Application Development in IDL II

R.M. Dimeo
NIST Center for Neutron Research

CONTENTS

INTRODUCTION

In this course we will be discussing a number of the more advanced topics in IDL programming. The end goal is to learn how to write complex multi-module applications using widget programming and facilitate effective communication between widgets. In the process of achieving the end goal we will cover some topics that can be troublesome in writing effective widget programs. We will begin with coverage of topics in conventional widget programming, the basis for the course titled Application Development in IDL I. Then we will discuss object programming in IDL and how to create widget objects. Finally the material will be tied together with a discussion of how to combine it all into a single multi-module application. Some of the detailed topics that we will cover will be:

- Allowing (forbidding) user to run more than one instance of your widget application
- Writing widget applications with resizable bases
- Circumventing pitfalls associated with incorporating two zoomable windows into a single widget application
- Differences between modal and non-modal dialog widgets
- Creating a HIDE/SHOW feature for detached control panels
- Why you might want to make multiple calls to the XMANAGER
- Overview of object programming in IDL
- Writing your widget application as an object
- Operating your widget object from the command line
- Creating a command line interface for your widget object allowing complete control without using widget controls
- A simplified non-modal widget program dialog using objects
- Tying it all together into a single application
- Getting your application to work in DAVE

All of the topics listed above will be described using code that we will be running in class. Although we will be covering a lot of ground, we will not be typing a lot of the code. In fact, most of it is contained in this set of notes. Rather we will run the code and discuss the features of the programs at length in class. From time to time I will ask you to do the exercises detailed in these notes which extend the code to add another feature. This will usually involve only a few lines of code. The goal is NOT for you to get tired hands! This will be a highly interactive class so please feel free to ask questions at any time.

You should download the programs used in the course via ftp at the following address:
ftp://ftp.ncnr.nist.gov/pub/staff/dimeo/idlprogs2.zip

There are numerous resources on-line containing IDL code. Some of the sites that I find very useful are listed below:

www.dfanning.com
David Fanning's extremely useful site has lots of widget programs, widget objects, code, and many useful technical tips. He has written an outstanding book about programming in IDL called IDL Programming Techniques.

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 a couple of nice widget and object graphics books: Application Development with IDL and Power Graphics with IDL. His web site contains freeware and code you can buy written in IDL.

cimss.ssec.wisc.edu/~gumley/index.html
This web site contains a number of useful IDL programs. Perhaps one of the most useful is IMDISP.PRO which is a general purpose image display program that takes care of the "dirty little details" of image display such as proper color and generating nice postscript output. Liam Gumley has also written a nice introductory-level book on IDL called Practical IDL Programming.

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

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

CONVENTIONAL WIDGET PROGRAMMING

§ A widget application for solving the 1-d Schrödinger Equation

We will review a simple widget application called SEUTILA.PRO that accepts user input for a 1-dimensional potential and solves the corresponding Schrödinger equation. The utility itself is not so important but the details in programming the application are important. We will examine this program here in broad strokes to acquaint ourselves with the paradigm of a conventional widget program. Note that when we use the term "conventional widget program" we are implying that no objects are involved. The combination of objects and widgets will be discussed later.

SEUTILA.PRO has been included in the software distribution for this course. Open it up, compile, and run it now. You should see an interface like that shown in figure 1.

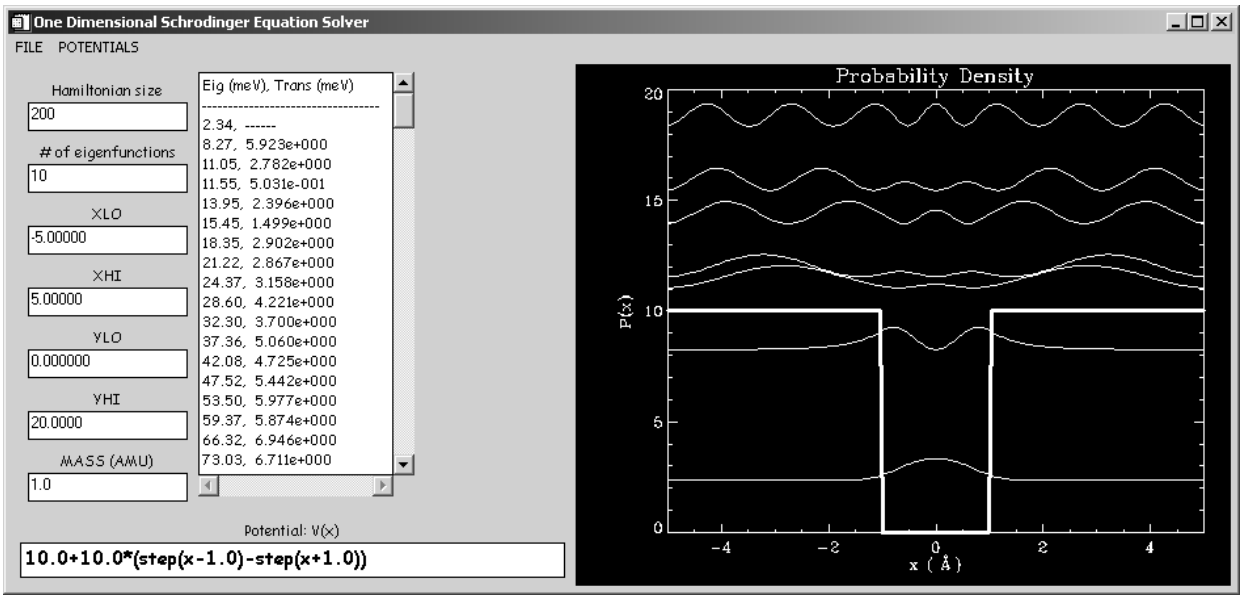

Fig. 1 Interface for SEUTILA.

We will not go into the details as to how this program solves the Schrödinger equation but rather concern ourselves with the logic behind this program. As you will recall applications written with widgets are **event-driven programs**. That is, they are in a perpetual wait state, waiting for user input, when events will be generated. Every element that you see on the interface in figure 1 (the text boxes, draw windows, and menu items) is called a **widget**, a control with which the user can interact. Play around with the program a bit to familiarize yourself with its capabilities. For instance, select a new pre-defined potential from the menu item named `POTENTIALS`. The program will

seem to pause for a moment while it calculates the new eigenvalues, transitions, and eigenfunctions. Also try typing in your own potential in the text field labeled `Potential: V(x)`. Make sure you hit a carriage return `<CR>` after typing in any of the text fields. This will initiate recalculation of the data. Use the mouse to zoom in and out of the display, too. A left mouse button press and hold allows you to drag the mouse to define a "zoombox". Release the mouse button to complete the process. A right mouse button press will autoscale the display. Finally you should be able to resize the application by dragging a corner or side of the interface. So, how does the program do all of this?

If you are a seasoned IDL widget programmer the implementation of these features should be pretty straightforward. For future reference and review, we will discuss the implementation of some of these features and how this widget program works.

There are 13 procedures and 2 functions in this application. The 2 functions are required for solution to the Schrödinger equation so will not be discussed here. The procedures are listed below:

```
seCleanup
seQuit
sePlot
seSolve
seDraw
seEvents
seDefaults
seLoadSHO
seLoadSquareWell
seLoadLinearWell
seLoadQuarticWell
seLoadDoubleWell
seutila
```

The two essential components to every widget application are (1) the widget definition module and (2) at least one event handler. The widget definition module here is named `seutila` and contains all of the code necessary for the interface layout and storing the information required in the calculations. There are a number of different ways to take care of event handling. One method is to assign an event handler to every widget. This specification of an individual event handler is how the widget button events are handled. This is not possible for the compound widget `CW_FIELD` which is used extensively in this code. Its events are handled in the event handler specified for the top-level base. Except for the draw widget, the event handler for the top level base dispatches events out to other event handlers in the program. Let's examine the widget definition module, `seutila,` first.

All of the statements in `seutila` leading up to `widget_control,tlb,/realize` define how the interface is displayed. First and foremost, all widget programs require a top-level base (tlb). This is the "canvas" onto which all of the *child* widgets will be arranged. The tlb is defined using the `widget_base` function with a number of arguments and keywords. The `tlb_size_events` keyword is set to allow events to be generated whenever this base is resized by the user (by dragging the corner or side with the mouse). There is one event handler associated with the tlb. It is specified in the call to `XMANAGER` just before the `END` statement in `seutila`. There, the `event_handler` keyword is set to "seEvents". Therefore, anytime an event is generated by a widget that does not explicitly have an event handler associated with it (via the `EVENT_PRO` or `EVENT_FUNC` keywords), `seEvents` will handle those events. The compound widget `CW_FIELD` is used a number of times to specify the Hamiltonian size, number of eigenfunctions to display, limits, etc. The keyword `return_events` has been set in `CW_FIELD` so that the event handler associated with the tlb (`seEvents`) is notified when a `<CR>` has been generated in any of these fields. The draw widget is defined last, its `button_events` are turned on, and an event handler called `seDraw` has been specified. This event handler, `seDraw`, handles all of the logic associated with zooming.

Just before the widgets are realized to the display, the vertical dimension of the window is resized based on the height of the bases created containing the text fields. This is a necessary task because we do not know beforehand how tall and wide the `CW_FIELD` widgets are (regardless if we specify `XSIZE` and `YSIZE`…it turns out that widget layouts are platform-dependent to some extent). Once we get the appropriate size information using the `geometry` keyword to `widget_info` we set the vertical draw widget size accordingly then realize the widgets to the display.

After the widgets have been realized, the window value is extracted from the draw widget using `widget_control` and then a pixmap of the same size as the draw widget is created. This pixmap will be used later for seamless display updates, in particular when zooming with a rubberband-type box.

Next all of the information required for program operation is packaged up in a `state` structure, and a pointer called `pState` is created. Many of the fields in this structure are empty pointers to be filled up only when the various event handlers are executed. The state pointer, `pState`, is then put into the `UVALUE` of the tlb so that any of this information can be extracted in the event handlers.

After setting the UVALUE of the tlb to the state pointer, we call the event handler `seSolve` with an argument shown below:

```
seSolve, {event,id:void,top:tlb,handler:0L}
```

We call this event handler here so that when the program is first displayed, the calculation with some default information is performed. However event handlers require a named event structure to be passed in as a parameter. Therefore we create one named `event` with the three required fields: `id`, `top`, and `handler`. It is important to note that we could run into problems if we ever tried to redefine a structure named `event` with more or less fields with different field names. This event structure is defined as such for the remainder of your IDL session. The widget identifiers used in each of the fields don't matter except for `top`. This one must be assigned the tlb since the event handler will expect to pull all of the state pointer information out of `event.top`. We use one of the button widgets' ids (the QUIT button) for the `id` field and use 0L for the event handler.

The final statement in the widget definition module is the call to `XMANAGER`. The purpose of `XMANAGER` is to register the widget with the local window manager and start the event loop (i.e. the wait state). We also specify the event handler here (as discussed previously) and the cleanup routine. Since there are numerous pointers and a pixmap that takes up memory, we must clean all of this up upon exiting the program. This is accomplished in `seCleanup` which contains multiple `ptr_free` statements and deletes the pixmap. The state pointer is freed here as well.

The event handlers are straightforward and we will discuss one, `seLoadSHO`, for brevity. In this handler, the state pointer is retrieved from the top field of the event structure, via

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

The remainder of this event handler simply calls another other event handler (`seDefaults` to assign default values for the calculation), set the potential display `CW_FIELD` to the appropriate string value, and calls the event handler, `seSolve`. Most of the other event handlers follow this prescription of pulling out the state pointer and changing some values of the widgets, extracting values from the widgets, and/or executing other event handlers.

The event handler `seEvents` dispatches all of the `<CR>` events generated in the `CW_FIELD` widgets as well as perform the resizing. We will hold off discussing the details of widget resizing until the next section where a very simple application is presented.

§ A simple two-window application

There are a number of challenges associated with writing a conventional widget program. Usually IDL is very good at picking limits for displaying your curve using the `PLOT` command (if you do not specify the `XRANGE` or `YRANGE` keywords).

However you can run into the problem of "axis confusion" if you try to do something as simple as displaying two zoomable draw widgets. To see an example of "axis confusion", open up and run the file called **XTWOWIN_A.PRO**. Both of the windows in the display should look like those shown in figure 2.

Now zoom into the left panel by pressing the left mouse button and dragging the cursor over the region in which you wish to zoom. You should see a magnified view of the region you selected with the mouse. Now try zooming into a portion of the curve in the right panel. It should be difficult (if not impossible) to display the correct view. I tried zooming into the lower left hand corner of the right hand plot after zooming into the left panel and the result is shown in figure 3. Why is there "axis confusion"?

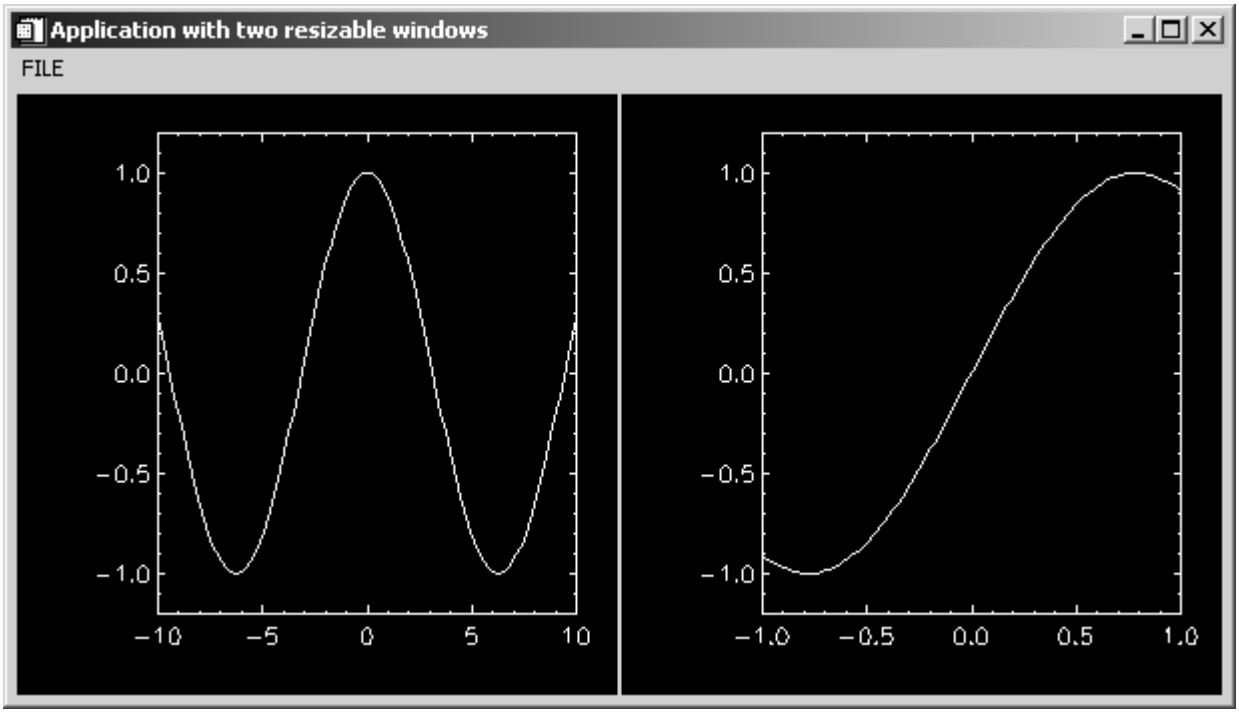

FIG 2. **Interface for XTWOWIN_A.PRO**

The answer lies in the system variables `!X` and `!Y`. These system variables are axis structures and directly affect how plots are displayed. There are numerous fields in these structures with which you are no doubt familiar (!X.MARGIN, !X.CRANGE, !X.RANGE, etc….see on-line help for more details). When data is drawn using the `PLOT` procedure, both `!X` and `!Y` are modified to reflect any keywords passed into the procedure and/or the details of the data itself. If different data with different limits are plotted to different windows then only the most recent system variables (i.e. those generated in the last plot) are "remembered". The solution to the confusion is to store both system variables into pointers associated with each window and restore (de-reference) these each time an event is generated in one of the windows.

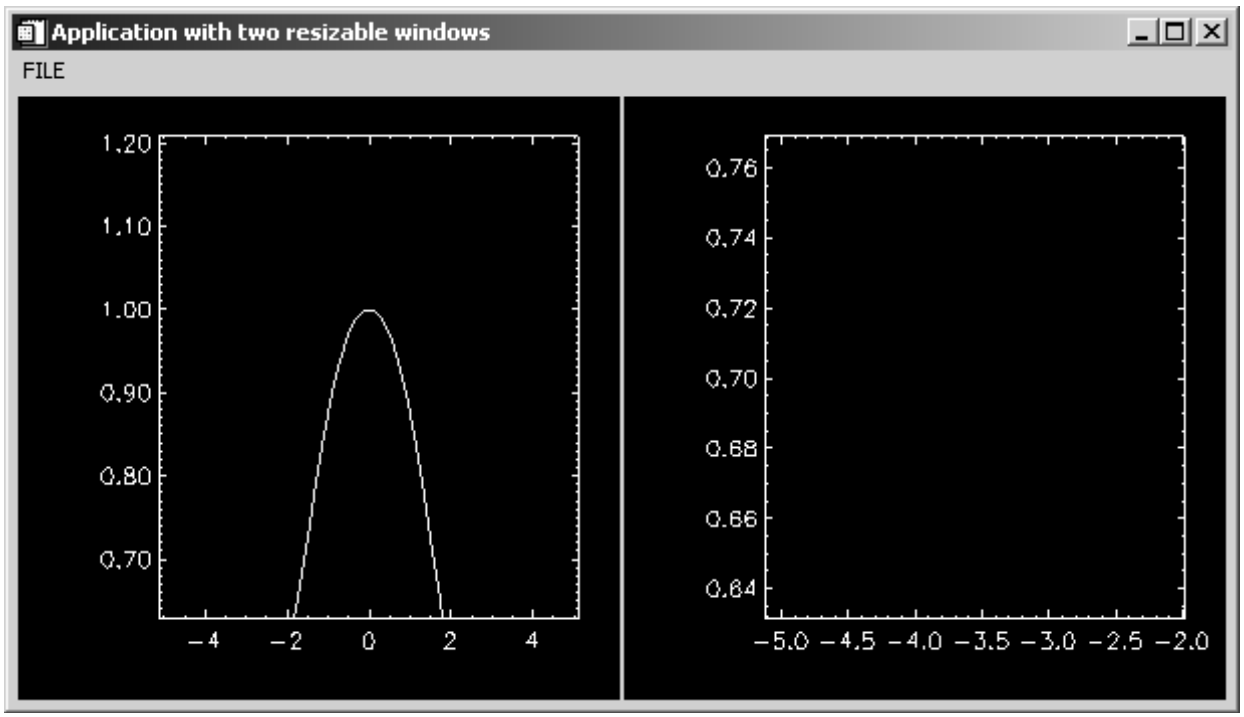

FIG3. Demonstration of "axis confusion".

Let's go through the existing code and add the relevant new code to eliminate "axis confusion". The first thing we'll do is add pointers designated `xPtr` and `yPtr` to the structures containing all of our window information (called `win1State` and `win2State`). Now go to the widget definition module (begins with `PRO XTWOWIN_A`) and add the following 4 lines of code shown in boldface:

```
win1State = {     win:win1,          $
                  winPix:winPix1,    $
                  winVis:winVis1, $
                  autoscale:1,       $
                  mouse:0B,          $
                  xPtr:ptr_new(/allocate_heap), $
                  yPtr:ptr_new(/allocate_heap), $
                  xrange:fltarr(2), $
                  yrange:fltarr(2), $
                  xpos:0.0,    $
                  ypos:0.0     $
            }
win2State = {     win:win2,          $
                  winPix:winPix2,    $
                  winVis:winVis2,    $
                  autoscale:1,       $
                  mouse:0B,          $
                  xPtr:ptr_new(/allocate_heap), $
                  yPtr:ptr_new(/allocate_heap), $
                  xrange:fltarr(2), $
                  yrange:fltarr(2), $
```

```
                xpos:0.0,   $
                ypos:0.0    $
        }
```

It is also a good idea to clean up these four new pointers now in xtwCleanup.

```
pro xtwCleanup,tlb
widget_control,tlb,get_uvalue = pState
wdelete,(*pState).win1State.winPix
wdelete,(*pState).win2State.winPix
ptr_free,(*pState).win1State.xPtr,(*pState).win2State.xPtr
ptr_free,(*pState).win1State.yPtr,(*pState).win2State.yPtr
ptr_free,(*pState).dataPtr1, (*pState).dataPtr2
ptr_free,pState
end
```

Next we need to assign the appropriate system variables to these pointers. This should be done after the `PLOT` command has been invoked. In this program there are two procedures that invoke the PLOT command: `xtwPlot1` and `xtwPlot2`. Both are shown below with the new code shown in boldface.

```
pro xtwPlot1,event
widget_control,event.top,get_uvalue = pState
data = *(*pState).dataPtr1
x = data[*,0] & y = data[*,1]
if (*pState).win1State.autoscale eq 1 then begin
  (*pState).win1State.xrange = [min(x),max(x)]
  dy = 0.1*(max(y)-min(y))
  (*pState).win1State.yrange = [min(y)-dy,max(y)+dy]
endif
plot,x,y,xrange = (*pState).win1State.xrange,xstyle = 1, $
  yrange = (*pState).win1State.yrange,ystyle = 1
*(*pState).win1State.xPtr = !x
*(*pState).win1State.yPtr = !y
end

pro xtwPlot2,event
widget_control,event.top,get_uvalue = pState
data = *(*pState).dataPtr2
x = data[*,0] & y = data[*,1]
if (*pState).win2State.autoscale eq 1 then begin
  (*pState).win2State.xrange = [min(x),max(x)]
  dy = 0.1*(max(y)-min(y))
  (*pState).win2State.yrange = [min(y)-dy,max(y)+dy]
endif
plot,x,y,xrange = (*pState).win2State.xrange,xstyle = 1, $
  yrange = (*pState).win2State.yrange,ystyle = 1
*(*pState).win2State.xPtr = !x
*(*pState).win2State.yPtr = !y
end
```

The final step is to de-reference these pointers in the draw widgets' event handlers, `xtwWin1Draw` and `xtwWin2Draw`. For convenience only the relevant portion of

`xtwWin1Draw` is shown below. The change to `xtwWin2Draw (not shown)` is similar except that you replace `(*pState).win1State` with `(*pState).win2State`.

```
pro xtwWin1Draw,event
widget_control,event.top,get_uvalue = pState
case event.type of
0:    begin       ; button press
        (*pState).win1State.mouse = event.press
        if (*pState).win1State.mouse eq 4 then begin
         (*pState).win1State.autoscale = 1

          !x = *(*pState).win1State.xPtr
          !y = *(*pState).win1State.yPtr
          wset,(*pState).win1State.winPix
          xtwPlot1,event
          wset,(*pState).win1State.winVis
          device,copy = $
            [0,0,!d.x_size,!d.y_size,0,0,(*pState).win1State.winPix]
        endif
        if (*pState).win1State.mouse eq 1 then begin
          (*pState).win1State.xPos = event.x
          (*pState).win1State.yPos = event.y
          !x = *(*pState).win1State.xPtr
          !y = *(*pState).win1State.yPtr
          wset,(*pState).win1State.winVis
          device,copy = $
            [0,0,!d.x_size,!d.y_size,0,0,(*pState).win1State.winPix]
          (*pState).win1State.autoscale = 0
        endif
      end
1:    begin ; button release
       if (*pState).win1State.mouse eq 1 then begin
        !x = *(*pState).win1State.xPtr
        !y = *(*pState).win1State.yPtr
        xll = Min([(*pState).win1State.xpos, event.x], Max=xur)
        yll = Min([(*pState).win1State.ypos, event.y], Max=yur)
        ll = convert_coord(xll,yll,/device,/to_data)
        ur = convert_coord(xur,yur,/device,/to_data)
        (*pState).win1State.xrange = [ll[0],ur[0]]
        (*pState).win1State.yrange = [ll[1],ur[1]]
        wset,(*pState).win1State.winPix
        xtwPlot1,event
        wset,(*pState).win1State.winVis
        device,copy = $
            [0,0,!d.x_size,!d.y_size,0,0,(*pState).win1State.winPix]
        (*pState).win1State.mouse = 0B
       endif
       if (*pState).win1State.mouse eq 4 then begin
        !x = *(*pState).win1State.xPtr
        !y = *(*pState).win1State.yPtr
        wset,(*pState).win1State.winPix
        xtwPlot1,event
        wset,(*pState).win1State.winVis
        device,copy = $
```

```
          [0,0,!d.x_size,!d.y_size,0,0,(*pState).win1State.winPix]
        (*pState).win1State.mouse = 0B
       endif
      end
```

The complete code with these changes can be found in **XTWOWIN.PRO,** included in the software distribution for this class. Now that these code changes have been made, try running the program again and zooming into both windows. The "axis confusion" should now be fixed.

