

Exploiting Graphs

George H. Wells, Jr.

July 12, 2001

Chapter 12 Title: Exploiting Graphs

12.1 Introduction

This chapter describes some things that can be done with graphs that go way beyond the mundane plotting of waveforms. In particular, we are interested in using graphs to draw complex animated images but we will also go into other topics, including the mundane plotting of waveforms. First we will cover the basic skills and then we will look at some examples from projects where these skills are used at NASA's Jet Propulsion Laboratory.

12.2 Basic Graph Skills

In this section, we will cover the basics of graphs and then the skills we need to exploit them.

12.2.1 Graph Basics

Some of the basics we need to exploit graphs are already covered in the LabVIEW documentation but we will review them again here and then go on to more topics.

12.2.1.1 Graph Data Types

XY Graphs accept plot data in one of two data types: an array of clusters containing X and Y values (Figure 1), and a cluster containing an array of X values and an array of Y values (Figure 2). Note that the terminal for the XY Graph changes to reflect the data type.

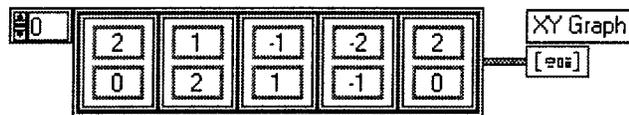


Figure 1: XY Graph data type 1 - An array of clusters containing X and Y values.

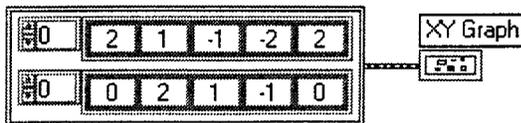


Figure 2: XY Graph data type 2 - A cluster containing an X array and a Y array.

These diagrams draw a four-sided irregular polygon. Note that the first point, (2,0), is replicated as the last point in order to close the outline. Figure 3 shows the graph that these diagrams produce.

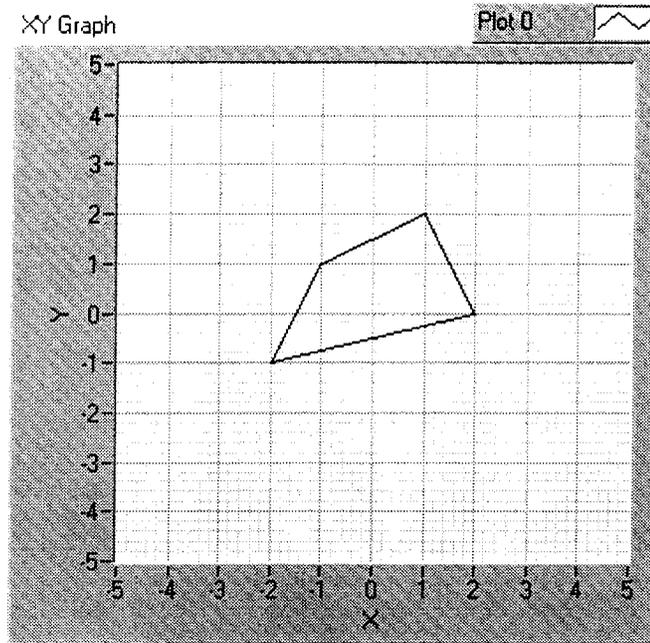


Figure 3: Graph produced by Figures 1 and 2 (and 10).

Although both of these data types are equivalent in terms of defining an outline, they have the disadvantage of taking up a lot of real estate on the diagram, and they can be a nightmare when it comes to performing the simple operation of translation (offsetting an object to a different location on the graph) or the much more cumbersome operation of rotation. (The operation of scaling, or sizing, is easy for any data type, since it involves only multiplying.)

12.2.1.2 Translation

To illustrate, Figures 4 and 5 show what it takes to translate or offset the object in Figure 3 by two units to the right and three units down. Figure 6 shows the resulting graph. Note that for type 1, you have to manually construct the right kind of cluster constant. You cannot just right-click on the unused input of the Add operator and select Create Constant because it will create an array of the same type that is present on the other input. However, you can CTRL-drag one of the cluster elements from within the array and place it near the unused Add input.

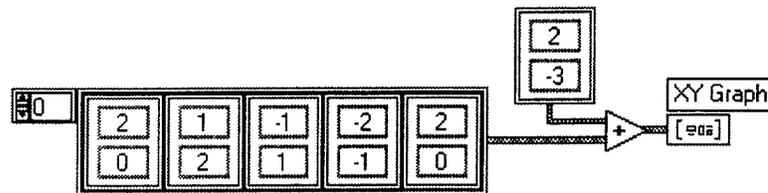


Figure 4: Translation for data type 1.

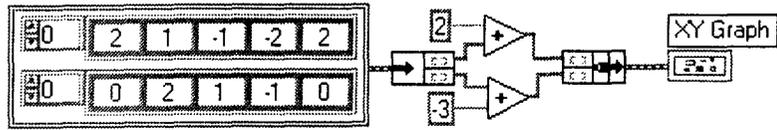


Figure 5: Translation for data type 2.

