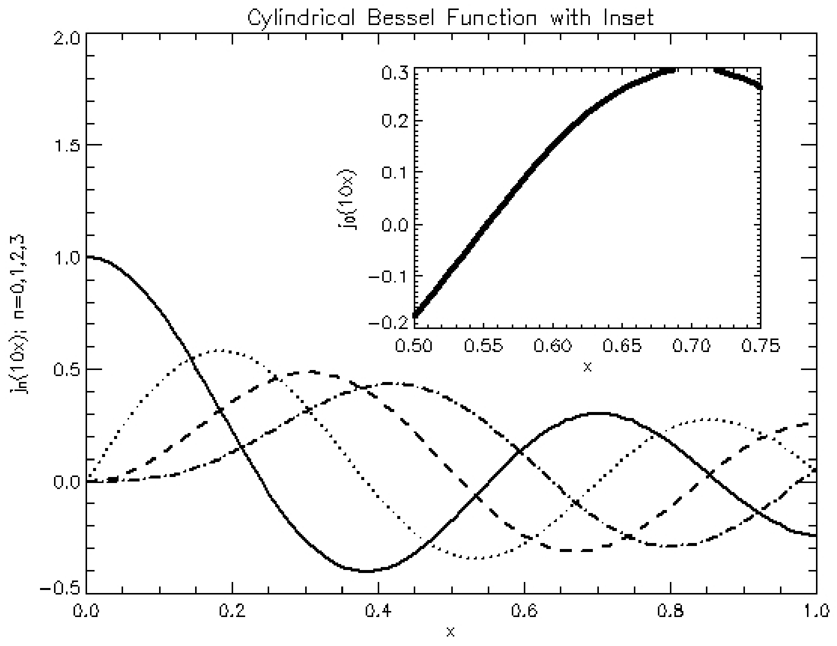

Figure 15 Same as Figure 14 except with an inset.

It is possible to display data using symbols as well as lines. In IDL the symbols are specified by the PSYM keyword and are defined as follows: 1-plus, 2-asterisk, 3-period, 4-diamond, 5-triangle, 6-square, 7-x, 8-user defined. The following example illustrates how we can plot a series of shifted sinusoids over one period using different symbols. When displaying plots that have different symbols and/or linestyles, it is useful to display a plot legend. Unfortunately there is no RSI-supplied routine that creates a plot legend in the direct graphics system. However there is a procedure called `MYLEGEND.PRO` which was written by a third party and is in the suite of programs included in this course distribution. For the sake of clarity, these commands have been formulated as a procedure.

Example:

Create a plot of shifted sinusoids using different symbols. This plot should also include a legend using the procedure MYLEGEND.PRO.

```
pro plot_example

x = makepts(xlo = 0.0,xhi = 3.0*!pi,npts = 100)
phase = makepts(xlo = 0.0,xhi = 1.0*!pi,npts = 8)

; Legend parameters
xpos = 0.5 & ypos = 0.9
charsize = 1.5
; Plot the shifted sinusoids
plot,x,sin(x+phase[0]),psym = 1,       $
   xtitle = 'x',ytitle = 'y(x)',       $
   title = 'Shifted sinusoids',        $
   xrange = [min(x),max(x)],/xstyle,   $
   yrange = [-1.0,3.0],/ystyle
   legend_text = strtrim(string(1),2)+'!est!n data set'

; Add the legend text for this curve
   mylegend,   xpos,ypos,              $
               legend_text,            $
               psym = 1,               $
               charsize = charsize

for i = 1,n_elements(phase)-2 do begin
   oplot,x,sin(x+phase[i]),psym = i+1
   legend_text = strtrim(string(i),2)+'!eth!n data set'
;  Add the legend text for this curve
   mylegend,   xpos,ypos-0.05*i,       $
               legend_text,            $
               psym = i+1,             $
               charsize = charsize
endfor
end
```

The result of the code above is shown in Figure 16.

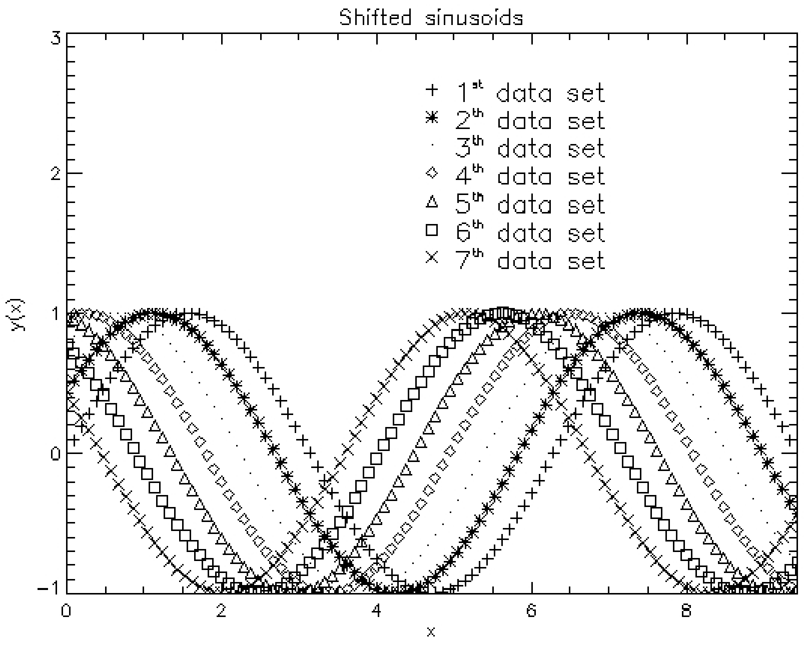

Figure 16 Example showing the various symbols available for plotting.

It is also possible to create user-defined symbols using the built-in IDL procedure called `USERSYM`. An example of how one can use this procedure to create a filled circle symbol is illustrated below.

```
IDL>  x = makepts(xlo = 0.0,xhi = 3.0*!pi,npts = 100)
IDL>  ntheta = 20
IDL>  theta = makepts(xlo = 0.0,xhi = 2.0*!pi,npts = ntheta)
IDL>  xc = cos(theta) & yc = sin(theta)
IDL>  usersym,xc,yc,/fill
IDL>  plot,x,sin(x),psym = 8,                         $
      xtitle = 'x',ytitle = 'y(x)',                   $
      title = 'Sinusoid with filled circle symbol'
```

The `USERSYM` procedure creates a symbol that is accessed by setting `PSYM = 8` in the `PLOT` call. The result of issuing the commands above is shown in Figure 17.

As a final example, we will show to create multiple plots on a single page. This can be done using the `NOERASE` keyword as described previously or, if the layout is grid-like, it is simpler to use a system variable called `!P.MULTI`. For example, if we wish to have a 2 × 2 grid of plots in one window then we would do so as follows.

```
IDL>  x = makepts(xlo = 0.0,xhi = 3.0*!pi,npts = 100)
IDL>  phase = makepts(xlo = 0.0,xhi = 1.0*!pi,npts = 8)
IDL>  !p.multi = [0,2,2]
IDL>  for i = 0,3 do $
IDL>  plot,x,sin(x+phase[i]),psym = i+4,  $
IDL>  xtitle = 'x',ytitle = 'y(x)',       $
IDL>  title = 'Shifted sinusoid',         $
IDL>  xrange = [min(x),max(x)],/xstyle
IDL>  !p.multi = 0
```

Note that we issue the final `!P.MULTI=0` command so that subsequent plots will not be arranged in a grid. Rather it will return to the default behavior of placing a single plot on the window. The result is shown in Figure 18.

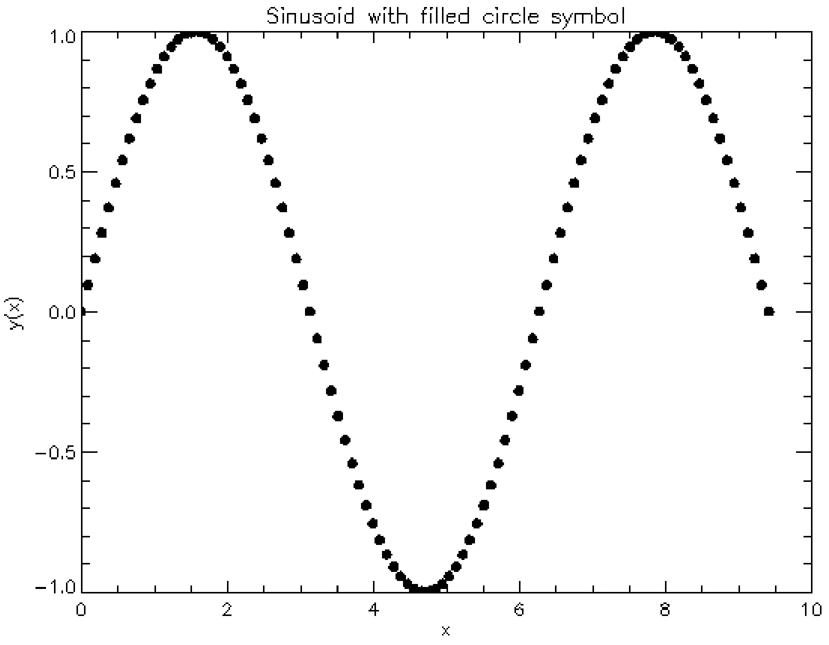

Figure 17 Illustration of how to implement a user-defined plotting symbol.

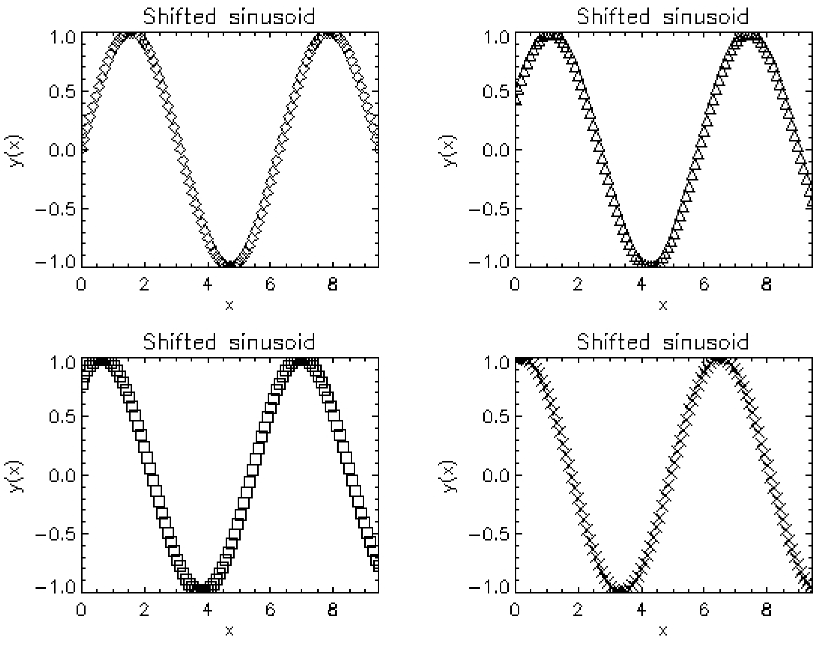

Figure 18 Illustration of multiple plots in a window using the `!P.MULTI` system variable.

The syntax for the !P.MULTI system variable is as follows:

- `!P.MULTI[0]` = number of plots remaining on the page
- `!P.MULTI[1]` = number of plot columns per page
- `!P.MULTI[2]` = number of plot rows per page
- `!P.MULTI[3]` = number of plots stacked in the z-dimension

Surface and image plots

Data that is a function of two independent variables can be visualized using contour, image, and/or surface plots. Many excellent texts exist that show how to display data in any of these forms so we will not go into great detail how to do these things. Instead we will show you how to create a false-color image plot of some data using a routine called `PLOTIMAGE` (developed by a third party) and briefly show you how to create a surface plot with a texture map.

`PLOTIMAGE` is a flexible image display routine that allows the user to plot an image representation of data with meaningful data axes surrounding the image. Note that this is not possible with the `TV` and `TVSCL` routines built in to IDL (without modification). The only requirements are (1) that the quantity to be displayed must be two-dimensional, (2) it must be regularly gridded, and (3) it must by scaled as a byte array from 0B to 255B.

As an example we can use the `PLOTIMAGE` procedure to display a two-dimensional function defined over the x-range of [-10,10] and over the y-range of [0,5]. The code below, `plotimage_example.pro` is the name of the example. The `IMGXRANGE` and `IMGYRANGE` keywords allow the user to specify the range over which the image is defined. In particular `IMGXRANGE[0]` specifies the lower boundary on the lowest bin and `IMGXRANGE[1]` specifies the upper boundary on the highest bin. We want to display this data so that each pixel is bin-centered. To do this we subtract off half a binwidth from lower range bound and add half a binwidth to the upper range bound.

```
pro plotimage_example
; Set up the colors for this example
device,decomposed = 0
loadct,15,/silent
; Specify the ranges of the function
xlo = -10.0 & xhi = 10.0 & nx = 101
ylo = 0.0 & yhi = 5.0 & ny = 51
x = makepts(xlo = xlo,xhi = xhi,npts = nx)
y = makepts(xlo = ylo,xhi = yhi,npts = ny)
; Create the function to be displayed
z = sin(x#y)+0.1*randomn(s,nx,ny)
; Convert the function into a byte-scaled image
image = bytscl(z)
; PLOTIMAGE requires that we specify the range
; over which to plot the data.  In order to ensure
; a "bin-centered" plot for each pixel we subtract
; off half a bin from the lower bound and add
```

```
      ; half a bin to the upper bound.
      dx = x[1] - x[0]
      dy = y[1] - y[0]
      imgxrange = [xlo-0.5*dx,xhi+0.5*dx]
      imgyrange = [ylo-0.5*dy,yhi+0.5*dy]

      ; Open the window
      window,0
      erase,255B  ; erase the window with white
      plotimage,image,imgxrange = imgxrange, $
               imgyrange = imgyrange,/noerase,color = 0
end
```

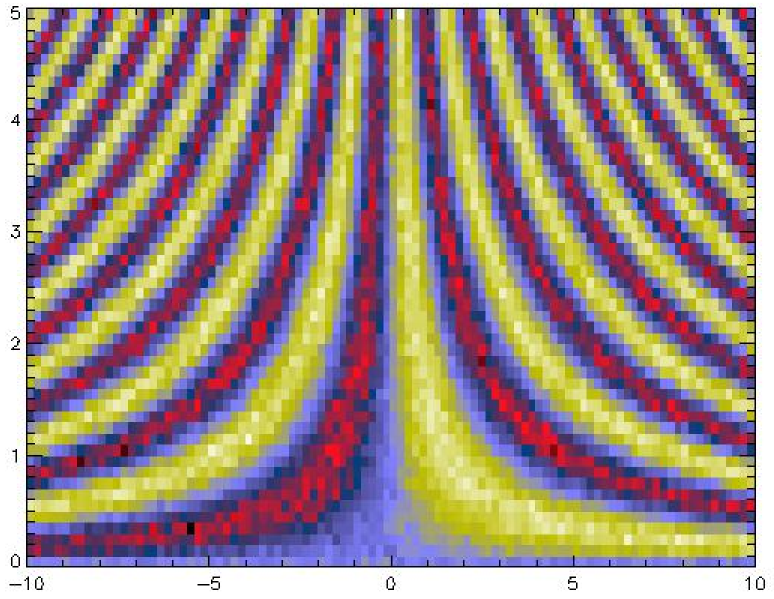

Figure 19 Output from the `plotimage_example.pro` program illustrating the use of the `PLOTIMAGE` procedure.

We can modify this program slightly to allow us to view a surface representation of this function as follows.

```
pro shade_surf_example
; Set up the colors for this example
device,decomposed = 0
loadct,15,/silent
; Specify the ranges of the function
xlo = -10.0 & xhi = 10.0 & nx = 100
ylo = 0.0 & yhi = 5.0 & ny = 50
x = makepts(xlo = xlo,xhi = xhi,npts = nx)
y = makepts(xlo = ylo,xhi = yhi,npts = ny)
; Create the function to be displayed
z = sin(x#y)+0.1*randomn(s,nx,ny)
shade_surf,z,x,y,shades = bytscl(z),ax  = 75.
end
```

In this example we used SHADE_SURF which has the following syntax:

```
SHADE_SURF, Z [, X, Y] [, AX=degrees] [, AZ=degrees] [, IMAGE=variable]
[, MAX_VALUE=value] [, MIN_VALUE=value] [, PIXELS=pixels] [, /SAVE]
[, SHADES=array] [, /XLOG] [, /YLOG]
```

In the example above we specified `z`, `x`, and `y`. Also we used the shades keyword to "drape" the image represenation of the surface on the surface itself. You can also change the orientation of the surface by specifying the `AX` and/or `AZ` keywords which represent the axial and azimuthal orientation, respectively.

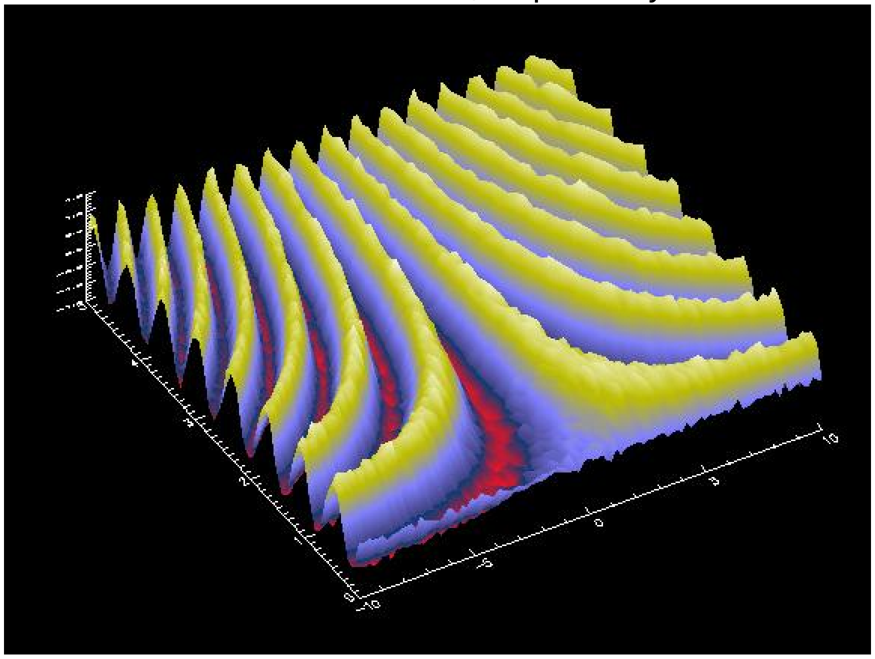

Figure 20 Output from the `shade_surf_example.pro` program illustrating the use of the `SHADE_SURF` procedure.

Animating data

Often it is illustrative to display a sequence of data as a function of some parameter such as time, concentration, etc. In these cases it is useful to animate the data sequence. This is possible in IDL by repeated updating of the plot window. However it is not as straightforward as just re-issuing a new plot command for each new frame. We must invoke the notion of a *pixmap*. A pixmap is a display device that exists in memory that is not rendered on the screen directly but is copied to the screen from memory using a technique called *double-buffering*.

This technique of double buffering is best illustrated with an example. Consider the code listed below.

```
IDL>  x = makepts(xlo = -5.0,xhi = 5.0,npts = 100)
IDL>  gauss = gaussian(x)
IDL>  window,0,xsize = 400,ysize = 400 ; create visible window
IDL>  window,/free,/pixmap,xsize = 400,ysize = 400 ; create the pixmap
IDL>  winpix = !d.window
IDL>  wset,winpix ; first plot it in the pixmap
IDL>  plot,x,gauss,thick = 2.0
IDL>  wset,0 ; now copy it from the pixmap to the visible window
IDL>  device,copy = [0,0,400,400,0,0,winpix]
IDL>  wdelete,winpix ; free the memory associated with the pixmap
```

In this example we create a visible window (0) and a pixmap (winpix). The plot is first rendered in the pixmap and then copied using the `device` procedure with the `copy` keyword set to an array composed of 7 elements. The first two elements are the x and y coordinates in pixels of the lower left corner of the source window. The third and fourth elements are the dimensions (in pixels) of the source window, the fifth and sixth elements are the coordinates (in pixels) of the lower left corner of the destination window, and the seventh element is the window identifier of the source window. Let's consider a slightly more complicated example that illustrates how this can be used for animating sequences of images.