An often-requested feature of widget programs is the ability to resize the base. This is easily done in IDL and you can go ahead and try it out with this current program. There are two simple code pieces necessary for resizing. The first piece involves turning on TLB_SIZE_EVENTS in the top level base. This is done in the widget definition module for **XTWOWIN** (and also for **XTWOWIN_A**). This code piece is shown below with the keyword set in boldface.

```
tlb = widget_base(/row,title = 'Application with two resizable windows', $
  /tlb_size_events,mbar = bar)
```

The final piece is to write the resize event handler. Since we are using the EVENT_PRO keyword to assign event handlers to the draw widgets we can use the event handler for the tlb specified in the call to XMANAGER to handle the resizing. The code is shown below.

```
pro xtwEvent,event
; This event handler will handle the resize events
widget_control,event.top,get_uvalue = pState
geom = widget_info(event.top,/geometry)
widget_control,(*pState).win1State.win,draw_xsize = fix(0.5*event.x), $
  draw_ysize = event.y
widget_control,(*pState).win2State.win,draw_xsize = fix(0.5*event.x), $
  draw_ysize = event.y
wdelete,(*pState).win1State.winPix
window,/free,/pixmap,xsize = fix(0.5*event.x),ysize = event.y
(*pState).win1State.winPix = !d.window
wdelete,(*pState).win2State.winPix
window,/free,/pixmap,xsize = fix(0.5*event.x),ysize = event.y
(*pState).win2State.winPix = !d.window
refresh1,event
refresh2,event
end
```

The idea is simple. The new base size is determined using a WIDGET_INFO call on the top-level base. Since TLB_SIZE_EVENTS are turned on, the x and y fields of the event structure passed into this event handler are the current size (in pixels) of the base. In fact the named event structure passed into this event handler is given by

```
{ WIDGET_BASE, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L }
```

If we want both windows to have the same horizontal size then we simply split `EVENT.X` in two making half of `EVENT.X` the new window `XSIZE`. The draw widgets `YSIZE` is simply `EVENT.Y`. Both of these steps are accomplished through the following calls:

```
widget_control,(*pState).win1State.win,draw_xsize = fix(0.5*event.x), $
  draw_ysize = event.y
widget_control,(*pState).win2State.win,draw_xsize = fix(0.5*event.x), $
  draw_ysize = event.y
```

Note that we also have to destroy the original pixmaps and create new ones of the appropriate size. The final step shown above is to redisplay the data in both of the resized draw widgets via the `refresh1` and `refresh2` calls.

It should be mentioned that this is a particularly simple example to demonstrate resizing. In general your application could have a complicated hierarchy of bases on bases and multiple draw widgets. Resizing more complicated widget layouts requires careful consideration of the sizes of all of the components on the top-level base. This necessarily requires multiple calls of `WIDGET_INFO(base,/geometry)`.

EXERCISE:

Modify SEUTILA.PRO so that running multiple instances of it does not result in "axis confusion".

§ Modal widget dialogs

All of us have used a modal widget dialog but probably didn't know that it was called **modal**. Dialog widgets are normally called from a parent program. In IDL a modal dialog is one that prevents the user from interacting with the calling program until the dialog has been dismissed. The main idea behind any dialog widget is that you want to collect information from the user and pass this along to the calling program. There are occasions when you would like the user to enter only the dialog information before proceeding. This is appropriate for a modal dialog.

To illustrate how to incorporate a modal dialog into a calling program, let's run a simple widget application called `XCURVECHANGE_A.PRO`. Press the button labeled MODAL DIALOG and you should see something like that displayed in Figure 4.

Try pressing the MODAL DIALOG button again and see what happens. How about if you try quitting the program from the QUIT selection under the FILE menu? What you are seeing is modal behavior. The operation of the calling program ceases until you press the DONE button on the modal dialog. The default curve that appears in the draw widget is a sine curve. Select `cos(x)` in the dialog and press DONE. You should see the curve update in the draw widget.

The modal dialog program itself is called MCURVECHANGE.PRO and is included in the software distribution for this course. Open this short file up and take a look at the code. The first thing to notice is that it is a function widget whose return value is the updated data array. When this program is exited, the calling program will plot the new

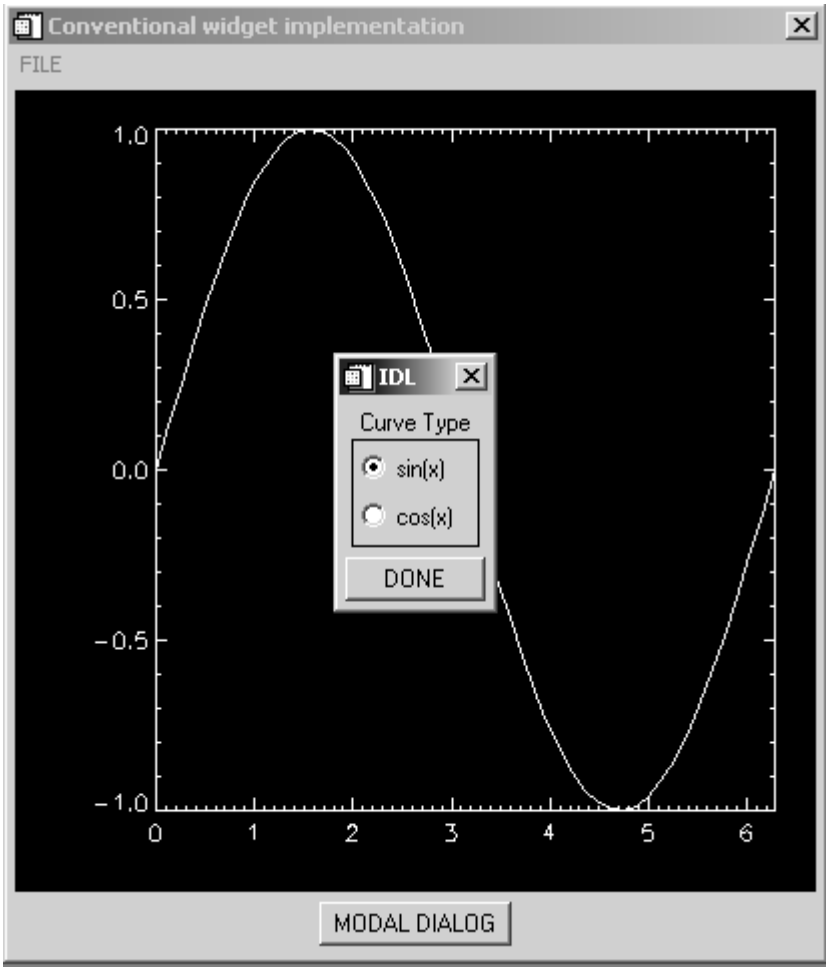

Fig. 4 Demonstration of a modal dialog from `XCURVECHANGE_A.PRO`.

data. The next thing to notice is the creation of the top-level base in the widget definition module (shown below).

```
function mCurveChange,parent = parent,x
if n_elements(parent) eq 0 then begin
  parent = 0L
  modal = 0
endif else begin
  modal = 1
endelse
tlb = widget_base(group_leader = parent,modal = modal,/col)
.
.
.
```

The parent keyword is optional here. When I write dialog widgets, I like to be able to test them out independent of the calling program. If this program is called with a parent (group_leader) then the top-level base (TLB) is called with its `MODAL` keyword set to one. This is necessary to get the modal behavior. However, if no parent is set then parent is set equal to 0L (long integer) which is equivalent to no parent and the modal keyword is set to 0 which means that it is no longer modal. This is perfectly fine for debugging purposes. You can test out this program independent of the calling program by just putting some dummy data in for the parameter x like,

```
IDL> y = mCurveChange(findgen(100))
```

You should see the little dialog box appear and allow you to change the button settings and also quit using the DONE button.

The main program, `XCURVECHANGE_A.PRO`, is a straightforward widget program. The event handler associated with the modal dialog button is called `xccModalLaunch` and is very simple.

```
pro xccModalLaunch,event
widget_control,event.top,get_uvalue = pState
data = *(*pState).dataPtr
x = data[*,0]
; The call below freezes this application up until the DONE button in the
; modal dialog is pressed.
newData = mCurveChange(parent = event.top,x)
; Get here if the DONE button was pressed.
*(*pState).dataPtr = newData
xccDisplayCurve,event
end
```

The x-values are simply de-referenced from the data pointer and mCurveChange is called with event.top as its parent and the x-values as the input parameter. The return value is then put back into the data pointer and the data is displayed in the draw widget. Note that we could have written the modal dialog as a procedure and used an output parameter too.

§ Non-modal widget dialogs

Often the user would like to enter some values and see the display change immediately or, at the very least, change the display upon pressing an ACCEPT button in the dialog. Only when the user presses a DISMISS button should the dialog disappear. This is the action of a **non-modal** widget program. These are a bit more challenging to write than modal dialogs because you do not want all of the program action to cease and wait for you to press DISMISS. Therefore you must have a way of conveying the message from the dialog to the calling program without closing the dialog. (We will see later that if you write your widget applications as objects, non-modal dialogs are extremely easy to write.)

In conventional widget programming the method that allows us to send messages involves the SEND_EVENT keyword in a call to WIDGET_CONTROL. The message that gets sent is a named event structure we make ourselves and pack into it whatever information we will need. Of course the three fields ID, TOP, and HANDLER (which are in every event structure) are necessary but anything else can be packaged into the event structure.

Open up the programs titled XCURVECHANGE_B.PRO and NMCURVECHANGE.PRO. XCURVECHANGE_B.PRO is similar to XCURVECHANGE_A.PRO but it has a button to launch the non-modal dialog and a new event handler that sends and receives events from the non-modal dialog. NMCURVECHANGE.PRO is the name of the non-modal dialog widget program.

Run the program, launch the non-modal dialog and see how it works. You should be able to see the plot in the draw widget update everytime you change the curve setting in the dialog. Also, quit the calling program by selecting QUIT from the FILE menu. The non-modal dialog will disappear when XCURVECHANGE_B.PRO disappears. This is a result of NMCURVECHANGE having the top-level base of XCURVECHANGE_B as its group leader.

Let's first take a look at XCURVECHANGE_B.PRO. The button labeled NON-MODAL DIALOG has an event handler associated with it called xccNonModalLaunch listed below.

```
pro xccNonModalLaunch,event
thisEvent = tag_names(event,/structure_name)
widget_control,event.top,get_uvalue = pState
case thisEvent of
'WIDGET_BUTTON': $
  begin
    x = (*(*pState).dataPtr)[*,0]
    nmCurveChange,x,group_leader = event.top,notifyIds = [event.id,event.top]
  end
```

```
'NMCURVEEVENT': $
  begin
      *(*pState).dataPtr = event.data
      xccDisplayCurve,event
  end
else:
endcase
end
```

This event handler is a bit different from those we have seen previously in the simpler widget programs. Here we are differentiating between two different types of events, `WIDGET_BUTTON` events and `NMCURVEEVENT` events. The `WIDGET_BUTTON` event is one we are all familiar with. It is just the name of the named event structure associated with a widget button. However there is no such event named `NMCURVEEVENT` in the IDL documentation. In fact this is an event structure of our own making, created and sent from the non-modal dialog program, `NMCURVECHANGE.PRO`. Let's take a closer look at what's happening in this event handler.

The first thing done in the event handler is to get the name of the event structure. If it is a `WIDGET_BUTTON` event then the independent variable data, x, is de-referenced from the data pointer. This is passed into the `NMCURVECHANGE.PRO` procedure along with the top-level base as the group leader and a two-element vector of widget IDs passed in via a keyword called `NOTIFYIDS`. This information in `NOTIFYIDS` is necessary so that `NMCURVECHANGE.PRO` knows which widget in the calling program to notify when it is sending an event. The first element of this vector is the ID of the button and the second is its top-level base.

If the event is named `NMCURVEEVENT` then the data pointer is updated by the field of the event structure called data via

```
*(*pState).dataPtr = event.data.
```

We will see that data is a field that we put into our structure called `NMCURVEEVENT`. Next the data is redisplayed in the draw widget of the calling program. And that is all there is to the event handler in the calling program.

The non-modal dialog widget is written a bit differently from a typical widget program. Let's look at `NMCURVECHANGE.PRO` in some detail. The beginning of the widget definition module is listed below:

```
pro nmCurveChange,x,group_leader = group_leader,notifyIds = notifyIds
; The first thing to check is if there is an instance of this program running
; associated with the calling program.  If so then proceed no further.
if n_elements(notifyIds) eq 0 then notifyIds = [0L,0L]
if n_elements(group_leader) eq 0 then begin
  group_leader = 0L
endif
tlb = widget_base(group_leader = group_leader,/col)
; Create a registered name that is unique for the calling program.  Use the
```

```
; group leader's id for instance...
registerName = 'nmcc'+strtrim(string(group_leader),2)
if xregistered(registerName) then return
.
.
.
xmanager,registerName,tlb,event_handler = 'nmccEvent',cleanup = 'nmccCleanup'
end
```

As in the modal dialog, this widget program has been written to allow you to launch it directly from the command line, independent of any calling program. You can try running the dialog by typing the following at the command line:

```
IDL> nmCurveChange,findgen(100)
```

The three allowed inputs are the parameter `x`, and the two keywords, `group_leader` and `notifyIds`. If no group leader is specified, a value of 0L is used. Also if no `notifyIds` are passed in then 0L is used for both elements of the vector.

There is an additional feature associated with this non-modal dialog. Since a non-modal dialog allows the user to interact with the calling program while the dialog is displayed it is also possible to press the `non-modal dialog` button to launch another instance of the non-modal dialog. It could be detrimental to launch multiple instances of the non-modal dialog. We can prevent this from happening by checking if the name of the program registered with the `XMANAGER` is currently registered elsewhere. If so then we return. The function call `XREGISTERED` performs this task and the argument is called `registerName`. Note that we have constructed the `registerName` variable using the group_leader as a component. The reason for doing this is that we want only one instance of the non-modal dialog to be launched per instance of `XCURVECHANGE_B.PRO`. Multiple instances of `XCURVECHANGE_B.PRO` should be allowed and each of those instances should be allowed to launch a single instance of `NMCURVECHANGE.PRO`. The group leader provides an identifier unique to each instance of `XCURVECHANGE_B.PRO` running.

The event handler for `NMCURVECHANGE.PRO` is where much of the action takes place and is shown below.

```
pro nmccEvent,event
uname = widget_info(event.id,/uname)
case uname of
'CURVEGROUP':     $
      begin
      widget_control,event.top,get_uvalue = pInfo
      if (*pInfo).notifyIds[0] ne 0L then nmSendInfo,pInfo
      end
'DONE':                $
      begin
      widget_control,event.top,get_uvalue = pInfo
```

```
      if (*pInfo).notifyIds[0] ne 0L then nmSendInfo,pInfo
      widget_control,event.top,/destroy
      end
else:
endcase
end
```

When the user presses either of the buttons `sin(x)` or `cos(x),` an event is generated. This event is detected through the UNAME of CURVEGROUP in the event handler and the procedure `NMSENDINFO` is called. As a brief aside, liberal use of the UNAME keyword in each of your widgets can actually save you from constructing an enormous state structure in which you place all of the widget identifiers. You can find out the UNAME of the widget causing an event via

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

You can also find out the widget id of a widget with a particular UNAME via

```
id = widget_info(event.top,find_by_uname = uname).
```

Judicious use of a widget username can prevent unnecessary proliferation of the state structure.

Now, back to the program. If the user presses the `DISMISS` button then the procedure `NMSENDINFO` is called and the non-modal widget is destroyed. So what does `NMSENDINFO` do?

`NMSENDINFO` is the procedure that creates our pseudo-event and sends it to the calling program. It is listed below.

```
pro nmSendInfo,pInfo
  widget_control,(*pInfo).curveGroup,get_value = index
  if index eq 0 then begin    ; SIN
      (*pInfo).data[*,1] = sin((*pInfo).data[*,0])
  endif else begin                    ; COS
      (*pInfo).data[*,1] = cos((*pInfo).data[*,0])
  endelse

  nmInfo = {NMCURVEEVENT,$
                        id:(*pInfo).notifyIDs[0],$
                        top:(*pInfo).notifyIDs[1],$
                        handler:0l,$
                        data:(*pInfo).data}
  if widget_info((*pInfo).notifyIDs[0],/valid_id) then begin $
    widget_control,(*pInfo).notifyIDs[0],send_event = nmInfo
  endif
end
```

In this procedure, the pointer to the state structure called `pInfo` is passed in as an argument. This allows local access to all of the widget information. The first thing that is done in the procedure is determine which button is currently set. Then we calculate

the data based on the button setting. Next a named event structure called NMCURVEEVENT is created whose ID, TOP, and HANDLER are filled out using the values from the NOTIFYIDS keywords. Note that IDL takes care of filling out the HANDLER field (since it now knows the ID and TOP fields) so using 0L is acceptable here. The ID should be the ID of the button from the calling program that launched this non-modal widget, and the TOP field should be the widget identifier of top-level base of the hierarchy to which ID belongs. Finally we add the field called DATA to the event structure because the calling program needs this information. The last step is to send this structure to the the widget known as ID. Since you can only send events to valid widget identifiers, the validity of ID has to be tested. Then a WIDGET_CONTROL statement is invoked to send the event to the ID of the widget that launched NMCURVE.PRO.

Sending events and other information is a powerful way to communicate between widgets. It also is the way to communicate among applications in a multi-module application as we will see later.

§ Hiding/showing control bases

One nice feature found in some applications is the ability to toggle between hiding and showing a base containing some type of controls, perhaps a set of buttons. The user might not want to display these controls all of the time because they might only be useful during a particular type of analysis, for instance. The real estate on the screen is reason enough to consider using such a feature. Implementing this feature is quite simple.

Open up, compile and run the program called DETACHED_WIDGET.PRO. Press the HIDE/SHOW button and you should see another control panel appear like that shown in figure 5. Pressing one of the buttons in the detached control panel should result in its value being displayed in the text field of the first panel.

Fig. 5 The detached widget example.

Pressing the HIDE/SHOW button again will result in the detached control panel disappearing.

The way you can make a base appear and disappear is by using the MAP keyword for a base widget. For example, we do not wish for the controls base (i.e. the one with the APPLE and ORANGE buttons) to appear upon realizing the main widget base. Therefore MAP=0 is set in the call to the widget_base function call in the widget definition module:

```
ctrlBase = widget_base(group_leader = tlb,/col,title = 'Controls', $
      map = 0,xsize = 100)
```

The toggle feature of the base is specified by the user-name of the HIDE/SHOW button. Initially the username is set to SHOW as shown below:

```
hide = widget_button(tlb,value = 'HIDE/SHOW',uname = 'SHOW')
```

As we will see in the event handler the user-name gets changed when the button gets pressed. This is accompanied by a call to map/unmap ctrlBase.

Note that we have also specified the group leader as the top-level base in the program. This has a number of consequences. The first is that we must issue a separate realize command as the one issued for the tlb. This is done in the widget definition module as follows:

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

The other consequence is that we have to set the user-value of ctrlBase to the state pointer so that the state information is accessible from the detached base. This is done after our usual step of setting the user-value of the tlb via:

```
widget_control,tlb,set_uvalue = pState
widget_control,(*pState).ctrlBase,set_uvalue = pState
```

Finally we must make two calls to XMANAGER so that the event loops for both tlb and ctrlBase are activated. The event handler for both are the same, dwEvent, so that events from both bases are processed in the same location. We could have specified separate event handlers for each base but , since the program is relatively small, there really is no need for the added complexity. The two calls to XMANAGER are shown below:

```
xmanager,'dw',tlb,cleanup = 'dwCleanup',/no_block, $
  event_handler = 'dwEvent'
xmanager,'dw',(*pState).ctrlBase,event_handler = 'dwEvent'
```

Note that we do not have to specify a cleanup procedure for `ctrlBase` since the tlb's cleanup does the job.

Calling `xmanager` multiple times in a single program allows you to register events from multiple bases with a single event handler. That event handler determines the widget that generated the event using the user-name as discussed in the previous example. The code is straightforward. The only new feature introduced in this code is the use of the map keyword to widget control shown in boldface, repeated below. Note also that the user-name of the `HIDE/SHOW` button gets toggled depending on whether `ctrlBase` is visible or not.

```
pro dwEvent,event
uname = widget_info(event.id,/uname)
case uname of
'QUIT':     $
  begin
    widget_control,event.top,/destroy
  end
'SHOW':     $
  begin
    widget_control,event.top,get_uvalue = pState
    widget_control,(*pState).ctrlBase,/map
    widget_control,event.id,set_uname = 'HIDE'
  end
'HIDE':     $
  begin
    widget_control,event.top,get_uvalue = pState
    widget_control,(*pState).ctrlBase,map = 0
    widget_control,event.id,set_uname = 'SHOW'
  end
; The following unames correspond to events coming from
; the control base.
'APPLE': $
  begin
    ; Note that here, EVENT.TOP is CTRLBASE itself.
      widget_control,event.top,get_uvalue = pState
    widget_control,(*pState).text,set_value = 'APPLE'
  end
'ORANGE': $
  begin
      widget_control,event.top,get_uvalue = pState
    widget_control,(*pState).text,set_value = 'ORANGE'
  end
else:
endcase

end
```

OBJECT WIDGET PROGRAMMING

§ Introduction to objects in IDL

There are a number of different ways to write a program in IDL. Perhaps one of the first types of programs you wrote was *procedural*, written in a top-down fashion. Event-driven programming, covered in the first half of this documentation, is another type of IDL program in which the program is in a perpetual "wait state", waiting for user input before proceeding to calculate and/or display something. **Object oriented programming** (OOP) offers another paradigm with which you can write programs in IDL. In OOP, an object combines data and the procedures and/or functions associated with that data into a single unit called an **object**. A procedure or function that operates on that data is known as a **method**. In a broad sense, a properly written widget application can be thought of as being written in an *object oriented* manner. The application itself is the object, the state structure passed around the various program modules is the object data, and the event handlers and associated code operating on the data are the methods. Programming with objects in IDL merely enforces certain rules. Perhaps the most important rule of IDL objects is that the data is shielded from the user. The user does not have access to manipulate the data outside of the object except through the methods that have been defined.

Rather than introduce definitions without context, we will build use an object class and then see how it is written. Definitions of various terms will be provided as needed.

Open up and compile the file called `DATA__DEFINE.PRO`. This file contains four procedures and one function which define the **object class** called `DATA`. The procedures are

```
data__define
data::display
data::getProperty
data::cleanup
```

and the function is

```
data::init
```

The purpose of this object class is to provide an **interface** for two dimensional data (i.e. data with two independent variables). In order to emulate the terminology of neutron scattering data we refer to the first independent variable as the *channel* and the second independent variable as the *group*. The interface is straightforward. We merely wish to load the data, display it with a color scheme of our choosing, and access the number of

channels and groups present in the data. Let's begin by creating some two-dimensional data:

```
IDL> d1 = hanning(40,20) & d2 = dist(60,60)
```

These are just some built-in IDL functions that we use for convenience. You can make up any kind of two-dimensional data that you wish and use it instead. Next **instantiate** (create) an object using the first set of data, d1:

```
IDL> o1 = obj_new("data",d1)
```

The `OBJ_NEW` command is the object creation command and the return value, `o1`, is the object reference. The object reference is a heap variable which you can confirm by probing the heap via:

```
IDL> help,/heap,/brief
```

You should see that there is a new object on the heap from the output log:

```
IDL> help,/heap,/brief
Heap Variables:
    # Pointer: 1
    # Object : 1
    # Bytes Heap Memory:  3216
```

What can we do with the object, `o1`, now? There is a method associated with it called `DISPLAY` allowing us to display an image of this data. Issue the following command at the command line to display the data:

```
IDL> o1 -> display
```

You should see a gray-scale image of the data in a new window. There is a useful keyword called COLORTABLE for this method. To change to a different (more colorful) colortable, issue the following command:

```
IDL> o1 -> display,colorTable = 5
```

You should see the image updated in its display window. Play around with this command and use any of the 41 color tables (use 0 to 40 for the colortable value passed into the method).

Next let's instantiate an object of the data given by d2:

```
IDL> o2 = obj_new("data",d2)
```

Display this new object using the command:

```
IDL> o2 -> display,colortable = 1
```

This should open a new window and display the data using a blue-white linear scale. Next issue a command to change the color table of the first object, `o1`:

```
IDL> o1 -> display,colortable = 26
```

This should affect only the object data, `o1`. Thus the two objects are independent of each other: the data are different, the plot windows are different, and the color tables are different. Yet each object updates the proper display with the correct data and color table. This is one of the strengths of objects…they remember! This is another way of saying that the object information is **persistent**.

