System Libraries and Computer Graphics

In this exercise you will make use of some simple computer graphics, and you will be introduced to the ideas of subroutine libraries and API's.

1. Problem

This assignment is the second half of the project begun in the last exercise: to plot a graph of the trajectory of an object fired into the air by a cannon. This program will read the data file produced by the previous exercise, putting the data in this file into a pair of arrays, and then pass these arrays to a subroutine which will draw the curve.

2. Computer Graphics

One of the most common uses of computers in scientific work is to plot data from experiments or theoretical calculations. Computer graphics programs can range from complete systems which do almost everything for you at the click of a mouse button, to collections of subroutines which perform simple, primitive operations such as drawing a line from one point on the screen to another. When writing programs which create computer graphics it is almost always the case that you will use a collection of pre-written subroutines to perform some set of tasks associated with drawing or rendering a graphical image.

For this assignment, as in a previous exercise, you will make use of a subroutine which someone else has already written. The subroutine `CURVE` takes as arguments two arrays containing the x and y values of a series of points, assumed to be in order, and then draws a series of line segments (called a "polyline") between all of the points in the list. If the points are close enough and there are enough of them, it's possible to draw what looks like a smooth curve.

The command to use the `CURVE` subroutine is,

```
CALL CURVE(NP, X, Y, IC)
```

where `NP` is an integer giving the number of points to be plotted in the polyline, and `X` and `Y` are real arrays containing the x and y values of the points to be plotted. These arrays must be at least `NP` long, but it is possible for them to be longer. `IC` is an integer which specifies the color of the (poly)line being drawn. The possible values are:

```
0 = BLACK      1 = RED
2 = GREEN      3 = YELLOW
```

```
4 = BLUE       5 = MAGENTA
6 = CYAN       7 = WHITE
```

The first time your program calls `CURVE` the graphics system is initialized and a window will appear on your screen. You can make as many calls to `CURVE` as you need to create your drawing, using as many colors as you like. Each call to `CURVE` will draw another polyline.

The viewing window in which all drawing takes place will be set from the minimum and maximum values of the x and y coordinates in the very first call to `CURVE`. Thus if you are drawing a complicated object you may want to first draw a "frame" around the area in which the drawing is to be made to get the viewing window set correctly. (The background color of the screen is black, so if you don't want this frame to be visible you could draw it also in black.)

When your program is finished drawing, it is necesary to terminate the graphics display, which will remove the window and "clean up" the graphics system. However, the program should first pause so that the user can look at the drawing. (If the program terminated right away the window would flash on the screen and then immediately disappear, which is not useful.) The subroutine `CURVE` will take care of both tasks if you call it with `NP` less than zero. This is the signal to the graphics system that all drawing has been completed, and the subroutine will then pause and wait for the user to push the ENTER key. Once the user presses ENTER, the graphics window will disappear.

Thus you should call `CURVE` at least twice: once to draw your curve, and then again with a negative value for `NP` to pause the program and then terminate the graphics when the user presses ENTER.

More information about `CURVE` is contained in the comments at the beginning of the subroutine, which is in the file `curve.f`. You can copy this file from my `fortran` directory, or download it from the web. Assuming we are on the same computer, the command to copy the file from my fortran subdirectory to your current directory is

```
% cp ~myers/fortran/curve.f  .
```

To copy them via the web you can right-click on the web link for the file and select the menu item labeled either "Save Link as.." (for Netscape) or "Save Target as..." (Internet Explorer).

You will also need several "header" files which are incorporated into `curve.f` at compile time via `INCLUDE` statements. The files you need are `CVstate.f`, `fvogl.h`, and `fvodevice.h`. The "`.h`" extension on the file names indicates that they are "header" files, not complete Fortran programs or subroutines.

3. API's

The specification of the arguments needed to call `CURVE` given above, along with the description of the resulting behaviour and output, are an example of an "API", which stands for "Application Programmer Interface". Since you are writing a program to do something useful (an "application"), you are the application programmer. The API tells you how to call the function(s) in a software package, and what each function call is supposed to do. The API does not tell you how the functions work inside. An API is only an abstract specification. The code that actually does the work is an "implementation" of the API. There may be several different implementations of the same API, and each may do the job a bit differently as long as they produce the same result.