Example:

Create an animated sequence of a Gaussian distribution translating and broadening with each successive frame. Compare the output to that of simply replotting the new frame on the same window to using double-buffering.

```
pro animate_example

nx = 200 & nframes = 50
x = makepts(xlo = -10.0,xhi = 10.0,npts = nx)
center = makepts(xlo = -10.0,xhi = 10.0,npts = nframes)
fwhm = makepts(xlo = 1.0,xhi = 9.0,npts = nframes)
gauss_mat = fltarr(nx,nframes)
for i = 0,nframes-1 do $
   gauss_mat[*,i] = gaussian(x,center = center[i],fwhm = fwhm[i])

; First do it the "intuitive" way
winvis = 0
window,winvis,xsize = 400,ysize = 400
for i = 0,nframes-1 do begin
   plot,x,gauss_mat[*,i],thick = 2.0,yrange = [0.0,2.0],/ystyle
endfor

; Now do it using double-buffering
window,/free,/pixmap,xsize = 400,ysize = 400
winpix = !d.window

for i = 0,nframes-1 do begin
   wset,winpix
   plot,x,gauss_mat[*,i],thick = 2.0,yrange = [0.0,2.0],/ystyle
   wset,winvis
   device,copy = [0,0,!d.x_size,!d.y_size,0,0,winpix]
endfor
; Note that we must delete the memory associated with the pixmap
wdelete,winpix

end
```

The first method in the example simply re-plots each successive frame of the curve's evolution in the same window. This results in an undesirable "flicker" as the data is displayed in the window. The second method, which uses double-buffering, results in a much smoother animated sequence. It should also be pointed out that the system variable `!D.WINDOW` has been used to get the window identifier for the pixmap. The `WINDOW` procedure to create the pixmap was called with the `FREE` keyword to get any available window identifier (i.e. corresponding to one that is not currently in use).

Example showing convergence of Fourier Series

As a final example of animating data we will illustrate the convergence of a Fourier Series approximation to a function as the number of terms in the sum increases. In particular, we consider a square wave function with the following Fourier Series representation:

$$\frac{1}{2}\left\{1+\frac{4}{\pi}\sum_{n=0}^{\infty}\frac{\sin((2n+1)x)}{2n+1}\right\}. \tag{5.1}$$

We will see that our knowledge of array operations will be useful in creating the summand and the quantity to be plotted without using loops. Of course we will require the main animation loop but we will not use any loops to create the elements in the animated sequence.

As described in the previous section we will create a visible window and a pixmap and then use the double-buffering to create the smooth animated sequence. After creating the windows, the "boxcar" function is created and then the summand is created based on the extents of x and the total number of terms in the series expansion. In the example presented here we choose 500 points in x and 100 terms in the expansion. Our strategy is to create the 2 dimensional summand array (500×100) using array operations and then, within the animation loop, use the `TOTAL` function to sum over the second dimension up to the term for display. For instance in the 40th iteration in the loop, we would want to sum the array over the second dimension up to the 39th term. This is done using the `TOTAL` function whose first argument is the 2 dimensional array and second argument is the dimension over which to sum (2). Once this has been calculated then the result is displayed in the pixmap and copied to the visible window using the double-buffering technique. Finally, after the animated sequence is finished, the integrated absolute value of the difference between the approximation and the actual function is plotted as a function of each successive term. This provides a simple visualization of the convergence of the Fourier Series. The code for this example is listed in the following box.

```
fourierProgram.pro
;
; Demonstration of the Fourier series approximation to
; a rectangle function as the number of terms in the
; series expansion increases.
; **********************************
pro fourierProgram
; Use the DEVICE procedure with the GET_SCREEN_SIZE
; keyword to determine the size (in pixels) of the
; screen.  Then make the window size 1/3 of this in
; both dimensions.
device,get_screen_size = screen_size
winxsize = 400 & winysize = 400
xsize = fix(screen_size[0]/3.0)
ysize = fix(screen_size[1]/3.0)

nx = 500
x = makepts(xlo = -0.5, xhi = 1.5*!pi, npts = nx)
y = heaviside(x)-heaviside(x-!pi) ; the function

nterms = 100 ; # of terms in the FS expansion
; Create the pixmap
window,/free,/pixmap,xsize = winxsize, ysize = winysize
winpix = !d.window
; Create the visible window
window,0,xsize = winxsize,ysize = winysize
winvis = !d.window

; Set up the display to tile one window with
; two plots: the function+approximation and the
; difference plot.
!p.multi = [0,1,2]
t = fltarr(nterms); integrated difference
yterms = fltarr(nx,nterms); summand
ux = 1+bytarr(nx)
integer = indgen(nterms)
yterms = sin(x#(2.0*integer+1))/(ux#(2.0*integer+1))
for i = 0,nterms-1 do begin
   if i lt 2 then begin
      ysum = yterms[*,0]
   endif else begin
      ysum = total(yterms[*,0:i-1],2)
   endelse
   ; Display the window
   wset,winpix
   plot,x,y,yrange = [-0.5,2.0],ystyle = 1,thick = 2.0, $
      title = strtrim(string(i+1),2)+'!3: Terms'
   oplot,x,0.5*(1.0+(4.0/!pi)*ysum),thick = 1.0
   plot,x,y-0.5*(1.0+(4.0/!pi)*ysum),yrange = [-0.25,0.25],  $
      ystyle = 1,xtitle = 'x',ytitle = 'Difference'
   wset,winvis
   device,copy = [0,0,winxsize,winysize,0,0,winPix]
   empty
   ; Integrate the difference
   t[i] = int_tabulated(x,abs(y-0.5*(1.0+(4.0/!pi)*ysum)))
endfor
wdelete,winpix

!p.multi = 0
```

```
window,1,xsize = winxsize,ysize = winysize
plot,t,xtitle = 'x',ytitle = 'Difference'
end
```

Chapter 6-Data Analysis

IDL contains many built-in data analysis features. Among these are curve fitting functions. However over time, other IDL programmers have developed a number of routines that extend the analysis capabilities of IDL. In this chapter we discuss a few of IDL's analysis routines, a set of robust fitting routines called MPFIT developed by a third party, and apply them to a realistic neutron scattering problem. Along the way we will have a brief diversion to see how you can view the intermediate fitting steps using animation techniques presented in Chapter 5-Data visualization, and also discuss a simple Monte-Carlo method for estimating uncertainty in your fit parameters.

Fitting a straight line with LINFIT

In this first section we consider one of the simplest types of data analysis problems: fitting a straight line to data. IDL has a routine called LINFIT that has the following syntax:

```
Result = LINFIT( X, Y [, CHISQ=variable] [, COVAR=variable] [, /DOUBLE]
[, MEASURE_ERRORS=vector] [, PROB=variable] [, SIGMA=variable]
[, YFIT=variable] )
```

The data are defined by the vectors `x` (the independent variable) and `y` (the dependent variable). If the measurement uncertainties are known then they can be specified by setting the keyword `MEASURE_ERRORS`. The goodness-of-fit as represented by the `CHISQ` and `PROB` output keywords. The uncertainty on the fit parameters can be accessed in the output keyword `SIGMA`.

```
pro fit_linear_data
; Create some fake linear data
offset = 1500.0 & slope = -0.25
npts = 100
x = makepts(xlo = -20.0,xhi = 30.0,npts = npts)
yline = offset+slope*x
; "Color" the data with Poisson noise
y = fltarr(npts)
for i = 0,npts-1 do y[i] = randomn(s,1,poisson = yline[i])
dy = sqrt(y)
plot,x,y,psym = 4
errplot,x,y-dy,y+dy,width = 0.0
; Fit it using the IDL LINFIT routine
fit_result = linfit(x, y, measure_errors = dy, yfit = yfit, sigma = sigma)
oplot,x,yfit,psym = 0,thick = 2.0
print,"Fit results"
print,"-----------"
print,'Offset: '+strtrim(string(fit_result[0]),2)+' +/-
'+strtrim(string(sigma[0]),2)
```

```
print,'Slope: '+strtrim(string(fit_result[1]),2)+' +/-
'+strtrim(string(sigma[1]),2)
end
```

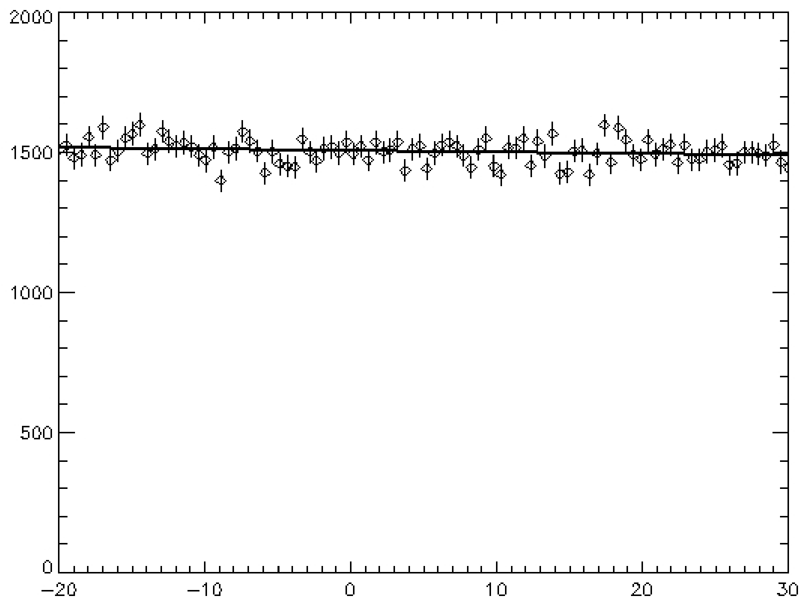

Figure 21 Fit of a straight line to synthesized linear data using IDL's LINFIT routine.

The text output from the program that resulted when I ran the program `fit_linear_data` was the following:

```
IDL> fit_linear_data
Fit results
-----------
Offset: 1504.51 +/- 4.10388
Slope: -0.541843 +/- 0.266056
```

Using CURVEFIT

We can use IDL's general-purpose curve fitting routine appropriately named `CURVEFIT` to fit the synthetic data to a line. This uses a gradient expansion of the fit function to first-order in the fit parameters to convert a non-linear least-squares problem into a linear least-squares problem. The syntax for `CURVEFIT` is as follows:

```
Result = CURVEFIT( X, Y, Weights, A [, Sigma] [, CHISQ=variable] [, /DOUBLE]
[, FITA=vector] [, FUNCTION_NAME=string] [, ITER=variable] [, ITMAX=value]
[, /NODERIVATIVE] [, STATUS={0 | 1 | 2}] [, TOL=value] [, YERROR=variable] )
```

We will only concern ourselves with the `CURVEFIT` arguments that are necessary to fit the straight line to the data. The first two parameters, `x` and `y`, are the independent and dependent data vectors respectively. The weights, used in the calculation of χ^2, are usually chosen as $1/\sigma^2$ where σ is a vector containing the measurement errors for the data. This is appropriate for “normal” measurement errors.

```
pro linear_function,x,p,yfit
; The model function that defines the
; line.
offset = p[0] & slope = p[1]
yfit = offset+slope*x
end

pro curvefit_linear_data
; Create some fake linear data
offset = 1500.0 & slope = -0.25
npts = 100
x = makepts(xlo = -20.0,xhi = 30.0,npts = npts)
yline = offset+slope*x
; "Color" the data with Poisson noise
y = fltarr(npts)
for i = 0,npts-1 do y[i] = randomn(s,1,poisson = yline[i])
dy = sqrt(y)
plot,x,y,psym = 4
errplot,x,y-dy,y+dy,width = 0.0
; Provide initial guess for the fit parameters
p = [100.0,1.0]
; Fit it using the CURVEFIT
yfit = curvefit(x,y,1d/dy^2,p,sigma,/noderivative, $
   function_name = 'linear_function')

fit_result = p

oplot,x,yfit,psym = 0,thick = 2.0
print,"Fit results"
print,"-----------"
print,'Offset: '+strtrim(string(fit_result[0]),2)+ $
   ' +/- '+strtrim(string(sigma[0]),2)
print,'Slope: '+strtrim(string(fit_result[1]),2)+  $
   ' +/- '+strtrim(string(sigma[1]),2)
end
```

Using MPCURVEFIT

One of the problems with IDL's `CURVEFIT` function is that it is impossible to use it to fit a function (or set of functions) that have been convolved with a resolution function. The robust fitting routines written by Craig Markwardt at NASA called `MPFIT` provide a mechanism for fitting data using a resolution function. These routines are an implementation of a variant on the Levenberg-Marquardt least-squares approach with a few modifications. It is an IDL adaptation of the MINPACK-1 software. In this section we will repeat the analysis of the previous section (i.e. fitting a straight line to synthesized linear data) to show how to use `MPCURVEFIT`, one of the routines from `MPFIT`. We will take a closer look at how to incorporate the instrumental resolution function into the fitting process using `MPCURVEFIT` in the next section.

The code below contains two procedures. The first procedure is the actual fit function named `linear_function` and accepts two input parameters, `x` and `p`, and returns a single output parameter, `yfit`. `yfit` is the model function evaluated at the points `x` with parameters `p` (`=[offset,slope]`). The second procedure, `mpcurvefit_linear_data`, is the "driver" program that creates the synthetic data and performs the fit.

The complete documentation for `MPCURVEFIT` can be seen in the header of that file but we will cover only the few keywords and parameters that are useful for the example. In our example the following syntax is used:

```
yfit = mpcurvefit(x,y,1d/dy^2,p,sigma,/noderivative,  $
   function_name = 'linear_function',/quiet)
```

In this function the return value `yfit` is the model function evaluated at `x`, the independent variable, with the final parameters specified in the output parameter, `p`. The data (independent and dependent variables) are the first two input parameters, and the weights for the data is the third input parameter. The weights are used to calculate the χ^2 value for the fit and, for "normal" errors, they are defined as $1/\sigma^2$ where the σ are the measurement uncertainties of the data points. The `noderivative` keyword is set which removes the usual requirement of least-squares routines to provide the analytic derivatives of the model function taken with respect to the parameters. The model function `linear_function` is specified with the `function_name` keyword. Finally, the `quiet` keyword is set to suppress intermediate reporting in the output log. If `quiet` is not set then at each decrease of χ^2, the parameters are printed to the output log. This information can be useful when diagnosing problems with fitting data.

```
pro linear_function,x,p,yfit
; The model function that defines the
; line.
offset = p[0] & slope = p[1]
yfit = offset+slope*x
end

pro mpcurvefit_linear_data
; Create some fake linear data
offset = 1500.0 & slope = -0.25
npts = 100
x = makepts(xlo = -20.0,xhi = 30.0,npts = npts)
yline = offset+slope*x
; "Color" the data with Poisson noise
y = fltarr(npts)
for i = 0,npts-1 do y[i] = randomn(s,1,poisson = yline[i])
dy = sqrt(y)
plot,x,y,psym = 4
errplot,x,y-dy,y+dy,width = 0.0
; Provide initial guess for the fit parameters
p = [100.0,1.0]
```

```
; Fit it using the MPCURVEFIT
yfit = mpcurvefit(x,y,1d/dy^2,p,sigma,/noderivative,  $
   function_name = 'linear_function',/quiet)
fit_result = p

oplot,x,yfit,psym = 0,thick = 2.0
print,"Fit results"
print,"-----------"
print,'Offset: '+strtrim(string(fit_result[0]),2)+ $
   ' +/- '+strtrim(string(sigma[0]),2)
print,'Slope: '+strtrim(string(fit_result[1]),2)+  $
   ' +/- '+strtrim(string(sigma[1]),2)
end
```

When I ran the program I obtained the following text output:

```
IDL> mpcurvefit_linear_data
Fit results
-----------
Offset: 1497.0937 +/- 4.09255
Slope: -0.37265026 +/- 0.265282
```

This was accompanied by the graphical output shown in Figure 22. It is clear that the fit to the data is very good.

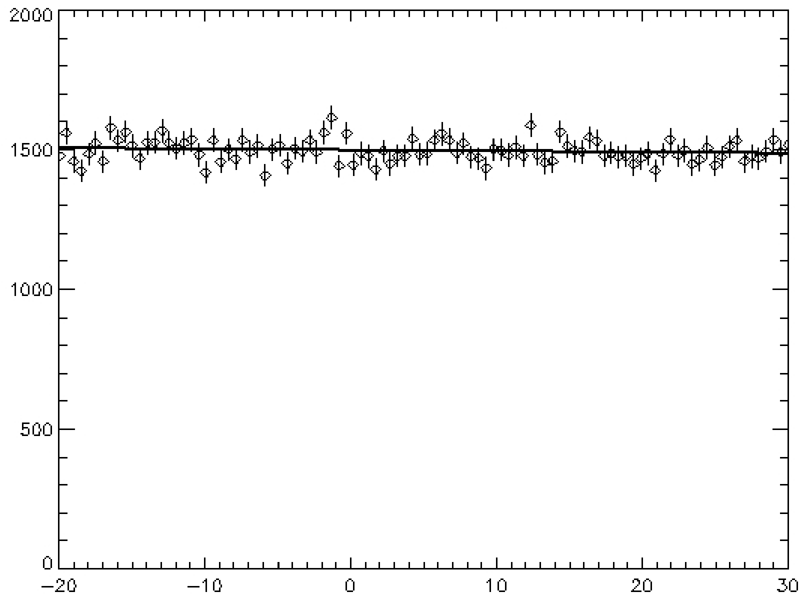

Figure 22 Result of using `MPCURVEFIT` to fit a straight line to synthetic linear data. Note that the results slightly differ from those found using `LINFIT` in the previous example due to differences in the synthetic data.

Analyzing neutron scattering data with MPCURVEFIT

In this section we will illustrate how to fit data using the `MPCURVEFIT` function which includes using convolution of the resolution function with the intrinsic function(s). The synthetic data that we will analyze is that from chapter 3. Recall that in that section we created a Gaussian resolution function and convolved this with an intrinsic function composed of a delta function plus a Lorentzian function. The procedure to create that data provided us with an option to write the “data” out to a file. For this exercise we will read this data in and fit the model function to it (with the instrumental resolution function convolved with the intrinsic function).

The analysis procedure is found in a program called `analyze_fake_data.pro` listed below.

```
; ******************************************** ;
pro scattering_function,x,p,yfit,_Extra = extra
bg = p[0] & area_el = p[1] & cen_el = p[2]
area_iel = p[3] & cen_iel = p[4] & fwhm_iel = p[5]
y_iel = lorentzian(x,area = area_iel,cen = cen_iel,fwhm = fwhm_iel)
nlo = 3 & nhi = n_elements(x) - nlo - 1
dx = x[1] - x[0]
con = convol(*extra.res_ptr,reverse(y_iel[nlo:nhi]),/edge_truncate) * dx
yfit = bg+gaussian(x,area = area_el,cen = cen_el,fwhm = 1.0)+con
end
; ******************************************** ;
pro analyze_fake_data
; First read in the data
dir = 'C:\RSI\dimeo_programs\idl_training\documentation\programs\'
filename = dir+'fake_data.txt'
count = 0L
dum1 = (dum2 = (dum3 = 0.0))
openr,lun,filename,/get_lun
while not eof(lun) do begin
   readf,lun,dum1,dum2,dum3
   if count eq 0L then begin
      x = dum1 & y = dum2 & dy = dum3
   endif else begin
      x = [x,dum1] & y = [y,dum2] & dy = [dy,dum3]
   endelse
   count++
endwhile
free_lun,lun,/force

; Now plot the data with the error bars
plot,x,y,psym = 4
errplot,x,y-dy,y+dy,width = 0.0

; We will assume that we know that the resolution
; function is a Gaussian with a FWHM of 1.0 and is
; defined over the same interval as the data.
res = gaussian(x,area = 1.0,center = 0.0, fwhm = 1.0)
; Create a pointer to the resolution function because we
; will not want to pass around the vector and keep making
; copies of it.  A single reference to a heap variable
; is all that's necessary.
res_ptr = ptr_new(res)
```

```
; Pick some initial guesses for the fit
bg = 10.0 & area_el = 500.0 & cen_el = 0.0
area_iel = 100.0 & cen_iel = 5.0 & fwhm_iel = 1.0
p = [bg,area_el,cen_el,area_iel,cen_iel,fwhm_iel]
; Impose constraints on each of the positive-definite
; parameters that they be greater than zero.
parinfo = replicate({limited:[0,0],limits:[0D,0D]},n_elements(p))
parinfo[0].limited[0] = 1
parinfo[0].limits[0] = 0.D
parinfo[1].limited[0] = 1
parinfo[1].limits[0] = 0.D
parinfo[3].limited[0] = 1
parinfo[3].limits[0] = 0.D
parinfo[5].limited[0] = 1
parinfo[5].limits[0] = 0.D

functargs = {res_ptr:res_ptr }
yfit = mpcurvefit(x,y,1d/dy^2,p, sigma, parinfo = parinfo, $
   function_name = 'scattering_function', $
   /noderivative,functargs = functargs)
oplot,x,yfit,psym = 0,thick = 2.0
for i = 0,n_elements(p)-1 do $
   print,strtrim(string(p[i]),2)+' +/- '+strtrim(string(sigma[i]),2)
ptr_free,res_ptr
end
```

The first procedure in this file is called `scattering_function` which is the model which will be fit to the data. This fit function is similar in syntax to `linear_function`, discussed in the the previous section except that it also accepts keywords. The function `MPCURVEFIT` provides the capability to pass in exogenous information into the fitting function via a keyword named `FUNCTARGS`. In our case we pass in a pointer reference to the resolution function. Why not pass in the resolution function as an array rather than a pointer to it? The Levenberg-Marquardt algorithm requires numerous evaluations of the fit function meaning that the information will be passed between routines very often. If the information is large then this will tend to slow down the fitting process. We can gain a bit of performance by passing a pointer reference rather than the full array. The other difference between `scattering_function` and `linear_function` is that the `CONVOL` routine is used to take into account the effects of instrumental resolution.