The other method in the data class is known as an **accessor** method, called `getProperty`. The keywords to this method are `nchan` and `ngroups`. To get the number of channels and groups for `o1` out simply type

```
IDL> o1 -> getProperty, nchan = nchan, ngroups = ngroups
IDL> print,nchan,ngroups
```

If you wanted to do so you could add a `data` keyword to the `getProperty` method which would allow the user (of the object) access to the data. However this conflicts with the philosophy of OOP since the data should be shielded from the user. By providing access to the data this shield is short-circuited. If there is something that the user wishes to do with the data then 99% of the time it is more appropriate to add a new method to the class definition that does the desired task.

Bottom line: ask yourself why you need access to the data before you add a data keyword to an accessor method.

When we are finished with the objects then we must destroy them (and any heap variables associated with them). This is done using the OBJ_DESTROY procedure as shown below.

```
IDL> obj_destroy,[o1,o2]
```

Now that we have used the object class with some concrete examples, let's look at the code in detail to figure out how it all works. It has been reproduced below. Object class definition programs such as this include at least one function, the `INIT` function, and one procedure, the definition module. In this case the definition module is called `DATA__DEFINE`. All object classes are named in this manner with a double-underscore separating the object name from "define". This module contains a named structure called DATA with six fields: `nchan`, `ngroups`, `winId`, `colorTable`, and

dataPtr. The purpose of this named structure is to specify the data types for each of the members of the class.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro data::cleanup
ptr_free,self.dataPtr
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro data::getProperty,  nchan = nchan,    $
                        ngroups = ngroups

if arg_present(nchan) then nchan = self.nchan
if arg_present(ngroups) then ngroups = self.ngroups
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro data::display,colorTable = colorTable
if n_elements(colorTable) ne 0 then self.colorTable = colorTable
loadct,self.colorTable,/silent
xsize = 400 & ysize = 400
if self.winId eq (-1L) then begin
  window,/free,xsize = xsize,ysize = ysize
  self.winId = !d.window
endif else begin
  wset,self.winId
endelse
tv,smooth(congrid(bytscl(*self.dataPtr),xsize,ysize),10)
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function data::init,data
device,decomposed = 0
loadct,0,/silent
dsize = size(data)
self.nchan = dsize[1]
self.ngroups = dsize[2]
self.dataPtr = ptr_new(data,/no_copy)
self.colorTable = 0
self.winId = -1L
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro data__define
define = {  data,       $
            nchan:0,    $
            ngroups:0,  $
            winId:0L,   $
            colorTable:0,     $
            dataPtr:ptr_new() $
         }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

The initialization module, called DATA::INIT, is executed when this class is instantiated. This initialization method must always be written as a function and return a value of 1 if successful or 0 if unsuccessful. Here the function expects a two-dimensional data parameter named DATA to be passed in. Each dimension of the array is deteremined and then placed into the appropriate field of the object, via the **SELF** construct. Since we will allow access to these dimensions it is necessary to have separate fields for them in the object definition. (Actually we can get this information

by invoking the `size` function in the accessor method when it is needed but here we choose instead to have separate fields in the object definition).

Here `SELF` is the reference of the particular object within the methods in the class definition. It is very important to note that `SELF` is not defined outside of the class. `SELF` is only exposed to the methods of a class. It has no meaning anywhere else. `SELF` is available everywhere within the class definition without the need to *get a value* (as in widget programming). `SELF` and its fields are always present. In the `init` method the `DATAPTR` field of `SELF` is then filled with the parameter passed into this method (function). Finally a default color table is selected (0 for black-white linear scale) and the window id, `-1L`, is selected because it means that there are no active windows.

The procedure that does the display work is called `DATA::DISPLAY`. This method sets the colortable to the currently selected value of the colortable or changes it if the input keyword was set to a different value. Note that only values between 0 and 40 are appropriate for the `COLORTABLE` keyword (but we have not implemented any error catching if the colortable is outside of those bounds). Finally we create a new window if necessary, set the current device to the window, and use `TV` to display a smoothed, expanded, and "byte-scaled" version of the data. Note that we "byte-scale" the data so that its values are stretched out over an integer range of 0 to 255. This coincides with the entries in the colortable ensuring that all colors in the colortable will be represented in the image that is displayed.

The accessor method, `DATA::GETPROPERTY`, allows the user access to the number of channels and number of groups in the data set. Note the use of `ARG_PRESENT` to determine which of the keywords was set. Since the overhead in processing speed is so small, we could eliminate the test using `ARG_PRESENT` and just populate the keywords `NCHAN` and `NGROUPS`. In general your accessor method might involve some computations for each keyword so it is probably in your best interests to compute those values that the user of the class requested.

The final method is called `DATA::CLEANUP`. Both `INIT` and `CLEANUP` are referred to as **lifecycle methods** since one is executed when the object is instantiated and one is executed when it is destroyed. In the `CLEANUP` there is one heap variable, `DATAPTR`, that must be freed upon destroying the object.

EXERCISE:

Modify `DATA__DEFINE` so that a new method, called `changeCT`, changes the current color table and updates the display. This should be written so that (once you have instantiated the data class) you can merely type `o1->changeCT,5` and the color table will be updated to color table 5 and the image will be redrawn.

§ A simple object class with derived classes

In the last section we used a single object class, DATA, to illustrate object creation, destruction, and how to invoke methods. In this section we will learn how to define and use a new object class illustrating a concept called **inheritance**.

One of the powerful ideas in OOP is the notion of a **base class** and its **derived classes**. A base class (a.k.a. **superclass**) can be thought of as containing common data (generalization) and methods for all of the more specific derived classes (specialization). The derived classes (a.k.a. **subclasses**) **inherit** data and methods from the base class, extend themselves with more data and/or methods, and/or **override** the base class methods. The relationships among the base class and superclass are best illustrated with an example. We will consider such a set of classes using IDL code.

Let's consider a base class called SHAPE and two derived classes, SQUARE and CIRCLE. We'd like to create an instance of either SQUARE or CIRCLE and display it to a plot window and obtain the area of that object. The notion of getting the area, and being able to plot these objects to a display are common methods but their implementation for a SQUARE and CIRCLE will be different. SQUARE and CIRCLE are then derived classes of SHAPE. Thus they inherit the data and methods associated with SHAPE. SHAPE is an **abstract class** in the sense that we would not instantiate the SHAPE class but rather one of its derived classes. The relationship among the objects can be represented by the diagram shown in figure 6.

In figure 6 the arrows denote an inheritance relationship: CIRCLE and SQUARE inherit the data and methods of SHAPE. The data and methods are shown to the right of each class in the diagram. The data in the SHAPE class are composed of two pointers, xPtr and yPtr, that point to the data defining the specific shape. These do not have to be pointers but making them so allows the flexibility of chaning the number of points defining the shape at runtime. CIRCLE and SQUARE, the derived classes, have the GETAREA and DRAW methods and three data fields each. The data in CIRCLE is composed of xc, yc, and radius, to specify the location and size of the circle. SQUARE has xc, yc, and side to specify the location and size of the square.

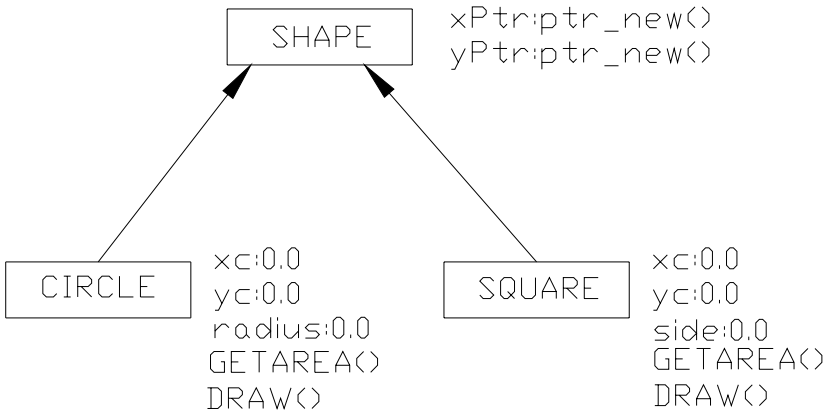

Fig. 6 Diagram of relationship between the base and derived classes in the example.

Open up the three files, `SHAPE__DEFINE.PRO`, `CIRCLE__DEFINE.PRO`, and `SQUARE__DEFINE.PRO`. Let's look at `SHAPE__DEFINE.PRO` first, listed below.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro shape::cleanup
ptr_free,self.xPtr, self.yPtr
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function shape::init
self.xPtr = ptr_new(/allocate_heap)
self.yPtr = ptr_new(/allocate_heap)
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro shape__define
define = {  shape,      $
            xPtr:ptr_new(),   $
            yPtr:ptr_new()    $
         }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

The last procedure in this file, `SHAPE__DEFINE`, contains a named structure definition called `SHAPE` with three fields: `area` (defined as a scalar float), and `xPtr` and `yPtr` (pointer variables). When the object gets instantiated this module defines the types of its data.

The initialization module only allocates heap to the pointers that will contain the data defining the shapes. As before this is accomplished using the appropriate fields of the `SELF` structure.

Next let's take a look at the code for `CIRCLE__DEFINE.PRO`, listed below.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function circle::getArea
return,!pi*self.radius^2
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
function circle::draw
plots,*self.xPtr,*self.yPtr,/device
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function circle::init,xc = xc,yc = yc, radius = radius
retVal =self -> shape::init()
if retVal ne 1 then return,retVal
if n_elements(xc) eq 0 then xc = 100.0
if n_elements(yc) eq 0 then yc = 100.0
if n_elements(radius) eq 0 then radius = 100.0
self.xc = xc
self.yc = yc
self.radius = radius
nth = 100
th = (2.0*!pi/(nth-1.))*findgen(nth)
*self.xPtr = xc + radius*cos(th)
*self.yPtr = yc + radius*sin(th)
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro circle__define
define = { circle, inherits SHAPE, $
            radius:0.0, $
            xc:0.0, $
            yc:0.0      }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

There are four modules to this object definition. The definition module, `CIRCLE__DEFINE`, is similar in format to that of `SHAPE__DEFINE`. One difference though is the statement just after the structure name: `inherits SHAPE`. This is the statement that allows this derived class of `SHAPE` to see all of the data in `SHAPE`. There are three other fields, `xc`, `yc`, and `radius` to specify the geometry and location of the shape.

The initialization module, `CIRCLE::INIT`, has three arguments, specified as optional keywords, `xc`, `yc`, and `radius`. The first statement executed in this module is a call to the initialization method of the base class, `SHAPE`. If there is an error in that initialization then no further action is taken and the program is exited. Next, defaults are used for geometric parameters if the keyword parameters were not explicitly stated. Finally the data determining the circle are calculated, including the points defining the circle. The coordinates defining the circle are de-referenced into `self.xPtr` and `self.yPtr` even though we never explicitly defined these fields for this derived class. Rather this derived class has *inherited* these fields.

We have defined a (function) method called `GETAREA` here that computes and returns the area of the object. Regardless of whether we created an instance of a `CIRCLE` class or `SQUARE` class, `GETAREA` returns the correct area. The idea that the `GETAREA` method works independent of whether the object is a circle or a square is a concept known as **polymorphism**. Naturally it contains different code in the `CIRCLE` and `SQUARE` class

definitions specific to each shape but users of this method need not know the shape to calculate the area.

No cleanup routine is necessary for this module because no new heap variables were created here. Of course if we had created any new pointers or objects here, we would have to write a cleanup routine. In the cleanup routine for SQUARE, we would have to call the cleanup on SHAPE explicitly via `self->shape::cleanup`, then "clean up" the heap variables created in SQUARE.

The object definition module for SQUARE, called SQUARE__DEFINE, is written similarly to CIRCLE__DEFINE. We will not go into the details of its construction because it is so similar.

Now let's see how this all works by instantiating the classes and invoking some methods. At the command prompt, create a circle by issuing the command:

```
IDL> ocircle = obj_new("circle")
```

Next examine the contents of the heap by the usual means (`help,/heap,/brief`). There should be no surprises: one object and two pointers. You can determine the class of this object with the following command:

```
IDL> print,obj_class(ocircle)
CIRCLE
```

You can also determine the base class of the CIRCLE class using the SUPERCLASS keyword to this command:

```
IDL> print,obj_class(ocircle,/superclass)
SHAPE
```

This command actually returns only the direct superclass of the object `ocircle`. If the SHAPE class was inheriting from another class, we would not know this from this command.

Next you can draw the circle to the screen via:

```
IDL> result = ocircle -> draw()
```

and print the area of the circle using

```
IDL> print,ocircle->getArea()
```

Finally, when you are finished with manipulating the object, you must destroy it,

```
IDL> obj_destroy,ocircle
```

Now repeat these steps for a square object:

```
IDL> osquare = obj_new("square")
```

```
IDL> print,oquare->getArea()
```

Notice that we used the same command (`getArea`) to get the area of the circle and the square but it was invoked for different objects. This is an example of the polymorphism where the command performs a different calculation depending on the identity of the object but the information is extracted by a command (method) of the same name.

Be sure to destroy the objects when you are finished.

EXERCISE:

Modify the code in CIRCLE__DEFINE.PRO so that you can change the radius of the circle. Hint: One approach might be to create a method called `circle::changeRadius` whose only argument is the desired radius. Within this procedure you will have to recompute the coordinates in `self.xPtr` and `self.yPtr` accordingly.

EXERCISE:

Modify the appropriate code so that you can change the thickness of the line that is used in the `DRAW` method (regardless if it is a circle or square).

§ A first widget object

The previous two sections have given some insight as to the power of OOP in IDL. The combination of widgets and objects allows a flexibility in program operation that is difficult to mimic using conventional widget programming. In this section we will run a simple object widget program to illustrate this flexibility then examine the code in detail to see how it works. Open up and compile the object class called `OBWID__DEFINE.PRO`. Instantiate this class by issuing the following command:

```
IDL> o = obj_new("obwid")
```

Press the button labeled `GENERATE RND` and a random number should appear in the text widget. Do this a number of times to see that the number changes every time you press the button. You can exit the program by pressing the `QUIT` button.

Once again, instantiate the class as you did above.

```
IDL> o = obj_new("obwid")
```

Now, instead of using the button labeled `GENERATE RND`, invoke the method directly via:

```
IDL> o -> genRand
```

The display should update in the same way as if you had pressed the `GENERATE RND` button. You can also quit the program by issuing the command:

```
IDL> o -> quit
```

Examine the heap to make sure that all pointers and objects have been freed/destroyed. The idea that you can run the widget program by invoking methods at the command line as well as using the usual widget controls implies a whole new flexibility in running your widget programs that is difficult to copy in a conventional widget program.

Now let's take a detailed look at how this program is written. The code that defines the `obwid` class is reproduced below for convenience.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro obWid::quit
widget_control,self.tlb,/destroy
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro obWid::genRand
rnd = randomn(s,1)
strout = strtrim(string(rnd),2)
widget_control,self.text,set_value = strout
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro obWidCleanup,tlb
widget_control,tlb,get_uvalue = self
obj_destroy,self
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro obWidEvents,event
widget_control,event.id,get_uvalue = cmd
call_method,cmd.method,cmd.object
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function obWid::init
; Create the widgets
self.tlb = widget_base(/col,title = 'First Object Widget')
self.text = widget_text(self.tlb,value = '',xsize = 50)
void = widget_button(self.tlb,value = 'Generate RND', $
           uvalue = {object:self,method:'genRand'})
void = widget_button(self.tlb,value = 'QUIT', $
           uvalue = {object:self,method:'quit'})
widget_control,self.tlb,/realize
```

```
widget_control,self.tlb,set_uvalue = self
xmanager,'obWid',self.tlb,event_handler = 'obWidEvents', $
  /no_block,cleanup = 'obWidCleanup'
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro obWid__define
define = {  obWid,                          $
            tlb:0L,                         $
            text:0L                         $
         }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

There are five procedures and one function (the `init` method). The object definition module, `obwid__define`, contains just two fields: `tlb`, which will be the top-level base, and `text`, which will be the widget id of the text field. These are the only two variables that must be present in the methods.

Here comes the hardest part of object widget programming…in an object widget program, **a method cannot be called as an event handler**. But when you press a button you want for something to happen. So what do you do? The way you circumvent this problem is by using a single event handler that dispatches the events to the appropriate method using `CALL_METHOD`. This is accomplished in this program with the procedure called `obWidEvents`, which was registered with the top-level base as the main event handler. The object reference and the appropriate method are stored in the user-value of the widget in the `INIT` method. These are retrieved in the event handler and then passed into `CALL_METHOD`. Thus the desired method gets executed right after the event is received in the event handler.

The remaining methods are relatively self-explanatory. The `quit` method simply destroys the top-level base whose `id` is a field of the persistent object reference `SELF`. The method `genRand` computes a random number, converts it to a string, and puts it into the text field (also available from the persistent object reference). The cleanup procedure (not a method!), as in a conventional widget program, has the top-level base as the only parameter and destroys the object reference, `SELF`. This procedure gets executed after the `quit` method as the top-level base is destroyed.

In this widget object the widgets were created in the INIT method. For applications with a more complicated graphical user interface you can create a separate method called `obWid::createWidgets` for instance. In the INIT method you would invoke the method as `self->createWidgets`.

It is interesting to compare the widget object code to a conventional widget program with the same interface. You can do this comparison by examining the conventional widget code called `conWid`, shown below. Note that it is shorter than the object widget code but it does not offer the same flexibility in running it from the command line.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro conWidQuit,event
widget_control,event.top,/destroy
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro genRand,event
rnd = randomn(s,1)
strout = strtrim(string(rnd),2)
text = widget_info(event.top,find_by_uname = 'TEXT')
widget_control,text,set_value = strout
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro conWid
tlb = widget_base(/col,title = 'First Object Widget')
text = widget_text(tlb,value = '',xsize = 50,uname = 'TEXT')
void = widget_button(tlb,value = 'Generate RND', $
            event_pro = 'genRand')
void = widget_button(tlb,value = 'QUIT', $
            event_pro = 'conWidQuit')
widget_control,tlb,/realize
xmanager,'conWid',tlb, /no_block
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

§ A more complicated widget object with a command line interface

We have seen how objects and widgets can be combined in a manner such that the events/methods can be processed through interaction with the widget controls and/or commands typed in at the command line (just seen in the class `OBWID`). In this section we are going to examine a more complicated widget object program and see how to write a command line interface wrapper. This will allow the user to interact with the widget controls directly or via commands typed at a custom command line, rather than the IDLDE command line.

Open up and compile the object class definition `COINTOSS__DEFINE.PRO`. Type the following command at the IDLDE command line and you should see an interface like that shown in figure 7.

```
IDL> coin_example
```

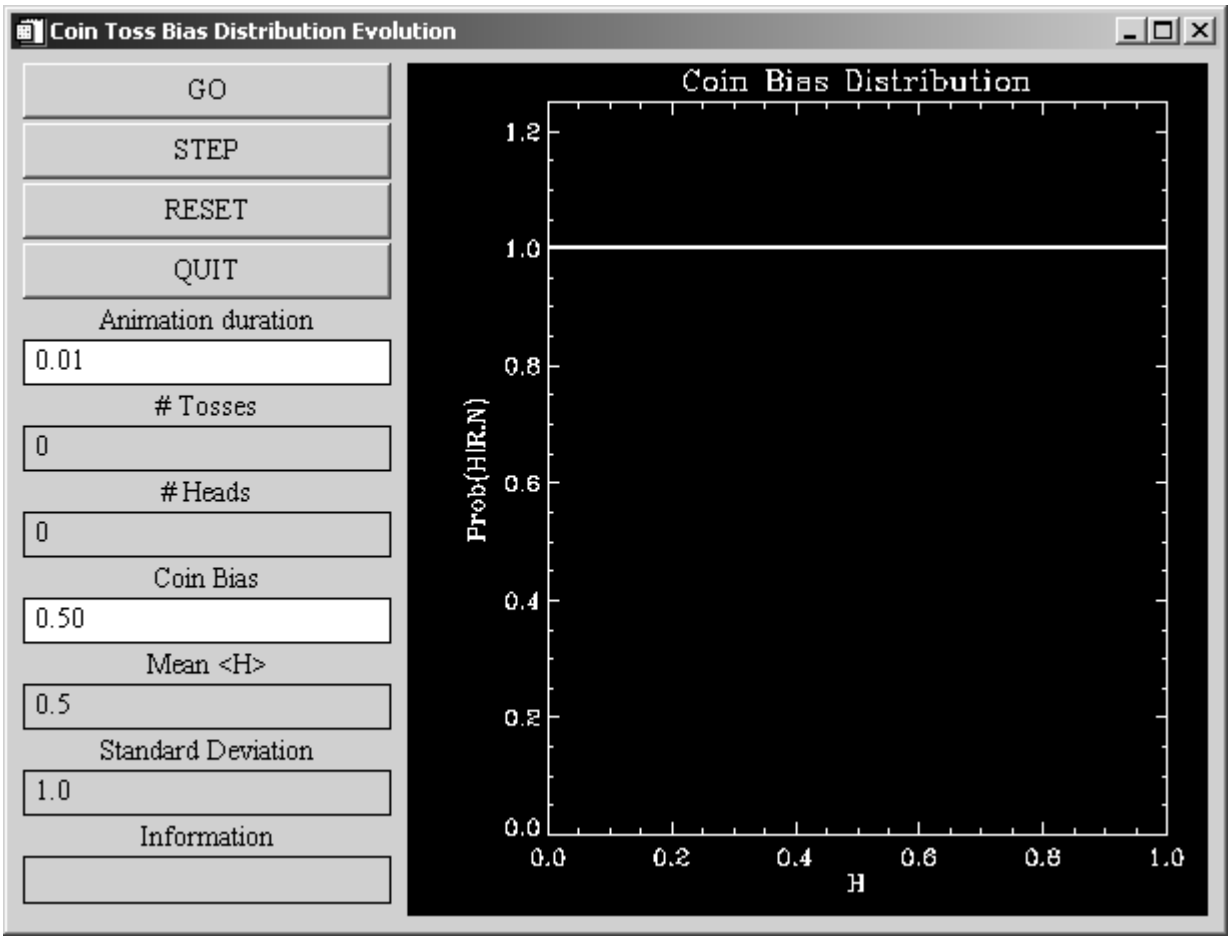

Fig. 7 Interface for an object instantiated from the COINTOSS class.

This whimsical widget application plots the evolution of the probability density of the bias of a coin determined through successive tosses. In the beginning there is complete ignorance as to the bias of the coin, so the density is a uniform distribution. However as the number of tosses increases, a better estimate of the bias is obtained as seen in the narrowing of the distribution. To see this narrowing, press the GO button and watch the probability density evolve. Note that this is not a true probability density since the area is not normalized to one. Rather it has been scaled so that the maximum value of the density is one for ease-of-viewing.

As the density evolves you should see it wobble around and get narrower. At any time you can press the `PAUSE` button. Pressing the `RESUME` button gets the evolution rolling again. If the evolution happened too quickly, press `RESET` and then `STEP` to step through the evolution one coin toss at a time. Alternatively you can type in a new time interval between animation updates (coin tosses) in the editable text field labeled `Animation duration`.

One of the (editable) text fields in the application interface is one labeled `COIN BIAS`. As the probability density evolves, the center of the density should converge to this value. Watch the density evolve for a few seconds then type in a new bias such as 0.2 and then hit a `<CR>`. You should see the whole distribution start to move towards the new bias. Next try a bias of 0.9 and see how the whole distribution moves towards this new bias. Note also that the application is resizable, even while the density is evolving.

After you have played around with the application and learned the functionality of the program, quit out of it, then instantiate the class at the command line and use the methods to control it. Examples of such commands are shown below.

```
IDL>  o = obj_new('cointoss')       ; instantiate the object class
IDL>  o -> startanimate             ; start animating the evolution
IDL>  o -> setbias,bias = 0.9       ; watch the distribution move right!
IDL>  o -> pause                    ; stop evolution temporarily
IDL>  o -> reset                    ; start over again
IDL>  o -> step                     ; step through evolution
IDL>  o -> resume                   ; resume animation of evolution
IDL>  o -> setbias,bias = 0.25      ; watch the distribution move left!
IDL>  o -> quit                     ; kill it
```

As in the class discussed in the previous section, `obWid`, it is easy to control this widget application from the IDL command line. However the end user of your application may not have access to the IDL command line when they run your application. (An example of this scenario is that you have given him/her the program with an embedded runtime license.) We would like to have a widget program giving the user the freedom to run the program with the widget controls or execute commands at a command line. The next step is to write this separate widget program so that the user can run the application from a command line not tied to the IDL Development Environment.

Open up and compile the program called `COINTOSSWRAPPER.PRO`. This is written as a conventional widget program because there is no real benefit gained from programming it as an object. After all, we want to run the widget object `COINTOSS` from a command line, not the program that is providing the command line.