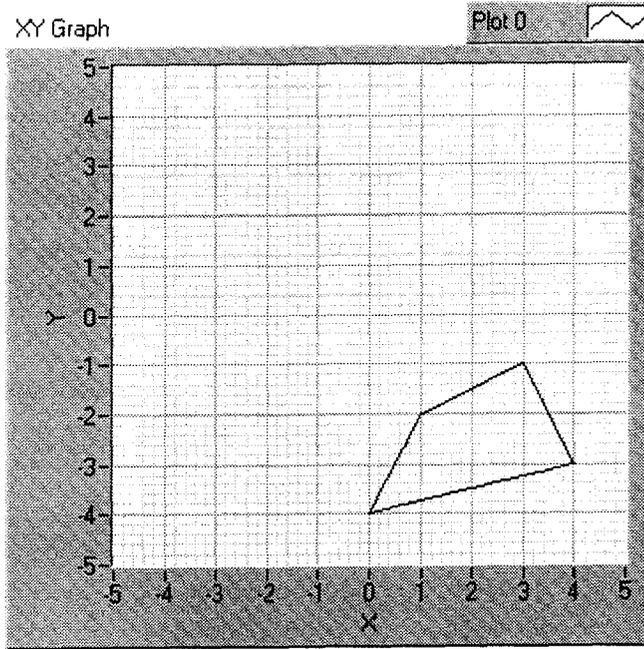


Figure 6: Graph illustrating translation produced by Figures 4 and 5 (and 11).

12.2.1.3 Rotation

Figures 7 and 8 illustrate what it takes to rotate an image in a counter-clockwise direction about the origin by 90 degrees (π divided by two) and Figure 9 shows the resulting graph.

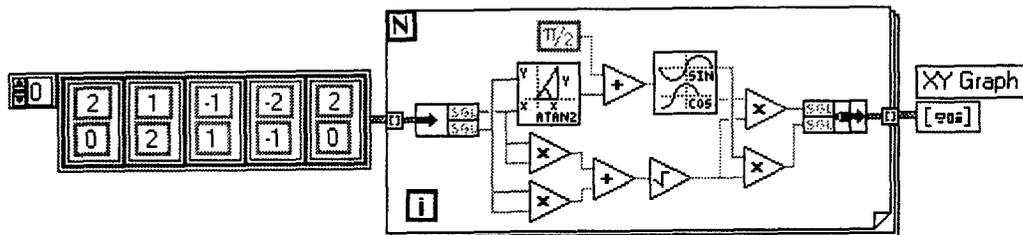


Figure 7: Rotation for data type 1.

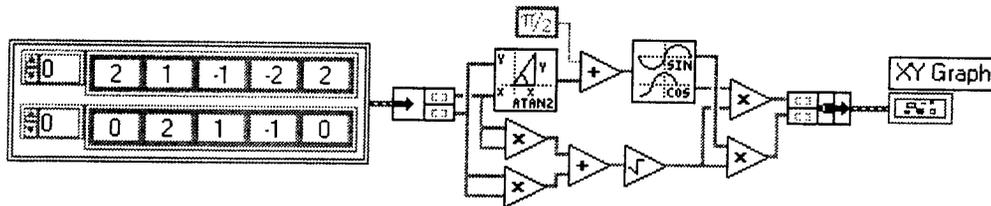


Figure 8: Rotation for data type 2.

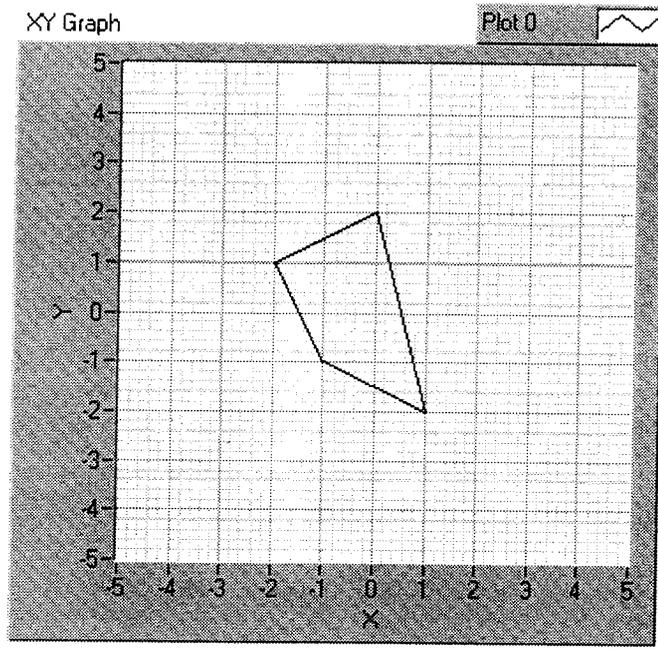


Figure 9: Graph illustrating rotation produced by Figure 7 and 8 (and 12).