The first task in `analyze_fake_data` is to read in the data saved in the file called `fake_data.txt`. The data is organized into three columns: x, y, dy, so we simply need to open up the file, read in three values on each line until we reach the end of the file. This is the subject of the first 15 lines or so of the `analyze_fake_data` procedure. This data is then plotted. Next the resolution function, assumed to be a Gaussian centered at 0.0 with a full-width-at-half-maximum of 1.0, is generated and a pointer reference to it is created. Then initial guesses for the parameters are specified and stored in the parameter `p`. Then an array of structures is created that enable us to impose a lower limit on certain parameters in the fit. For instance, we know that the area of each of the functions must be greater than 0 so a lower limit of zero is imposed on those parameters. Likewise a lower limit of 0 is imposed on the full-width-at-half-maximum for the Lorentzian is imposed. Next, the structure of exogenous function information is created called FUNCTARGS and the pointer to the resolution function is put into it. Then the MPCURVEFIT function is called with the following syntax:

```
yfit = mpcurvefit(x,y,1d/dy^2,p, sigma, parinfo = parinfo, $
   function_name = 'scattering_function', $
   /noderivative,functargs = functargs)
```

Here the constraint information embodied in `parinfo` is passed in as an input keyword, as is `functargs`. The `noderivative` keyword is also set because we have not supplied the analytic derivatives of the fit function. Otherwise the syntax does not differ from that of CURVEFIT very much. Then the resulting fit is plotted on top of the original data and the parameter estimates and their uncertainties are printed to the output log. Finally the pointers are freed to avoid memory leaks.

The parameter estimates from the analysis are displayed in the following table along with the actual values used in the synthesis. The estimated values are very close to the actual values.

Parameter name	**Fit value/uncertainty**	**Actual value**
Background	19.141001 +/- 0.439675	20.0
Elastic peak area	588.62901 +/- 8.03496	600.0
Elastic peak center	-0.00288 +/- 0.00626	0.0
Inelastic peak area	304.65080 +/- 8.57177	300.0
Inelastic peak center	5.0082212 +/- 0.0215845	5.0
Inelastic peak width	1.0527218 +/- 0.0603475	1.0

The graphical output is shown in the figure below. It is clear that the fit is excellent.

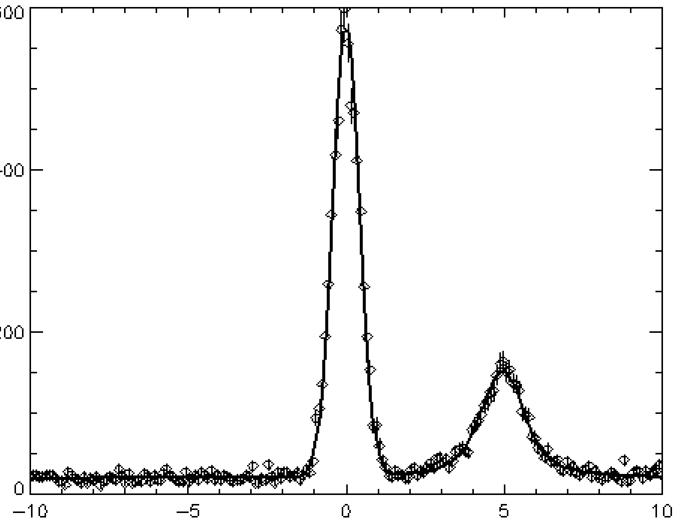

Figure 23 Result of analyzing the synthesized neutron scattering data using MPCURVEFIT, taking into account the instrumental resolution. Solid line is the resulting fit to the data.

Visualizing the intermediate fitting steps

It can be instructive to view the intermediate steps in the least-squares fit process in order to see if the fit is converging and how fast it converges. Unfortunately this is not possible with the built-in IDL curve-fitting routines. Here we will modify the example from the previous section so that you can see a (sort-of) animated view of the progress of the algorithm.

The mechanism by which we can see the intermediate steps is through the ITERPROC procedure and the ITERARGS argument in MPCURVEFIT. ITERPROC can contain any type of update code that you wish. You can plot the χ^2 value as a function of iteration if you want to see a value that describes the goodness-of-fit. In our case we will plot the current fit on top of the data at each iteration.

The ITERPROC procedure must be declared with a number of arguments and keywords as shown in my_iterproc below. Exogenous arguments necessary for the update routine are passed into ITERPROC via the ITERARGS structure. This is done through keyword inheritance (the _Extra keyword mechanism). The required arguments for the procedure are fnc (the function to be minimized), p (the current parameter set), iter (the current iteration), and fnorm (the current value of χ^2).

In our existing code as presented in the last section we add the following procedure named my_iterproc. This procedure should be located just before the analyze_fake_data source code. The fields that comprise ITERARGS in our implementation here include `x,y,dy` (the data), `res_ptr` (the pointer to the resolution function), `winpix` and `winvis` (used to plot the intermediate steps). The code that actually plots the data and fit to the pixmap and copies this to the window should be familiar to you at this point.

```
pro my_iterproc,  fnc,p,iter,fnorm,    $
                  dof = dof,           $
                  quiet = quiet,       $
                  parinfo = parinfo,   $
                  _Extra = extra
; Routine for performing a custom update procedure
scattering_function,extra.x,p,yfit, $
   _Extra = {res_ptr:extra.res_ptr}
wset,extra.winpix
plot,extra.x,extra.y,psym = 4
errplot,extra.x,extra.y-extra.dy,extra.y+extra.dy,width = 0.0
oplot,extra.x,yfit,psym = 0,thick = 2.0
wset,extra.winvis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,extra.winpix]

end
```

The code from the previous section is repeated here with the new additions emphasized in boldface.

```
; ********************************************** ;
pro scattering_function,x,p,yfit,_Extra = extra
bg = p[0] & area_el = p[1] & cen_el = p[2]
area_iel = p[3] & cen_iel = p[4] & fwhm_iel = p[5]
y_iel = lorentzian(x,area = area_iel,cen = cen_iel,fwhm = fwhm_iel)
nlo = 3 & nhi = n_elements(x) - nlo + 1
dx = x[1] - x[0]
con = convol(*extra.res_ptr,reverse(y_iel[nlo:nhi]),/edge_truncate) * dx
yfit = bg+gaussian(x,area = area_el,cen = cen_el,fwhm = 1.0)+con
end
; ********************************************** ;
pro my_iterproc,  fnc,p,iter,fnorm,    $
                  dof = dof,           $
                  quiet = quiet,       $
                  parinfo = parinfo,   $
                  _Extra = extra
; Routine for performing a custom update procedure
scattering_function,extra.x,p,yfit, $
   _Extra = {res_ptr:extra.res_ptr}
wset,extra.winpix
plot,extra.x,extra.y,psym = 4
errplot,extra.x,extra.y-extra.dy,extra.y+extra.dy,width = 0.0
oplot,extra.x,yfit,psym = 0,thick = 2.0
wset,extra.winvis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,extra.winpix]

end
; ********************************************** ;
pro analyze_fake_data
; First read in the data
```

```
dir = 'C:\idl_training\ornl\documentation\programs\'
filename = dir+'fake_data.txt'
count = 0L
dum1 = (dum2 = (dum3 = 0.0))
openr,lun,filename,/get_lun
while not eof(lun) do begin
   readf,lun,dum1,dum2,dum3
   if count eq 0L then begin
      x = dum1 & y = dum2 & dy = dum3
   endif else begin
      x = [x,dum1] & y = [y,dum2] & dy = [dy,dum3]
   endelse
   count++
endwhile
free_lun,lun,/force

; Now plot the data with the error bars
winvis = 0
window,0,xsize = 400,ysize = 400
window,/free,/pixmap,xsize = 400,ysize = 400
winpix = !d.window
wset,winpix
plot,x,y,psym = 4
errplot,x,y-dy,y+dy,width = 0.0
wset,winvis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,winpix]

; We will assume that we know that the resolution
; function is a Gaussian with a FWHM of 1.0 and is
; defined over the same interval as the data.
res = gaussian(x,area = 1.0,center = 0.0, fwhm = 1.0)
; Create a pointer to the resolution function because we
; will not want to pass around the vector and keep making
; copies of it.  A single reference to a heap variable
; is all that's necessary.
res_ptr = ptr_new(res)

; Pick some initial guesses for the fit
bg = 10.0 & area_el = 500.0 & cen_el = 0.0
area_iel = 100.0 & cen_iel = 5.0 & fwhm_iel = 1.0
p = [bg,area_el,cen_el,area_iel,cen_iel,fwhm_iel]
; Impose constraints on each of the positive-definite
; parameters that they be greater than zero.
parinfo = replicate({limited:[0,0],limits:[0D,0D]},n_elements(p))
parinfo[0].limited[0] = 1
parinfo[0].limits[0] = 0.D
parinfo[1].limited[0] = 1
parinfo[1].limits[0] = 0.D
parinfo[3].limited[0] = 1
parinfo[3].limits[0] = 0.D
parinfo[5].limited[0] = 1
parinfo[5].limits[0] = 0.D

iterargs = {x:x,y:y,dy:dy,winpix:winpix,winvis:winvis,res_ptr:res_ptr}
p = double(p)
functargs = {res_ptr:res_ptr}
yfit = mpcurvefit(x,y,1d/dy^2,p, sigma, parinfo = parinfo, $
   function_name = 'scattering_function', $
   /noderivative,functargs = functargs ,  $
   iterargs = iterargs,iterproc = 'my_iterproc')

wdelete,winpix

~~oplot,x,yfit,psym = 0,thick = 2.0~~
for i = 0,n_elements(p)-1 do $
   print,strtrim(string(p[i]),2)+' +/- '+strtrim(string(sigma[i]),2)

ptr_free,res_ptr
end
```

Since we want to have smooth updates of the plot window then we create a pixmap window to which we plot the data and copy the pixmap onto the visible window. The additional code includes the creation, use, and final deletion of the pixmap window. The only new additions to the call to the `MPCURVEFIT` function are the inclusion of the `ITERARGS` and `ITERPROC` keywords.

A diversion: Monte-Carlo estimation of parameter uncertainties

The uncertainties in the parameters that are returned by `MPCURVEFIT` are based on the curvature of the χ^2 surface. Implicit in this analysis is that the χ^2 surface is parabolic near the minimum. However this is only strictly true if the model function has a linear dependence on the parameters. Nevertheless this approximation is useful for many situations where the model function has a non-linear dependence on the parameters but the χ^2 surface is sufficiently parabolic near the minimum to warrant the approximation.

In fact you may come across analysis problems in your work that merit going beyond this approximation. The good news is that you can get reliable estimates on the uncertainties by a Monte-Carlo method. In this technique you generate a *large* number of synthetic data sets that are based on the original data and their error bars. In particular, if data point y_i has an error bar σ_i then you use a random number generator to create a new synthetic data point $\tilde{y}_i = y_i + r\sigma_i$ where r is a random number drawn from a normal distribution of unit standard deviation. After you create a large number of synthetic data sets you apply the same fitting algorithm to each data set and record the parameters determined from the fit. Finally you compute the standard deviation of each of the recorded parameters over the entire set of Monte-Carlo estimates. In the case of a linear dependence on the parameters the results from this method should converge with those of the least-squares technique.

In order to see how this works, add the following code to your program named `analyze_fake_data.pro` just before the line freeing `res_ptr`:

```
; First create the synthetic data
nmc = 300
nx = n_elements(x)
```

```
data = rebin(y,nx,nmc,/sample) + $
   (rebin(dy,nx,nmc,/sample)) * randomn(s,nx,nmc)
nparams = n_elements(p)
mc_params = fltarr(nparams,nmc)
; Now loop through NMC times and gather the statistics
for i = 0,nmc-1 do begin
   yfit = mpcurvefit(x,data[*,i],1d/dy^2,p, sigma, $
      parinfo = parinfo, $
      function_name = 'scattering_function', $
      /noderivative,functargs = functargs,/quiet)
   mc_params[*,i] = p
endfor
; Calculate the statistics on the Monte-Carlo estimates
sigma_p = fltarr(nparams)
for i = 0,nparams-1 do begin
   sigma_p[i] = sqrt((moment(mc_params[i,*]))[1])
endfor
print
print
for i = 0,nparams-1 do $
   print,strtrim(string(sigma[i]),2)+ $
   ' , '+strtrim(string(sigma_p[i]),2)
```

The last line prints the results comparing the uncertainties returned from `MPCURVEFIT` and those from the Monte-Carlo analysis. When I ran the program I got the following output:

```
0.43948982 , 0.441838
8.0348532 , 7.71067
0.0063221015 , 0.00693790
8.5615994 , 9.07471
0.021549678 , 0.0215111
0.060263775 , 0.0691997
```

The parameter uncertainties found using the two methods in general are quite close. In fact the relative uncertainties (uncertainty normalized to the parameter estimate) are virtually indistinguishable. We can conclude from this that the parabolic estimate for χ^2 is very good in this analysis.

Chapter 7-User Interface basics

Up until this point we have written procedural programs. As a user of these procedures you are expected to know how to “run” these programs. If you develop software for a group of individuals such as users at a neutron scattering center then one of the requirements is that your software must be user-friendly. More often than not this means that you need to develop a user-interface (UI) which “wraps” the overall functionality (or purpose) of the program. In IDL UI controls are called *widgets*. Widgets include buttons, check-boxes, sliders, interactive plot windows, text fields, droplists, menus, and many more. We cannot cover all widget types in IDL but we will cover enough to (1) get you started writing your own widget programs and (2) give you enough exposure to widget programming so that you can easily incorporate those widget types not covered in this manual in your own programs. In this chapter we will describe widget programming through a series of example programs. Each successive widget example will be more complicated and new features will be introduced.

Event-driven programming

Event-driven programs (EDP) are substantially different from procedural programs because EDP are in a state of waiting for information from the user. This information is transmitted to the program via the widgets (or controls). There are two essential components in a widget program: (1) the widget creation module which can be a procedure or function, and (2) the event handler. The widget creation module is where all of the widgets are created and rendered to the computer screen. In addition, the *event loop* is started in the widget creation module. When you run a widget program where input is expected from the user, the program is in a state of waiting for this input: this wait state is the event loop. The event handler can be either a procedure or a function. Its purpose is to execute the appropriate code for a given event. For example, if the user presses the QUIT button in an application then the corresponding event handler code would destroy the top-level base. The minimal requirements for a widget program are that it have both a widget creation module and an event handler. Most of our widget applications will have a single main event handler that dispatch the events to secondary event procedures or functions. This is better programming practice than lumping a huge set of commands in one event handler. This is done in the interest of modularity and clarity of code.

In the widget creation module the widgets for the application are defined. Functions define the widgets and the function’s return values are long integers known as widget identifiers (or ids for short). A widget id is extremely important because it is the *handle* needed to modify a widget, query a widget, or seek information about a widget. Usually the means by which you affect a widget is using `WIDGET_CONTROL` or

WIDGET_INFO. One of the requirements for mastering widget programming is to have working knowledge of accessing widget ids from various event handlers. We will see one way of how to do this in the examples presented in this chapter.

The first widget typically defined in any application is the top-level base which is created using the WIDGET_BASE function. This base widget holds all of the other widgets that you wish to expose to the users. The syntax for creating a base widget is as follows:

```
tlb = widget_base(/col,title = 'First Widget')
```

In this function two keywords are specified but neither of them are necessary. The first keyword is column which has been abbreviated to col in this example. The other keyword is title that will appear in the native WINDOWS frame for the widget. Consult the on-line help for the WIDGET_BASE function for a complete set of keywords and parameters. It is worth mentioning that one parameter that WIDGET_BASE accepts is a parent widget. By definition, the first widget (the top-level base), does not have a parent. The widget hierarchy is best described in terms of a parent-child-sibling model. The top-level base is the parent for all of the widgets that it contains. A button widget whose *parent* is the top-level base is a *child* of the top-level base. If there are two buttons, each of which have the top-level base as their parent, then they are *siblings* of each other. Sometimes it is useful to use the parent-child-sibling relationship in accessing and controlling widgets.

After the widgets have been defined they only exist in memory…they have not been displayed yet. The procedure named WIDGET_CONTROL is used to *realize* the top-level base on the screen. When the top-level base is realized on the screen then all of its children (and their children, and their children’s children, etc.) are realized too. The way to do this is with the command,

```
widget_control,tlb,/realize
```

The last required ingredient for a widget creation module is to register the top-level base with the local windows manager and start the event loop. This is done with the XMANAGER procedure. (Actually you don’t need to put in the XMANAGER procedure if you do not wish for the program to respond to events…it would not be a very useful program then would it?) The syntax for XMANAGER is the following

```
xmanager,’widget_program’, tlb,event_handler = ‘widget_handler’, $
      cleanup = ‘widget_cleanup’,/no_block
```

In the call to xmanager we have specified a number of keywords and set others. There are more keywords available to you in XMANAGER but we will only specify those that we will use often in our programs here. Please consult the on-line help for additional information.

The first argument is the name of the procedure that is being registered with the event loop. In the example given above this is called `widget_program`. The assumption is that we have written a program whose widget definition module is named `widget_program`. The second argument is the widget id of the top-level base defined in the widget creation module. In this example this widget id is `tlb`. These first two parameters are required. The remaining keywords are optional. Remember that order of keywords is not important. The first keyword specified is the `event_handler`. In our example above we name this `widget_handler` which is a procedure. The user must supply a procedure named `widget_handler`. Whenever any event gets generated by one of the widgets in the widget creation module then it will be dispatched to the event handler for further processing as defined by the programmer. If we had not specified the name of the event handler with this keyword then IDL will automatically expect that the user supplies a routine whose name is composed of the widget creation module name with `_event` appended to it. In the case above the user would have to supply a procedure named `widget_program_event`. The next keyword specified is `cleanup`. This is a user-supplied procedure that is executed just after the top-level base is destroyed. The purpose of this is for memory management. Often we pass information from program module to another and this can include heap variables or pixmaps. If we pass heap variable around or create pixmaps then this is the procedure that we use to free up that memory. There is no default expectation (if no cleanup routine is specified in the `XMANAGER` procedure) for the cleanup. The final keyword set in our example is the `NO_BLOCK` keyword. Setting this keyword allows the user to execute commands at the IDL command line while the UI is running. The default behavior is to *block* the command line. It is also good practice to invoke non-blocking behavior if you wish to have multiple applications running simultaneously. Otherwise the the blocking widget program will not relinquish control until it is destroyed and you will not be able to access other widget programs running concurrently.