4. Libraries

The subroutine `CURVE` uses a set of further subroutines to actually draw your lines and curves. For example, you will note in `curve.f` that the variable `IC` is simply passed to a subroutine called `COLOR`, which changes the color in which lines are drawn. This and all of the other subroutines used by `CURVE` are part of a complete package of graphics subroutines which are known as VOGL.

It would be awkward to copy all of the VOGL subroutines to your own directory and compile them along with your program, and besides, your program will probably only use a small subset of the VOGL routines. Instead, it is possible to compile all of the VOGL routines once, and put them in a library, and then have the ones which your program needs included in the executable when your program is compiled and loaded. The VOGL library has already been compiled on the class computer, and a copy is available to all users in a standard place. When you want to compile a program which uses routines from such a library, you have simply to tell the compiler (in the `f77` command) which libraries you are using. So to load the program for this assignment, once you have compiled `curve.f` and `mavassar15.f` (with `f77 -c`), you would say

```
f77 mavassar15.o curve.o -lvogl -L/usr/X11R6/lib -lX11 -o ctest
```

Because the files are already compiled, Fortran skips the compilation step and goes straight to the loading step. The `-lvogl` argument tells the loader to look for the library called `vogl` and to get subroutines from this library whenever they cannot be found in the program itself. This library is contained in a file called `libvogl.a` in the directory `/usr/local/lib`. The `-L` argument tells the loader to look for subsequent libraries in the directory it names. The directory `/usr/X11R6/lib` is where all the libraries for X11 are kept on Linux systems. The `-lX11` argument tells the loader to also look in the X11 library (in the file `libX11.a`), which contains graphics routines for the X windowing system. Since these functions are called by the VOGL library functions, it is important to list the `-lvogl` argument before the `-lX11` argument, not the other way around.

5. VOGL and X11

To complete this exercise you will have to log on to noether from some other workstation, such as helios, and run your program on noether but observe the graphics output on helios. One further complication is that VOGL has to be told to use X11 as the output medium. Because VOGL can run on many types of computer systems it does not assume that you are using X11. VOGL determines how to display its final output from the environment variable VDEVICE, which should be set to "X11". You can test this with the `echo` command, '`echo $VDEVICE`. If the variable is not set then you should set it. For C-shells `csh` and `tcsh` the command is

```
% setenv VDEVICE X11
```

while for Bourne shells `sh` and `bash` the command is

```
% export VDEVICE=X11
```

(If you are working on noether.vassar.edu you are probably using the tcsh shell).

Two very useful X11 applications you may like to learn more about are `xfig` and `xmgr` (known now as "grace" or `xmgrace`). The `xfig` program is a simple drawing program which can be used to produce Encapsulated PostScript drawings suitable for inclusion in a physics paper. The drawing of the airplane in Exercise 05 was done using `xfig`. The `xmgr` program is a useful graphing program which will read in data such as the pairs of numbers you generated in the last exercise and plot them with axis scaling, error bars, and other useful features. You may want to use `xmgr` to view your data file while you debug your program.

6. Input/Output

Your program should simply ask for the name of the data file which is to be read (this makes it more general), and then it should read in the x,y data from that file, one pair of numbers per line. Your program should use the `CURVE` routine both to plot the curve and to then wait for the user to press the ENTER key before deleting the graphics window.

7. Optional improvement

One optional improvement to your program which will make it a more generally useful tool is to have it draw multiple lines from the same file. Your program would stop drawing a line whenever a blank line is encountered, change the pen color, and then start on a new line.

This is trickier than it sounds. You first need to read in an entire line as a character string to see if it is blank. If it is not blank then it has to be re-interpreted to read the two data items, (x, y coordinates). One way to do this is with an "internal" READ statement,

where you read the data from a character string rather than from a file. To find out more about this technique look up "internal files" in your textbook.