Run `COINTOSSWRAPPER` and press the button labeled `Launch Object Widget Application`. You should see the application identical to that in figure 7 appear on the screen. Next type `startanimate` in the command line in the wrapper program and hit `<CR>`. You should see the evolution begin. The way this is written you only need to type the methods, not the object reference with the method invocation (i.e. not `o->method`). Therefore you can type things like

```
pause
resume
setbias,bias = 0.1
quit
```

making sure to hit a <CR> after typing in each command. Don't worry if you make a typographical error. The program will "swallow" these errors and you should not see any effect.

Let's take a look at the program and see how it works. Since it is so short we repeat it below for convenience.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ctaCleanup,tlb
widget_control,tlb,get_uvalue = pState
obj_destroy,(*pState).o
ptr_free,pState
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ctaEvents,event
uname = widget_info(event.id,/uname)
case uname of
'LAUNCH':   $
      begin
        widget_control,event.top,get_uvalue = pState
        if obj_valid((*pState).o) then return
        (*pState).o = obj_new('cointoss',group_leader = event.top)
      end
'QUIT':     widget_control,event.top,/destroy
'CMDLINE':  $
      begin
      widget_control,event.top,get_uvalue = pState
      widget_control,event.id,get_value = cmd
      o = (*pState).o
      result = execute('o->'+cmd[0],1)
      widget_control,event.id,set_value = ''
      end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinTossWrapper
tlb = widget_base(/col,title = 'Coin Toss Application')
void = widget_button(tlb,value = 'Launch Object Widget Program', $
      uname = 'LAUNCH')
font1 = "Comic Sans MS*16*BOLD"
font2 = "Comic Sans MS*16"
cmdline = cw_field(tlb,value = '',xsize = 50,title = 'COMMAND', $
      /return_events,uname = 'CMDLINE',/column,font = font1, $
      fieldfont = font2)
quit = widget_button(tlb,value = 'QUIT',uname = 'QUIT')

widget_control,tlb,/realize
state = {o:obj_new()}
pState = ptr_new(state)

widget_control,tlb,set_uvalue = pState
xmanager,'cta',tlb,event_handler = 'ctaEvents',cleanup = 'ctaCleanup'
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

The logic of the program is straightforward. When the button labeled `Launch Object Widget Program` is pressed, the object class `cointoss` is instantiated. The object reference is stored in the state pointer since it will be needed when commands are typed in the command line. In this program the command line is just a text widget from which we retrieve the contents whenever a `<CR>` is generated. The event handler determines the origin (id) of the (widget causing the) events using `UNAME` to avoid putting widget ids into the state structure and then performs the required function.

There are only three modules in this program: the widget definition module (`coinTossWrapper`), the event handler (`ctaEvents`), and the cleanup routine (`ctaCleanup`). The widget definition module is straightforward. The only new piece of code is in the construction of the state structure. The variable, `o`, is typed here as a null object via `o:obj_new()` and this null object is created here for later use.

The event handler handles three event types. The `QUIT` button event type should need no explanation. If the `LAUNCH OBJECT WIDGET PROGRAM` button is pressed then the class `COINTOSS` is instantiated and its object reference is stored in the `o` field of the state structure. This launches the widget object. The other event handled is when a `<CR>` is generated in the command line text field. The text in the field is pulled out and a command based on the object and the method typed in the command line is constructed:

```
widget_control,event.id,get_value = cmd
o = (*pState).o
result = execute('o->'+cmd[0],1)
```

If the command is not completed successfully then the program continues gracefully without an error or the program crashing. Finally the command line is cleared by setting its value to a null string.

The only other module is the cleanup routine which simply destroys the object and frees the state pointer.

§ A widget object with modal and non-modal dialog calls

We have seen how to program and incorporate modal and non-modal widgets in a conventional widget application. It is a simple matter to incorporate a modal widget program but we saw that it can be somewhat more challenging to add a non-modal widget. It turns out that if your application is written as a widget object, it is extremely simple to add a widget with non-modal functionality. Modal widgets are just as simple to call from a conventional widget program as a widget object application. Instead we will focus on how to program and include non-modal widgets in a widget object

application. The program will be identical in functionality to XCURVECHANGE discussed in the section on conventional widget programming.

Open up and compile the files OCURVECHANGE__DEFINE.PRO, NMOCURVE.PRO, and MCURVECHANGE.PRO. Note that MCURVECHANGE.PRO is the same program as the one used in XCURVECHANGE.PRO. To launch the main widget program remember to instantiate the class via

```
IDL> o = obj_new("ocurvechange")
```

Alternatively you can just type `curvechange` at the IDLDE command line and it will launch. The reason that this command works is that I have included a procedure (all the way at the bottom of the `OCURVECHANGE__DEFINE.PRO` after the `OCURVECHANGE__DEFINE` module) that instantiates the class for you. Launch the modal dialog and see that it works as it did in `XCURVECHANGE`. Launch the non-modal dialog and see that it has the proper non-modal functionality. Also, try to launch multiple instances of the non-modal dialog. Can you launch more than one?

The widget object class `OCURVECHANGE__DEFINE` is written in a manner similar to `OBWID__DEFINE`. There is a method called `CREATEWIDGETS` that is called by `INIT`, thus defining the widget layout. There are two keyword parameters in the init method: `group_leader` and `count`. The role of the group leader has been discussed previously in the context of `NMCURVECHANGE`. The keyword `count` is for use by the `group_leader` so that it can keep track of how many instances of the program are running. It is further used in this program as part of the title in the top-level base.

The `QUIT`, `CLEANUP`, and `DRAW` methods have been discussed before (in the context of the `SHAPE` class and its derived classes) so we will not repeat the explanations. Furthermore the event handler, `ocurveEvents`, is similar to the event handler discussed in `OBWID__DEFINE`. The methods that we are concerned with are `ocurveChange::nonModalDialog`, `ocurveChange::modalDialog`, `ocurveChange::setProperty`, and `ocurveChange::calcFun`. The method `setProperty` allows one to redefine the data and draw the redefined data to the draw widget. In general a setProperty that allows access is not a good idea because it breaks the "rule" that the data is shielded from the user. This `setProperty` method should be thought of as a *private* method, i.e. one that is not generally available by users of the object class.

The modal dialog is called from this class by the following method:

```
pro ocurveChange::modalDialog,event = event
newData = mCurveChange(parent = self.tlb,(*self.dataPtr)[*,0])
; Get here if the DONE button was pressed.
self->setProperty,data = newData,/draw
end
```

Note that the call to MCURVECHANGE requires passing in the parent (the top-level base here) and the array representing the independent variable. Upon exiting the modal dialog by pressing the DONE button, the new data is set using the setProperty method and the keyword set /DRAW tells the program to plot the redefined data.

The non-modal dialog is called from this class by the following method:

```
pro ocurveChange::nonModalDialog,event = event
nmoCurve,group_leader = self.tlb
end
```

That's all there is to it! We don't have to wait to receive an event structure from the non-modal dialog here as in the conventional widget program. So why is this so short and simple?

The answer is that the non-modal dialog is calling a method on the calling program (OCURVECHANGE) in response to its local events. So when you press one of the buttons labeled SIN or COS in the non-modal dialog, a local event triggers a method call. Either the string 'SIN' or 'COS' is passed into the method called CALCFUN and the calculation is performed in the object. Let's see how that's done in detail by looking at the non-modal code, NMOCURVE.PRO, repeated below for convenience.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro nmoCurveCleanup,tlb
widget_control,tlb,get_uvalue = pInfo
ptr_free,pInfo
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro nmoCurveEvent,event
uname = widget_info(event.id,/uname)
case uname of
'CURVEGROUP':     $
      begin
      widget_control,event.top,get_uvalue = pInfo
      if (*pInfo).group_leader ne 0L then begin
        widget_control,event.id,get_value = index
        widget_control,(*pInfo).group_leader,get_uvalue = object
        if index eq 0 then funcName = 'SIN' else funcName = 'COS'
        object->calcFun,funcName
      endif
      end
'DONE':                $
      begin
      widget_control,event.top,/destroy
      end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro nmoCurve,group_leader = group_leader
if n_elements(group_leader) eq 0 then begin
  group_leader = 0L
  floating = 0
```

```
endif else begin
  floating = 1
endelse
tlb = widget_base(group_leader = group_leader,/col,floating = floating)
; The next thing to check is if there is an instance of this program running
; already associated with the calling program.  If so then proceed
; no further.
registerName = 'nmo'+strtrim(string(group_leader),2)
if xregistered(registerName) then return
curveType = ['sin(x)','cos(x)']
curveGroup = cw_bgroup(tlb, curveType, /col, /exclusive,$
         label_top='Curve Type',/no_release,$
         frame=1,set_value=0,/return_index,uname = 'CURVEGROUP')
void = widget_button(tlb,value = 'DONE',uname = 'DONE')
widget_control,tlb,/realize

info = {group_leader:group_leader}

pInfo = ptr_new(info,/no_copy)
widget_control,tlb,set_uvalue = pInfo
xmanager,registerName,tlb,event_handler = 'nmoCurveEvent', $
  cleanup = 'nmoCurveCleanup'
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

This is a simple conventional widget program with three modules: the widget definition module, an event handler, and a cleanup routine. We begin the widget definition module as we did NMCURVECHANGE.PRO by determining if a widget like this one associated with the same group leader has been launched already. We do not allow multiple instances of this non-modal dialog. Next we set up the widgets exactly as in NMCURVECHANGE in which the events are differentiated based on the UNAME of each widget. The event handler handles two events. The DONE event is easy. If the user generates an event by changing the selection from SIN to COS or vice-versa then (1) the object reference of the calling program is pulled out of the group_leader (communicated within this program in the state pointer) and (2) the method CALCFUN is executed with either 'SIN' or 'COS' passed in as a string parameter. Within CALCFUN the function is executed using the built-in IDL function called CALL_FUNCTION and then the updated data is redrawn in the draw widget. When all this is performed the widget is not destroyed and remains until the user presses the DONE button. This is the non-modal behavior that we wanted to implement.

CREATING A SINGLE MULTI-MODULE APPLICATION

Up to this point we have discussed how to write a widget or object widget program that is effectively standalone. Some of these applications had features that allowed you to launch modal or non-modal widget dialogs. Launching any number of (formerly) standalone applications is, in general, just like incorporating a non-modal widget into a conventional widget application. Here we will discuss how to write a single application capable of launching multiple instances of a number of standalone widget applications.

Open up and compile the program `CLASSAPPLICATIONA.PRO`. This program is a conventional widget program that launches `XCURVECHANGE` and instantiates the class `OCURVECHANGE`. Run the program and launch some instances of each application. How many instances can you launch? What indication do you have in the application that there are more than one instance running? While you have more than one of these applications open, quit the main program. What happens?

In both of these applications, there is no information being communicated "up to the calling program," i.e. to `CLASSAPLICATION`. The only information being communicated is being sent to the applications in the form of the keywords `group_leader` and `count`. The group leader is present so that all applications close gracefully if the main application is closed before the others are closed. The keyword `count` is a counter that is incremented everytime a new instance of the application is launched. In these programs the communication is "one-way" in the sense that the individual applications that are launched from `CLASSAPPLICATIONA` get the `group_leader` and counter information from the calling program. However they do not communicate anything back to the calling program. If you want to include programs that do not have two-way communication like these, their implementation is easy. We'll discuss how to implement an application that has two-way communication in a few moments but let's briefly discuss how this one-way communication is done in the context of the two existing applications.

The event handler, `caEvent`, handles three types of button events (again, via the `UNAME` of the button widgets): 'QUIT', 'APP1', and 'APP2'. The `QUIT` event is simple. APP1 launches `xcurvechange` and APP2 instantiates the `ocurvechange` class. The programs `xcurvechange` and `ocurvechange` have been modified to accommodate these keywords. Note that nowhere in `CLASSAPPLICATIONA` is it necessary to destroy the object that was created when `ocurvechange` was instantiated. This is taken care of in its widget cleanup routine. Also note that the top-level bases of each application that is launched now displays a title with the value of the counter to indicate how many

times the application has been launched. This is a "dumb" counter in the sense that it does not decrement when you quit one of the applications.

The two applications that can be launched from CLASSAPPLICATIONA were very simple to implement in a multi-module program. Now let's try something more challenging. Let's modify SEUTILA.PRO so that it can be launched from this program. Furthermore let's modify SEUTILA so that the lowest eigenvalue gets sent from SEUTILA up to the calling program and appears in a text box. In particular, whenever the Schrödinger equation is solved due to some update in one of the values or potential, CLASSAPPLICATIONA will get sent a message. The first two applications in CLASSAPPLICATIONA did not have two-way communication but this will give us an opportunity to implement one that does.

Let's first modify the widget definition module for SEUTILA and add the new keywords, group_leader and notifyIds, shown below in boldface.

```
pro seutila,group_leader = group_leader,notifyIds = notifyIds
loadct,0,/silent
; Widget definition module
if n_elements(group_leader) eq 0 then group_leader = 0L
if n_elements(notifyIds) eq 0 then notifyIds = [0,0]
tlb = widget_base(/row,title='One Dimensional Schrodinger Equation Solver', $
      /tlb_size_events,mbar = bar,group_leader = group_leader)
.
.           [ WIDGET DEFINITION CODE HERE ]
.
state = {winVis:winVis, $
         winPix:winPix, $
.
.           [ MORE STATE STRUCTURE DEFINITIONS HERE]
.
         group_leader:group_leader, $
         notifyIds:notifyIds, $
         xPtr:ptr_new(/allocate_heap), $
         vPtr:ptr_new(/allocate_heap), $
         evalsPtr:ptr_new(/allocate_heap), $
         probPtr:ptr_new(/allocate_heap)}
```

Although we are modifying this program to be launched from another application, we also want to maintain the capability to launch it as a standalone application. This is the reason for the two new if-then statements at the top of the widget definition module. Also add the group_leader and notifyIds to the state structure. Look familiar? It should be since it is identical to what we did when incorporating the non-modal dialog in xcurvechange.

Next we must write the procedure that will notify the calling widget (CLASSAPPLICATION) the value of the minimum eigenvalue. The code for the

procedure, called seSendInfo, is shown below and it has a single argument, the state pointer.

```
pro seSendInfo,pState
theseVals = min(*(*pState).evalsPtr)
seInfo = {  seEvent,    $
            id:(*pState).notifyIds[0], $
            top:(*pState).notifyIds[1], $
            handler:0L, $
            minEigVal:theseVals     }
if widget_info((*pState).notifyIds[0],/valid) then begin
  widget_control,(*pState).notifyIds[0],send_event = seInfo
endif
end
```

The code for seSendInfo should also be familiar to you by now. It is virtually identical to the procedure, nmSendInfo, that was used in nmcurvechange.pro to send information to xcurvechange.pro. The argument is the state pointer which is packaged up along with a pseudo-event structure and sent to the calling widget (here the calling widget is the button labeled 'Schrodinger Equation Utility' and whose user name is SEUTIL). When is this procedure called? Whenever we solve the Schrödinger equation and wish to update the field. This is done in seSolve.

Now at the bottom of the event handler called seSolve, make the following additions (shown in boldface).

```
pro seSolve,event
!except = 0
widget_control,event.top,get_uvalue = pState
strout = ['Solving the Schrodinger equation','Please wait...']
widget_control,(*pState).info,set_value = strout
.
.           (UNMODIFIED CODE HERE)
.
wset,(*pState).winPix
sePlot,event
wset,(*pState).winVis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).winPix]

; If this program was called from another and the calling program
; wants to be notified, send an event.
if total((*pState).notifyIds) gt 0 then seSendInfo,pState
end
```

Recall that seSolve is the procedure that solves the Schrödinger equation whenever one of the parameters such as the potential is modified. When the solution is completed, seSendInfo is invoked, thus notifying the calling program to update the text field.

Those are all the necessary changes in SEUTILA. You can find the finished program in the course software distribution but it is named SEUTIL. Now take a look at CLASSAPPLICATIONA.PRO. First add a text field to the widget definition module to the state pointer as shown below. This will be where the minimum eigenvalue from SEUTIL will be displayed. Again, for event handling purposes, we specify a UNAME for the text field so that we don't have to carry around its widget ID in the state pointer.

```
pro classApplicationa
; Widget definition module
tlb = widget_base(/col,title = 'Class Application')
void = widget_button(tlb,value = 'Conventional Widget App',uname = 'APP1')
void = widget_button(tlb,value = 'Widget Object App',uname = 'APP2')
void = widget_button(tlb,value = 'Schrodinger Equation Utility', $
  uname = 'SEUTIL')
void = widget_button(tlb,value = 'QUIT',uname = 'QUIT')
txtField = cw_field(tlb,title = 'Min Eigenvalue',value = '', $
  uname = 'TXTFIELD')

widget_control,tlb,/realize
; Create a couple of counters for the conventional widget app and the
; object widget app.
state = {wCounter:0, oCounter:0 }
widget_control,tlb,set_uvalue = ptr_new(state,/no_copy)

xmanager,'CA',tlb,event_handler = 'caEvent',cleanup = 'caCleanup',/no_block
end
```

Finally modify the event handler so that it responds to events coming from SEUTILA and also include the ability to launch SEUTILA from this program (passing in group_leader and the notifyIds as keywords).

```
pro caEvent,event
thisEvent = tag_names(event,/structure_name)
if strupcase(thisEvent) eq 'SEEVENT' then begin
  widget_control,event.top,get_uvalue = pState
  txtField = widget_info(event.top,find_by_uname = 'TXTFIELD')
  widget_control,txtField,set_value = $
    strtrim(string(event.minEigVal),2)
  return
endif
uname = widget_info(event.id,/uname)
case uname of
'QUIT':     widget_control,event.top,/destroy
'APP1':     begin
            widget_control,event.top,get_uvalue = pState
            xcurveChange,group_leader = event.top,count = (*pState).wCounter
            (*pState).wCounter = (*pState).wCounter + 1
      end
'SEUTIL':   begin
            widget_control,event.top,get_uvalue = pState
            seutila,group_leader = event.top,notifyIds = [event.id,event.top]
            end
'APP2':     begin
```

```
            widget_control,event.top,get_uvalue = pState
            o = obj_new('ocurveChange',group_leader = event.top, $
              count = (*pState).oCounter)
            (*pState).oCounter = (*pState).oCounter + 1
      end
else:
endcase
end
```

Now try your modified program out. You should be able to launch a number of instances of SEUTIL from CLASSAPPLICATIONA. To see how it works, launch two instances of SEUTIL. Move the CLASSAPPLICATIONA widget out of the way of the two instances of SEUTIL so that you can clearly see the text field. Modify one of the parameters in SEUTIL and see if the text field gets updated. Repeat this with the other instance of SEUTIL. To sum up, we have taken a single standalone application (SEUTILA) and modified it to run in a multi-module program (CLASSAPPLICATION), sending messages to it at appropriate times.

EXERCISE:

Modify the appropriate code so that you can launch the COINTOSSWRAPPER program from CLASSAPPLICATION.PRO. Note that you also want to make sure that COINTOSSWRAPPER can be launched from the IDLDE as a standalone application.

PLUGGING YOUR APPLICATION INTO DAVE

§ What is DAVE?

DAVE, short for the Data Analysis and Visualization Environment, is a set of software tools for many of the inelastic spectrometers at the NIST Center for Neutron Research. Those of you who have used DAVE will now recognize that it is nothing more than a multi-module application similar to the one discussed in the previous section. One of the unique features of DAVE is that, if you have just reduced a set of data, this data is present when you launch the analysis program PAN for instance. This persistence of the data is a desirable feature in any multi-module application in which data of similar type are being manipulated, visualized, and/or analyzed. In this section we will discuss the program module DAVE.PRO and how to get your standalone widget program to work in DAVE.

The main program, DAVE.PRO, is written exactly like any other standalone (conventional) widget application. It is composed of a widget definition module and some event handlers. As always, the widget definition module provides the widget layout, defines the menus, and creates a state pointer containing information for the application. Each menu item is really a widget button with an event handler associated with it specified using the event_pro keyword. The event handlers for each menu item are usually in a file separate from DAVE.PRO since there are so many applications.

As in other conventional widget programs we use a state pointer to store essential program information. This state pointer, pState, points to a structure in which one of the fields is called davePtr (or just the DAVE pointer). The DAVE pointer is the internal standard data file format and its content is discussed in the next section. This DAVE pointer can be retrieved and modified by any of the programs launched from DAVE. The ability to retrieve this information is the main difference between the programs that you have seen previously in this course. (I didn't say it was any more difficult than anything we have covered so far…just different!)

If you have written an application for DAVE that will never require the state pointer then you needn't worry about retrieving it. Let's say that it is some kind of general tool, a procedure called xTool. You can simply add a menu item under GENERAL TOOLS in the widget definition module for DAVE.PRO as follows:

```
myAppButton = widget_button(genTools,value = 'My Application', $
      event_pro = 'launchMyApp')
```

Then you add the appropriate event handler called `launchMyApp.pro`, which might look something like the following:

```
pro launchMyApp,event
xTool,group_leader = event.top
end
```

Often however you will need the state pointer information for either retrieval (in an analysis or visualization application) or to populate (in a data reduction module). At runtime, the state pointer contains no real information other than a hierarchy of fields defining a data structure. Let's say that our fictitious application `xTool` needs the state pointer. Since the state pointer for `DAVE.PRO` is carried around in the user value for its top-level base and `xTool` has `DAVE`'s tlb for its `group_leader`, then retrieval is straightforward. You simply retrieve it by getting the user-value of the group_leader. The first few lines of the widget definition module for `xTool` might look like the following:

```
pro xTool, group_leader = group_leader
widget_control,group_leader,get_uvalue = statePtr
davePtr = (*statePtr).davePtr
.
.
.
```

You will want to be able to pass `davePtr` around within your `xTool` program so you will need to store `davePtr` in `xTool`'s local state pointer if it is a conventional widget program or as data associated with an object class if it is an object widget program. Note however that when you change any of the contents of the DAVE pointer, you have irrevocably changed it for all current applications as well as those that will be launched in the future. These changes might not be reflected until you cause an event to occur in the other previously launched applications but those other applications will eventually "find out" that you've changed the DAVE pointer. If your intent is to modify some of the contents of the DAVE pointer then you need not send any information "up to the `group_leader`." Since it is a pointer, the calling program (DAVE.PRO) will "know."

§ Notifying multiple applications

If you wish for your program to notify other applications that it has completed some task or the user has made some selection that another program should know about then you can use a technique similar to the one described in the section titled *Creating a single multi-module application*. You must append the necessary information to a custom event structure (just like our modification to `SEUTILA.PRO`). This event structure must then be sent back to the calling program via the event handler (as was done with the procedure `seSendInfo` in `SEUTIL`). If you wish to notify more than one program then you can send in a vector of widget ids in the `notifyIds` keyword. Recall that the first

element of the keyword is `event.id` and the second is `event.top`. To notify multiple applications, we must pass in a 2×n array of widget ids rather than the 2×1 array passed in to notify a single application where n is the number of widgets to be notified

So now that you know how to retrieve the information from the DAVE pointer in your application if it is called from DAVE, if you want to use this information then you need to understand the contents of the pointer. This is discussed in the following section.

§ Contents of the DAVE pointer

The remainder of this section contains information about the structure of the data being passed around within the DAVE program. It is useful for reference if you are writing an application that will use the data structure but is not a necessary part of this course.

Once you have access to the `DAVE` pointer you must have knowledge of its structure. The code in DAVE.PRO that defines the DAVE pointer is shown below:

```
commonStr = {instrument:'',$
             histPtr:ptr_new(/allocate_heap),$
             xlabel:'',$
             ylabel:'',$
             xtype:'',$
             ytype:'',$
             xunits: '',$
             yunits: '',$
             histLabel:'',$
             histUnits:'',$
             treatmentPtr:ptr_new(/allocate_heap),$
             commonStrPtr:ptr_new()}

dataStr = {commonStr:commonStr,$
           specificPtr:ptr_new(/allocate_heap)}

dataStrPtr = ptr_new(dataStr,/no_copy)

panContainer = obj_new('IDL_CONTAINER')
panInfo = {panContainer:panContainer}

; DAVE developers can put whatever they like into
; this extra structure
extra = { panInfo:panInfo }
extraPtr = ptr_new(extra,/no_copy)

daveStr =    {dataStrPtr:dataStrPtr, $
              descriPtr:ptr_new(/allocate_heap)}

davePtr = ptr_new(daveStr,/no_copy)
state = {davePtr:davePtr,extraPtr:extraPtr}

pState = ptr_new(state,/no_copy)
```

The code above implies that one accesses the DAVE pointer using the following dereference of pState:

```
davePtr = (*pState).davePtr
```

The pointer dataStrPtr points to a structure with two fields: `commonStr` and `specificPtr`. The first structure in the code above, `commonStr`, contains information that is common to data collected on any spectrometer and is required. Such information as the values in the histogram, the units of the two independent variables, the types for the independent variables, the two independent variables: x=energy transfer, channel number, time-of-flight, etc, y=group number, detector angle, wavevector transfer, etc. are contained in this structure. This structure is typically populated in the data reduction modules for the individual spectrometers.