As a first example of an event-driven program we'll create a base with a QUIT button on it. The source code for the example is listed in the box below. The only event type possible in this program will be button events and receipt of one of those events will cause the widget to be destroyed and the program will end. This program satisfies the minimal requirements for a widget program in that it contains two components: a widget creation module (`widget_ex_1`) and an event handler (`widget_ex_1_event`). Note that we have not specified a cleanup routine here because there is no information (i.e. heap variables) being passed from one module to another that require freeing up in the interest of proper memory management.

```
; **************************** ;
pro widget_ex_1_event,event
; This is the main event handler.
; Since there is only a single widget
; except for the base, there is only one
; event type that can occur and transfer program
; control to this handler.  So no matter
; what we will know that this is a
; QUIT button event.  Therefore we simply
; destroy the widget hierarchy.
widget_control,event.top,/destroy
end
; **************************** ;
pro widget_ex_1
; This is a first widget example
;
; Create the top-level base
tlb = widget_base(/col,title = 'First Widget')
; Create a QUIT button on the top-level base
quit = widget_button(tlb,value = 'QUIT',xsize = 200)
; Render the widgets on the screen
widget_control,tlb,/realize
; Start up the XMANAGER which manages the
; widget events
xmanager,'widget_ex_1',tlb,   $
   event_handler = 'widget_ex_1_event',/no_block

end
```

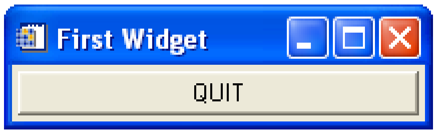

Figure 24 Screen output for our first widget application.

The widget creation module has four statements in it. The first statement defines the top-level base which we name `tlb`. As in the example presented we use the `WIDGET_BASE` function to create the top-level base. Note that we specify the `TITLE` keyword so that ‘First Widget’ appears in the title portion of the base (see Figure 24).

The second statement defines the QUIT button using the `WIDGET_BUTTON` function. In this function we specify the parent via the first parameter passed into the function, `tlb`. Next the `VALUE` keyword is set to ‘QUIT’ which is the name to be displayed on the button. We also set the `XSIZE` keyword to 200 which forces the button to be 200 pixel units wide.

The third statement, `WIDGET_CONTROL,TLB,/REALIZE`, tells IDL to display the tlb (and its child, the button) on the screen. The final statement is the call to `XMANAGER` which specifies the event handler as `widget_ex_1_event` and also makes it a non-blocking widget.

When you run this program and press the QUIT button, the widget disappears from the screen. But you might be wondering what information is sent to the event handler. In fact an event structure is passed to the event handler everytime you press a widget button (or, in general, interact with any widget). In the example `widget_ex_1` the widget button press causes a named event structure to be sent to the event handler.

The event handler is a procedure here that has a single input parameter: event. All event handlers are expected to be able to receive a single event structure. In this case the name of the event structure is WIDGET_BUTTON and it has four fields. The first three fields are named ID, TOP, and HANDLER. The last field is called SELECT. The first three fields are required fields for all widgets but there will usually be more fields. ID is the identifier (long integer) of the widget that generated the event. TOP is the widget id of the base at the top of the widget hierarchy that includes ID. HANDLER is the widget id associated with the event handler. In our example ID is quit, TOP is tlb, and HANDLER is tlb. The SELECT field has a value of 1 indicating that the button was pressed. We can make a few additions to the program to see these. Run the program after adding three lines shown below in boldface.

```
; **************************** ;
pro widget_ex_1_event,event
; This is the main event handler.
; Since there is only a single widget
; except for the base, there is only one
; event type that can occur and transfer program
; control to this handler.  So no matter
; what we will know that this is a
; QUIT button event.  Therefore we simply
; destroy the widget hierarchy.
print,tag_names(event,/structure_name)
print,tag_names(event)
widget_control,event.id,get_value = widget_value
print,'Widget_value: ',widget_value
widget_control,event.top,/destroy
end
```

When I ran this program and pressed the QUIT button I got the following text output:

```
IDL> widget_ex_1
WIDGET_BUTTON
ID TOP HANDLER SELECT
Widget_value: QUIT
```

The name of the event structure is WIDGET_BUTTON, there are four fields in the event structure (ID, TOP, HANDLER, and SELECT) and the *value* of the widget that generated the event is 'QUIT'. The importance of being able to identify different widgets in the hierarchy in the event handler will become important in the subsequent examples.

A UI with user input

The example in the previous section illustrated the fundamentals in creating a simple UI. In this section we present a more complicated UI which accepts user input from one widget and uses that information to update another widget.

In the example from the last section it was not necessary to distinguish between the types of events in the event handler because there was only one type of event. This event was generated by pressing the QUIT widget button. Applications that you write will likely have more than a single widget on the top-level base so it is necessary to have a mechanism to distinguish between different widgets. There are a number of ways to do this but we will select one in particular. This is not because it is better than any other way but it is usually good practice to stick with one method. In this case we specify a user-name for each widget which is accomplished by setting the `UNAME` keyword in the widget to an appropriate string. With this implemented then we can determine the id of any widget by using the `WIDGET_INFO` function with the `FIND_BY_UNAME` keyword:

```
id = widget_info(event.top,find_by_uname = ‘some_widget_uname’)
```

If there is no such widget in the hierarchy (rooted at event.top) whose `uname` is ‘some_widget_uname’ then the return value will be 0L (long 0). Note in the example above we used the top field of the event structure as the root of our search because it corresponds to the widget id of the top-level base.

The event handler for a program in which the `UNAMES` are specified for each widget can be written with the `CASE` statement to distinguish between the widgets that generated the events. In the example shown below, since the sequence of commands to be executed in the event handler are not too lengthy, we will keep all commands in the event handler.

```
; ***************************** ;
pro widget_ex_2_event,event
; This is the main event handler.  We
; parse the event by the user-name of
; the widget that generated it.
uname = widget_info(event.id,/uname)
case uname of
'QUIT':  widget_control,event.top,/destroy
'INPUT_TEXT':  $
   begin
      ; We first need to get the widget identifier
      ; for the output text field
      out_label_id = widget_info(event.top, $
         find_by_uname = 'OUTPUT_TEXT')
      ; Get the current text value from the input
      ; text widget
      widget_control,event.id,get_value = val
      ; Set the current text value from the input
      ; text widget in the output text widget
      widget_control,out_label_id,set_value = val
   end
else:
endcase
end
; ***************************** ;
pro widget_ex_2
; This is the second widget example
;
```

```
; Create the top-level base
tlb = widget_base(/col,title = 'Second Widget')
; Create a label
input_label_id = widget_label(tlb,value = 'Input')
; Create an editable input text widget.
; Moreover set the ALL_EVENTS keyword so that
; any change in the text field generates an
; event.
input_text_id = widget_text(tlb,value = '',/editable, $
   uname = 'INPUT_TEXT',xsize = 50,/all_events)
; Create the output label
out_label_id = widget_label(tlb,value = 'Output')
; Create an uneditable output text widget
output_text_id = widget_text(tlb,value = '', $
   uname = 'OUTPUT_TEXT',xsize = 50)

; Create a QUIT button on the top-level base
quit = widget_button(tlb,value = 'QUIT',uname = 'QUIT')

; Render the widgets on the screen
widget_control,tlb,/realize
; Start up the XMANAGER which manages the
; widget events
xmanager,'widget_ex_2',tlb,   $
   event_handler = 'widget_ex_2_event',/no_block

end
```

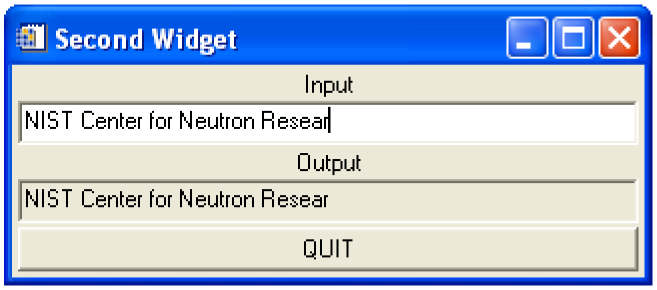
Second Widget
Input
NIST Center for Neutron Resear
Output
NIST Center for Neutron Resear
QUIT

Figure 25 Screen output for the second widget program.

When you run the program you will be allowed to type in the first editable field. IDL tells you that it is editable because it is white (in contrast to the second text field which is not editable and is the same color as the base). When you type in the first field labeled Input your typing will be mirrored in the second text box. This behavior is intentional. Typing information in the first text box creates events and the event structure, named WIDGET_TEXT_CH, has six fields. The first three fields are ID, TOP, and HANDLER (as usual) and the remaining three fields are TYPE, OFFSET, and CH. The TYPE field specifies which type of text event is created: for typing events this is 0. OFFSET specifies where the text is being inserted into the existing value. The final field specifies which character is being typed (the byte representation of the character).

The widget creation module is still short in this application. We define a top-level base, define five widgets, and start up the event loop with XMANAGER. The five widgets are composed of two label widgets, two text widgets, and a QUIT button. Note the order in which the widgets are created. Since the top-level base is a column base the order specified where each widget appears vertically. The two label widgets do not generate events and their only purpose is cosmetic. Their first parameter argument is the parent (i.e. tlb) and the keyword parameter is the value which is the string to be displayed as the label. The two widget labels are for each of the two text widgets.

The first text widget is the input text widget. The first parameter is the top-level base. The value is initialized to a null string. The EDITABLE keyword is set so that the user can modify the contents of the text box. The user name is specified with the UNAME keyword ('INPUT_TEXT' in this case). The horizontal width in characters is specified using the XSIZE keyword. Finally we set the ALL_EVENTS keyword so that any change in the contents of the text widget will generate an event.

```
input_text_id = widget_text(tlb,value = '',/editable, $
   uname = 'INPUT_TEXT',xsize = 50,/all_events)
```

The second text widget is the output text widget and is called as follows.

```
output_text_id = widget_text(tlb,value = '', $
   uname = 'OUTPUT_TEXT',xsize = 50)
```

This is called exactly the same way as the input text widget except the EDITABLE keyword is not set and the UNAME is set to 'OUTPUT_TEXT'.

The QUIT button is specified as in the example from the last section except we also specify the UNAME as 'QUIT' so that we can parse this widget event in the event handler.

The last two statements in the widget creation module should now be familiar to you. The first realizes the top-level base and its children to the screen and the second registers the top-level base with the local windows manager and starts the event loop.

There is a big difference in the event handler in this example compared to that in the example of the previous section. In the event handler, widget_ex_2_event, the uname of the widget that created the event is determined via WIDGET_INFO

```
uname = widget_info(event.id,/uname)
```

The parameter in the WIDGET_INFO function is event.id which is the widget id of the widget that generated the event. This function returns the UNAME because the UNAME keyword was set. Then we use a CASE statement to sort all possible widget ids via their unames. In this application ony two widgets can create events: the button widget and the input text box. We have already examined the code for the QUIT button in the previous example so we will not discuss it further. Events generated from the input text widget result in the following three commands (comments omitted):

```
out_label_id = widget_info(event.top, $
   find_by_uname = 'OUTPUT_TEXT')
widget_control,event.id,get_value = val
widget_control,out_label_id,set_value = val
```

First the id of the output text widget is found from using its user-name and the `FIND_BY_UNAME` keyword to the function `WIDGET_INFO`. Then we must get the current value of the input text widget. Since the id of the widget that got us to this point in the event handler, it is `event.id`. We use `WIDGET_CONTROL` to extract the value from `event.id`. Finally we use `WIDGET_CONTROL` to set the new value of the output widget to the value from `event.id`.

Exercise: Modify `widget_ex_2.pro` so that when the user presses the QUIT button, the application pauses for 2 seconds prior to the disappearance of the UI. Hint: you will want use the `WAIT` procedure.

Interacting with a plot window

One of the more interesting things that you might want to do in a widget application is enable the user to interact with a graphical image using the mouse. In general this is pretty simple to do and falls naturally in the realm of event-driven programming. The difference between this type of event-driven program and the simple ones presented thus far is that the events that provide the interaction with the plot window are mouse motion events and mouse button events.

We will specify the application first. The application that we write here will include a UI composed of a quit button, a randomizer button that, when pressed, will create a set of 50 random numbers drawn from a normal distribution, and a plot window that displays the current set of random numbers. The user will be able to move the cursor over the data using the mouse and the nearest data values will be displayed in the title of the plot. In addition, a thicker symbol will be displayed at the nearest data position in the plot. This is not a particularly complicated nor ambitious example but it will introduce you to the fundamental aspects of interacting with graphics using the mouse.

Because this program has a higher level of complexity than the previous ones we will be doing things a bit differently. First whenever there is interaction with a plot window it is standard practice to use a pixmap window so that plot window updates are smooth and not choppy. We will need to define the pixmap in the widget creation module and destroy it in a cleanup routine. Next we will need to have some mechanism to pass the information around between modules. We will define a structure in the widget creation module (composed of the pixmap, a window reference, and the data), create a pointer to this structure, and put the pointer reference into the user-value (UVALUE) of the top-level base. We have already seen that the programmer can put whatever string he/she wants into the UNAME of a widget. The programmer is not limited to putting just strings into the UVALUE. In fact any quantity can be put into a widget's UVALUE. We put a pointer reference to a structure into the UVALUE of the top-level base because all event handlers have access to the widget at the top of the hierarchy: the top-level base. Thus all event handlers have access to the information contained in this structure. The code for the program is shown in the box below.

There are seven modules (procedures) in this application: `widget_ex_3`, `wex3_event`, `wex3_rand_events`, `wex3_win_events`, `wex3_plot_refresh`, `wex3_plot_rand`, and `wex3_cleanup`. The purpose of each is listed below:

• `widget_ex_3`:	widget creation module
• `wex3_event`:	main event handler
• `wex3_rand_events`:	creates a new set of random numbers
• `wex3_win_events`:	processes window events such as cursor position
• `wex3_plot_refresh`:	copies pixmap contents to the visible window
• `wex3_plot_rand`:	plots the series of random numbers
• `wex3_cleanup`:	deletes the pixmap and frees the pointer

We will look at each of these procedures in detail.

```
; **************************** ;
pro wex3_cleanup,tlb
widget_control,tlb,get_uvalue = pstate
wdelete,(*pstate).winpix
ptr_free,pstate
end
; **************************** ;
pro wex3_plot_rand,event
widget_control,event.top,get_uvalue = pstate
plot,(*pstate).data,psym = -4,   $
   title = (*pstate).plot_title
end
; **************************** ;
pro wex3_plot_refresh,event
widget_control,event.top,get_uvalue = pstate
wset,(*pstate).winpix
wex3_plot_rand,event
wset,(*pstate).winvis
device,copy = [0,0,!d.x_size,!d.y_size,0,0, $
               (*pstate).winpix]
end
; **************************** ;
pro wex3_win_events,event
widget_control,event.top,get_uvalue = pstate
```

```
coords = convert_coord(event.x,event.y,   $
   /device,/to_data)
x = 49 < (round(coords[0]) > 0)
y = ((*pstate).data)[x]
; Update the title to include the latest position
; of the mouse (nearest to the data)
(*pstate).plot_title = $
      strtrim(string(x,format = '(I3)'),2)+   $
      ','+strtrim(string(y,format = '(f15.5)'),2)
wex3_plot_refresh,event
; Plot a thick symbol at the current closest
; data point.
plots,x,y,psym = 4,thick = 4.0,/data
end
; **************************** ;
pro wex3_rand_events,event
widget_control,event.top,get_uvalue = pstate
(*pstate).data = randomn(s,50)
wex3_plot_refresh,event
end
; **************************** ;
pro wex3_event,event
uname = widget_info(event.id,/uname)
case uname of
'QUIT':  widget_control,event.top,/destroy
'WIN':   wex3_win_events,event
'RAND':  wex3_rand_events,event
else:
endcase
end
; **************************** ;
pro widget_ex_3
tlb = widget_base(/row,title = '')
col_base = widget_base(tlb,/col)
void = widget_button(col_base,value = 'Randomize', $
   uname = 'RAND')
void = widget_button(col_base,value = 'Quit',      $
   uname = 'QUIT')
win = widget_draw(tlb,xsize = 300,ysize = 300,  $
   /motion_events,uname = 'WIN')

widget_control,tlb,/realize
widget_control,win,get_value = winvis
window,/free,/pixmap,xsize = 300,ysize = 300
winpix = !d.window
data = randomn(s,50)

state =  {  winvis:winvis,             $
            winpix:winpix,             $
            plot_title:'',             $
            data:data                  }
pstate = ptr_new(state)
widget_control,tlb,set_uvalue = pstate

xmanager,'widget_ex_3',tlb,/no_block,     $
   event_handler = 'wex3_event',   $
   cleanup = 'wex3_cleanup'
```

```
; Create a pseudo-event so that we can use
; the plot refresh event handler to plot
; the data in the window
pevent = {event,id:win,top:tlb,handler:0L}
wex3_plot_refresh,pevent
end
```

In the widget creation module we create the top-level base and make it a row base by specifying the `ROW` keyword. Next we create a column base using `WIDGET_BASE`, specifying its parent as the top-level base, and setting the `COL` keyword. We put two widget buttons in the column base by specifying their parent as the top-level base. The values for these are 'Randomize' and 'Quit' and the `UNAMES` are 'RAND' and 'QUIT' respectively. Next we use the `WIDGET_DRAW` function to create a plot window. Its parent is the top-level base. Since the top-level base is a *row* base then the plot window is plotted to the right of the column base. The keywords set in the `WIDGET_DRAW` function are `XSIZE` and `YSIZE` (set to 300 pixels for each dimension), `MOTION_EVENTS` and `UNAME` (set to 'WIN'). The `MOTION_EVENTS` keyword tells IDL to generate an event whenever the cursor moves over the draw widget. The user-name is specified so that we can distinguish between events that are initiated from it and the other two widgets. The top-level base (and all of its children) is realized next. After the widgets are realized then we need to get the window index which we can obtain with the draw widget id and the `WIDGET_CONTROL` procedure.

```
widget_control,win,get_value = winvis
```

In the command above, the widget id is `win` and the resulting window index is `winvis`. We will need this index so that we can plot data into it. Next we create a pixmap window of the same size as the draw widget. We already saw how to do this in Chapter 5-Data visualization. After this we assign the pixmap window to the `winpix` variable. Next we create the normally distributed random data and assign the 50-element vector to the data variable.

The next statement creates the *state structure* which contains `winvis, winpix, data`, and a variable called `plot_title` (just an empty string…we'll see its purpose later). Then a pointer is created, `pstate`, into which the state structure is assigned. Next we set the `UVALUE` of the top-level base to `pstate`.

The call to `XMANAGER` is no different from what we've seen so far except we now specify a cleanup routine. The final command to issue in this module is to draw the data in the draw widget. To make things easier we will call the procedure `wex3_plot_refresh`. This procedure requires an event structure passed in to it. Since one is not available in the widget creation module we must create one. We create a named pseudo-event structure that has the required fields: id, top, and handler. For simplicity we use the draw widget identifier as id, tlb for the top field, and 0L for the handler. (It really doesn't matter what we use for the handler field so 0L is as good as any other widget identifier) Next we call the procedure wex3_plot_refresh with the new pseudo-event structure as the argument.

From this point on all program control goes to the event handler whenever any events are generated. The main event handler, wex3_event, is very simple…simpler even than the example from the last section. This event handler is similar to the one in the previous example in that is distinguishes the origin of events based on the UNAME of the widgets. The three events are (1) from the 'Quit' button, (2) from the 'Randomize' button, and (3) events generated by moving the cursor over the draw widget. For 'Quit' we simply destroy the top-level base. For an event that is generated from the 'Randomize' button we invoke the wex3_rand_events procedure. This simply populates the data variable with a new set of random numbers. Note that when we call the procedure from here, we pass the event structure on.

The procedure wex3_rand_events is written as an event handler which is evident from the fact that the only parameter expected is an event structure. The first statement in that procedure gets the pointer to the state structure (pstate) from the top-level base (actually from the top field of the event structure). This is obtained from the GET_UVALUE keyword to the WIDGET_CONTROL procedure:

```
widget_control,event.top,get_uvalue = pstate
```

Next we create a new set of 50 random numbers and replace the current data field from the state structure. Note the use of pointer de-referencing here:

```
(*pstate).data = randomn(s,50)
```

The final part of this procedure is to display the data by invoking the wex3_plot_refresh procedure. Again the event structure is passed on into this procedure.

The wex3_plot_rand procedure plots the data to the current plotting device. Note that this procedure does not specify the device because this procedure is supposed to be used in a modular way. For instance this procedure is called from the wex3_plot_refresh procedure in order to update the pixmap (winpix) and then copy it to the plot window (winvis).

The procedure wex3_win_events contains the code to deal with draw widget events. Since we set the MOTION_EVENTS keyword in the call to WIDGET_DRAW, then we can expect that the event structure from the draw widget will contain x and y fields (see the on-line help for WIDGET_DRAW). These fields are the cursor position in pixel units. The first statement in this procedure extracts the state pointer from the top field of the event structure. The second statement converts the cursor coordinates from pixel units to data units using the CONVERT_COORD function. The next statement ensures that the horizontal cursor position (in data coordinates) is not larger than 49 and not less than 0 and assigns it to the variable x.

```
x = 49 < (round(coords[0]) > 0)
```

The next statement assigns the data point at that x coordinate to `y`. Next the title field of the state pointer is set to display the x and y coordinates. Next the data is displayed in the window and the new title is displayed at the top of the window. Finally a thicker symbol is plotted at the corresponding x and y coordinates using the `PLOTS` command.

The last routine we discuss is the cleanup routine which contains a single argument. For reasons that we will not go into here, the argument is not an event structure but rather is the widget id for the top-level base. Therefore to get the state pointer it is necessary only to use `widget_control` to extract it from `tlb`. Then we can delete the pixmap and free the state pointer `pstate`.

When I ran the program and moved the cursor around the plot I got the following output shown in Figure 26. You should see the little white ball jump from one data point to another as you move the cursor about.

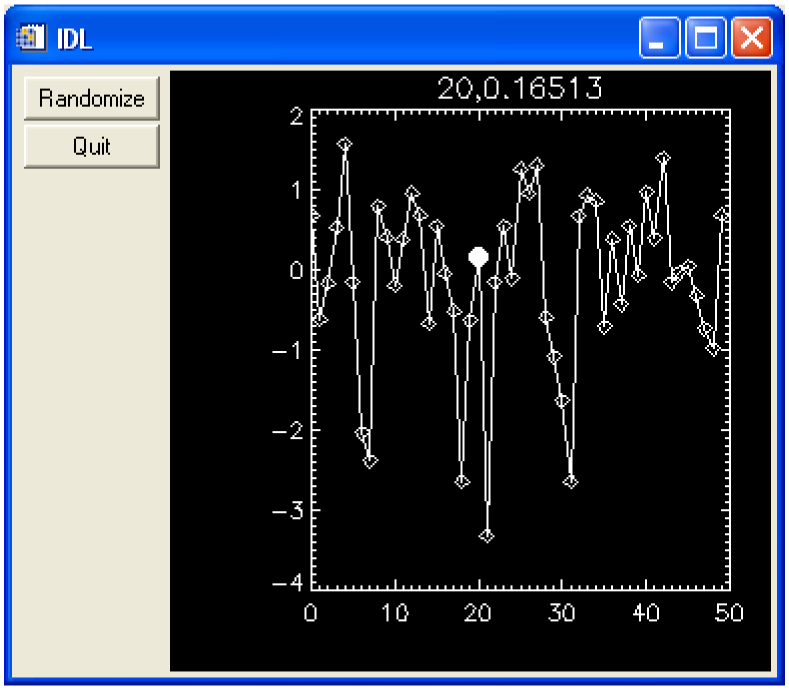

Figure 26 Screen shot of `widget_ex_3.pro`.

Distributing your application with the IDL Virtual Machine

After you have written an application you may wish to distribute it to users who do not have an IDL license. One option that became available recently is to distribute it so that it works in the IDL Virtual Machine (VM). This is a free option and anyone can download the VM from the Research Systems website (http://www.rsinc.com/download/).

The first step is to compile your program which you can do from the command line as follows:

```
IDL> .compile widget_ex_3
```

Next resolve all of the routines with which `widget_ex_3` depends:

```
IDL> resolve_all
```

Finally save the routines into a file:

```
IDL> save,/routines,filename = 'c:\widget_ex_3.sav'
```

Now you can run the VM and choose the file that you just created. The program will run without an IDL license. You can even create a standalone icon on your desktop (if you're running WINDOWS, that is) that will launch the IDL VM with your particular application (i.e. without needing to select the file). You can do this by doing the following:

1. Make a copy of the IDL VM on your desktop (<u>not</u> a shortcut).
2. Right click on the new icon and select PROPERTIES.
3. In the text field labeled Target add the fully path-qualified filename jusft after the "…vm=" string. For instance on my machine the original string in the Target text field is C:\RSI\IDL61\bin\bin.x86\idlrt.exe -vm=. I changed this to
   `C:\RSI\IDL61\bin\bin.x86\idlrt.exe -vm=widget_ex_3.sav`
4. Change the icon symbol to something else if you wish.

Finally you should be able to double-click the new icon and your program should start running (after the VM splash screen…no way to get rid of that!).

Applications such as DAVE require an embedded IDL license-a license to distribute code with an embedded IDL license which precludes the end user from needing to purchase a copy of IDL. Why not distribute DAVE with the VM? The reason is that there are certain limitations of the VM. For instance, the use of the EXECUTE function is forbidden with the VM. Since there are a number of applications in the DAVE distribution that require the use of the EXECUTE function, it is impossible to use the VM in this case.

<u>Exercise:</u> Create a distribution of widget_ex_1.pro and widget_ex_2.pro suitable for the IDL Virtual Machine.

Chapter 8-Writing a GUI application

In this chapter we will design and construct a rather sophisticated data display application. We will begin with a description of the functionality desired and then describe the plan for implementation in IDL. After we write the application we will finish with some exercises for you to extend the application's functionality.

Design specifications and functional requirements

The application that we wish to write will allow the user to load in some (synthetic) two-dimensional neutron scattering data, visualize it in an image representation, and display dynamics "cuts" through the image based on the current cursor position. The user will also be able to zoom into a region of the image and change the color table used to display it. In order to zoom into the image plot we will choose the following mechanism: (1) user presses the left mouse button and holds it down, (2) a rubber-band box is expanded based on where the user moves the mouse in the draw widget (still holding the left mouse button down), and (3) when user releases the mouse button the image display is updated to show the appropriate *magnified* view. There will be three plot windows (draw widgets) configured as shown in Figure 27. As we design the application please keep in mind the numbering scheme shown in the figure.

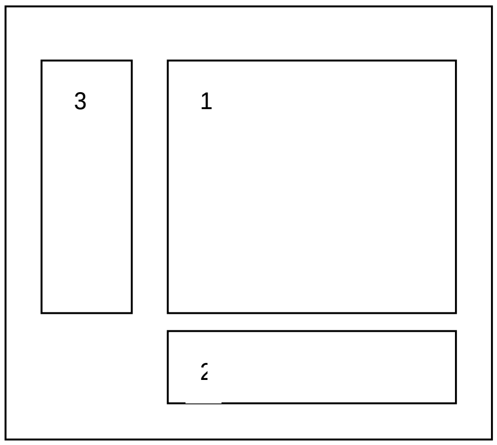

Figure 27 Schematic layout of the draw widgets in this application.

The main image will be displayed in window 1. When the cursor is positioned over window 1, a vertical line will extend in up and down to the extents of the image and the horizontal line will extend to the left and right to the extents of the image. This active cross-hairs will determine which "cuts" are displayed in windows 2 and 3. The data displayed in window 2 will be a "constant-y" cut and the data displayed in window 3 will be a "constant-x" cut. Just below window 3 the data coordinates corresponding to the cursor's position over window 1 will be updated dynamically.

Implementation plan

The functionality described in the last section gives us the framework for developing an outline of the code: procedures/functions and the data necessary for implementation. There are a few new things that we will learn here regarding widget programming. For instance the visualization technique of zooming into an image is very useful, especially when you are looking for small features in the data. In addition we will see how to change the color table. This will be done in an interactive manner, again taking advantage of a widget application developed by a third party. This particular color table selection application is called `XCOLORS.PRO` written by David Fanning and is part of the distribution for this course. We will also use `PLOTIMAGE` (covered in chapter 5) for display of the data in window 1. Finally we will see how to construct a menu bar at the top of the application. This menu bar will contain a selection for loading data, exiting the application, and changing the color table.

From the specifications given above we can write down the routines that we will need to write along with their purposes. We will name this application `IMAGE_GUI.PRO`.

Procedure/Function	**Purpose**
`IMAGE_GUI`	Widget creation module
`IMAGE_GUI_EVENT`	Main event handler/dispatcher
`IMAGE_GUI_CLEANUP`	Frees up heap variables and deletes pixmaps
`IMAGE_GUI_LOAD_DATA`	Self explanatory
`IMAGE_GUI_DRAW1`	Plots data for window 1
`IMAGE_GUI_DRAW2`	Plots data for window 2
`IMAGE_GUI_DRAW3`	Plots data for window 3
`IMAGE_GUI_REFRESH1`	Updates pixmap 1 and copies to window 1
`IMAGE_GUI_REFRESH2`	Updates pixmap 2 and copies to window 2
`IMAGE_GUI_REFRESH3`	Updates pixmap 3 and copies to window 3
`IMAGE_GUI_WIN_EVENTS`	Handles updating coordinates output and zooming events
`IMAGE_GUI_CHANGE_COLORS`	Enables user to change color tables using `XCOLORS`.

Table 1 Procedures (and purpose) included in the IMAGE_GUI application.

GUI Layout

The draw widgets are arranged in a precise manner as shown in Figure 27 because the aesthetics of the layout are important. The way to implement this particular layout is two-fold: (1) make the dimensions of each of the three windows correspond appropriately and (2) construct the bases sensibly. We address point (2) by selecting the top-level base to be a row base and then create two column bases, both of whom have the top-level base as their parent. For the sake of screen real-estate we will make window 1 square (horizontal dimension = vertical dimension) and the exact dimension will rely on the size of the display currently in use. We can access the current display size using the IDL command `device, get_screen_size = screen_size`. When I invoked this command on my PC I got the following output:

```
IDL> device,get_screen_size = screen_size
IDL> print,screen_size
    1024     768
```

The variable screen_size is a two element integer array whose first element is the horizontal size in pixels and whose second element is the vertical size in pixels. We will choose a edge size for window 1 that is less than half the vertical screen size. Moreover we will choose that the “cut” windows (2 and 3) have dimensions consistent with the Golden Ratio ($\left(1+\sqrt{5}\right)/2$) which is considered to be aesthetically pleasing (so sue me for being odd). We will place three label widgets below window 3 for displaying the current cursor coordinates in data units. The following code for the widget definition module arranges the widgets per our specification.

```
pro image_gui
; Widget creation module
; Initialize the color table to blue-white
device,decomposed = 0
loadct,1,/silent
; Get the screen size to make the draw widgets
; appropriately sized.
device,get_screen_size = screen_size
; Define the sizes.  And just for fun we'll use the
; Golden Ratio to determine the size of the
; windows into which the cuts will be displayed.
golden_ratio = 0.5*(1.0+sqrt(5.))
xsize1 = (ysize1 = fix(0.35*screen_size[1]))
xsize2 = xsize1 & ysize2 = fix(ysize1/golden_ratio)
ysize3 = ysize1 & xsize3 = fix(xsize1/golden_ratio)
; Create the top-level base, make it a row-base, and
; specify that we want a menu bar
tlb = widget_base(/row,title = 'Data Display Application', $
   mbar = bar)
; Create the filemenu
filemenu = widget_button(bar,value = 'File',/menu)
; Create the first column base which holds window 3
; and the label widget that will display the cursor
; coordinates in data units.
col1 = widget_base(tlb,/col)
win3 = widget_draw(col1,xsize = xsize3,ysize = ysize3)
xlabel = widget_label(col1,value = 'x: ',/dynamic_resize, $
   uname = 'XLABEL',/align_left)
ylabel = widget_label(col1,value = 'y: ',/dynamic_resize, $
   uname = 'YLABEL',/align_left)
zlabel = widget_label(col1,value = 'z: ',/dynamic_resize, $
   uname = 'ZLABEL',/align_left)
col2 = widget_base(tlb,/col)
win1 = widget_draw(col2,xsize = xsize1,ysize = ysize1)
win2 = widget_draw(col2,xsize = xsize2,ysize = ysize2)
```

```
; Let's center the top-level base on the screen
geom = widget_info(tlb,/geometry)
widget_control,tlb,  $
   xoff = (screen_size[0]/2)-(geom.scr_xsize/2),$
   yoff = (screen_size[1]/2)-(geom.scr_ysize/2)
widget_control,tlb,/realize

end
```

We also implemented a menu bar by setting the output keyword `mbar` in the top-level base. The output keyword, which we assign to `bar`, is a widget id into which we can place additional widget buttons with the `menu` keyword set. We created a menu heading on this list named FILE (whose widget id is `filemenu`) using the `WIDGET_BUTTON` function (whose parent is `bar`) and setting the `menu` keyword. Try running the program and you should see an interface similar to the one shown in the following figure.

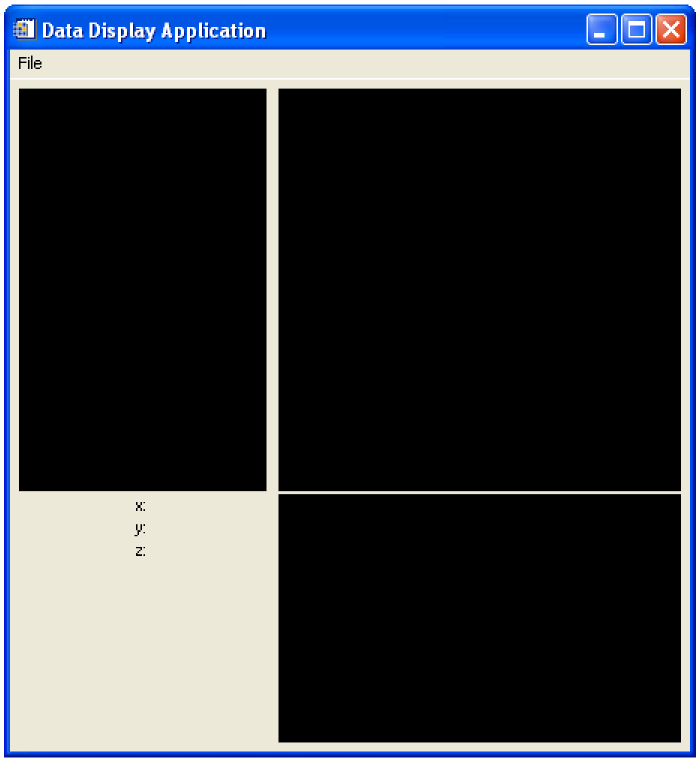

Figure 28 Screen shot of the user-interface layout for `IMAGE_GUI`.

Event handler, data, and auxiliary procedures

In this application we will need to pass data from one module to another. We will store this information in a structure and create a pointer to it and place it in the `UVALUE` for the top-level base, thus making accessible to any procedure having access to an event structure (via `event.top`). The following list of variables represents the information necessary for this application.

Variable name	**Purpose**
winvis1	Window ID for draw widget 1
winvis2	Window ID for draw widget 2
winvis3	Window ID for draw widget 3
winpix1	Pixmap ID corresponding to draw widget 1
winpix2	Pixmap ID corresponding to draw widget 2
winpix3	Pixmap ID corresponding to draw widget 3
color_ptr	Pointer to a structure containing the current color triplet (r,g,b) for the image displayed in window 1
xtitle	Label for the x-axis in the display
ytitle	Label for the y-axis in the display
ztitle	Label for the z-axis in the display
mouse	Contains last button event from the mouse
Autoscale	Flag which determines if autoscaling of the axes should occur
xbox	Horizontal coordinates of the lower and upper bounds of the rubberband zoom box
ybox	Vertical coordinates of the lower and upper bounds of the rubberband zoom box
xrange	Horizontal zoom range in data coordinates
yrange	Vertical zoom range in data coordinates
eindex	Index specifiying the element in the energy-vector for the Q-cut
qindex	Index specifying the element in the q-vector for the E-cut
zptr	Pointer to the data array
dzptr	Pointer to the uncertainty in the data array
eptr	Pointer to the x-values corresponding to the data array
qptr	Pointer to the y-values corresponding to the data array

Table 2 Table of variables included in the state structure and their purposes.

The code for the entire widget creation module is shown below. The statements added to this from the previous listing are shown in boldface.

```
pro image_gui
; Widget creation module
; Initialize the color table to blue-white
device,decomposed = 0
loadct,1,/silent
; Load the colors into variables that we'll use
; to "color-protect" the cut data: i.e. we wish
; to maintain the white on black appearance of
; the cut plots but not affect the color table
; of the image plot.
tvlct,r,g,b,/get
color_ptr = ptr_new({r:r,g:g,b:b},/no_copy)
; Get the screen size to make the draw widgets
; appropriately sized.
device,get_screen_size = screen_size
```

```
; Define the sizes.  And just for fun we'll use the
; Golden Ratio to determine the size of the
; windows into which the cuts will be displayed.
golden_ratio = 0.5*(1.0+sqrt(5.))
xsize1 = (ysize1 = fix(0.35*screen_size[1]))
xsize2 = xsize1 & ysize2 = fix(ysize1/golden_ratio)
ysize3 = ysize1 & xsize3 = fix(xsize1/golden_ratio)
; Create the top-level base, make it a row-base, and
; specify that we want a menu bar
tlb = widget_base(/row,title = 'Data Display Application', $
   mbar = bar)
; Create the filemenu
filemenu = widget_button(bar,value = 'File',/menu)
void = widget_button(filemenu,value = 'Load data',uname = 'LOAD')
void = widget_button(filemenu,value = 'Change colors', $
      uname = 'CHANGE_COLORS')
void = widget_button(filemenu,value = 'Quit',uname = 'QUIT')
; Create the first column base which holds window 3
; and the label widget that will display the cursor
; coordinates in data units.
col1 = widget_base(tlb,/col)
win3 = widget_draw(col1,xsize = xsize3,ysize = ysize3)
xlabel = widget_label(col1,value = 'x: ',/dynamic_resize, $
   uname = 'XLABEL',/align_left)
ylabel = widget_label(col1,value = 'y: ',/dynamic_resize, $
   uname = 'YLABEL',/align_left)
zlabel = widget_label(col1,value = 'z: ',/dynamic_resize, $
   uname = 'ZLABEL',/align_left)
col2 = widget_base(tlb,/col)
win1 = widget_draw(col2,xsize = xsize1,ysize = ysize1,   $
   /motion_events,/button_events,uname = 'WIN')
win2 = widget_draw(col2,xsize = xsize2,ysize = ysize2)


; Let's center the top-level base on the screen
geom = widget_info(tlb,/geometry)
widget_control,tlb,  $
   xoff = (screen_size[0]/2)-(geom.scr_xsize/2),   $
   yoff = (screen_size[1]/2)-(geom.scr_ysize/2)
widget_control,tlb,/realize

; Get the window ids out and create the pixmaps
widget_control,win1,get_value = winvis1
widget_control,win2,get_value = winvis2
widget_control,win3,get_value = winvis3
window,/free,/pixmap,xsize = xsize1,ysize = ysize1
winpix1 = !d.window
window,/free,/pixmap,xsize = xsize2,ysize = ysize2
winpix2 = !d.window
window,/free,/pixmap,xsize = xsize3,ysize = ysize3
winpix3 = !d.window

; Initialize the pointers
zptr = ptr_new(/allocate_heap)
dzptr = ptr_new(/allocate_heap)
eptr = ptr_new(/allocate_heap)
qptr = ptr_new(/allocate_heap)
```

```
state = {   winvis1:winvis1,              $
            winvis2:winvis2,              $
            winvis3:winvis3,              $
            winpix1:winpix1,              $
            winpix2:winpix2,              $
            winpix3:winpix3,              $
            color_ptr:color_ptr,          $
            xtitle:'E (meV)',             $
            ytitle:'Q (!3!sA!r!u!9 %!3!n!E-1!N)',  $
            ztitle:'Intensity',           $

            mouse:0B,                     $
            autoscale:1B,                 $
            xbox:[0,0],                   $
            ybox:[0,0],                   $
            xrange:[0.,0.],               $
            yrange:[0.,0.],               $
            eindex:0,                     $
            qindex:0,                     $
            zptr:zptr,                    $
            dzptr:dzptr,                  $
            eptr:eptr,                    $
            qptr:qptr                     }

pstate = ptr_new(state,/no_copy)
widget_control,tlb,set_uvalue = pstate,/no_copy

xmanager,'image_gui',tlb,             $
   event_handler = 'image_gui_event',  $
   cleanup = 'image_gui_cleanup',      $
   /no_block

end
```

The first addition to the code is the creation of a pointer to a color structure. The reason for this is as follows. IDL can only keep a single color table loaded at any one time. Yet in this application we wish to be able to specify any of the 42 color tables for the image display (window 1) and maintain a white-on-black color scheme for the cuts displayed in windows 2 and 3. The default plotting scheme for the PLOT procedure is that the background is always the bottom of the color table (color 0) whereas the text and symbols are displayed with the color at the top of the color table (color 255). For many of the color tables this corresponds to black (0) and white (255) respectively. However this is not true for color table 6. Therefore, since we anticipate giving the users the option to select any color table for the display in window 1, we must ensure that the colors in windows 1 and 2 are always white on a black background. We do this by *protecting* the colors in the widget creation module. The color table loaded here, 2, is the blue-white color table. In this color table black is at the bottom of the table and white is at the top. Since this satisfies our color requirements for the colors in the cuts plots then we can simply save the resulting r-g-b color triplet. This is the reason why we call to `TVLCT, r, g, b, /get` and then store this information in a structure (subsequently referenced with a pointer). Note that we also use the no_copy keyword in the creation of the pointer to the color structure. The reason for doing this is that we do not wish to create a copy of the structure with three 256-element vectors in it if we do not need to (in the interest of maintaining a philosophy of good memory management). The no_copy keyword simply creates a pointer to the original memory location for the r, g, and b color vectors. The pointer is subsequently placed into an anonymous structure whose variable name is state. A pointer named pstate is created which contains the state structure.