The field in `commonStr` called `histPtr` contains the numeric values of the "data." The fields in this structure are `qty`, `err`, `x`, and `y`. In general `qty` and `err` are two-dimensional arrays where the first dimension represents the variation in `x` and the second represents the variation in `y`. Note that both `x` and `y` could be two-dimensional arrays as well. For example the fields of `histPtr` should have the following dimensions:

	dimensions
`qty`	(nx,ny)
`err`	(nx,ny)
`x`	nx′ or (nx′,ny′)
`y`	ny′ or (nx′,ny′)

The variable type should be defined as follows:

if xtype="points", nx′=nx
if xtype="histo", nx′=nx+1
if ytype="points",ny′=ny
if ytype="histo",ny′=ny+1

Examples of the other components of the structure commonStr are shown below:

```
instrument: 'dcs'
xlabel: 'energy transfer (meV)'
ylabel: 'Q'
xtype: 'histo'
ytype: 'points'
xunits: 'energy:meV'
yunits: 'angle:degree'
```

```
      histLabel: 'S(Q,w)'
      histUnits: 'Arbitrary units'
```

Although the DAVE pointer is straightforward in its structure, it can be a bit of a challenge for novices to get accustomed to pulling out values. However it is just a matter of remembering the rules of pointer dereferencing. Let's consider an example in which we wish to plot the x-variation of the data (with error bars) for the third value of `y`. We assume that `davePtr` is known perhaps via a retrieval from the `group_leader` as discussed above.

```
x = (*(*(*davePtr).dataStrPtr).commonStr.histPtr).x
y =((*(*(* davePtr).dataStrPtr).commonStr.histPtr).y)[3]
z = ((*(*(*davePtr).dataStrPtr).commonStr.histPtr).qty)[*,3]
zerr = ((*(*(*davePtr).dataStrPtr).commonStr.histPtr).err)[*,3]
plot,x,z,psym = 4,title = 'y='+strtrim(string(y),2)
errplot,x,z-zerr,z+zerr,width = 0
```

These commands will plot the data with error bars (no caps on the error bars as specified by the `width` keyword) and the title of the plot shows the particular value of `y` used in the cut of the data.

The other pointer, `specificPtr`, points to a structure containing fields specific to the neutron spectrometer from which the data was collected. For a time-of-flight neutron spectrometer for instance one might include the sample-to-detector distances, the chopper speeds, monochromator information, etc. In a backscattering spectrometer, one might include the Doppler velocity profile as well as the monitor spectra. The data reduction applications are the only routines that really should use information from the `specificPtr`. The general tools such as the visualization and analysis applications should <u>never</u> use any of this information. Inclusion of this information in the general tools would be at odds with the internal standard data format.

There is another pointer called `descriPtr` which is a pointer to a structure containing sample information such as the sample environment (temperature, applied magnetic field) or other descriptive sample conditions such as concentration. This information could be used to assemble large numbers of data files in which one of these parameters is varied systematically and the experimentalist wishes to analyze the data as a function of this variable. The construction of the structure to which `descriPtr` points is listed below:

```
void = {descriPtrStr, $
        name:'', $     ; name of series variable eg 'HField'
        units:'', $    ; units, eg 'T'
        legend:'', $   ; descriptive info, eg 'Average Sample Field'
        qty: 0.0, $    ; value, eg 0.05
        err: 0.0 $     ; 0.0
       }
```

The other pointer created when constructing the DAVE pointer is called `extraPtr`. This pointer points to a structure called `extra` that can contain whatever the DAVE developers like. There is a container object called `panContainer` for instance that keeps track of how many instances of the program PAN are running. PAN is a widget object program so a container object is appropriate for this task.

DESSERT

Now that we have worked through some fairly difficult material I would like to discuss an IDL application that is a bit more fun than the ones we have discussed thus far. Back in 1991, a scientist at the Johns Hopkins Applied Physics Laboratory named Ray Sterner wrote a version of the computer game TETRIS to illustrate the capabilities of IDL. If you are unfamiliar with the game, he describes the program as follows in his help utility:

> Tetris has 7 different playing pieces which drop down from the top of the screen. Points are scored by fitting these pieces together to form horizontal rows having no gaps. Such complete rows dissolve away and add to the player's score. Pieces may be moved left and right and rotated to fit together. The more rows completed the higher the score each newly completed row is worth. Extra credit is given for completing 4 rows at the same time. Upper or lower case key commands may be used, except that the Q (quit) command must be upper case. Both the current game scores and the highest score during the current session of IDL are displayed.
>
> The first version of this project was written using PC IDL in an afternoon as a test of the capabilities of IDL on a 386 class machine.

The original program was written in an afternoon. It is a conventional procedural program (uses no widgets) and allows you to control the orientation and program operation with the keyboard. Furthermore, two programs are necessary for its operation, (TETRIS.PRO and SPRINT.PRO) both of which use common blocks.

The purpose of the auxiliary procedure SPRINT is to allow easy updates of the scores in the window. The position of the scores on the screen must be stored in a global memory location, thus the need for a common block. Also, TETRIS stores all of its variables in a global memory location, requiring another common block. By now you should realize that there is no real need for a common block in a widget program since we can store all of the required program information in the user-value of the top-level base. Also, since we can easily write a widget object program, the object reference itself can contain the "global" information. In either case (conventional widget program or widget object program), we do not need common blocks. We can replace SPRINT with an object class that gets instantiated in the widget program. This object class can include methods allowing the exact same functionality (and memory!) as SPRINT.

In this final chapter for the course, we will look at how to rewrite TETRIS.PRO as a widget program (called XTETRIS.PRO) and removing common blocks. In doing so we will see how to replace SPRINT with an object class (PRINTOBJ__DEFINE.PRO). Due

to the length of XTETRIS.PRO, it is included in the course software distribution but not listed here. We will simply list some of the major code snippets that illustrate a particular point.

Some things that you should learn from this section are:

- Registering keyboard events in a draw widget (a "hack" is necessary for versions of IDL prior to IDL 5.6)
- Eliminating common blocks

Open up the file called TETRIS.PRO and SPRINT.PRO. Try running TETRIS. The controls for the game are located on the right hand side of the display window. Since there is a common block in the TETRIS code, you can run into conflicts between two (or more) instances of the same program.

Now open up XTETRIS.PRO and PRINTOBJ__DEFINE.PRO and run XTETRIS. Try launching a second instance of XTETRIS. What happens?

You cannot launch another instance of the application because this action is prevented using the XREGISTERED function as seen in some of our previous applications. The relevant lines of the widget definition module for XTETRIS are shown below:

```
pro xTetris
; Widget definition module
device,decomposed = 0
registerName = 'xtetris'
if xregistered(registerName) then return
.
.     [LOTS MORE CODE HERE]
.
xmanager,registerName,tlb,event_handler = 'xtEvents', $
  cleanup = 'xtCleanup',/no_block
end
```

So the first question you might have is how does one process keyboard events in a draw widget? This is not a feature that is supported in IDL prior to release 5.6. The technique used here is a clever hack suggested by J.D. Smith, a frequent newsgroup (comp.lang.idl-pvwave) poster. The idea is to create a bulletin-board base (i.e. one in which you have not specified either `row` or `column` keywords) and then put a draw widget and a text widget on it. The code below creates this bulletin-board base and puts the draw widget named `win` and text widget named `hiddenTextId` (1 pixel by 1 pixel in size!) on top of it.

```
xsize = 650 & ysize = 500
; Now create a "bulletin board" type base and put a draw widget and
; a text widget on it.
```

```
bulBase = widget_base(tlb);         bulletin board base
win = widget_draw(bulBase,xsize = xsize,ysize = ysize)
hiddenTextId = widget_text(bulBase,scr_xsize = 1,scr_ysize = 1,/all_events)
```

Note that the `all_events` keyword is set so that any new text the user types will generate an event. For this to work properly though we must ensure that the focus is always on the text widget. This is accomplished using a keyword to widget_control as follows:

```
widget_control,hiddenTextId,/input_focus
```

The id of the text widget is located in the state pointer so it is always available. Since it is possible to change input focus in any number of ways, we need to ensure that the focus is on the text widget at appropriate times. This means that we must repeat the procedure above in the event handler. The event handler code is listed below:

```
pro xtEvents,event
widget_control,event.top,get_uvalue = pState
thisEvent = tag_names(event,/structure_name)
case thisEvent of
'WIDGET_BUTTON': $
                  begin
                    uname = widget_info(event.id,/uname)
                    if uname eq 'EASY' then (*pState).duration = 0.5
                    if uname eq 'HARD' then (*pState).duration = 0.1
                  end
'WIDGET_TEXT_CH': $
                  begin
                    case strupcase(event.ch) of

                    ' ':  begin
                              if (*pState).loop eq 1 then xtRotate,event
                          end
                    'H':      xtHelp,event
                    'P':      (*pState).loop = 0
                    'R':      (*pState).loop = 1
                    'Q':      begin
                                xtQuit,event
                                return
                              end
                    'A':      begin
                              if (*pState).loop eq 1 then xtLeft,event
                              end
                    'F':      begin
                              if (*pState).loop eq 1 then xtRight,event
                              end
                    else:
                    endcase
                  end
else:  widget_control,(*pState).hiddenTextId,/input_focus
endcase

if (*pState).loop eq 1 then begin   ; update display
```

```
  xt_drop,event, done = d, range = r
  if d eq 1 then begin  ; piece is done moving
      xtFinish,event,r = r
  endif

  widget_control,(*pState).timerId,timer = (*pState).duration
endif

end
```

In this event handler we first determine if the event is a button event or a text widget event. The only button events handled in this event handler are if the user changes the skill level between EASY and HARD. This simply changes the duration between widget timer events. Note that TETRIS.PRO relied on the `WAIT` procedure for timing but we can use timer events here.

If the event is a keyboard event then we determine which key was pressed. The event is then dispatched to an appropriate event handler whether it rotates the current piece, moves it left or right, displays help, etc. These key presses are sorted out using a CASE statement. Prior to exiting the CASE statement, the input focus is allocated to the text widget again.

Next, if the game is in progress then the current piece drops another position and collision or game completion events get handled and dispatched appropriately. Finally the timer is fired again (if game is in progress).

In TETRIS.PRO there is a pretty significant common block called t_com, shown below:

```
common t_com, t_nx, t_ny, t_brd, t_p, t_seed, t_r, t_x, t_y, $
    t_pxa, t_pya, t_px, t_py, t_ca, t_c, $
    t_bell, t_wait, t_pflst, t_pfxa, t_pfya, t_pfx, t_pfy, $
    t_pc, t_lpc, t_hpc, t_ln, t_lln, t_hln, t_sc, t_lsc, t_hsc
```

We replace the common block with a substantial state pointer, shown below:

```
state = {   winPix:winPix,          $
            win:win,                $
            winVis:winVis,          $
            duration:0.05,          $
            symsize:2.0,            $
            hiddenTextId:hiddenTextId,    $
            direction:'S',          $
            xincrement:8,           $
            yincrement:8,           $
            xpos:xpos,              $
            ypos:ypos,              $
            loop:0,                 $
            timerID:bulBase,  $
            oText:obj_new('printObj'),$
```

```
            t_init_flag:0,          $
            wt:(-1),    $
            t_wait:0.1, $
            lev:0,      $
            t_ln:0,     $
            t_pc:0,     $
            t_sc:0,     $
            t_lln:0,    $
            t_lpc:0,    $
            t_lsc:0,    $
            t_hln:0,    $
            t_hpc:0,    $
            t_hsc:0,    $

            t_p:0,      $
            t_r:0,      $
            t_x:0,      $
            t_y:0,      $
            t_px:ptr_new(0),  $
            t_py:ptr_new(0),  $
            t_c:0,            $
            t_pfx:ptr_new(0), $
            t_pfy:ptr_new(0), $

            top:ptr_new(/allocate_heap),        $
            t_nx:t_nx,                          $
            t_ny:t_ny,                          $
            t_brd:ptr_new(/allocate_heap),      $
            t_ca:(1+indgen(8)),                 $
            t_pxa:ptr_new(/allocate_heap),      $
            t_pya:ptr_new(/allocate_heap),      $
            t_pflst:[3,3,7,7,7,5,5],            $
            t_pfxa:ptr_new(/allocate_heap),     $
            t_pfya:ptr_new(/allocate_heap)      $
            }

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

The only object in this conventional widget program is a field in the state structure called `oText` which belongs to the `PRINTOBJ` class. In the original TETRIS program, a procedure called SPRINT is used that has its own common block,

```
common sprint_com, xx, yy, ss, cc, ee, txt
```

which keeps track of the order in which text elements are added to the plot window. Whenever the score changes and the screen text needs to be updated, only the index identifying the text needs to be specified (and the new text of course). This common block doesn't have to appear in the application from which it's called (i.e. TETRIS.PRO).

We can replace this procedure with an object class whose data are the text itself, the index for each entry, color, erase color, position in device coordinates and whose

methods enable the user to add text, change text, clear all of the text (to start fresh), and display all of the text currently stored. Within the object, PRINTOBJ, all of the information is persistent and the methods provide access to the appropriate values. As an example consider the following sequence of commands:

```
IDL>  o = obj_new('printobj')       ; instantiate the object class
IDL>  window,0,xsize = 400,ysize = 400
```

Initialize a text string, 'Hi there', to appear at (325,310) in device coordinates.

```
IDL>  o-> addText,txtSize=1.2, x=325, y=310, text='Hi'
```

At this point, there is a single string appearing in the window. This string can be accessed using an index 1.

Initialize a text string, 'Hello', to appear at (125,100) in device coordinates

```
IDL>  o-> addText,txtSize = 3.4,x = 125,y = 100,text = 'Hello'
```

Now there are two strings. This latest one can be accessed using an index 2.

We can change the first string from 'Hi' to 'Oh no' by accessing index 1 and setting the keyword text to the new string.

```
IDL>  o -> changeText,index = 1,text = 'Oh no'
```

Be sure to destroy the object when you are finished.

```
IDL>  obj_destroy,o
```

COMPLETE CODE LISTINGS

The complete code listings are presented in the remainder of these course notes. They are presented here in their entirety for reference but they can also be downloaded from the ftp site as described in the introduction.

XTWOWIN.PRO
XCURVECHANGE.PRO
MCURVECHANGE.PRO
NMCURVECHANGE.PRO
DETACHEDWIDGET.PRO

DATA__DEFINE.PRO
SHAPE__DEFINE.PRO
CIRCLE__DEFINE.PRO
SQUARE__DEFINE.PRO
OBWID__DEFINE.PRO
COINTOSSWRAPPER.PRO
OCURVECHANGE__DEFINE.PRO
NMOCURVE.PRO

CLASSAPPLICATIONA.PRO
SEUTIL.PRO
COINTOSS__DEFINE.PRO

SPRINT.PRO
PRINTOBJ__DEFINE.PRO

XTWOWIN.PRO

```
; XTWOWIN.PRO
;
; Simple widget application that has two zoomable, resizable windows.
;
; Written by R.M. Dimeo (11/27/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xtwCleanup,tlb
widget_control,tlb,get_uvalue = pState
wdelete,(*pState).win1State.winPix
wdelete,(*pState).win2State.winPix
ptr_free,(*pState).win1State.xPtr,(*pState).win2State.xPtr
ptr_free,(*pState).win1State.yPtr,(*pState).win2State.yPtr
ptr_free,(*pState).dataPtr1, (*pState).dataPtr2
ptr_free,pState
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xtwQuit,event
widget_control,event.top,/destroy
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xtwPlot1,event
widget_control,event.top,get_uvalue = pState
data = *(*pState).dataPtr1
x = data[*,0] & y = data[*,1]
if (*pState).win1State.autoscale eq 1 then begin
  (*pState).win1State.xrange = [min(x),max(x)]
  dy = 0.1*(max(y)-min(y))
  (*pState).win1State.yrange = [min(y)-dy,max(y)+dy]
endif
plot,x,y,xrange = (*pState).win1State.xrange,xstyle = 1, $
  yrange = (*pState).win1State.yrange,ystyle = 1
*(*pState).win1State.xPtr = !x
*(*pState).win1State.yPtr = !y
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xtwPlot2,event
widget_control,event.top,get_uvalue = pState
data = *(*pState).dataPtr2
x = data[*,0] & y = data[*,1]
if (*pState).win2State.autoscale eq 1 then begin
  (*pState).win2State.xrange = [min(x),max(x)]
  dy = 0.1*(max(y)-min(y))
  (*pState).win2State.yrange = [min(y)-dy,max(y)+dy]
endif
plot,x,y,xrange = (*pState).win2State.xrange,xstyle = 1, $
  yrange = (*pState).win2State.yrange,ystyle = 1
*(*pState).win2State.xPtr = !x
*(*pState).win2State.yPtr = !y
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro refresh1,event
widget_control,event.top,get_uvalue = pState
wset,(*pState).win1State.winPix
xtwPlot1,event
wset,(*pState).win1State.winVis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win1State.winPix]
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro refresh2,event
widget_control,event.top,get_uvalue = pState
wset,(*pState).win2State.winPix
xtwPlot2,event
wset,(*pState).win2State.winVis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win2State.winPix]
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xtwWin1Draw,event
widget_control,event.top,get_uvalue = pState
case event.type of
0:      begin          ; button press
          (*pState).win1State.mouse = event.press
          if (*pState).win1State.mouse eq 4 then begin
           (*pState).win1State.autoscale = 1

            !x = *(*pState).win1State.xPtr
            !y = *(*pState).win1State.yPtr
```

```
          wset,(*pState).win1State.winPix
          xtwPlot1,event
          wset,(*pState).win1State.winVis
          device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win1State.winPix]
        endif
        if (*pState).win1State.mouse eq 1 then begin
          (*pState).win1State.xPos = event.x
          (*pState).win1State.yPos = event.y
          !x = *(*pState).win1State.xPtr
          !y = *(*pState).win1State.yPtr
          wset,(*pState).win1State.winVis
          device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win1State.winPix]
          (*pState).win1State.autoscale = 0
        endif
      end
1:    begin   ; button release
       if (*pState).win1State.mouse eq 1 then begin
        !x = *(*pState).win1State.xPtr
        !y = *(*pState).win1State.yPtr
     xll = Min([(*pState).win1State.xpos, event.x], Max=xur)
     yll = Min([(*pState).win1State.ypos, event.y], Max=yur)
        ll = convert_coord(xll,yll,/device,/to_data)
        ur = convert_coord(xur,yur,/device,/to_data)
        (*pState).win1State.xrange = [ll[0],ur[0]]
        (*pState).win1State.yrange = [ll[1],ur[1]]
        wset,(*pState).win1State.winPix
        xtwPlot1,event
        wset,(*pState).win1State.winVis
        device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win1State.winPix]
        (*pState).win1State.mouse = 0B
       endif
       if (*pState).win1State.mouse eq 4 then begin
        !x = *(*pState).win1State.xPtr
        !y = *(*pState).win1State.yPtr
        wset,(*pState).win1State.winPix
        xtwPlot1,event
        wset,(*pState).win1State.winVis
        device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win1State.winPix]
        (*pState).win1State.mouse = 0B
       endif
      end
2:    begin   ; mouse motion
        if (*pState).win1State.mouse eq 1 then begin
             xc = [(*pState).win1State.xpos,event.x,event.x,$
                   (*pState).win1State.xpos,$
                   (*pState).win1State.xpos]
             yc = [(*pState).win1State.ypos,(*pState).win1State.ypos,$
                   event.y,event.y,$
                   (*pState).win1State.ypos]
             wset,(*pState).win1State.winVis
             device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win1State.winPix]
             plots,xc,yc,/device
        endif
      end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xtwWin2Draw,event
widget_control,event.top,get_uvalue = pState
case event.type of
0:    begin          ; button press
        (*pState).win2State.mouse = event.press
        if (*pState).win2State.mouse eq 4 then begin
         (*pState).win2State.autoscale = 1

          !x = *(*pState).win2State.xPtr
          !y = *(*pState).win2State.yPtr
          wset,(*pState).win2State.winPix
          xtwPlot2,event
          wset,(*pState).win2State.winVis
          device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win2State.winPix]
        endif
        if (*pState).win2State.mouse eq 1 then begin
          (*pState).win2State.xPos = event.x
          (*pState).win2State.yPos = event.y
          !x = *(*pState).win2State.xPtr
          !y = *(*pState).win2State.yPtr
          wset,(*pState).win2State.winVis
          device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win2State.winPix]
          (*pState).win2State.autoscale = 0
```

```
          endif
        end
1:      begin   ; button release
         if (*pState).win2State.mouse eq 1 then begin
          !x = *(*pState).win2State.xPtr
          !y = *(*pState).win2State.yPtr
      xll = Min([(*pState).win2State.xpos, event.x], Max=xur)
      yll = Min([(*pState).win2State.ypos, event.y], Max=yur)
          ll = convert_coord(xll,yll,/device,/to_data)
          ur = convert_coord(xur,yur,/device,/to_data)
          (*pState).win2State.xrange = [ll[0],ur[0]]
          (*pState).win2State.yrange = [ll[1],ur[1]]
          wset,(*pState).win2State.winPix
          xtwPlot2,event
          wset,(*pState).win2State.winVis
          device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win2State.winPix]
          (*pState).win2State.mouse = 0B
         endif
         if (*pState).win2State.mouse eq 4 then begin
          !x = *(*pState).win2State.xPtr
          !y = *(*pState).win2State.yPtr
          wset,(*pState).win2State.winPix
          xtwPlot2,event
          wset,(*pState).win2State.winVis
          device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win2State.winPix]
          (*pState).win2State.mouse = 0B
         endif
        end
2:      begin   ; mouse motion
          if (*pState).win2State.mouse eq 1 then begin
               xc = [(*pState).win2State.xpos,event.x,event.x,$
                     (*pState).win2State.xpos,$
                     (*pState).win2State.xpos]
               yc = [(*pState).win2State.ypos,(*pState).win2State.ypos,$
                     event.y,event.y,$
                     (*pState).win2State.ypos]
               wset,(*pState).win2State.winVis
               device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).win2State.winPix]
               plots,xc,yc,/device
          endif
        end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xtwEvent,event
; This event handler will handle the resize events
widget_control,event.top,get_uvalue = pState
geom = widget_info(event.top,/geometry)
widget_control,(*pState).win1State.win,draw_xsize = fix(0.5*event.x), $
  draw_ysize = event.y
widget_control,(*pState).win2State.win,draw_xsize = fix(0.5*event.x), $
  draw_ysize = event.y
wdelete,(*pState).win1State.winPix
window,/free,/pixmap,xsize = fix(0.5*event.x),ysize = event.y
(*pState).win1State.winPix = !d.window
wdelete,(*pState).win2State.winPix
window,/free,/pixmap,xsize = fix(0.5*event.x),ysize = event.y
(*pState).win2State.winPix = !d.window
refresh1,event
refresh2,event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xTwoWin
device,decomposed = 0
loadct,0,/silent

tlb = widget_base(/row,title = 'Application with two resizable windows', $
  /tlb_size_events,mbar = bar)
filemenu = widget_button(bar,value = 'FILE',/menu)
void = widget_button(filemenu,value = 'QUIT',event_pro = 'xtwQuit')
xsize = 300 & ysize = 300
win1 = widget_draw(tlb,xsize = xsize,ysize = ysize,/button_events, $
  /motion_events,event_pro = 'xtwWin1Draw')
win2 = widget_draw(tlb,xsize = xsize,ysize = ysize,/button_events, $
  /motion_events,event_pro = 'xtwWin2Draw')
widget_control,tlb,/realize

widget_control,win1,get_value = winVis1
widget_control,win2,get_value = winVis2
window,/free,/pixmap,xsize = xsize,ysize = ysize
```

```
winPix1 = !d.window
window,/free,/pixmap,xsize = xsize,ysize = ysize
winPix2 = !d.window