The next additions to the code are the menu items for (1) changing the color table and (2) exiting the program. As in the program from the last chapter we assign unames to each of these widgets so that we distinguish between events in the main event handler and dispatch the events to the appropriate procedures accordingly.

We also modify the creation of the draw widget by setting the button_events and motion_events keywords as well as specifying a uname (for distinguishing between events in the main event handler). We gained some experience with setting the motion_events keyword in the last program from the last chapter. However here we also need to set the button_events keyword so that whenever we press a button on the mouse, an event is generated. Recall from our specifications stated previously that we need to be able to capture both motion events and button events.

The final statements added to the widget creation module come after the top-level base is realized (i.e. widget_control,tlb,/realize). The statements that immediately follow (1) get the window IDs for each of the draw widgets (we will need them all so that we can tell where to draw different plots) and (2) create pixmaps to enable smooth screen updates via the double buffering mechanism with which you should now be familiar.

Next we initialize the data pointers. We set the keyword allocate_heap in each of them although this is not strictly necessary. It is a personal preference of mine to do so so that when data is loaded in, we simpley de-reference the data into the pointer (rather than create a new one everytime a new dataset is loaded in). Next we define the state structure composed of all of the variables listed in table 2. A pointer to this structure is then created and stored in the uvalue of the top-level base for easy retrieval from the event handler.

The final statement in the program is the call to the xmanager procedure. There is nothing new or different about this call as compared to that in widget_ex_3, for instance. We specify the main event handler, a cleanup routine, and the fact that it is a non-blocking widget.

The next important procedure in the program is the main event handler, image_gui_event. It is listed below:

```
pro image_gui_event,event
uname = widget_info(event.id,/uname)
```

```
case uname of
'LOAD':           image_gui_load_data,event
'QUIT':           widget_control,event.top,/destroy
'WIN':            image_gui_win_events,event
'CHANGE_COLORS':  image_gui_change_colors,event
'SHOW_SURF':      image_gui_show_surf,event
else:
endcase
end
```

As done in a previous widget program we are distinguishing between all of the possible events based on the `unames` of the widgets that generate the events.

The first logical event that will be generated with this application will be to load data. The procedure that handles loading data, `image_gui_load_data`, is shown below:

```
pro image_gui_load_data,event
widget_control,event.top,get_uvalue = pstate
filename = dialog_pickfile(dialog_parent = event.top, $
   filter = '*.sav',path = sourceroot())
if ~file_test(filename) then return
restore,filename = filename
*(*pstate).zptr = y
*(*pstate).dzptr = dy
*(*pstate).eptr = e
*(*pstate).qptr = q
image_gui_refresh3,event
image_gui_refresh2,event
image_gui_refresh1,event
end
```

In this module we used a built-in IDL widget called `dialog_pickfile` which allows the user to select a file with an interface that is native to the platform (operating system) upon which you are running. Next we check if the file selected with dialog_pickfile indeed exists. If so it is opened and the data values from the state pointer are de-referenced with these values. Note that we are assuming that the data was stored in IDL's native binary format using the program you created in the exercise at the end of Chapter 3. It is further assumed that the variables saved in your exercise were named `y, dy, e,` and `q`. The last three statements simply refresh each of the three plot widgets.

Each of the three "refresh" routines, `image_gui_refresh1`, `image_gui_refresh2`, and `image_gui_refresh3`, are all identical in function and the only difference is the source of the graphics and the destination of the graphics. We only list the "refresh" routine for the draw widget 1. There should be no surprises here since we are simply doing the same double-buffering for a smooth graphics display.

```
pro image_gui_refresh1,event
widget_control,event.top,get_uvalue = pstate
wset,(*pstate).winpix1
```

```
image_gui_draw1,event
wset,(*pstate).winvis1
device,copy = [0,0,!d.x_size,!d.y_size,   $
               0,0,(*pstate).winpix1]
end
```

The three procedures that do the actual plot creation, `image_gui_draw1`, `image_gui_draw2`, and `image_gui_draw3`, are different from each other and we should address the content of each one in turn.

```
pro image_gui_draw1,event
widget_control,event.top,get_uvalue = pstate
e = *(*pstate).eptr
q = *(*pstate).qptr
y = *(*pstate).zptr
if (*pstate).autoscale then begin
   (*pstate).xrange = [min(e),max(e)]
   (*pstate).yrange = [min(q),max(q)]
endif
de = e[1] - e[0]
dq = q[1] - q[0]
imgxrange = [min(e)-0.5*de,max(e)+0.5*de]
imgyrange = [min(q)-0.5*dq,max(q)+0.5*dq]
plotimage,bytscl(y),imgxrange = imgxrange,   $
         imgyrange = imgyrange,              $
         xrange = (*pstate).xrange,          $
         yrange = (*pstate).yrange,          $
         xtitle = (*pstate).xtitle,          $
         ytitle = (*pstate).ytitle
end
```

This plotting procedure uses the `plotimage` routine to render the data in an image representation with meaningful axes surrounding it. The data are first de-referenced from the state pointer. Next the horizontal and vertical data ranges are determined based on if the `autoscale` keyword is set or not. If `autoscale` is set then the horizontal and vertical ranges extend to the limits of the data. Next the `imgxrange` and `imgyrange` values are determined based on the actual data range. In order to get the data to be displayed “bin-centered” we subtract off half a channel (bin) from the lower bound and add half a channel (bin) to the upper bound. Finally the data are displayed using `plotimage` with the appropriate range and label keywords set.

```
pro image_gui_draw2,event
widget_control,event.top,get_uvalue = pstate
ycut = (*(*pstate).zptr)[*,(*pstate).qindex]
dycut = (*(*pstate).dzptr)[*,(*pstate).qindex]
dely = max(ycut)+max(dycut)
tvlct,r,g,b,/get
tvlct,(*(*pstate).color_ptr).r,  $
      (*(*pstate).color_ptr).g,  $
      (*(*pstate).color_ptr).b
plot,*(*pstate).eptr,ycut,psym = 4, $
   xrange = (*pstate).xrange,/xsty, $
   yrange = [0.0,1.1*dely],/ysty,   $
   xtitle = (*pstate).xtitle,       $
   ytitle = (*pstate).ztitle
errplot,*(*pstate).eptr,ycut-dycut,ycut+dycut,  $
   width = 0.0
; Now plot the vertical line
plots,[(*(*pstate).eptr)[(*pstate).eindex], $
       (*(*pstate).eptr)[(*pstate).eindex]],!y.crange,/data
tvlct,r,g,b
end
```

In `image_gui_draw2`, we come across our first instance of "color-protection." First from the data and uncertainty arrays, we get the appropriate single q-cut with energy as the independent variable. For aesthetic reasons we always want to plot the y-range over a range greater than the extents of the data. This is accomplished by creating the variable `dely` which is based on the largest data point and the largest "error bar." This is then used in the `yrange` keyword in the subsequent `plot` statement. Next we get the current r-g-b color triplet and save the variables r, g, and b (i.e. save the color table for the image plot in window 1 and then load the stored color triplet in which will ensure that the plot is white on black. The "error bars" are then plotted on top of the data. Then the vertical line is drawn over the data (using `PLOTS`) which will be coincident with the horizontal location of the mouse position in window 1. Finally the color table is restored via the final call to the procedure `TVLCT`.

```
pro image_gui_draw3,event
widget_control,event.top,get_uvalue = pstate
xcut = (*(*pstate).zptr)[(*pstate).eindex,*]
dxcut = (*(*pstate).dzptr)[(*pstate).eindex,*]
q = *(*pstate).qptr
delx = max(xcut)

tvlct,r,g,b,/get
tvlct,(*(*pstate).color_ptr).r,  $
      (*(*pstate).color_ptr).g,  $
      (*(*pstate).color_ptr).b
plot,xcut,*(*pstate).qptr,psym = 4, $
   yrange = (*pstate).yrange,/ysty, $
   xrange = reverse([0.0,1.1*delx]),/xsty, $
   xtitle = (*pstate).ztitle,       $
   ytitle = (*pstate).ytitle
for i = 0,n_elements(xcut)-1 do $
   plots,[xcut[i]-dxcut[i],xcut[i]+dxcut[i]], $
         [q[i],q[i]],/data
; Now plot the vertical line
plots,!x.crange,[(*(*pstate).qptr)[(*pstate).qindex], $
       (*(*pstate).qptr)[(*pstate).qindex]],/data
tvlct,r,g,b
end
```

The procedure to draw the plot in window 3 is slightly more challenging than that for window 2 because we want to draw the plot "on its side." We want to rotate a "normal" plot counter-clockwise by 90 degrees. We accomplish this by (1) swapping the x and y arguments to the PLOT procedure and (2) reversing the xrange variable for the plot. Also, the errplot routine will not accommodate "horizontal" error bars. Therefore we can use our own procedure which is just a loop over each of the data values and a call to the PLOTS procedure to create the horizontal lines of the right length and at the correct location. One other point to note about both image_gui_draw2 and image_gui_draw3 is that in both cases the range for the independent variable is determined by the current range (magnification) of window 1.

```
pro image_gui_win_events,event
widget_control,event.top,get_uvalue = pstate
if n_elements(*(*pstate).zptr) eq 0 then return
case event.type of
0: begin ; button press
      (*pstate).mouse = event.press
      if event.press eq 1 then begin
         ; Left mouse button
         (*pstate).xbox[0] = event.x
         (*pstate).ybox[0] = event.y
         image_gui_refresh1,event
         (*pstate).autoscale = 0B
      endif
   end
1: begin ; button release
      if (*pstate).mouse eq 4 then begin
         (*pstate).autoscale = 1B
         image_gui_refresh1,event
      endif
      if (*pstate).mouse eq 1 then begin
         xll = (*pstate).xbox[0] < (*pstate).xbox[1]
         yll = (*pstate).ybox[0] < (*pstate).ybox[1]
         w = abs((*pstate).xbox[1] - (*pstate).xbox[0])
         h = abs((*pstate).ybox[1] - (*pstate).ybox[0])
         xur = xll + w
         yur = yll + h
         ll = convert_coord(xll,yll,/device,/to_data)
         ur = convert_coord(xur,yur,/device,/to_data)
         (*pstate).xrange = [ll[0],ur[0]]
         (*pstate).yrange = [ll[1],ur[1]]
         image_gui_refresh1,event
      endif
      (*pstate).mouse = 0B
   end
2: begin ; motion events
      if (*pstate).mouse eq 1 then begin
         (*pstate).xbox[1] = event.x
         (*pstate).ybox[1] = event.y
         xc = [(*pstate).xbox[0],event.x,event.x,$
               (*pstate).xbox[0],$
               (*pstate).xbox[0]]
         yc = [(*pstate).ybox[0],(*pstate).ybox[0],$
               event.y,event.y,$
               (*pstate).ybox[0]]
         wset,(*pstate).winvis1
         device,copy = [0,0,!d.x_size,!d.y_size, $
            0,0,(*pstate).winpix1]
         plots,xc,yc,/device
      endif
      if (*pstate).mouse eq 0 then begin
         ; We haven't pressed a button.  The things that we
         ; need to do here are (1) display the current cursor
         ; coordinates in the widget labels and (2) generate
         ; the cuts.
         coords = convert_coord(event.x,event.y, $
            /device,/to_data)
         elo = min(*(*pstate).eptr,max = ehi)
         qlo = min(*(*pstate).qptr,max = qhi)
         condition = (coords[0] lt elo) or (coords[0] gt ehi) or $
                     (coords[1] lt qlo) or (coords[1] gt qhi)
         if condition then return
         e = *(*pstate).eptr
         q = *(*pstate).qptr
         eindex = (where(abs(e-coords[0]) eq min(abs(e-coords[0]))))[0]
         qindex = (where(abs(q-coords[1]) eq min(abs(q-coords[1]))))[0]

         xlabel = widget_info(event.top,find_by_uname = 'XLABEL')
         ylabel = widget_info(event.top,find_by_uname = 'YLABEL')
         zlabel = widget_info(event.top,find_by_uname = 'ZLABEL')
         widget_control,xlabel,set_value = 'x: '+ $
            strtrim(string(coords[0]),2)
         widget_control,ylabel,set_value = 'y: '+ $
            strtrim(string(coords[1]),2)
         widget_control,zlabel,set_value = 'z: '+  $
            strtrim(string((*(*pstate).zptr)[eindex,qindex]),2)
         (*pstate).qindex = qindex
         (*pstate).eindex = eindex
         image_gui_refresh3,event
         image_gui_refresh2,event
         image_gui_refresh1,event
         ; Plot the cross-hairs
         plots,!x.crange,replicate((*(*pstate).qptr)[qindex],2),/data
         plots,replicate((*(*pstate).eptr)[eindex],2),!y.crange,/data
      endif
   end
else:
endcase
end
```

The procedure that handles motion and button events from the mouse is called `image_gui_win_events`. This procedure distinguishes events based on the `type` field from the event structure. The procedure uses a `CASE` statement to distinguish between button press events, button release events, and mouse motion events. Just a reminder that these events can only be generated from window 1.

Let's start with button press events. If the left button is pressed then we store that information (the *button press* information) in the `mouse` field of the state pointer and also store the device coordinates of the current mouse position in the window in the `xbox` and `ybox` fields of the state pointer.

When either of the buttons (left or right) are released we use the mouse field of the state pointer to determine which button was released. If the right button is released then we set the autoscale field of the state pointer to be 1 so that subsequent refreshes of the windows are “auto-scaled.”

If the left mouse button is released then we do something quite a bit different. In this case the user has just completed selection of a region-of-interest using the rubberband box type zooming. The subsequent commands create a properly ordered set of two coordinates (device coordinates) that represent the lower left and upper right hand corners of the zoom box. These coordinates are converted to data coordinates and the xrange and yrange fields of the state pointer are updated accordingly. Window 1 is refreshed once more and the mouse field of the state pointer is set to zero.

The last event type that is handled in image_gui_win_events is motion events. There are two circumstances for motion events that we need to distinguish: mouse button pressed and no mouse button pressed. If no mouse button is pressed ((*pstate).mouse=0) then we do two things: (1) update the coordinates displayed in the label widgets and (2) update the cuts being displayed in windows 2 and 3 based on the cursor position. If the left mouse button is pressed ((*pstate).mouse=1) then we create the dynamically changing rubberband zoom box based on the cursor coordinates in window 1. We’ll consider the second situation first.

When the left mouse button is pressed and the cursor is being moved across window 1, events are being generated of type 2. Recall that we already have recorded where the user initially pressed the button for event.type of 0 where we updated the first positions of xbox and ybox in the state pointer. Here we wish to update the remaining two elements in xbox and ybox with the current value of the cursor position in device coordinates. Next we create the extents of the rubberband zoom box in device coordinates using the updated xbox and ybox values in the state pointer. Finally the current pixmap contents are copied to window 1 and the rubberband zoom box is plotted over this using the PLOTS procedure.

When no mouse button is pressed but the cursor is moving over window 1 then events are being generated which should lead to the data coordinates being updated in the label widgets and the cuts generated and displayed in windows 2 and 3. First we check to see if the cursor is still in the plot range using a sequence of conditional statements. If not then we return. If the cursor is in the plot range then the index into the e-vector is found as is the index into the q-vector. These are then used to display the coordinates in the label widgets. The refresh procedures are called for all windows so that the cuts can be updated with the new e and q indices. Finally the PLOTS command is used to display a cross-hairs that is displayed at the appropriate e and q values depending on the current e and q indices.

The last procedure in the program that we have not yet discussed is image_gui_change_colors which calls the program XCOLORS to allow the user to change the current color table for the image displayed in window 1. This code is remarkably short and is shown below.

```
pro image_gui_change_colors,event
widget_control,event.top,get_uvalue = pstate
this_event = tag_names(event,/structure_name)
case this_event of
'WIDGET_BUTTON':  $
   xcolors,group_leader = event.top,   $
      notifyid = [event.id,event.top]
'XCOLORS_LOAD':   image_gui_refresh1,event
else:
endcase
end
```

In this procedure there are two possible events: (1) a widget button event coming from the user pressing the button in our application labeled “Change colors” in the menu bar and (2) an event from the `XCOLORS` program that loads in a new color table. If we have the latter then we must issue a screen refresh to show the image display with the new colors. I like this operation in particular because it allows us to see the image with the new color table without needing to exit the `XCOLORS` program.

In the first case where the procedure responds to a button press, we call the `XCOLORS` program with the group_leader keyword set to event.top (the top-level base) and a keyword `notifyid` set to a 2-element long array of widget ids: the id of the button that created the event that got us to this point in the program and the id of the top-level base. This notion of using a `notifyid` keyword is very useful when you want to implement your own non-modal widgets to be called from other applications, especially if you want them to run concurrently.