; Create some stuff to plot
nx = 100
xlo = -10.0 & xhi = 10.0 & dx = (xhi-xlo)/(nx-1.)
x1 = xlo+dx*findgen(nx)
y1 = cos(0.5*x1)
data1 = [[x1],[y1]]
xlo = -1.0 & xhi = 1.0 & dx = (xhi-xlo)/(nx-1.)
x2 = xlo+dx*findgen(nx)
y2 = sin(2.0*x2)
data1 = [[x1],[y1]]
data2 = [[x2],[y2]]
dataPtr1 = ptr_new(data1,/no_copy)
dataPtr2 = ptr_new(data2,/no_copy)

win1State = {  win:win1,              $
               winPix:winPix1,        $
               winVis:winVis1, $
               autoscale:1,   $
               mouse:0B,              $
               xPtr:ptr_new(/allocate_heap), $
               yPtr:ptr_new(/allocate_heap), $
               xrange:fltarr(2),      $
               yrange:fltarr(2),      $
               xpos:0.0,      $
               ypos:0.0       $
               }
win2State = {  win:win2,              $
               winPix:winPix2,        $
               winVis:winVis2,        $
               autoscale:1,   $
               mouse:0B,              $
               xPtr:ptr_new(/allocate_heap), $
               yPtr:ptr_new(/allocate_heap), $
               xrange:fltarr(2),      $
               yrange:fltarr(2),      $
               xpos:0.0,      $
               ypos:0.0       $
               }

state = {      win1State:win1State,   $
               win2State:win2State,   $
               dataPtr1:dataPtr1,             $
               dataPtr2:dataPtr2              $
               }
pState = ptr_new(state,/no_copy)
widget_control,tlb,set_uvalue = pState

pEvent = {event,id:(*pState).win1State.win,top:tlb,handler:0L}
refresh1,pEvent
refresh2,pEvent

xmanager,'xtw',tlb,event_handler = 'xtwEvent',cleanup = 'xtwCleanup'
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

XCURVECHANGE.PRO

```
; XCURVECHANGE.PRO
;
; This widget program demonstrates the use of a non-modal and modal dialog
; widget utility.  This program is a non-blocking widget but it can call
; both a modal and non-modal widget program.
;
; Written by R.M. Dimeo (11/25/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xccCleanup,tlb
widget_control,tlb,get_uvalue = pState
wdelete,(*pState).winPix
ptr_free,(*pState).dataPtr
ptr_free,pState
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xccQuit,event
widget_control,event.top,/destroy
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xccDisplayCurve,event
widget_control,event.top,get_uvalue = pState
data = *(*pState).dataPtr
x = (*(*pState).dataPtr)[*,0]
y = (*(*pState).dataPtr)[*,1]
wset,(*pState).winPix
plot,x,y,xrange = [min(x),max(x)],/xsty
wset,(*pState).winVis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).winPix]
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xccNonModalLaunch,event
thisEvent = tag_names(event,/structure_name)
widget_control,event.top,get_uvalue = pState
case thisEvent of
'WIDGET_BUTTON': $
  begin
    x = (*(*pState).dataPtr)[*,0]
    nmCurveChange,x,group_leader = event.top,notifyIds = [event.id,event.top]
  end
'NMCURVEEVENT': $
  begin
       *(*pState).dataPtr = event.data
       xccDisplayCurve,event
  end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xccModalLaunch,event
widget_control,event.top,get_uvalue = pState
data = *(*pState).dataPtr
x = data[*,0]
; The call below freezes this application up until the DONE button in the
; modal dialog is pressed.
newData = mCurveChange(parent = event.top,x)
; Get here if the DONE button was pressed.
*(*pState).dataPtr = newData
xccDisplayCurve,event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro xcurveChange,group_leader = group_leader, count = count
; Widget definition module
; Note the group_leader keyword is optional so that this program may be
; called by other programs.
device,decomposed = 0
loadct,0,/silent
if n_elements(count) eq 0 then count = 0
thisTitle = 'Conventional widget implementation ('+strtrim(string(count),2)+')'
if n_elements(group_leader) eq 0 then group_leader = 0L
tlb = widget_base(group_leader = group_leader,/col,/tlb_frame_attr, $
    mbar = bar,title = thisTitle)

filemenu = widget_button(bar,value = 'FILE',/menu)
void = widget_button(filemenu,value = 'QUIT',event_pro = 'xccQuit')
xsize = 400 & ysize = 400
```

```
win = widget_draw(tlb,xsize = xsize,ysize = ysize)
rowBase = widget_base(tlb,/row,/align_center)
void = widget_button(rowBase,value = 'MODAL DIALOG',event_pro = 'xccModalLaunch')
void = widget_button(rowBase,value = 'NON-MODAL DIALOG', $
  event_pro = 'xccNonModalLaunch')
widget_control,tlb,/realize
; create a sin function to display
n = 50
x = (2.0*!pi/(n-1.))*findgen(n)
y = sin(x)
data = [[x],[y]]

widget_control,win,get_value = winVis
window,/free,/pixmap,xsize = xsize,ysize = ysize
winPix = !d.window

state = {      win:win,                              $
               winPix:winPix,                        $
               winVis:winVis,                        $
               dataPtr:ptr_new(data,/no_copy)               $
        }
pState = ptr_new(state,/no_copy)
widget_control,tlb,set_uvalue = pState

; Display curve
pseudoEvent = {event,id:win,top:tlb,handler:0L}
xccDisplayCurve,pseudoEvent

xmanager,'xc',tlb,/no_block,cleanup = 'xccCleanup', $
  event_handler = 'xccEvent'
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

MCURVECHANGE.PRO

```
; MCURVECHANGE.PRO
;
; Modal widget dialog that ceases operation of the calling widget program and
; passes out the new data upon exiting.
;
; Written by R.M. Dimeo (11/25/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro mccEvent,event
widget_control,event.top,get_uvalue = pInfo
uname = widget_info(event.id,/uname)
if uname eq 'DONE' then begin
  widget_control,(*pInfo).curveGroup,get_value = index
  if index eq 0 then begin    ; SIN
       (*pInfo).data[*,1] = sin((*pInfo).data[*,0])
  endif else begin     ; COS
       (*pInfo).data[*,1] = cos((*pInfo).data[*,0])
  endelse
  widget_control,event.top,/destroy
endif
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function mCurveChange,parent = parent,x
if n_elements(parent) eq 0 then begin
  parent = 0L
  modal = 0
endif
tlb = widget_base(group_leader = parent,modal = modal,/col)
curveType = ['sin(x)','cos(x)']
curveGroup = cw_bgroup(tlb, curveType, /col, /exclusive,$
         label_top='Curve Type',/no_release,$
         frame=1,set_value=0,/return_index,uname = 'CURVEGROUP')
void = widget_button(tlb,value = 'DONE',uname = 'DONE')
widget_control,tlb,/realize
y = fltarr(n_elements(x))
data = [[x],[y]]

info = {       curveGroup:curveGroup, $
               data:data             $
       }

pInfo = ptr_new(info,/no_copy)
widget_control,tlb,set_uvalue = pInfo
; Since this is a modal widget program, no actions will occur
; past this xmanager call until the DONE button is pressed.
xmanager,'mcurvechange',tlb,event_handler = 'mccEvent'

data = (*pInfo).data
ptr_free,pInfo
return,data

end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

NMCURVECHANGE.PRO

```
; NMCURVECHANGE.PRO
;
; Non-modal widget dialog.
;
; Written by R.M. Dimeo (11/25/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro nmSendInfo,pInfo
  widget_control,(*pInfo).curveGroup,get_value = index
  if index eq 0 then begin    ; SIN
       (*pInfo).data[*,1] = sin((*pInfo).data[*,0])
  endif else begin                    ; COS
       (*pInfo).data[*,1] = cos((*pInfo).data[*,0])
  endelse

  nmInfo = {NMCURVEEVENT,$
                        id:(*pInfo).notifyIDs[0],$
                        top:(*pInfo).notifyIDs[1],$
                        handler:0l,$
                        data:(*pInfo).data}
  if widget_info((*pInfo).notifyIDs[0],/valid_id) then begin $
    widget_control,(*pInfo).notifyIDs[0],send_event = nmInfo
  endif
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro nmccCleanup,tlb
widget_control,tlb,get_uvalue = pInfo
ptr_free,pInfo
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro nmccEvent,event
uname = widget_info(event.id,/uname)
case uname of
'CURVEGROUP':  $
       begin
       widget_control,event.top,get_uvalue = pInfo
       if (*pInfo).notifyIds[0] ne 0L then nmSendInfo,pInfo
       end
'DONE':               $
       begin
       widget_control,event.top,get_uvalue = pInfo
       if (*pInfo).notifyIds[0] ne 0L then nmSendInfo,pInfo
       widget_control,event.top,/destroy
       end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro nmCurveChange,x,group_leader = group_leader,notifyIds = notifyIds
; The first thing to check is if there is an instance of this program running
; associated with the calling program.  If so then proceed no further.
if n_elements(notifyIds) eq 0 then notifyIds = [0L,0L]
if n_elements(group_leader) eq 0 then begin
  group_leader = 0L
endif
tlb = widget_base(group_leader = group_leader,/col)
; Create a registered name that is unique for the calling program.  Use the
; group leader's id for instance...
registerName = 'nmcc'+strtrim(string(group_leader),2)
if xregistered(registerName) then return

curveType = ['sin(x)','cos(x)']
curveGroup = cw_bgroup(tlb, curveType, /col, /exclusive,$
         label_top='Curve Type',/no_release,$
         frame=1,set_value=0,/return_index,uname = 'CURVEGROUP')
void = widget_button(tlb,value = 'DISMISS',uname = 'DONE')
widget_control,tlb,/realize
y = fltarr(n_elements(x))
data = [[x],[y]]

info = {       curveGroup:curveGroup,        $
               notifyIds:notifyIds,          $
               data:data                     $
       }

pInfo = ptr_new(info,/no_copy)
widget_control,tlb,set_uvalue = pInfo
xmanager,registerName,tlb,event_handler = 'nmccEvent',cleanup = 'nmccCleanup'
```

```
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

DETACHEDWIDGET.PRO

```
; DETACHEDWIDGET.PRO
;
; This simple widget program shows how you can use a "HIDE/SHOW" feature
; to selectively display a detached widget control base.  This program
; uses MAPPING to toggle hiding and showing the base.  Some notable features
; are that it calls XMANAGER twice to register two bases with the local windows
; manager and it also sets the UVALUE of each base to the state pointer.
;
; Written by R.M. Dimeo (11/26/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro dwCleanup,tlb
widget_control,tlb,get_uvalue = pState
ptr_free,pState
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro dwEvent,event
uname = widget_info(event.id,/uname)
case uname of
'QUIT': $
  begin
    widget_control,event.top,/destroy
  end
'SHOW': $
  begin
    widget_control,event.top,get_uvalue = pState
    widget_control,(*pState).ctrlBase,/map
    widget_control,event.id,set_uname = 'HIDE'
  end
'HIDE': $
  begin
    widget_control,event.top,get_uvalue = pState
    widget_control,(*pState).ctrlBase,map = 0
    widget_control,event.id,set_uname = 'SHOW'
  end
; The following unames correspond to events coming from
; the control base.
'APPLE': $
  begin
    ; Note that here, EVENT.TOP is CTRLBASE itself.
       widget_control,event.top,get_uvalue = pState
    widget_control,(*pState).text,set_value = 'APPLE'
  end
'ORANGE': $
  begin
       widget_control,event.top,get_uvalue = pState
    widget_control,(*pState).text,set_value = 'ORANGE'
  end
else:
endcase

end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro detachedWidget
; Widget definition module
tlb = widget_base(/col,title = 'Detached widget example')
text = widget_text(tlb,value = '',xsize = 35)
hide = widget_button(tlb,value = 'HIDE/SHOW',uname = 'SHOW')
quit = widget_button(tlb,value = 'QUIT',uname = 'QUIT')
; Create the CTRLBASE, do not map it, but put the value of the TLB into the
; UVALUE of CTRLBASE.
ctrlBase = widget_base(group_leader = tlb,/col,title = 'Controls',map = 0)
apple = widget_button(ctrlBase,value = 'APPLE',uname = 'APPLE')
orange = widget_button(ctrlBase,value = 'ORANGE',uname = 'ORANGE')

; We must realize BOTH bases!
widget_control,tlb,/realize
widget_control,ctrlBase,/realize

state = {ctrlBase:ctrlBase,text:text}
pState = ptr_new(state,/no_copy)

; Set the UVALUES of both tlb and ctrlBase to pState!
widget_control,tlb,set_uvalue = pState
widget_control,(*pState).ctrlBase,set_uvalue = pState
; Now we must register BOTH bases with the local windows manager using xmanager.
xmanager,'dw',tlb,cleanup = 'dwCleanup',/no_block, $
  event_handler = 'dwEvent'
```

```
xmanager,'dw',(*pState).ctrlBase,event_handler = 'dwEvent'
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

DATA__DEFINE.PRO

```
; DATA__DEFINE.PRO
;
; DATA object class.
;
; Written by R.M. Dimeo (12/03/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro data::cleanup
ptr_free,self.dataPtr
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro data::getProperty, nchan = nchan, $
                       ngroups = ngroups

if arg_present(nchan) then nchan = self.nchan
if arg_present(ngroups) then ngroups = self.ngroups
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro data::display,colorTable = colorTable
if n_elements(colorTable) ne 0 then self.colorTable = colorTable
loadct,self.colorTable,/silent
xsize = 400 & ysize = 400
if self.winId eq (-1L) then begin
  window,/free,xsize = xsize,ysize = ysize
  self.winId = !d.window
endif else begin
  wset,self.winId
endelse
tv,smooth(congrid(bytscl(*self.dataPtr),xsize,ysize),10)
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function data::init,data
device,decomposed = 0
loadct,0,/silent
dsize = size(data)
self.nchan = dsize[1]
self.ngroups = dsize[2]
self.dataPtr = ptr_new(data,/no_copy)
self.colorTable = 0
self.winId = -1L
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro data__define
define = {     data,           $
               nchan:0,        $
               ngroups:0,      $
               winId:0L,       $
               colorTable:0,   $
               dataPtr:ptr_new()      $
        }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

SHAPE__DEFINE.PRO

```
; SHAPE__DEFINE.PRO
;
; Definition of the abstract class SHAPE.
;
; Written by R.M. Dimeo (12/01/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro shape::cleanup
ptr_free,self.xPtr, self.yPtr
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function shape::draw
plots,*self.xPtr,*self.yPtr,/device
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function shape::init
self.xPtr = ptr_new(/allocate_heap)
self.yPtr = ptr_new(/allocate_heap)
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro shape__define
define = {     shape,  $
               xPtr:ptr_new(),        $
               yPtr:ptr_new() $
         }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

CIRCLE__DEFINE.PRO

```
; CIRCLE__DEFINE.PRO
;
; Definition of the concrete class CIRCLE, a derived class of SHAPE.
;
; Written by R.M. Dimeo (12/01/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function circle::getArea
return,!pi*self.radius^2
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function circle::draw
plots,*self.xPtr,*self.yPtr,/device
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function circle::init,xc = xc,yc = yc, radius = radius
retVal =self -> shape::init()
if retVal ne 1 then return,retVal
if n_elements(xc) eq 0 then xc = 100.0
if n_elements(yc) eq 0 then yc = 100.0
if n_elements(radius) eq 0 then radius = 100.0
self.xc = xc
self.yc = yc
self.radius = radius
nth = 100
th = (2.0*!pi/(nth-1.))*findgen(nth)
*self.xPtr = xc + radius*cos(th)
*self.yPtr = yc + radius*sin(th)
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro circle__define
define = { circle, inherits SHAPE, $
               radius:0.0, $
               xc:0.0, $
               yc:0.0 }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

SQUARE__DEFINE.PRO

```
; SQUARE__DEFINE.PRO
;
; Definition of the concrete class SQUARE, a derived class of SHAPE.
;
; Written by R.M. Dimeo (12/01/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function square::getArea
return,self.side^2
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function square::draw
plots,*self.xPtr,*self.yPtr,/device
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function square::init,xc = xc,yc = yc, side = side
retVal =self -> shape::init()
if retVal ne 1 then return,retVal
if n_elements(xc) eq 0 then xc = 100.0
if n_elements(yc) eq 0 then yc = 100.0
if n_elements(side) eq 0 then side = 100.0
self.xc = xc
self.yc = yc
self.side = side
d = self.side
x = [self.xc-0.5*d,self.xc+0.5*d,self.xc+0.5*d,self.xc-0.5*d,self.xc-0.5*d]
y = [self.yc-0.5*d,self.yc-0.5*d,self.yc+0.5*d,self.yc+0.5*d,self.yc-0.5*d]
*self.xPtr = x
*self.yPtr = y
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro square__define
define = { square, inherits SHAPE, $
               side:0.0, $
               xc:0.0, $
               yc:0.0  }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

OBWID__DEFINE.PRO

```
; OBWID__DEFINE.PRO
;
; A first object widget program.
;
; Written by R.M. Dimeo (12/03/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro obWid::quit
widget_control,self.tlb,/destroy
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro obWid::genRand
rnd = randomn(s,1)
strout = strtrim(string(rnd),2)
widget_control,self.text,set_value = strout
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro obWidCleanup,tlb
widget_control,tlb,get_uvalue = self
obj_destroy,self
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro obWidEvents,event
widget_control,event.id,get_uvalue = cmd
call_method,cmd.method,cmd.object
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function obWid::init
; Create the widgets
self.tlb = widget_base(/col,title = 'First Object Widget')
self.text = widget_text(self.tlb,value = '',xsize = 50)
void = widget_button(self.tlb,value = 'Generate RND', $
               uvalue = {object:self,method:'genRand'})
void = widget_button(self.tlb,value = 'QUIT', $
               uvalue = {object:self,method:'quit'})
widget_control,self.tlb,/realize
widget_control,self.tlb,set_uvalue = self
xmanager,'obWid',self.tlb,event_handler = 'obWidEvents', $
  /no_block,cleanup = 'obWidCleanup'
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro obWid__define
define = {     obWid,                        $
               tlb:0L,                       $
               text:0L                       $
         }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COINTOSSWRAPPER.PRO

```
; COINTOSSWRAPPER.PRO
;
; This application provides a command line for the user to run the
; cointoss object widget application.
;
; Written by R.M. Dimeo (11/26/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ctaCleanup,tlb
widget_control,tlb,get_uvalue = pState
obj_destroy,(*pState).o
ptr_free,pState
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ctaEvents,event
uname = widget_info(event.id,/uname)
case uname of
'LAUNCH':      $
        begin
          widget_control,event.top,get_uvalue = pState
          if obj_valid((*pState).o) then return
          (*pState).o = obj_new('cointoss',group_leader = event.top)
        end
'QUIT':  $
widget_control,event.top,/destroy
'CMDLINE':     $
        begin
        widget_control,event.top,get_uvalue = pState
        widget_control,event.id,get_value = cmd
        o = (*pState).o
        result = execute('o->'+cmd[0],1)
        widget_control,event.id,set_value = ''
        end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinTossWrapper
tlb = widget_base(/col,title = 'Coin Toss Application')
void = widget_button(tlb,value = 'Launch Object Widget Program',uname = 'LAUNCH')
font1 = "Comic Sans MS*16*BOLD"
font2 = "Comic Sans MS*16"
cmdline = cw_field(tlb,value = '',xsize = 50,title = 'COMMAND', $
        /return_events,uname = 'CMDLINE',/column,font = font1, $
        fieldfont = font2)
quit = widget_button(tlb,value = 'QUIT',uname = 'QUIT')

widget_control,tlb,/realize
state = {o:obj_new()}
pState = ptr_new(state)

widget_control,tlb,set_uvalue = pState
xmanager,'cta',tlb,event_handler = 'ctaEvents',cleanup = 'ctaCleanup'
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

OCURVECHANGE__DEFINE.PRO

```
; OCURVECHANGE__DEFINE.PRO
;
; Object widget program that illustrates how to implement modal and non-modal
; dialog widget programs.
;
; Written by R.M. Dimeo (12/25/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ocurveCleanup,tlb
widget_control,tlb,get_uvalue = self
obj_destroy,self
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ocurveChange::cleanup
wdelete,self.winPix
ptr_free,self.dataPtr
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ocurveChange::quit, event = event
widget_control,self.tlb,/destroy
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ocurveChange::nonModalDialog,event = event
nmoCurve,group_leader = self.tlb
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ocurveChange::modalDialog,event = event
newData = mCurveChange(parent = self.tlb,(*self.dataPtr)[*,0])
; Get here if the DONE button was pressed.
self->setProperty,data = newData,/draw
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ocurveChange::draw,event = event
data = *self.dataPtr
x = data[*,0]
y = data[*,1]
wset,self.winPix
plot,x,y,psym = 0,xrange = [min(x),max(x)],/xsty
wset,self.winVis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,self.winPix]
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ocurveChange::setProperty,data = data,draw = draw
if n_elements(data) ne 0 then *self.dataPtr = data
if keyword_set(draw) then self->draw
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ocurveChange::calcFun,funcName
(*self.dataPtr)[*,1] = call_function(funcName,(*self.dataPtr)[*,0])
self->draw
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
PRO ocurveEvents, event
; This is the only event handler necessary for the program.  It makes a
; generic method call for an object, the information being provided
; by the UVALUE of the calling widgets.
widget_control, event.id, get_uvalue=cmd
call_method, cmd.method,cmd.object,event = event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ocurveChange::createWidgets
; Widget definition module
thisTitle = 'Object widget implementation ('+strtrim(string(self.count),2)+')'
if self.group_leader eq 0L then begin
  self.tlb = widget_base(/col,/tlb_frame_attr,mbar = bar,title = thisTitle)
endif else begin
  self.tlb = widget_base(group_leader = self.group_leader,/col, $
    /tlb_frame_attr,mbar = bar,title = thisTitle)
endelse
filemenu = widget_button(bar,value = 'FILE',/menu)
void = widget_button(filemenu,value = 'QUIT', $
       uvalue = {object:self,method:'quit'})

xsize = 400 & ysize = 400
win = widget_draw(self.tlb,xsize = xsize,ysize = ysize)
rowBase = widget_base(self.tlb,/row,/align_center)

void = widget_button(rowBase,value = 'MODAL DIALOG', $
```

```
        uvalue = {object:self,method:'modalDialog'})
void = widget_button(rowBase,value = 'NON-MODAL DIALOG', $
        uvalue = {object:self,method:'nonModalDialog'})
widget_control,self.tlb,/realize
widget_control,win,get_value = winVis
self.winVis = winVis
window,/free,/pixmap,xsize = xsize,ysize = ysize
self.winPix = !d.window
widget_control,self.tlb,set_uvalue = self
xmanager,'oc',self.tlb,$
         event_handler = 'ocurveEvents',$
         cleanup = 'ocurveCleanup',$
         /no_block
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function ocurveChange::init,group_leader = group_leader,count = count
device,decomposed = 0
loadct,0,/silent
if n_elements(group_leader) eq 0 then group_leader = 0L
self.group_leader = group_leader
if n_elements(count) eq 0 then count = 0
self.count = count
; create a sin function to display
n = 50
x = (2.0*!pi/(n-1.))*findgen(n)
y = sin(x)
data = [[x],[y]]
self.dataPtr = ptr_new(data,/no_copy)
self->createWidgets
self->draw
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro ocurveChange__define
define =       {       ocurveChange,                 $
                       group_leader:0L,              $
                       count:0,                      $
                       winVis:0L,                    $
                       winPix:0L,                    $
                       tlb:0L,                       $
                       dataPtr:ptr_new()             $
               }

end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro curveChange
o = obj_new('ocurveChange')
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

NMOCURVE.PRO

```
; NMOCURVE.PRO
;
; Non-modal widget dialog for the object program called ocurveChange.
;
; Written by R.M. Dimeo (11/25/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro nmoCurveCleanup,tlb
widget_control,tlb,get_uvalue = pInfo
ptr_free,pInfo
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro nmoCurveEvent,event
uname = widget_info(event.id,/uname)
case uname of
'CURVEGROUP':  $
       begin
       widget_control,event.top,get_uvalue = pInfo
       if (*pInfo).group_leader ne 0L then begin
         widget_control,event.id,get_value = index
         widget_control,(*pInfo).group_leader,get_uvalue = object
         if index eq 0 then funcName = 'SIN' else funcName = 'COS'
         object->calcFun,funcName
       endif
       end
'DONE':               $
       begin
       widget_control,event.top,/destroy
       end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro nmoCurve,group_leader = group_leader
if n_elements(group_leader) eq 0 then begin
  group_leader = 0L
  floating = 0
endif else begin
  floating = 1
endelse
tlb = widget_base(group_leader = group_leader,/col,floating = floating)
; The next thing to check is if there is an instance of this program running
; already associated with the calling program.  If so then proceed no further.
registerName = 'nmo'+strtrim(string(group_leader),2)
if xregistered(registerName) then return
curveType = ['sin(x)','cos(x)']
curveGroup = cw_bgroup(tlb, curveType, /col, /exclusive,$
         label_top='Curve Type',/no_release,$
         frame=1,set_value=0,/return_index,uname = 'CURVEGROUP')
void = widget_button(tlb,value = 'DONE',uname = 'DONE')
widget_control,tlb,/realize

info = {group_leader:group_leader}

pInfo = ptr_new(info,/no_copy)
widget_control,tlb,set_uvalue = pInfo
xmanager,registerName,tlb,event_handler = 'nmoCurveEvent', $
  cleanup = 'nmoCurveCleanup'
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