Although we have listed all of the code for this application, for completeness we display the code in it entirety below.

```
; ************************************** ;
pro image_gui_cleanup,tlb
widget_control,tlb,get_uvalue = pstate
wdelete,(*pstate).winpix1
wdelete,(*pstate).winpix2
wdelete,(*pstate).winpix3
heap_free,pstate
help,/heap,/brief
end
; ************************************** ;
pro image_gui_draw1,event
widget_control,event.top,get_uvalue = pstate
e = *(*pstate).eptr
q = *(*pstate).qptr
y = *(*pstate).zptr
if (*pstate).autoscale then begin
   (*pstate).xrange = [min(e),max(e)]
   (*pstate).yrange = [min(q),max(q)]
```

```
endif
de = e[1] - e[0]
dq = q[1] - q[0]
imgxrange = [min(e)-0.5*de,max(e)+0.5*de]
imgyrange = [min(q)-0.5*dq,max(q)+0.5*dq]
plotimage,bytscl(y),imgxrange = imgxrange,   $
         imgyrange = imgyrange,              $
         xrange = (*pstate).xrange,          $
         yrange = (*pstate).yrange,          $
         xtitle = (*pstate).xtitle,          $
         ytitle = (*pstate).ytitle
end
; *************************************** ;
pro image_gui_draw2,event
widget_control,event.top,get_uvalue = pstate
ycut = (*(*pstate).zptr)[*,(*pstate).qindex]
dycut = (*(*pstate).dzptr)[*,(*pstate).qindex]
dely = max(ycut)+max(dycut)
tvlct,r,g,b,/get
tvlct,(*(*pstate).color_ptr).r,  $
      (*(*pstate).color_ptr).g,  $
      (*(*pstate).color_ptr).b
plot,*(*pstate).eptr,ycut,psym = 4, $
   xrange = (*pstate).xrange,/xsty, $
   yrange = [0.0,1.1*dely],/ysty,   $
   xtitle = (*pstate).xtitle,       $
   ytitle = (*pstate).ztitle
errplot,*(*pstate).eptr,ycut-dycut,ycut+dycut,  $
   width = 0.0
; Now plot the vertical line
plots,[(*(*pstate).eptr)[(*pstate).eindex], $
       (*(*pstate).eptr)[(*pstate).eindex]],!y.crange,/data
tvlct,r,g,b
end
; *************************************** ;
pro image_gui_draw3,event
widget_control,event.top,get_uvalue = pstate
xcut = (*(*pstate).zptr)[(*pstate).eindex,*]
dxcut = (*(*pstate).dzptr)[(*pstate).eindex,*]
q = *(*pstate).qptr
delx = max(xcut)

tvlct,r,g,b,/get
tvlct,(*(*pstate).color_ptr).r,  $
      (*(*pstate).color_ptr).g,  $
      (*(*pstate).color_ptr).b
plot,xcut,*(*pstate).qptr,psym = 4, $
   yrange = (*pstate).yrange,/ysty, $
   xrange = reverse([0.0,1.1*delx]),/xsty, $
   xtitle = (*pstate).ztitle,       $
   ytitle = (*pstate).ytitle
for i = 0,n_elements(xcut)-1 do $
   plots,[xcut[i]-dxcut[i],xcut[i]+dxcut[i]], $
         [q[i],q[i]],/data
; Now plot the vertical line
plots,!x.crange,[(*(*pstate).qptr)[(*pstate).qindex], $
       (*(*pstate).qptr)[(*pstate).qindex]],/data
```

```
tvlct,r,g,b
end
; ****************************************** ;
pro image_gui_refresh1,event
widget_control,event.top,get_uvalue = pstate
wset,(*pstate).winpix1
image_gui_draw1,event
wset,(*pstate).winvis1
device,copy = [0,0,!d.x_size,!d.y_size,   $
               0,0,(*pstate).winpix1]
end
; ****************************************** ;
pro image_gui_refresh2,event
widget_control,event.top,get_uvalue = pstate
wset,(*pstate).winpix2
image_gui_draw2,event
wset,(*pstate).winvis2
device,copy = [0,0,!d.x_size,!d.y_size,   $
               0,0,(*pstate).winpix2]
end
; ****************************************** ;
pro image_gui_refresh3,event
widget_control,event.top,get_uvalue = pstate
wset,(*pstate).winpix3
image_gui_draw3,event
wset,(*pstate).winvis3
device,copy = [0,0,!d.x_size,!d.y_size,   $
               0,0,(*pstate).winpix3]
end
; ****************************************** ;
pro image_gui_change_colors,event
widget_control,event.top,get_uvalue = pstate
this_event = tag_names(event,/structure_name)
case this_event of
'WIDGET_BUTTON':  $
   xcolors,group_leader = event.top,   $
      notifyid = [event.id,event.top]
'XCOLORS_LOAD':   image_gui_refresh1,event
else:
endcase
end
; ****************************************** ;
pro image_gui_win_events,event
widget_control,event.top,get_uvalue = pstate
if n_elements(*(*pstate).zptr) eq 0 then return
case event.type of
0: begin ; button press
      (*pstate).mouse = event.press
      if event.press eq 1 then begin
         ; Left mouse button
         (*pstate).xbox[0] = event.x
         (*pstate).ybox[0] = event.y
         image_gui_refresh1,event
         (*pstate).autoscale = 0B
      endif
   end
1: begin ; button release
```

```
      if (*pstate).mouse eq 4 then begin
         (*pstate).autoscale = 1B
         image_gui_refresh1,event
      endif
      if (*pstate).mouse eq 1 then begin
         xll = (*pstate).xbox[0] < (*pstate).xbox[1]
         yll = (*pstate).ybox[0] < (*pstate).ybox[1]
         w = abs((*pstate).xbox[1] - (*pstate).xbox[0])
         h = abs((*pstate).ybox[1] - (*pstate).ybox[0])
         xur = xll + w
         yur = yll + h
         ll = convert_coord(xll,yll,/device,/to_data)
         ur = convert_coord(xur,yur,/device,/to_data)
         (*pstate).xrange = [ll[0],ur[0]]
         (*pstate).yrange = [ll[1],ur[1]]
         image_gui_refresh1,event
      endif
      (*pstate).mouse = 0B
   end
2: begin ; motion events
      if (*pstate).mouse eq 1 then begin
         (*pstate).xbox[1] = event.x
         (*pstate).ybox[1] = event.y
         xc = [(*pstate).xbox[0],event.x,event.x,$
               (*pstate).xbox[0],$
               (*pstate).xbox[0]]
         yc = [(*pstate).ybox[0],(*pstate).ybox[0],$
               event.y,event.y,$
               (*pstate).ybox[0]]
         wset,(*pstate).winvis1
         device,copy = [0,0,!d.x_size,!d.y_size, $
            0,0,(*pstate).winpix1]
         plots,xc,yc,/device
      endif
      if (*pstate).mouse eq 0 then begin
         ; We haven't pressed a button.  The things that we
         ; need to do here are (1) display the current cursor
         ; coordinates in the widget labels and (2) generate
         ; the cuts.
         coords = convert_coord(event.x,event.y, $
            /device,/to_data)
         elo = min(*(*pstate).eptr,max = ehi)
         qlo = min(*(*pstate).qptr,max = qhi)
         condition = (coords[0] lt elo) or (coords[0] gt ehi) or $
                     (coords[1] lt qlo) or (coords[1] gt qhi)
         if condition then return
         e = *(*pstate).eptr
         q = *(*pstate).qptr
         eindex = (where(abs(e-coords[0]) eq min(abs(e-coords[0]))))[0]
         qindex = (where(abs(q-coords[1]) eq min(abs(q-coords[1]))))[0]

         xlabel = widget_info(event.top,find_by_uname = 'XLABEL')
         ylabel = widget_info(event.top,find_by_uname = 'YLABEL')
         zlabel = widget_info(event.top,find_by_uname = 'ZLABEL')
         widget_control,xlabel,set_value = 'x: '+ $
            strtrim(string(coords[0]),2)
         widget_control,ylabel,set_value = 'y: '+ $
```

```
            strtrim(string(coords[1]),2)
         widget_control,zlabel,set_value = 'z: '+  $
            strtrim(string((*(*pstate).zptr)[eindex,qindex]),2)
         (*pstate).qindex = qindex
         (*pstate).eindex = eindex
         image_gui_refresh3,event
         image_gui_refresh2,event
         image_gui_refresh1,event
         ; Plot the cross-hairs
         plots,!x.crange,replicate((*(*pstate).qptr)[qindex],2),/data
         plots,replicate((*(*pstate).eptr)[eindex],2),!y.crange,/data
      endif
   end
else:
endcase
end
; *************************************** ;
pro image_gui_load_data,event
widget_control,event.top,get_uvalue = pstate
filename = dialog_pickfile(dialog_parent = event.top, $
   filter = '*.sav',path = sourceroot())
if ~file_test(filename) then return
restore,filename = filename
*(*pstate).zptr = y
*(*pstate).dzptr = dy
*(*pstate).eptr = e
*(*pstate).qptr = q
image_gui_refresh3,event
image_gui_refresh2,event
image_gui_refresh1,event
end
; *************************************** ;
pro image_gui_show_surf,event
; Use FSC_SURFACE to display an interactive
; fully orientable 3D surface representation
; of the data.
widget_control,event.top,get_uvalue = pstate
if n_elements(*(*pstate).zptr) eq 0 then return
z = *(*pstate).zptr
x = *(*pstate).eptr
y = *(*pstate).qptr
nx = n_elements(x) & ny = n_elements(y)
; For aesthetic reasons, display the data as a square
; matrix.  Use congrid for the interpolation.
if nx ge ny then begin
   xi = x
   yi = makepts(xlo = min(y),xhi = max(y),npts = nx)
   zi = congrid(z,nx,nx,/cubic)
endif else begin
   xi = makepts(xlo = min(x),xhi = max(x),npts = ny)
   yi = y
   zi = congrid(z,ny,ny,/cubic)
endelse
fsc_surface, zi, xi, yi, group_leader = event.top, $
   /shaded,/elevation_shading,                  $
   xtitle = (*pstate).xtitle,                   $
   ytitle = (*pstate).ytitle,                   $
   colortable = 15
end
; **************************************** ;
pro image_gui_event,event
uname = widget_info(event.id,/uname)
case uname of
'LOAD':           image_gui_load_data,event
'QUIT':           widget_control,event.top,/destroy
'WIN':            image_gui_win_events,event
'CHANGE_COLORS':  image_gui_change_colors,event
'SHOW_SURF':      image_gui_show_surf,event
else:
endcase
end
; **************************************** ;
pro image_gui
; Widget creation module
; Initialize the color table to blue-white
device,decomposed = 0
loadct,1,/silent
; Load the colors into variables that we'll use
; to "color-protect" the cut data: i.e. we wish
; to maintain the white on black appearance of
; the cut plots but not affect the color table
; of the image plot.
tvlct,r,g,b,/get
color_ptr = ptr_new({r:r,g:g,b:b},/no_copy)
; Get the screen size to make the draw widgets
; appropriately sized.
device,get_screen_size = screen_size
; Define the sizes.  And just for fun we'll use the
; Golden Ratio to determine the size of the
; windows into which the cuts will be displayed.
golden_ratio = 0.5*(1.0+sqrt(5.))
xsize1 = (ysize1 = fix(0.35*screen_size[1]))
xsize2 = xsize1 & ysize2 = fix(ysize1/golden_ratio)
ysize3 = ysize1 & xsize3 = fix(xsize1/golden_ratio)
; Create the top-level base, make it a row-base, and
; specify that we want a menu bar
tlb = widget_base(/row,title = 'Data Display Application', $
   mbar = bar)
; Create the filemenu
filemenu = widget_button(bar,value = 'File',/menu)
void = widget_button(filemenu,value = 'Load data',uname = 'LOAD')
void = widget_button(filemenu,value = 'Change colors',uname =
'CHANGE_COLORS')
void = widget_button(filemenu,value = 'Surface representation',uname =
'SHOW_SURF')
void = widget_button(filemenu,value = 'Quit',uname = 'QUIT')
; Create the first column base which holds window 3
; and the label widget that will display the cursor
; coordinates in data units.
col1 = widget_base(tlb,/col)
win3 = widget_draw(col1,xsize = xsize3,ysize = ysize3)
xlabel = widget_label(col1,value = 'x: ',/dynamic_resize, $
   uname = 'XLABEL',/align_left)
ylabel = widget_label(col1,value = 'y: ',/dynamic_resize, $
   uname = 'YLABEL',/align_left)
zlabel = widget_label(col1,value = 'z: ',/dynamic_resize, $
   uname = 'ZLABEL',/align_left)
col2 = widget_base(tlb,/col)
win1 = widget_draw(col2,xsize = xsize1,ysize = ysize1,   $
   /motion_events,/button_events,uname = 'WIN')
win2 = widget_draw(col2,xsize = xsize2,ysize = ysize2)


; Let's center the top-level base on the screen
geom = widget_info(tlb,/geometry)
widget_control,tlb,  $
   xoff = (screen_size[0]/2)-(geom.scr_xsize/2),   $
   yoff = (screen_size[1]/2)-(geom.scr_ysize/2)
widget_control,tlb,/realize

; Get the window ids out and create the pixmaps
widget_control,win1,get_value = winvis1
widget_control,win2,get_value = winvis2
widget_control,win3,get_value = winvis3
window,/free,/pixmap,xsize = xsize1,ysize = ysize1
winpix1 = !d.window
window,/free,/pixmap,xsize = xsize2,ysize = ysize2
winpix2 = !d.window
window,/free,/pixmap,xsize = xsize3,ysize = ysize3
winpix3 = !d.window

; Initialize the pointers
zptr = ptr_new(/allocate_heap)
dzptr = ptr_new(/allocate_heap)
eptr = ptr_new(/allocate_heap)
qptr = ptr_new(/allocate_heap)

state = {   winvis1:winvis1,              $
            winvis2:winvis2,              $
            winvis3:winvis3,              $
            winpix1:winpix1,              $
            winpix2:winpix2,              $
            winpix3:winpix3,              $
            color_ptr:color_ptr,          $
            xtitle:'E (meV)',             $
            ytitle:'Q (!3!sA!r!u!9 %!3!n!E-1!N)',  $
            ztitle:'Intensity',           $

            mouse:0B,                     $
            autoscale:1B,                 $
            xbox:[0,0],                   $
            ybox:[0,0],                   $
            xrange:[0.,0.],               $
            yrange:[0.,0.],               $
            eindex:0,                     $
            qindex:0,                     $
            zptr:zptr,                    $
            dzptr:dzptr,                  $
            eptr:eptr,                    $
            qptr:qptr                     }
```

```
pstate = ptr_new(state,/no_copy)
widget_control,tlb,set_uvalue = pstate,/no_copy

xmanager,'image_gui',tlb,              $
   event_handler = 'image_gui_event',  $
   cleanup = 'image_gui_cleanup',      $
   /no_block

end
```

A screen shot of this application is shown below in Figure 29.

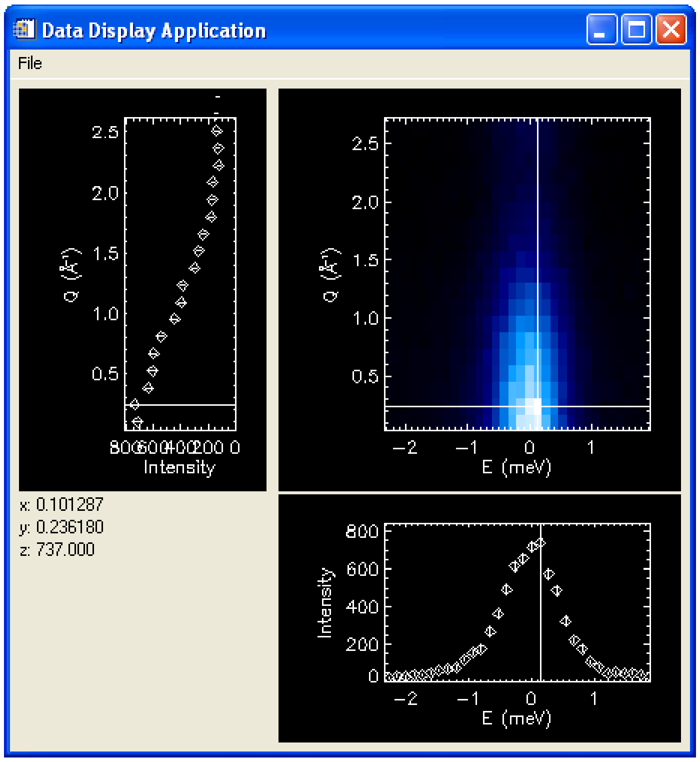

Figure 29 Screen shot of the application with data loaded.

Exercise: Add the ability to display the data as a 3-D surface using the procedure called FSC_SURFACE written by David Fanning. Give users of your application access to this functionality via the menu bar on the top-level base.

Chapter 9-Writing a multi-module GUI application

Up until this point we have discussed how to create a single application. However you may wish to distribute a number of "standalone applications" in one program suite. The DAVE suite of neutron scattering applications is an example. In this brief chapter we will show how to create a simple program that launches the widget programs developed thus far.

Design specifications and functional requirements

The new application, `prog_menu.pro`, will expose a menu of buttons which allow the user to launch `image_gui` and `widget_ex_3`. These programs can be run simultaneously. A QUIT button will also close all applications.

Implementation plan

The outline of the code for this launcher program is straightforward. There are two procedures necessary: a widget creation module and an event handler to handle all of the possible button events. In this program these two modules are named `prog_menu` and `prog_menu_event`, respectively.

We *could* just launch the two programs with no programmatic link between the launcher and the individual applications. The code for that situation is shown below:

```
; ******************************** ;
pro prog_menu_event,event
case widget_info(event.id,/uname) of
'QUIT':        widget_control,event.top,/destroy
'IMAGE_GUI':   image_gui
'RANDOM_GUI':  widget_ex_3
else:
endcase
end
; ******************************** ;
pro prog_menu
; Widget creation module
tlb = widget_base(/col,title = 'Multi-Apps')
void = widget_button(tlb,value = 'Image GUI', $
   uname = 'IMAGE_GUI')
void = widget_button(tlb,value = 'Random Data Display',  $
   uname = 'RANDOM_GUI')
void = widget_button(tlb,value = 'Quit',uname = 'QUIT')
centertlb,tlb
```

```
widget_control,tlb,/realize

xmanager,'prog_menu',tlb,  $
   event_handler = 'prog_menu_event',/no_block
end
```

This is a standard widget program and you should be comfortable with everything in it. The main event handler distinguishes between button identities based on the `uname` as usual. We actually use a nice widget utility here called `centertlb.pro` that centers the top-level base on the screen. The source code for `centertlb` is listed below.

```
pro centertlb,tlb
; This procedure centers the entire widget (the
; top-level-base) on the screen (independent of
; the platform).
device,get_screen_size = screenSize
geom = widget_info(tlb,/geometry)
widget_control,tlb,  $
   xoff = (screenSize[0]/2)-(geom.scr_xsize/2),$
   yoff = (screenSize[1]/2)-(geom.scr_ysize/2)
end
```

When you press the Image GUI button the event handler just calls the `IMAGE_GUI` procedure. The same thing happens with the Random Data Display button. Run the `prog_menu` program and launch `image_gui`. Now press the quit button in `prog_menu`. You will notice the undesired behavior that `image_gui` does not close. At the moment the programs are completely independent even though one *calls* the other. There is no programmatic means by which the programs are linked.

We can link the programs using the `group_leader` keyword. However we must modify the existing `image_gui` and `widget_ex_3` programs to accomplish this. Open up both `image_gui` and `widget_ex_3` and make the following changes to their respective widget creation modules.

```
pro image_gui,group_leader = group_leader
if n_elements(group_leader) eq 0 then group_leader = 0L
.
.
tlb = widget_base(/row,title = 'Data Display Application', $
   mbar = bar,group_leader = group_leader)
.
.
.
```

```
pro widget_ex_3,group_leader = group_leader
if n_elements(group_leader) eq 0 then group_leader = 0L
tlb = widget_base(/row,title = '',group_leader = group_leader)
.
.
.
```

The additional keyword for these programs, `group_leader`, tells IDL to specify a group leader (widget ID) at the top of the widget hierarchy. In our case the group leader is the top-level base for `prog_menu`. If the user does not pass in a `group_leader` then the default is set to 0L (long zero). This is never a valid widget ID and its inclusion in the individual top-level base's function call is swallowed without an exception.

The necessary modification to the event handler for `prog_menu` is as follows:

```
pro prog_menu_event,event
case widget_info(event.id,/uname) of
'QUIT':  widget_control,event.top,/destroy
'IMAGE_GUI':   $
   image_gui,group_leader = event.top
'RANDOM_GUI':  $
   widget_ex_3,group_leader = event.top
else:
endcase
end
```

Now run `prog_menu` and then press the quit button in `prog_menu`. You should see all applications disappear which is the desired behavior using the `group_leader` keyword.

Suppressing multiple instances of an application

If you press the buttons in the `prog_menu` GUI for each application you will find that you can quickly create many simultaneous instances of the same program. This may not be the behavior you wish. For instance you may wish to allow only a single instance of each program during a session. This can be done using the `XREGISTERED` function. We will show how to do this for `widget_ex_3`.

```
pro widget_ex_3,group_leader = group_leader
if n_elements(group_leader) eq 0 then group_leader = 0L
if xregistered('widget_ex_3') then return
.
.
.
```

The addition of the line in boldface to `widget_ex_3` is all that is necessary to ensure that you cannot launch more than one instance of this particular application. The argument in the `XREGISTERED` function is the name of the widget program. Note that this is the same name as used as the first argument in the call to `XMANAGER`.

Exercise: Modify `image_gui.pro` so that you cannot launch more than one instance of it from `prog_menu`.

Chapter 10-Plugging your application into DAVE

What is DAVE?

DAVE (the Data Analysis and Visualization Environment-version 1.x) is mainly a collection of (almost independent) application modules that are glued together through a single main widget application. The DAVE application suite can be encapsulated in the following statements:

1. A basic framework that allows applications to be easily plugged in.
2. All the main functionality (such as data reduction and visualization tasks) are implemented as modules. Little or no constraints are placed on the design and operation of the modules. This decision was made in order to foster the creation of new functionality within DAVE, which at conception had only a few developers working on the project in their spare time.
3. Applications within the framework can share, or have access to, a single dataset. This dataset is created and owned by the main framework even though it can be read or modified by any interested module.