CLASSAPPLICATION.PRO

```
; CLASSAPPLICATION_NEW.PRO
;
; This is the main menu module for the multi-module application.  It will launch
; both object widget programs and conventional-style widget programs.
;
; Written by R.M. Dimeo (11/26/02)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro caCleanup,tlb
widget_control,tlb,get_uvalue = pState
ptr_free,pState
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro caEvent,event
thisEvent = tag_names(event,/structure_name)
if strupcase(thisEvent) eq 'SEEVENT' then begin
  widget_control,event.top,get_uvalue = pState
  txtField = widget_info(event.top,find_by_uname = 'TXTFIELD')
  widget_control,txtField,set_value = $
    strtrim(string(event.minEigVal),2)
  return
endif
uname = widget_info(event.id,/uname)
case uname of
'QUIT':        widget_control,event.top,/destroy
'APP1': begin
               widget_control,event.top,get_uvalue = pState
               xcurveChange,group_leader = event.top,count = (*pState).wCounter
               (*pState).wCounter = (*pState).wCounter + 1
        end
'SEUTIL':      begin
               widget_control,event.top,get_uvalue = pState
               seutil,group_leader = event.top,notifyIds = [event.id,event.top]
        end
'APP2': begin
               widget_control,event.top,get_uvalue = pState
               o = obj_new('ocurveChange',group_leader = event.top, $
                 count = (*pState).oCounter)
               (*pState).oCounter = (*pState).oCounter + 1
        end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro classApplication
; Widget definition module
tlb = widget_base(/col,title = 'Class Application')
void = widget_button(tlb,value = 'Conventional Widget App',uname = 'APP1')
void = widget_button(tlb,value = 'Widget Object App',uname = 'APP2')
void = widget_button(tlb,value = 'Schrodinger Equation Utility',uname = 'SEUTIL')
void = widget_button(tlb,value = 'QUIT',uname = 'QUIT')
txtField = cw_field(tlb,title = 'Min Eigenvalue',value = '',uname = 'TXTFIELD')

widget_control,tlb,/realize
; Create a couple of counters for the conventional widget app and the
; object widget app.
state = {wCounter:0, oCounter:0}
widget_control,tlb,set_uvalue = ptr_new(state,/no_copy)

xmanager,'CA',tlb,event_handler = 'caEvent',cleanup = 'caCleanup',/no_block
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

SEUTIL.PRO

```
;+
; NAME:
;       SEUTIL
;
; PURPOSE:
;
;      This simple widget program solves the one-dimensional Schrodinger
;      equation using the discrete variable approximation.  User can type in
;      a function of x that represents the potential, in IDL syntax, and the
;      eigenvalues and probability density are calculated for that potential.
;      The accuracy increases as the size of the Hamiltonian increases but at
;      the cost of processing speed.
;
; AUTHOR:
;
;      Robert M. Dimeo, Ph.D.
;      NIST Center for Neutron Research
;      100 Bureau Drive
;      Gaithersburg, MD 20899
;      Phone: (301) 975-8135
;      E-mail: robert.dimeo@nist.gov
;      http://www.ncnr.nist.gov/staff/dimeo
;
; CATEGORY:
;
;      Widgets, mathematics
;
; CALLING SEQUENCE:
;
;       SEUTIL
;
;
; INPUT FIELDS:
;
;              Note that a carriage return in any of the text fields below
;              begins the calculation
;
;              HAMILTONIAN SIZE:      size of the matrix to be diagonalized (default: 200)
;                                     Increase this to improve accuracy.
;              # EIGENFUNCTIONS:      Number of eigenfunctions to plot to the screen.
;              XLO:    lower bound for evaluation of the wavefunctions and potential
;              XHI:    upper bound for evaluation of the wavefunctions and potential
;              YLO:    lower bound for vertical display
;              YHI:    upper bound for vertical display
;              MASS:   mass of particle in atomic mass units (AMU)
;              POTENTIAL:     function describing the potential (meV)
;
; EXAMPLES FOR POTENTIALS (must use IDL syntax as shown below):
;
;              0.5*x^2                          ; simple harmonic oscillator
;              (abs(2.0*x))^15                  ; approximation to the infinite square well
;              abs(x)                           ; linear potential
;              20.0*((x/2.0)^2-1.0)^2 ; quartic potential (exhibits tunneling)
;
;      I have also included a step function in this program so that you
;      can see the effects of confinement in a square well.  An example
;      is listed below and is the default when the program is launched.
;
;              10.0+10.0*(step(x-1.0)-step(x+1.0))   ; finite square well
;
; DISCLAIMER
;
;      This software is provided as is without any warranty whatsoever.
;      Permission to use, copy, modify, and distribute modified or
;      unmodified copies is granted, provided this disclaimer
;      is included unchanged.
;
; MODIFICATION HISTORY:
;
;       Written by Rob Dimeo, September 30, 2002.
;              (10/01/02)     -Added built-in potential functions as menu items under
;                             POTENTIALS
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function step,x
a = 1D & b = 0D & c = 0.5D
y = (x gt 0)*a + ((x lt 0))*b + c*(x eq 0)
return,y
```

```
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function kronMat,n,i,j
i = fix(i) & j = fix(j)
k = 0*indgen(n,n)
ni = n_elements(i)
for ii = 0,ni-1 do begin
    if (i[ii] lt n) and (i[ii] ge 0) and $
       (j[ii] lt n) and (j[ii] ge 0) then begin
       k[i[ii],j[ii]] = 1
    endif
endfor
return,k
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro seSendInfo,pState
theseVals = min(*(*pState).evalsPtr)
seInfo = {     seEvent,       $
               id:(*pState).notifyIds[0], $
               top:(*pState).notifyIds[1], $
               handler:0L, $
               minEigVal:theseVals    }
if widget_info((*pState).notifyIds[0],/valid) then begin
  widget_control,(*pState).notifyIds[0],send_event = seInfo
endif
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro seCleanup,tlb
widget_control,tlb,get_uvalue = pState
ptr_free,(*pState).evalsPtr,(*pState).probPtr,(*pState).xPtr,(*pState).vPtr
wdelete,(*pState).winPix
ptr_free,pState
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro seQuit,event
widget_control,event.top,/destroy
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro sePlot,event
widget_control,event.top,get_uvalue = pState
if (n_elements(*(*pState).vPtr) eq 0) or $
   (n_elements(*(*pState).xPtr) eq 0) then return
evals = *(*pState).evalsPtr
evecs = *(*pState).probPtr
x = *(*pState).xPtr
v = *(*pState).vPtr
if (*pState).autoscale eq 1 then begin
  widget_control,(*pState).yhi,get_value = yhi
  widget_control,(*pState).ylo,get_value = ylo
  xlo = min(x) & xhi = max(x)
  (*pState).xrange = [xlo,xhi]
  (*pState).yrange = [float(ylo[0]),float(yhi[0])]
endif

plot,x,v,xrange = (*pState).xrange,yrange = (*pState).yrange, $
     xstyle = 1,ystyle = 1,xtitle = '!6x ( !6!sA!r!u!9 %!6!n )', $
     ytitle = '!6P(x)', title = '!6Probability Density',linestyle = 0, $
     thick = 3.0

widget_control,(*pState).nvals,get_value = nvals
nvals = fix(nvals[0]) < n_elements(v)
;nvals = (*pState).numVals2Plot
for i = 0,nvals-1 do begin
  oplot,x,evals[i]+evecs[*,i]
endfor
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro seSolve,event
!except = 0
widget_control,event.top,get_uvalue = pState
strout = ['Solving the Schrodinger equation','Please wait...']
widget_control,(*pState).info,set_value = strout

widget_control,(*pState).hsize,get_value = nx & nx = fix(nx[0])
widget_control,(*pState).xlo,get_value = xlo & xlo = double(xlo[0])
widget_control,(*pState).xhi,get_value = xhi & xhi = double(xhi[0])
widget_control,(*pState).pot,get_value = pot & pot = string(pot[0])
widget_control,(*pState).mass,get_value = m & m = double(m[0])
hbarc = 1973d3 & mc2 = m*931.5d9
beta = -(hbarc^2)/(2.0*mc2)
```

```
dx = (xhi-xlo)/(nx-1.0)
x = xlo+dx*dindgen(nx)
*(*pState).xPtr = x
vstring = 'v='+strtrim(pot,2)
r = execute(vstring,1)
if r ne 1 then begin
  strout = 'Input error'
  void = dialog_message(dialog_parent = event.top,strout)
  return
endif

*(*pState).vPtr = v
; Build the hamiltonian
h = dblarr(nx,nx)
ux = 1+bytarr(nx)
ix = indgen(nx)

h = (ux#(v-2.0*beta/dx^2))*kronMat(nx,ix,ix) + $
       (ux#(beta/dx^2)#ux)*kronMat(nx,ix,ix+1) + $
       (ux#(beta/dx^2)#ux)*kronMat(nx,ix,ix-1)


tired,h,evals,e,/double
triql,evals,e,h,/double

esort = sort(evals)
evals = evals[esort]
evecs = h[*,esort]
*(*pState).evalsPtr = evals
prob = evecs*evecs

widget_control,(*pState).ylo,get_value = ylo
widget_control,(*pState).yhi,get_value = yhi
ylo = float(ylo[0]) & yhi = float(yhi[0])

yfactor = yhi-ylo
nvals = (*pState).numVals2Plot
for i = 0,nvals-1 do begin
  sf = 0.05*yfactor/(max(prob[*,i])-min(prob[*,i]))
  prob[*,i] = sf*prob[*,i]
endfor

*(*pState).probPtr = prob
trans = string(evals[1:nx-1]-evals[0:nx-1])
feig = '(f10.2)'
f = '(e10.3)'
trans = ['--',trans]
strout = strarr(nx+2)
strout[0] = 'Eig (meV), Trans (meV)'
strout[1] = '---------------------------------'
strout[2] = strtrim(string(evals[0],format = feig),2)+ $
              ',  ------'
for i = 1,nx-1 do begin
  strout[2+i] = strtrim(string(evals[i],format = feig),2)+ $
              ',  '+strtrim(string(trans[i],format = f),2)
endfor
widget_control,(*pState).info,set_value = strout

wset,(*pState).winPix
sePlot,event
wset,(*pState).winVis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).winPix]

; If this program was called from another and the calling program
; wants to be notified, send an event.
if total((*pState).notifyIds) gt 0 then seSendInfo,pState
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro sedraw,event
widget_control,event.top,get_uvalue = pState

case event.type of
0:     begin          ; button press
         (*pState).mouse = event.press
         if (*pState).mouse eq 4 then begin
           (*pState).autoscale = 1

           wset,(*pState).winPix
           sePlot,event
           wset,(*pState).winVis
           device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).winPix]
```

```
          endif
          if (*pState).mouse eq 1 then begin
            (*pState).xbox[0] = event.x
            (*pState).ybox[0] = event.y
            wset,(*pState).winVis
            device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).winPix]
            empty
            (*pState).autoscale = 0
            widget_control,(*pState).win,/draw_motion_events
          endif
        end
1:      begin   ; button release
         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]]
          wset,(*pState).winPix
          sePlot,event
          wset,(*pState).winVis
          device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).winPix]
          (*pState).mouse = 0B
          widget_control,(*pState).win,draw_motion_events = 0
         endif
         if (*pState).mouse eq 4 then begin
          wset,(*pState).winPix
          sePlot,event
          wset,(*pState).winVis
          device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).winPix]
          (*pState).mouse = 0B
          widget_control,(*pState).win,draw_motion_events = 0
         endif
        end
2:      begin   ; mouse motion
          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).winVis
                device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).winPix]
                plots,xc,yc,/device
                empty
          endif
        end
else:
endcase
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro seEvents,event
widget_control,event.top,get_uvalue = pState
if (event.id eq (*pState).hsize) or $
   (event.id eq (*pState).xlo) or $
   (event.id eq (*pState).xhi) or $
   (event.id eq (*pState).mass) or $
   (event.id eq (*pState).pot) then begin
   seSolve,event
   return
endif
if (event.id eq (*pState).ylo) or $
   (event.id eq (*pState).nvals) or $
   (event.id eq (*pState).yhi) then begin
   wset,(*pState).winPix
   sePlot,event
   wset,(*pState).winVis
   device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).winPix]
   return
endif
; Handle the resizing events
ctrlgeom = widget_info((*pState).base,/geometry)
tlbgeom = widget_info(event.top,/geometry)
```

```
xsize = event.x
ysize = event.y

newxsize = xsize-ctrlgeom.xsize
newysize = ysize

widget_control,(*pState).win,draw_xsize = newxsize, $
               draw_ysize = newysize
wdelete,(*pState).winPix
window,/free,/pixmap,xsize = newxsize,ysize = newysize
(*pState).winPix = !d.window

wset,(*pState).winPix
sePlot,event
wset,(*pState).winVis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,(*pState).winPix]

end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro SEDefaults,event
widget_control,event.top,get_uvalue = pState
widget_control,(*pState).xlo,set_value = '-5.0'
widget_control,(*pState).xhi,set_value = '5.0'
widget_control,(*pState).hsize,set_value = '200'
widget_control,(*pState).nvals,set_value = '10'
widget_control,(*pState).ylo,set_value = '0.0'
widget_control,(*pState).yhi,set_value = '20.0'
widget_control,(*pState).mass,set_value = '1.0'
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro SEloadSHO,event
widget_control,event.top,get_uvalue = pState
seDefaults,event
strout = '0.5*x^2'
widget_control,(*pState).pot,set_value = strout
seSolve,event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro SEloadSquareWell,event
widget_control,event.top,get_uvalue = pState
seDefaults,event
strout = '10.0+10.0*(step(x-1.0)-step(x+1.0))'
widget_control,(*pState).pot,set_value = strout
seSolve,event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro SEloadLinearWell,event
widget_control,event.top,get_uvalue = pState
seDefaults,event
strout = 'abs(4.0*x)'
widget_control,(*pState).pot,set_value = strout
seSolve,event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro SEloadQuarticWell,event
widget_control,event.top,get_uvalue = pState
seDefaults,event
strout = '10.0*((x/2.0)^2-1.0)^2'
widget_control,(*pState).pot,set_value = strout
seSolve,event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro SEloadDoubleWell,event
widget_control,event.top,get_uvalue = pState
seDefaults,event
widget_control,(*pState).mass,set_value = '5.0'
strout = '10.0+10.0*(step(x-2.0)-step(x+2.0)) + 5.0*(step(x+0.5)-step(x-0.5))'
widget_control,(*pState).pot,set_value = strout
seSolve,event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro seutil,group_leader = group_leader,notifyIds = notifyIds
loadct,0,/silent
; Widget definition module
if n_elements(group_leader) eq 0 then group_leader = 0L
if n_elements(notifyIds) eq 0 then notifyIds = [0,0]
tlb = widget_base(/row,title = 'One Dimensional Schrodinger Equation Solver', $
      /tlb_size_events,mbar = bar,group_leader = group_leader)

filemenu = widget_button(bar,value = 'FILE',/menu)
potMenu = widget_button(bar,value = 'POTENTIALS',/menu)
```

```
void = widget_button(potMenu,value = 'Simple harmonic oscillator', $
       event_pro = 'SEloadSHO')
void = widget_button(potMenu,value = 'Finite square well', $
       event_pro = 'SEloadSquareWell')
void = widget_button(potMenu,value = 'Linear well', $
       event_pro = 'SEloadLinearWell')
void = widget_button(potMenu,value = 'Quartic well', $
       event_pro = 'SEloadQuarticWell')
void = widget_button(potMenu,value = 'Double square well', $
       event_pro = 'SEloadDoubleWell')

if !d.name eq 'WIN' then begin
  thisFont = "Comic Sans MS*16"
  bigFont = "Comic Sans MS*20*Bold"
endif else begin
  thisFont = -1
  bigFont = -1
endelse

base = widget_base(tlb,/col)
newBase = widget_base(base,/row)
thisbase = widget_base(newbase,/col,/grid_layout)
hsize = cw_field(thisbase,/col,title = 'Hamiltonian size',value = 200,$
        /string,font = thisFont,/return_events)
nvals = cw_field(thisbase,/col,title = '# of eigenfunctions',value = 10,$
        /string,font = thisFont,/return_events)
xlo = cw_field(thisbase,/col,title = 'XLO',value = -5.0,/string,$
      font = thisFont,/return_events)
xhi = cw_field(thisbase,/col,title = 'XHI',value = 5.0,/string,font = thisFont, $
      /return_events)
ylo = cw_field(thisbase,/col,title = 'YLO',value = 0.0,/string,$
      font = thisFont,/return_events)
yhi = cw_field(thisbase,/col,title = 'YHI',value = 20.0,/string,font = thisFont, $
      /return_events)
mass = cw_field(thisbase,/col,title = 'MASS (AMU)',value = '1.0',/string,$
       font = thisFont,/return_events)

void = widget_button(filemenu,value = 'QUIT',event_pro = 'seQuit', $
       font = thisFont)
info = widget_text(newbase,value = '',/editable,/scroll,xsize = 20,$
       ysize = 20,font = thisFont)
pot = cw_field(base,/col,title = 'Potential: V(x)', $
      value = '10.0+10.0*(step(x-1.0)-step(x+1.0))',/string,$
      fieldfont = bigFont,xsize = 45,/return_events, $
      font = thisFont)
xsize = 500 & ysize = 400
win = widget_draw(tlb,xsize = xsize,ysize = ysize,/button_events, $
      event_pro = 'sedraw')

geom = widget_info(base,/geometry)
newysize = geom.ysize
widget_control,win,draw_ysize = newysize
widget_control,tlb,/realize


widget_control,win,get_value = winVis
window,/free,/pixmap,xsize = xsize,ysize = newysize
winPix = !d.window

state = {winVis:winVis, $
         winPix:winPix, $
         autoscale:1, $
         xrange:[0.0,1.0], $
         yrange:[0.0,1.0], $
         xbox:[0.0,1.0], $
         ybox:[0.0,1.0], $
         mouse:0B, $
         numVals2Plot:100, $
         win:win, $
         xlo:xlo, $
         xhi:xhi, $
         ylo:ylo, $
         yhi:yhi, $
         pot:pot, $
         nvals:nvals, $
         mass:mass, $
         base:base, $
         hsize:hsize, $
         info:info, $
         group_leader:group_leader, $
```

```
         notifyIds:notifyIds, $
         xPtr:ptr_new(/allocate_heap), $
         vPtr:ptr_new(/allocate_heap), $
         evalsPtr:ptr_new(/allocate_heap), $
         probPtr:ptr_new(/allocate_heap)}


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

seSolve,{event,id:void,top:tlb,handler:0l}

xmanager,'seutil',tlb,event_handler = 'seEvents',$
         cleanup = 'seCleanup', /no_block

return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COINTOSS__DEFINE.PRO

```
;+
; NAME:
;       COINTOSS__DEFINE
;
; PURPOSE:
;
;       This simple widget program is a demonstration of combining objects and
;       widgets in an application.  The program simulates flipping a (possibly
;       biased) coin and displays the distribution (normalized so that its
;       maximum value is 1 for ease of visualization).
;
;       The particular distribution here is the conditional probability density
;       that one obtains R heads in N tosses of a coin.  This is of course the
;       binomial distribution.  Widget timer events are used to illustrate the
;       evolution of the probability density with successive coin tosses.  As the
;       number of coin tosses increases, the distribution becomes narrower indicating
;       more confidence in the mean as the actual bias for the coin.
;
;       The idea for this program originated in an example in the math text:
;       "Data Analysis: A Bayesian Tutorial", by Devinder Sivia (Oxford University
;       Press, 1996).
;
;       Users of the program can control the program using the widget controls
;       (which call the object's methods) or they can call the object's methods
;       directly (see example below).  Some example code is included in the
;       procedure called COIN_EXAMPLE at the after the object definition module.
;       To run the code, compile this file and type COIN_EXAMPLE.
;
; AUTHOR:
;
;       Robert M. Dimeo, Ph.D.
;       NIST Center for Neutron Research
;       100 Bureau Drive
;       Gaithersburg, MD 20899
;       Phone: (301) 975-8135
;       E-mail: robert.dimeo@nist.gov
;       http://www.ncnr.nist.gov/staff/dimeo
;
; CATEGORY:
;
;       Objects, widgets
;
; CALLING SEQUENCE:
;
;       object = obj_new('COINTOSS')
;
;
; INPUT PARAMETERS:
;
;       NONE
;
; INPUT KEYWORDS:
;
;       ntoss:  number of coin tosses at the outset of the program.
;
; REQUIRED PROGRAMS:
;
;        NONE
;
; COMMON BLOCKS:
;
;       NONE
;
; RESTRICTIONS
;
;       NONE
;
; OBJECT METHODS:
;
; There are no explicitly private object methods in IDL but the
; methods used in this class are divided into PUBLIC and PRIVATE
; to indicate which ones should be used by users who wish to run
; the program from the command line, for instance.  Execution of
; the program's controls from the command line is described in the
; example below.
;
; PUBLIC OBJECT PROCEDURE METHODS:
;
```

```
;   setBias -- Allows user to change value of the coin's bias ( 0 < bias < 1.0 ).
;                       Can be called with BIAS keyword set to a value or it reads the
;              current value of biasField if no keyword value is supplied.
;
;   pause -- Pauses evolution of probability density.
;
;   resume -- Resumes evolution of probability density if animation running.
;
;   reset -- Resets probability density parameters so no coins have been tossed.
;
;   startAnimate -- Commences the animation process.
;
;   setDuration -- Allows user to change the interval between animation
;                      frames.  Can be called with the DURATION keyword set
;                      to a value or it reads the current value of durField
;                      if no keyword value is supplied.
;
;   quit -- Destroy widget hierarchy and destroy object.
;
;
; PRIVATE OBJECT PROCEDURE METHODS:
;
;   toss_the_coin -- Calculates the new distribution after simulating the toss
;                    of the coin.
;
;   resize -- Allows dynamic resizing of the GUI interface.
;
;   animateCoinToss -- Executes object methods after animation has been started.
;
;   draw -- Displays current probability density in draw widget.
;
;   updateTimer -- Increments the widget timer (for animation).
;
;   displayInfo -- Displays information about the widget under the cursor.
;
; EXAMPLE
;
;       IDL>    o = obj_new('cointoss')       ; instantiate the object class
;
;               ;  The user can manipulate the widget controls or call the methods directly
;               ;  from the command line as shown below:
;
;       IDL>    o -> startanimate             ; start animating the evolution
;       IDL>    o -> setbias,bias = 0.9       ; watch the distribution move to the right!
;       IDL>    o -> pause                    ; stop evolution temporarily
;       IDL>    o -> reset                    ; start over again
;       IDL>    o -> step                     ; step through evolution
;       IDL>    o -> resume                   ; resume animation of evolution
;       IDL>    o -> setbias,bias = 0.25      ; watch the distribution move to the left!
;       IDL>    o -> quit                     ; kill it
;
; DISCLAIMER
;
;       This software is provided as is without any warranty whatsoever.
;       Permission to use, copy, modify, and distribute modified or
;       unmodified copies is granted, provided this disclaimer
;       is included unchanged.
;
; MODIFICATION HISTORY:
;
;       Written by Rob Dimeo, September 2, 2002.
;       RMD-09/04/02
;               Used the suggestion by David Fanning to
;               incorporate the object and method into the UVALUE of each widget.
;               This results in only a single short event handler that calls a
;               general object method (specified in the UVALUE) of the calling
;               widget.
;       RMD-09/06/02
;               Implemented exact expression for mean and standard deviation
;               for the distribution for small number of coin flips.  Also
;               modified procedures (where relevant) so that EVENT is a keyword
;               not necessary in any of the methods except the widget resizing.
;       RMD-11/26/02
;               Added group leader to the widget definition method (INIT) to allow
;               this program to be run from another.  Also wrote a separate widget
;               program that has a command line and allows the user to operate
;               this widget program using the commands above.  This program wrapper
;               is called COINTOSSAPPLICATION.PRO.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function bindist,h,r,n,$
```

```
                 mean = mean,$
                 st_dev = st_dev
; Calculate the mean and standard deviation of the distribution
; using the Gaussian approximation.


if n lt 125 then begin ; IDL can evaluate this for relatively small values
                       ; of n without a problem.  We can use the known exact
                       ; expressions for the mean and standard deviation for
                       ; the binomial distribution.
  dist = (double(h)^r)*(1.0-double(h))^(n-r)
  dmax = max(dist)
  dist = dist/dmax
  area = beta(r+1,n-r+1)
  mean = 1.0*(r+1.0)/(n+2.0)
  st_dev = sqrt((1.0*(r+2.0)*(r+1.0)/((n+3.0)*(n+2.0)))-mean^2)
endif else begin       ; IDL has a rough time for large values of n so
                                      ; use Gaussian approximation.
  mean = 1.0*r/n
  st_dev = sqrt(mean*(1.0-mean)/n)
  dist = exp(-0.5*((h-mean)/st_dev)^2)
endelse

return,dist
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinTossCleanup,tlb
widget_control,tlb,get_uvalue = self
obj_destroy,self
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::quit,event = event
strout = ['Exit the program']
if n_elements(event) ne 0 then begin
  if tag_names(event,/structure_name) eq 'WIDGET_TRACKING' then begin
    self->displayInfo,strout
    return
  endif
endif
widget_control,self.tlb,/destroy
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::cleanup
ptr_free,self.h,self.dist
wdelete,self.winPix
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::setproperty,buttonValue = buttonValue
if n_elements(buttonValue) ne 0 then $
  widget_control,self.goButton,set_value = buttonValue
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::getproperty,buttonValue = buttonValue
if arg_present(buttonValue) then $
  widget_control,self.goButton,get_value = buttonValue
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::setbias,bias = bias,event = event
strout = ['Change the coin bias']
if n_elements(event) ne 0 then begin

if tag_names(event,/structure_name) eq 'WIDGET_TRACKING' then begin
  self->displayInfo,strout
endif
endif
 if n_elements(bias) eq 0 then begin
  widget_control,self.biasField,get_value = bias
  bias = bias[0]
  bias = (bias gt 1.0) ? 0.5 : bias   ; default is 0.5 if user enters invalid bias
  bias = (bias lt 0.0) ? 0.5 : bias ; default is 0.5 if user enters invalid bias
  self.bias = bias
  thisFormat = '(f8.2)'
  widget_control,self.biasField,$
                 set_value = strtrim(string(bias,format = thisformat),2)
 endif else begin
  bias = (bias gt 1.0) ? 0.5 : bias ; default is 0.5 if user enters invalid bias
```

```
  bias = (bias lt 0.0) ? 0.5 : bias ; default is 0.5 if user enters invalid bias
  self.bias = bias
  thisFormat = '(f8.2)'
  widget_control,self.biasField,$
               set_value = strtrim(string(bias,format = thisformat),2)
 endelse
;endelse
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::setduration,duration = duration,event = event
strout = ['Change animation interval']
if n_elements(event) ne 0 then begin
  if tag_names(event,/structure_name) eq 'WIDGET_TRACKING' then begin
    self->displayInfo,strout
  endif
endif
 if n_elements(duration) eq 0 then begin
  widget_control,self.durField,get_value = duration
  self.duration = float(duration[0])
 endif else begin
  self.duration = duration
  thisFormat = '(f8.2)'
  widget_control,self.durField,$
               set_value = strtrim(string(duration,format = thisformat),2)
 endelse
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::toss_the_coin,event = event
; This method simulates tossing a biased coin
self.n = self.n + 1L
widget_control,self.ntosses,set_value = strtrim(string(self.n),2)
r = randomn(s,1,/uniform)
if r[0] lt self.bias then self.r = self.r + 1L
*self.dist = bindist(*self.h,self.r,self.n,mean = mean,st_dev = st_dev)
self.mean = mean
self.st_dev = st_dev
thisFormat = '(f10.4)'
widget_control,self.meanField,set_value = $
               strtrim(string(self.mean,format = thisformat),2)
widget_control,self.stdevField,set_value = $
               strtrim(string(self.st_dev,format = thisformat),2)
widget_control,self.nheads,set_value = $
               strtrim(string(self.r),2)
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::resize,xsize = xsize,ysize = ysize,event = event
; This method controls how the GUI is resized.  Note that we have to consider
; the screen real-estate taken by self.butBase and subtract this from the
; value of xsize and ysize that are passed in here (assumed to be event.x
; and event.y from the BASE widget's event structure) in order to get the
; proper new size for the draw window.

if n_elements(event) ne 0 then begin
  xsize = event.x
  ysize = event.y
endif
geom = widget_info(self.butBase,/geometry)

widget_control,self.win,draw_xsize = xsize-geom.xsize,draw_ysize = ysize
; Don't forget to resize the pixmap window!
wdelete,self.winPix
window,/free,/pixmap,xsize = xsize-geom.xsize,ysize = ysize
self.winPix = !d.window
self->draw
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::draw,event = event
xtitle = '!6H'
ytitle = '!6Prob(H!3|!6R,N)'
title = '!6Coin Bias Distribution'
wset,self.winPix
plot,*self.h,*self.dist,psym = 0,thick = 2.0,xrange = [0.0,1.0],xstyle = 1,$
     yrange = [0.0,1.25],ystyle = 1,xtitle = xtitle,ytitle = ytitle,title = title

wset,self.winVis
device,copy = [0,0,!d.x_size,!d.y_size,0,0,self.winPix]
```

```
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::updateTimer,event = event
; Fire off the timer after a period of self.duration
widget_control,self.timerID,timer = self.duration
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::animateEvents,event = event
if n_elements(event) ne 0 then begin
  if tag_names(event,/structure_name) eq 'WIDGET_TRACKING' then begin
    strout = ['Start/pause/resume animation']
    self->displayInfo,strout
    return
  endif
endif
self->getproperty,buttonValue = val
case val of
    'GO':      begin
               self->startanimate
               end
    'PAUSE':   begin
               self->pause
               end
    'RESUME':  begin
               self->resume
               end
else:
endcase
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::startAnimate,event = event
self->setbias
self.loop = 1
self->setproperty,buttonValue = 'PAUSE'
self->updateTimer
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::pause,event = event
;if n_params() eq 0 then event = {pseudo_event,x:0.0}
if self.loop eq 1 then begin
   self.loop = 0
   self->setproperty,buttonValue = 'RESUME'
endif
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::resume,event = event
if self.loop eq 0 then begin
  self->setbias
  self.loop = 1
  self->setproperty,buttonValue = 'PAUSE'
  self->updateTimer
endif
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro cointoss::step,event = event
if n_elements(event) ne 0 then begin
  if tag_names(event,/structure_name) eq 'WIDGET_TRACKING' then begin
    strout = ['Step through evolution']
    self->displayInfo,strout
    return
  endif
endif
  self->setbias
  self.loop = 0
  self->setproperty,buttonValue = 'RESUME'
  self->toss_the_coin
  self->draw
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro cointoss::animateCoinToss,event = event
if self.loop eq 1 then begin
  self->updateTimer
  self->toss_the_coin
  self->draw
```

```
endif
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coinToss::reset,event = event
strout = ['Reset evolution']
if n_elements(event) ne 0 then begin
  if tag_names(event,/structure_name) eq 'WIDGET_TRACKING' then begin
    self->displayInfo,strout
    return
  endif
endif
  npts = 200
  hlo = 0.0 & hhi = 1.0 & dh = (hhi-hlo)/(npts-1)
  h = hlo+dh*dindgen(npts)
  ones = 1.0+0.0*dindgen(npts)
  *self.dist = ones
  self.n = 0L
  self.r = 0L
  self.loop = 0
  self.mean = 0.5
  self.st_dev = 1.0
  widget_control,self.goButton,set_value = 'GO'
  widget_control,self.ntosses,set_value = strtrim(string(self.n),2)
  widget_control,self.meanField,set_value = $
               strtrim(string(self.mean,format = thisformat),2)
  widget_control,self.stdevField,set_value = $
               strtrim(string(self.st_dev,format = thisformat),2)
  self->draw
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro cointoss::displayInfo,strout
widget_control,self.infoField,set_value = strout
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
PRO coinEvents, event
; This is the only event handler necessary for the program.  It makes a
; generic method call for an object, the informating begin provided
; by the UVALUE of the calling widgets.
;
; Check for resize events
if tag_names(event,/structure_name) eq 'WIDGET_BASE' then begin
  widget_control,event.top,get_uvalue = self
  self->resize,event = event
  return
endif
; Handle the methods
Widget_Control, event.id, Get_UValue=cmd
call_method, cmd.method,cmd.object,event = event
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function coinToss::init,ntoss = ntoss,group_leader = group_leader
!except = 0
; First initialize the data
; Note that the distribution is normalized to a peak of 1.0 (always)
npts = 200
hlo = 0.0 & hhi = 1.0 & dh = (hhi-hlo)/(npts-1)
h = hlo+dh*dindgen(npts)
self.h = ptr_new(h,/no_copy)
ones = 1.0+0.0*dindgen(npts)
self.dist = ptr_new(ones,/no_copy)
self.loop = 0

; Build the GUI interface
smFont = "Times New Roman*16"
bsize = 75
tsize = 25
if n_elements(group_leader) eq 0 then begin
  tlb = widget_base(/row,title = 'Coin Toss Bias Distribution Evolution',$
        tlb_size_events = 1)
endif else begin
  tlb = widget_base(/row,title = 'Coin Toss Bias Distribution Evolution',$
        tlb_size_events = 1,group_leader = group_leader)
endelse
self.tlb = tlb

xsize = 400 & ysize = 300
butbase = widget_base(tlb,/col)
self.butbase = butbase
```

```
goButton = widget_button(butbase,value = 'GO',font = smFont,xsize = bsize,$
           uvalue = {method:'animateevents',object:self,name:'START'},$
           /tracking_events)

self.goButton = goButton

void = widget_button(butbase,value = 'STEP',font = smFont,xsize = bsize,$
       uvalue = {object:self,method:'step',name:'STEP'},/tracking_events)
void = widget_button(butbase,value = 'RESET',font = smFont,xsize = bsize,$
       uvalue = {object:self,method:'reset',name:'RESET'},/tracking_events)
void = widget_button(butbase,value = 'QUIT',font = smFont,xsize = bsize,$
       uvalue = {object:self,method:'quit',name:'QUIT'},/tracking_events)

void = widget_label(butbase,value = 'Animation duration',font = smFont)
durField = widget_text(butbase,value = '0.01',font = smFont,$
           /editable,xsize = tsize,/tracking_events,$
           uvalue = {object:self,method:'setduration',name:'DURATION'})
self.durField = durField

void = widget_label(butbase,value = '# Tosses',font = smFont)
ntosses = widget_text(butbase,value = '0',font = smFont,xsize = tsize)
self.ntosses = ntosses

void = widget_label(butbase,value = '# Heads',font = smFont)
nheads = widget_text(butbase,value = '0',font = smFont,xsize = tsize)
self.nheads = nheads

void = widget_label(butbase,value = 'Coin Bias',font = smFont)
biasField = widget_text(butbase,/editable,value = '0.5',$
            font = smFont,xsize = tsize,/tracking_events,$
            uvalue = {object:self,method:'setbias',name:'BIAS'})
self.biasField = biasField

void = widget_label(butbase,value = 'Mean <H>',font = smFont)
meanField = widget_text(butbase,value = '0.5',font = smFont,xsize = tsize)
self.meanField = meanField

void = widget_label(butbase,value = 'Standard Deviation',font = smFont)
stdevField = widget_text(butbase,value = '1.0',font = smFont,xsize = tsize)
self.stdevField = stdevField

void = widget_label(butbase,value = 'Information',font = smFont)
infoField = widget_text(butbase,value = '',font = smFont,xsize = tsize,$
            ysize = 1)
self.infoField = infoField

; Set the window base but don't map it.  After realization we'll
; determine how big the control base is and resize the draw window,
; then map the window base.
winBase = widget_base(self.tlb,/col,map = 0,$
      uvalue = {object:self,method:'animatecointoss',$
                name:'animatecointoss'})
self.timerID = winBase
win = widget_draw(winBase,xsize = xsize,ysize = ysize)
self.win = win

widget_control,self.tlb,/realize

; Get the size of the controls base.
geom = widget_info(butbase,/geometry)
; If the vertical dimension is larger than the draw widget's vertical
; dimension then resize the draw widget.
ysize = ysize > geom.ysize
widget_control,win,draw_ysize = ysize
widget_control,winBase,map = 1


self->setbias
self->setduration
widget_control,win,get_value = winVis
self.winVis = winVis
window,/free,/pixmap,xsize = xsize,ysize = ysize
self.winPix = !d.window

if n_elements(ntoss) ne 0 then begin
  if ntoss gt 0 then begin
    ntoss = ntoss > 1L
    if ntoss gt 1 then begin
      self.n = ntoss - 1L
      toss = randomn(s,self.n,/uniform)
```

```
      dummy = where(toss gt self.bias,r)
      self.r = r
    endif
    self->toss_the_coin
  endif
endif
self -> draw
widget_control,self.tlb,set_uvalue = self
xmanager,'coinToss::init',self.tlb,$
         event_handler = 'coinEvents',$
         cleanup = 'coinTossCleanup',$
         /no_block

return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro cointoss__define
; Definition of the COINTOSS class
define = {cointoss, $
          tlb:0L, $
          butBase:0L, $
          winVis:0L, $
          winPix:0L, $
          infoField:0L, $     ; text field that displays helpful messages
          durField:0L, $      ; input field for animation duration
          ntosses:0L, $       ; text field to display number of coin tosses
          nheads:0L, $        ; text field to display number of heads
          meanField:0L, $     ; text field for display of mean
          biasField:0L, $     ; text field for coin bias
          bias:0.0, $         ; numerical value for coin bias
          win:0L, $           ; draw window's widget id
          stdevField:0L, $    ; text field for display of standard deviation
          timerID:0L,$        ; the top-level base will contain timer information
          goButton:0L, $      ; go button which will change to pause/resume as
                              ; required by program operation.
          duration:0.0, $     ; timer interval
          loop:0, $           ; variable stating if we're evolving the distribution
                              ; 1 -> animating, 0 -> paused
          mean:0.0, $         ; mean
          st_dev:0.0, $       ; standard deviation
          dist:ptr_new(), $ ; probability coin has bias h
          h:ptr_new(), $      ; independent variable (heads)
          n:0L, $             ; number of coin tosses
          r:0L $              ; number of heads in n tosses
          }

return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro coin_example
o = obj_new('cointoss')
return
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

SPRINT.PRO

```
;-------------------------------------------------------------
;+
; NAME:
;       SPRINT
; PURPOSE:
;       Print text on screen, allows easy update.
; CATEGORY:
; CALLING SEQUENCE:
;       sprint, n, txt
; INPUTS:
;       n = text index number.    in
;       txt = new text.           in
; KEYWORD PARAMETERS:
;       Keywords:
;         INDEX=in on text setup returned text index.
;         SIZE=s text size (def=1).
;         COLOR=c text color (def=max).
;         ERASE=e erase color (def=0).
; OUTPUTS:
; COMMON BLOCKS:
;       sprint_com
;       sprint_com
; NOTES:
;       Notes:
;         To setup text do:
;         sprint, x, y, txt
;           x,y = screen coordinates of text.     in
;           txt = initial text at that location.  in
;         To display all current text do:
;         sprint
; MODIFICATION HISTORY:
;       R. Sterner, 6 Oct, 1991
;       R. Sterner, 1998 Jan 15 --- Dropped use of !d.n_colors.
;
; Copyright (C) 1991, Johns Hopkins University/Applied Physics Laboratory
; This software may be used, copied, or redistributed as long as it is not
; sold and this copyright notice is reproduced on each copy made.  This
; routine is provided as is without any express or implied warranties
; whatsoever.  Other limitations apply as described in the file disclaimer.txt.
;-
;-------------------------------------------------------------

        pro sprint, p1, p2, p3, index=ind, color=color, $
          erase=erase,  size=size, help=hlp

        common sprint_com, xx, yy, ss, cc, ee, txt

        if keyword_set(hlp) then begin
          print,' Print text on screen, allows easy update.'
          print,' sprint, n, txt'
          print,'   n = text index number.    in'
          print,'   txt = new text.           in'
          print,' Keywords:'
          print,'   INDEX=in on text setup returned text index.'
          print,'   SIZE=s text size (def=1).'
          print,'   COLOR=c text color (def=max).'
          print,'   ERASE=e erase color (def=0).'
          print,' Notes:'
          print,'   To setup text do:'
          print,'   sprint, x, y, txt'
          print,'     x,y = screen coordinates of text.     in'
          print,'     txt = initial text at that location.  in'
          print,'   To display all current text do:'
          print,'   sprint'
          return
        endif

        ;----------  Initialize  --------------
        if n_params(0) eq 3 then begin
```

```
  if n_elements(color) eq 0 then color = !p.color
  if n_elements(size)  eq 0 then size=1.
  if n_elements(erase) eq 0 then erase=0
  xyouts, /dev, p1, p2, p3, color=color, size=size
  if n_elements(xx) eq 0 then begin     ; Must start common.
    xx = intarr(1)+p1
    yy = intarr(1)+p2
    ss = fltarr(1)+size
    cc = intarr(1)+color
    ee = intarr(1)+erase
    txt = strarr(1)
    txt(0) = p3
  endif else begin                      ; Common exists.
    xx = [xx,p1]
    yy = [yy,p2]
    ss = [ss, size]
    cc = [cc, color]
    ee = [ee, erase]
    txt = [txt,p3]
  endelse
  ind = n_elements(xx)-1
  return
endif

;-----------  Update text  -----------
if n_params(0) eq 2 then begin
  if (p1 lt 0) or (p1 gt n_elements(xx)-1) then return
  if n_elements(erase) eq 0 then erase = ee(p1)
  xyouts, /dev, xx(p1), yy(p1), txt(p1), size=ss(p1), color=erase
  if n_elements(color) eq 0 then color = cc(p1)
  if n_elements(size) eq 0 then size = ss(p1)
  xyouts, /dev, xx(p1), yy(p1), p2, size=size, color=color
  txt(p1)=p2
  ss(p1)=size
  cc(p1)=color
  ee(p1)=erase
  return
endif

if n_params(0) eq 0 then begin
  for i = 0, n_elements(xx)-1 do begin
    xyouts, /dev, xx(i), yy(i), txt(i), size=ss(i), color=cc(i)
  endfor
  return
endif

end
```

PRINTOBJ__DEFINE.PRO

```
;+
; NAME:
;       PRINTOBJ__DEFINE
;
; PURPOSE:
;
;       This object class mimics the functionality of the routine SPRINT.PRO
;       written by Ray Sterner for use in the program TETRIS.PRO.  This object
;       class removes the common block.  This object class is used in the
;       widget program XTETRIS.PRO.
;
; AUTHOR:
;
;       Robert M. Dimeo, Ph.D.
;       NIST Center for Neutron Research
;       100 Bureau Drive
;       Gaithersburg, MD 20899
;       Phone: (301) 975-8135
;       E-mail: robert.dimeo@nist.gov
;       http://www.ncnr.nist.gov/staff/dimeo
;
; CATEGORY:
;
;       Objects, widgets
;
; CALLING SEQUENCE:
;
;       object = obj_new('PRINTOBJ')
;
;
; INPUT PARAMETERS:
;
;       NONE
;
; INPUT KEYWORDS:
;
;       NONE
;
; REQUIRED PROGRAMS:
;
;       NONE
;
; COMMON BLOCKS:
;
;       NONE
;
; RESTRICTIONS
;
;       NONE
;
; OBJECT METHODS:
;
; There are no explicitly private object methods in IDL but the
; methods used in this class are divided into PUBLIC and PRIVATE
; to indicate which ones should be used by users who wish to run
; the program from the command line, for instance.
;
; PUBLIC OBJECT PROCEDURE METHODS:
;
;   addText --          initializes text at device coordinates (x,y).
;
;   changeText --       changes text whose index is given by the order in which
;                       it was added using addText method.
;
;   displayAll --       displays all of the text that has been added using addText.
;
;   clear --            clears all of the text from memory that has been added
;                       using addText.
```

```
;
;
; PRIVATE OBJECT PROCEDURE METHODS:
;
;   cleanup -- frees the pointers
;
;   init -- standard object class initialization
;
; EXAMPLE
;
;       IDL>    o = obj_new('printobj')              ; instantiate the object class
;       IDL>    window,0,xsize = 400,ysize = 400
;               Initialize a text string, 'Hi there', to appear at (325,310) in device coordinates
;       IDL>    o-> addText,txtSize=1.2, x=325, y=310, text='Hi'
;               Initialize a text string, 'Hello', to appear at (125,100) in device coordinates
;       IDL>    o-> addText,txtSize = 3.4,x = 125,y = 100,text = 'Hello'
;               Change the first string from 'Hi' to 'Oh no'
;       IDL>    o -> changeText,index = 1,text = 'Oh no'
;               Destroy the object
;       IDL>    obj_destroy,o
;
;
; DISCLAIMER
;
;       This software is provided as is without any warranty whatsoever.
;       Permission to use, copy, modify, and distribute modified or
;       unmodified copies is granted, provided this disclaimer
;       is included unchanged.
;
; MODIFICATION HISTORY:
;
;       Written by Rob Dimeo, December 7, 2002.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro printObj::cleanup
compile_opt idl2,hidden
ptr_free,self.xPtr,self.yPtr
ptr_free,self.sizePtr,self.textPtr
ptr_free,self.colorPtr,self.indexPtr
ptr_free,self.erasePtr
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro printObj::addText, x = x, $
                                          y = y, $
                                          txtSize = txtSize,    $
                                          text = text, $
                                          color = color, $
                                          erase = erase
compile_opt idl2,hidden
if n_elements(erase) eq 0 then erase = 0
if n_elements(txtSize) eq 0 then txtSize = 1
if n_elements(color) eq 0 then color = !p.color

if n_elements(*self.xPtr) eq 0 then begin    ; first one
  *self.xPtr = [x]
  *self.yPtr = [y]
  *self.textPtr = [text]
  *self.colorPtr = [color]
  *self.sizePtr = [txtSize]
  *self.indexPtr = [1]
  *self.erasePtr = [erase]
endif else begin       ; adding another
  *self.xPtr = [*self.xPtr,x]
  *self.yPtr = [*self.yPtr,y]
  *self.textPtr = [*self.textPtr,text]
  *self.colorPtr = [*self.colorPtr,color]
  *self.sizePtr = [*self.sizePtr,txtSize]
  *self.erasePtr = [*self.erasePtr,erase]
  newIndex = n_elements(*self.indexPtr) + 1
  *self.indexPtr = [*self.indexPtr,newIndex]
endelse
```

```
; Display the latest addition
xyouts,/dev,x,y,text,color = color,charSize = txtSize

end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro printObj::changeText,     text = text, $
                                                     index = index, $
                                                     color = color, $
                                                     txtSize = txtSize, $
                                                     erase = erase
compile_opt idl2,hidden
if n_elements(erase) eq 0 then erase = (*self.erasePtr)[index - 1]
; First erase the old text
x = (*self.xPtr)[index - 1]
y = (*self.yPtr)[index - 1]
oldText = (*self.textPtr)[index - 1]
xyouts,/dev,x,y,oldText,color = erase, $
               charsize = (*self.sizePtr)[index - 1]

; Now display the new text
if n_elements(color) eq 0 then color = (*self.colorPtr)[index - 1]
if n_elements(txtSize) eq 0 then txtSize = (*self.sizePtr)[index - 1]
xyouts,/dev,x,y,text,charsize = txtSize,color = color
(*self.colorPtr)[index-1] = color
(*self.sizePtr)[index-1] = txtSize
(*self.textPtr)[index-1] = text
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro printObj::displayAll
compile_opt idl2,hidden
n = n_elements(*self.xPtr)
if n eq 0 then return
for i = 0,n-1 do begin
       x = (*self.xPtr)[i]
       y = (*self.yPtr)[i]
       text = (*self.textPtr)[i]
       color = (*self.colorPtr)[i]
       txtSize = (*self.sizePtr)[i]
       xyouts,/dev,x,y,text,color = color,charSize = txtSize
endfor
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro printObj::clear
ptr_free,self.xPtr,self.yPtr
ptr_free,self.sizePtr,self.textPtr
ptr_free,self.colorPtr,self.indexPtr
ptr_free,self.erasePtr
self.xPtr = ptr_new(/allocate_heap)
self.yPtr = ptr_new(/allocate_heap)
self.sizePtr = ptr_new(/allocate_heap)
self.textPtr = ptr_new(/allocate_heap)
self.colorPtr = ptr_new(/allocate_heap)
self.indexPtr = ptr_new(/allocate_heap)
self.erasePtr = ptr_new(/allocate_heap)
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
function printObj::init
compile_opt idl2,hidden
self.xPtr = ptr_new(/allocate_heap)
self.yPtr = ptr_new(/allocate_heap)
self.sizePtr = ptr_new(/allocate_heap)
self.textPtr = ptr_new(/allocate_heap)
self.colorPtr = ptr_new(/allocate_heap)
self.indexPtr = ptr_new(/allocate_heap)
self.erasePtr = ptr_new(/allocate_heap)
return,1
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro printObj__define
compile_opt idl2,hidden
```

```
define = {      printObj,                    $
                xPtr:ptr_new(),              $
                yPtr:ptr_new(),              $
                sizePtr:ptr_new(),     $
                textPtr:ptr_new(),     $
                colorPtr:ptr_new(),    $
                erasePtr:ptr_new(),    $
                indexPtr:ptr_new()     $
                 }
end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```