There are some shortcomings to this design (such as the single dataset limitation, lack of a standard look and feel, and some duplication of effort) which will be addressed in a future version of the application. Nevertheless, DAVE has experienced a tremendous growth in features over a brief timescale since its inception in 2001 as a result of the initial decisions that were made.

Adding your IDL application to DAVE

There are three main steps necessary to include a functioning application into DAVE.

1. Modifications to the application, necessary for communicating with DAVE.
2. Write a launch procedure that acts as a bridge between DAVE and the module. The launch procedure follows a well defined API and is only a few lines of code.
3. Insert a new menu entry in DAVE for launching the application. This is also a couple of lines of code.

Modifications to the application

The following changes to your IDL application are necessary to facilitate incorporation into DAVE:

- o Event passing – the module needs to be able to send events to others (usually an exit event). We will not demonstrate this in this chapter.
- o Data sharing – where applicable, the module needs to view or even modify the DAVE dataset.
- o Group leader widget ID in the case of a GUI application.

The required changes include additional parameters and/or keywords to the main entry procedure or function for the application.

As an example, we will incorporate our application `IMAGE_GUI` into DAVE so that it can display DAVE data. We will add a menu item to the menu bar which allows us to load in a DAVE data set (rather than reading the fake data set). Note that in DAVE there are two menu options that can be exposed to the user. In the "classic" DAVE the menu selections are composed of buttons in the menu bar in the DAVE interface. In "tree-view" DAVE the menu selections are tree widget leaves. `DAVE.PRO` has two menu-building modules depending on the preference of the user: `DAVE_BUILD_MENU` and `DAVE_BUILD_TREE_MENU`. We will need to add code to both of those sections. In the `DAVE.PRO` module we would add the following line (shown in boldface) in the procedure named `DAVE_BUILD_MENU`:

```
.
.
visMenu = widget_button(bar, value = 'Visualization', /menu)
   twoDRen = widget_button(visMenu, value = 'Trifenestra (Three Windows)' $
     , event_pro = 'fdaveTAS', uvalue='PSD')
   void = widget_button(visMenu,value = 'Simple image slicer',event_pro = 'daveLaunchIS')
        void = widget_button(visMenu,value='Data Browser',event_pro='launch_dave_db')
   void = widget_button(visMenu,value = 'DAVE PEEK',event_pro = 'launch_dave_peek')
   void = widget_button(visMenu,value = 'Image Viewer',event_pro = 'launch_image_gui')
.
.
```

Similarly we add the following line to the procedure in `DAVE.PRO` named `DAVE_BUILD_TREE_MENU`:

```
.
.
visualization = widget_tree(file_tree,value = 'Data Visualization',/folder,$
   event_pro = 'dave_do_nothing',bitmap = vis_icon)
void = widget_tree(visualization,value = 'Trifenestra (Three Windows)',$
   event_pro = 'fdaveTAS',uvalue = 'PSD',bitmap = app_icon)
void = widget_tree(visualization,value = 'Simple image slicer',$
   event_pro = 'daveLaunchIS',bitmap = app_icon)
void = widget_tree(visualization,value = 'Data Browser',$
   event_pro = 'launch_dave_db',bitmap = app_icon)
void = widget_tree(visualization,value = 'DAVE PEEK',$
   event_pro = 'launch_dave_peek',bitmap = app_icon)
void = widget_tree(visualization,value = 'Image Viewer',$
   event_pro = 'launch_image_gui',bitmap = app_icon)
.
.
```

The launcher application, `launch_image_gui`, is a simple event handler that launches `IMAGE_GUI` with the appropriate arguments. The code for it is shown below. Note that we used the existing group_leader keyword in `IMAGE_GUI` so that the group leader for our application is the top-level base of the DAVE application. This ensures that our application will be closed if the DAVE application is closed (while both are running). We also added the keyword `notify_ids` which we saw when we implemented the `XCOLORS` program into `IMAGE_GUI`. Again `notify_ids` is a two-element long array composed of a widget id for the widget that got us into the event handler and the second element is the top-level base in `DAVE.PRO`.

```
pro launch_image_gui,event
thisEvent = tag_names(event,/structure_name)
case thisEvent of
'WIDGET_BUTTON': $
   image_gui,  group_leader = event.top,        $
               notify_ids = [event.id,event.top]

'WIDGET_TREE_SEL': $
   image_gui,  group_leader = event.top,        $
               notify_ids = [event.id,event.top]
else:
endcase
end
```

In addition to adding the `notify_ids` keyword to the procedure's interface we also add this keyword to the state structure so that it is available throughout this application. Although we will not do it here, it is possible to send information from this application that is "plugged into" DAVE up to DAVE so that it is available for other DAVE applications. Therefore the addition of `notify_ids` is not strictly necessary in our case. The modifications to the widget creation module in `IMAGE_GUI` regarding these changes are shown below.

```
; ****************************************** ;
pro image_gui,group_leader = group_leader, notify_ids = notify_ids
if n_elements(notify_ids) ne 2 then notify_ids = [0L,0L]
if n_elements(group_leader) eq 0 then group_leader = 0L
if xregistered('image_gui') then return
; Widget creation module
; Initialize the color table to blue-white
device,decomposed = 0
loadct,1,/silent
.
.
state = {   winvis1:winvis1,             $
            winvis2:winvis2,             $
            winvis3:winvis3,             $
            winpix1:winpix1,             $
            winpix2:winpix2,             $
            winpix3:winpix3,             $
            color_ptr:color_ptr,         $
            xtitle:'E (meV)',            $
            ytitle:'Q (!3!sA!r!u!9 %!3!n!E-1!N)',  $
            ztitle:'Intensity',          $

            mouse:0B,                    $
            autoscale:1B,                $
            xbox:[0,0],               $
            ybox:[0,0],               $
            xrange:[0.,0.],           $
            yrange:[0.,0.],           $
            eindex:0,                 $
            qindex:0,                 $
            zptr:zptr,                $
            dzptr:dzptr,              $
            eptr:eptr,                $
            qptr:qptr,                $
            notify_ids:notify_ids     }
.
.
```

At this point the application `IMAGE_GUI` will run just fine as a part of DAVE. However we cannot read in any DAVE data into `IMAGE_GUI` which would make it truly useful. To do so we need to add a new procedure into `IMAGE_GUI` that allows us to load in a DAVE data set.

First we need to include a new menu option to load in a DAVE file. In the widget creation module for `IMAGE_GUI`, add the following line shown below in boldface.

```
.
.
filemenu = widget_button(bar,value = 'File',/menu)
void = widget_button(filemenu,value = 'Load data',uname = 'LOAD')
void = widget_button(filemenu,value = 'Load DAVE data',uname = 'LOAD_DAVE')
void = widget_button(filemenu,value = 'Change colors',uname = 'CHANGE_COLORS')
.
.
```

We also must update the event handler for `IMAGE_GUI` as follows:

```
pro image_gui_event,event
uname = widget_info(event.id,/uname)
case uname of
'LOAD':           image_gui_load_data,event
'QUIT':           widget_control,event.top,/destroy
'WIN':            image_gui_win_events,event
'CHANGE_COLORS':  image_gui_change_colors,event
'SHOW_SURF':      image_gui_show_surf,event
'LOAD_DAVE':      image_gui_load_dave,event
else:
endcase
end
```

Finally we write the procedure that loads in the DAVE data, `image_gui_load_dave`. In this procedure we query the user for a DAVE data file using the `DIALOG_PICKFILE` routine, restore this file, extract the contents of the DAVE data file using a function called `GET_DAVE_PTR_CONTENTS`, fill up the local variables with the values from the DAVE pointer and then plot the data.

```
pro image_gui_load_dave,event
widget_control,event.top,get_uvalue = pstate
```

```
filename = dialog_pickfile(dialog_parent = event.top, $
           path = sourceroot(), filter = '*.dave',    $
           /read,title = 'Please select a DAVE data file')
if ~file_test(filename) then return
restore,filename
; The statement above restores a variable called
; DAVEPTR.  We can extract the variables for
; displaying using a function called
; GET_DAVE_PTR_CONTENTS.
ret_val = get_dave_ptr_contents(daveptr,  $
          qty = z,                        $
          err = dz,                       $
          yvals = q,                      $
          xvals = e,                      $
          xlabel = xlabel,                $
          ylabel = ylabel                 )
; Free the DAVE pointer to prevent memory
; leaks.
heap_free,daveptr
*(*pstate).eptr = e
*(*pstate).qptr = q
*(*pstate).zptr = z
*(*pstate).dzptr = dz
image_gui_refresh3,event
image_gui_refresh2,event
image_gui_refresh1,event
end
```

Now you should be able to run this application from DAVE, load in DAVE data, and visualize it.

More details of the DAVE pointer

In the last section we saw how to read in a DAVE formatted data file that was saved to disk without knowing any of the details of the data structure. In general there are a few simple functions that allow reading/writing information from/to the DAVE data files. In this section we will go into some depth to describe the structure.

DAVE 1.x uses a simple data structure to store experimental data. The structure was conceived early on in the development of DAVE and at the time emphasis was placed on ensuring that the *plottable data* is well represented. The complete data structure is created and saved on file as an IDL pointer (heap variable) hence it is commonly referred to as a *davePtr* (pronounced DAVE pointer). DAVE 1.x files are saved using IDL's sav binary format.

Essentially several data structures have been defined to store the relevant information within the DAVE pointer. The complete structure is outlined in the diagram below using the following rules.

- Each rectangular shape contains the definition of an IDL structure.
- Structure member names are in the left column (in blue).
- Their IDL data type and description are in the right column.

- Highlighted structures member names (bold-underlined) are themselves structures or pointers to structures.

A hierarchical description of the structure now follows.

davePtr

The `davePtr` points to an IDL structure consisting of two other pointers: ***dataStrPtr*** and ***descriPtr***. The experimental data and all relevant metadata are stored in the `dataStrPtr`. The `descriPtr` is used for labeling or tagging the dataset.

descriPtr

The `descriPtr` points to a structure as described in the diagram. It is used for labeling or tagging the `davePtr`. This tagging can be user-defined and consists of a name, description and value. For example a user might make several measurements of a particular sample at different temperatures. The `descriPtr` may be used to identify these measurements as part of a series. In this case the `(*descriPtr).name` and `(*descriPtr).units` fields for all measurements could be 'Temperature' and 'K' respectively while the actual temperature for each measurement would be stored in the corresponding `(*descriPtr).qty` field. Although the `descriPtr` is optional, if present some applications within DAVE do make use of it. For example the Data Browser (visualization module) is able to combine datasets that belong to the same series to produce a composite dataset.

davePtr

dataStrPtr	ptr; experimental data
descriPtr	ptr; arbitrary label/tag for this data

dataStrPtr

commonStr	struct; plottable data + attributes
specificPtr	ptr; instrument dependent data+metadata

descriPtr

name	string; name of tag eg 'Temperature'
units	string; units eg 'K'
legend	string; descriptive info eg 'Sample Temp'
qty	float; value of variable eg 250.0
err	float; uncertainty in value eg 0.5

commonStr

histPtr	ptr; to data (dependent+independent)
treatmentPtr	ptr; to string array of data treatment history
instrument	string; instrument name eg 'DCS'
xtype	string; 'points' | 'histogram'
xlabel	string; x-axis label eg 'Energy Transfer'
xunits	string; x-axis units eg 'meV'
ytype	string; 'points' | 'histogram'
ylabel	string; y-axis label eg 'Wavevector'
yunits	string; y-axis units eg 'A-1'
histLabel	string; data label eg 'S(Q,w)'
histUnits	string; data units eg 'Arbitrary units'

specificPtr

ptr to an arbitrary structure. Contains any instrument specific data/metadata.

histPtr

qty	float[nx,ny]; data, counts, S(q,w) etc
err	float[nx,ny]; uncertainty in qty
x	float[nx']; x-axis data
y	float[ny']; y-axis data

dataStrPtr

The experimental data is contained within the `dataStrPtr`. The `dataStrPtr` points to a structure consisting of two fields: *commonStr* and *specificPtr*. `commonStr` contains the plottable data and is the only mandatory component of the `davePtr`. The `specificPtr` is optional and is included as a placeholder for any instrument specific information.

specificPtr

Points to an instrument-dependent structure that can be used to store any relevant information about that instrument. The definition of the structure is unspecified and hence any information stored here can only be useful for instrument specific modules.

commonStr

The `commonStr` is an IDL structure designed to hold plotable data and additional basic attributes required to display it. A structure member, *histPtr,* holds the data itself with the remaining fields storing various useful attributes for displaying the data. The meaning of the attributes are easily understood as described in the diagram. The `treatmentPtr` points to a string array of arbitrary length that describes all the treatment details applied so far to the DAVE pointer. Hence, all program modules that modify a DAVE pointer should ensure that the `treatmentPtr` is updated accordingly with concise human readable textual information about the modification.

histPtr

The actual data is stored in the `histPtr`, a four member structure as indicated in the diagram. The dependent variable (eg counts) is stored in the `qty` field and the associated uncertainty in `err`. The first independent variable is stored in the `x` field. If *qty* is two dimensional then the second independent variable is stored in the `y` field. The dimensions of the fields should obey these rules:

histPtr field name	***dimension***	***dimension scale***
qty	1D or 2D	(nx) or (nx,ny)
err	1D or 2D	(nx) or (nx,ny)
x	1D	nx' where nx'=nx; if commonStr.xtype='points' nx'=nx+1; if commonStr.xtype='histogram'
y	1D	ny' where ny'=ny; if commonStr.ytype='points' ny'=ny+1; if commonStr.ytype='histogram'

Working with the davePtr

Once you become familiar with the `davePtr` structure, it is quite straightforward to directly manipulate it's contents. Simply remember:

- the syntax for accessing structure members (structure.field).
- a pointer variable has to be dereferenced to access its contents (all pointers within the DAVE pointer have a Ptr suffix).
- to be careful about inadvertently deleting heap memory within the dataset. For example never directly assign a local pointer to one in the data structure and then subsequently freeing the local pointer – this would result in deletion of content from the dataset.

A few simple examples will now be given to illustrate reading/writing from/to a DAVE pointer. In all examples it will be assume that the DAVE pointer is available as a locale variable called `davePtr` – the internal structure will already be properly defined for you. A separate document is available that deals with attaching an application module to the DAVE suite , obtaining a reference to the DAVE pointer and reading/writing to a file.

Example 1

Reading the plottable data (counts, errors, independent variable(s)) .

```
IDL>data  = (*(*(*davePtr).dataStrPtr).commonStr.histPtr).qty
IDL>error = (*(*(*davePtr).dataStrPtr).commonStr.histPtr).err
IDL>xval  = (*(*(*davePtr).dataStrPtr).commonStr.histPtr).x
IDL>yval  = (*(*(*davePtr).dataStrPtr).commonStr.histPtr).y
```

The local variables `data`, `error`, `xval` and `yval` should now contain *copies* of the data, associated uncertainties, x- and y-axis data. Note that if data is 1D then `yval` will be scalar (value 0.0) instead of a vector.

Example 2

Reading the treatment history. This contains an account of the data processing that the dataset has an undergone.

```
history = (*(*(*davePtr).dataStrPtr).commonStr.treatmentPtr)
```

The local variable, `history`, should contain the treatment history as a string array.

Example 3

Obtaining plot attributes.

```
IDL>instr_name = (*(*davePtr).dataStrPtr).commonStr.instrument
IDL>xtype = (*(*davePtr).dataStrPtr).commonStr.xtype
IDL>xlabel = (*(*davePtr).dataStrPtr).commonStr.xlabel
IDL>xunits = (*(*davePtr).dataStrPtr).commonStr.xunits
```

`xtype` will be 'points' or 'histo*gram', etc

Example 4
Updating data. counts and xdata are local variables containing the new data. Can either make direct assignments like

```
IDL>(*(*(*davePtr).dataStrPtr).commonStr.histPtr).qty = counts
IDL>(*(*(*davePtr).dataStrPtr).commonStr.histPtr).x = xdata
```

or (to emphasize a point) make use of an intermediate local variable for `histPtr`

```
IDL>local_histPtr = (*(*davePtr).dataStrPtr).commonStr.histPtr
IDL>(*local_histPtr).qty = counts
IDL>(*local_histPtr).x = xdata
```

Both methods are equivalent. However, in the second case, the local variable `local_histPtr` should not be freed (ie don't use: `ptr_free, local_histPtr`) since this is the same heap memory that is being used by the `histPtr` field within `davePtr`.

Example 5
Appending information to the `treatmentPtr`. This should be done whenever a noteworthy modification is made to the dataset.

```
IDL>old_rec = (*(*(*davePtr).dataStrPtr).commonStr.treatmentPtr)
IDL>new_rec = ['First additional line – counts scaled by 2',
              'Second line as required, etc',
              '-------------------------------------------']
IDL>new_rec = [old_rec,new_rec]
IDL>(*(*(*davePtr).dataStrPtr).commonStr.treatmentPtr) = new_rec
```

String arrays are used for storing treatment information. The style is arbitrary but the language should be clear and concise.

While it is straightforward to work with the DAVE pointer, the syntax is very verbose and as such it is easy to make mistakes. For these reasons, a set of functions have been created for writing to and reading from a DAVE pointer. You have already used `GET_DAVE_PTR_CONTENTS`. Brief references for these functions now follow.

```
function create_dave_pointer, davePtr, instrument=instrument, qty=qty, qtunits=qtunits, $
qtlabel=qtlabel, err=err, xvals=xvals, xtype=xtype, xunits=xunits, xlabel=xlabel, yvals=yvals, $
ytype=ytype, yunits=yunits, ylabel=ylabel, specificstr=specificstr, treatment=treatment, $
dname=dname, dunits=dunits, dlegend=dlegend, dqty=dqty, derr=derr, ermsg=errmsg
```

- all keywords except `ermsg` specify input quantities
- `davePtr` parameter and `qty` keyword are required

```
function get_dave_pointer_contents, davePtr, instrument=instrument, qty=qty, qtunits=qtunits, $
qtlabel=qtlabel, err=err, xvals=xvals, xtype=xtype, xunits=xunits, xlabel=xlabel, yvals=yvals, $
ytype=ytype, yunits=yunits, ylabel=ylabel, specificstr=specificstr, treatment=treatment, $
dname=dname, dunits=dunits, dlegend=dlegend, dqty=dqty, derr=derr, ermsg=errmsg
```

- all keywords specify output quantities
- `davePtr` parameter is required

Both functions return a 1 if successful or 0 otherwise. Meaning of parameter/keywords:

`davePtr`	DAVE pointer. For `create_dave_pointer()` it can either be an input or output parameter. For `get_dave_pointer_contents()` it is an input parameter. `davePtr` must be a valid DAVE pointer when it is used as an input parameter.
`instrument`	string variable describing instrument.
`qty`	double or float array containing the data variable
`qtunits`	string variable specifying the units of `qty`
`qtlabel`	string variable specifying the plot label for `qty`
`err`	double of float array containing the error associated with `qty`. It must have the same dimension and size as `qty`.
`xvals`	double or float array containing the first independent variable for `qty`. If not provided then a simple index array is determined based on the size of `qty`.
`xtype`	string variable specifying if the first independent variable, `xvals`, is "POINTS" or "HISTOGRAM".
`xunits`	string variable specifying the units of `xvals`.
`xlabel`	string variable specifying the plot label for the first independent variable, `xvals`.
`yvals`	y-axis equivalent of `xvals`
`ytype`	y-axis equivalent of `xtype`
`yunits`	y-axis equivalent of `xunits`
`ylabel`	y-axis equivalent of `xlabel`
`specificstr`	structure containing any instrument specific information to be include in the DAVE pointer. Can contain any variable type in its fields.
`treatment`	string array of any length containing the treatment of the data.
`dname`	string name of the `davePtr` tag (`descriPtr`).
`dunits`	string units of the `davePtr` tag (`descriPtr`).
`dlegend`	string description of the `davePtr` tag (`descriPtr`).
`dqty`	value of the `davePtr` tag (`descriPtr`).
`derr`	uncertainty in the value of the `davePtr` tag (`descriPtr`).
`ermsg`	output keyword. Contains error message if the function is unsuccessful.