Clearly the second data type is a little easier on which to perform rotation but it has the disadvantage of making data entry more difficult for large numbers of points because you have to be careful to keep the two separate arrays aligned.

12.2.2 Complex Numbers to the Rescue

The solution to all these problems is to use complex numbers. Complex numbers are a convenient way both to represent two-dimensional outlines or shapes and to perform all the operations of translation, rotation and sizing. They also make it easy to animate complicated objects.

Complex numbers have a real and an imaginary part. When you view a complex number in LabVIEW you will see the real part on the left and the imaginary part on the right followed by the letter "i". The imaginary part always displays a sign that also serves to separate the two parts. To get an imaginary number on the diagram, simply put down a numeric constant and change its representation to CSG. There is

no point in using a higher precision for numbers that will eventually be displayed on a graph.

There are several ways to enter values into complex number constants. You can always triple-click on the number, highlighting the entire display, and type in the real part followed by the imaginary part (with its sign) and the letter "i". Don't leave off the "i" or the values will get corrupted. You can also double-click on just the real or imaginary part and enter them separately. This has the advantage of not requiring you to enter the letter "i". There is a special trick that works for complex numbers with a precision of zero (as our examples have). If you double-click to the left of the real part when it is only one digit, both the real and imaginary parts get highlighted without the "i". Now you can type in the real and imaginary parts (separated by a sign) and hit return without typing an "i". Is this a feature or a bug? Well, it seems to work only with version 6i on a PC, so it probably is a bug, but take advantage of it while you can.

When used to represent points on a graph, the real part of a complex number is the horizontal or X component and the imaginary part is the vertical or Y component. Before a complex array can be drawn to a graph it must be converted to one of the two types acceptable to an XY graph. The obvious choice is the one shown in Figure 2, a cluster containing two arrays, the horizontal (real) array and the vertical (imaginary) array. Figure 10 illustrates how to make the conversion. Note that using a complex array always keeps the two components together (unlike the arrays in Figure 2). The plot drawn by this figure is identical to Figure 3.

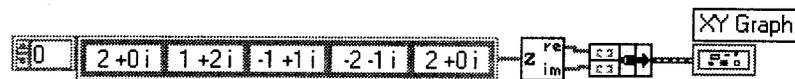


Figure 10: Complex number equivalent to Figures 1 and 2 (creating the outline in Figure 3).

Of course it could be argued that having to convert the complex array to the correct data type for a graph is too much of a burden to make it worthwhile but the real advantage comes when you want to perform operations. Compare Figure 11, the complex number solution to performing translation with Figure 4 or 5.

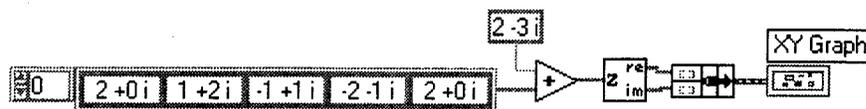


Figure 11: Complex number equivalent to Figures 4 and 5 (performing the offset in Figure 6).

If that won't persuade you, compare Figure 12, the complex number solution to performing rotation with Figure 7 or 8.

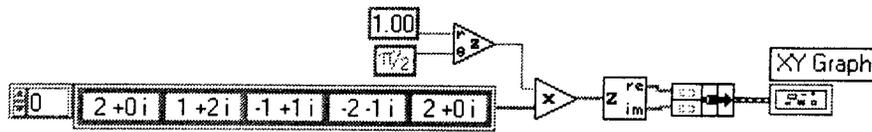


Figure 12: Complex number equivalent to Figures 7 and 8 (performing the rotation in Figure 9).

The key to understanding how rotation works is to realize that complex numbers can be represented as either having a real and an imaginary part (the only way LabVIEW displays them) or as a magnitude and an angle. Multiplying two complex numbers is performed by multiplying the magnitudes of the two numbers and adding their angles. In this example, we use a magnitude of 1.00 for the multiplier and any desired phase (in radians). LabVIEW takes care of all the arithmetic and trigonometric operations identical to what is shown in Figures 7 and 8. As an added benefit, we can set the constant to something other than 1.00 to resize the object for free.

By the way, if we ever wanted to resize an object that was represented with an array of complex numbers, we would multiply the array by a complex constant with the real part set to the size factor and the imaginary part set to zero. Actually, we could use a simple numeric constant rather than a complex one and let LabVIEW coerce it to complex inside the multiply function.

Since we often will want to enter angles in degrees rather than radians, we will present a simple way to convert from degrees to radians. The straightforward way is to divide the value in degrees by 180 and multiply by pi. A simple way that takes up less real estate is to use a constant (or a control) that has a “deg” unit label applied to it and then use a Convert Unit set to “rad” as shown in Figure 13.



Figure 13: A straightforward and a simple way to convert degrees to radians.

This scheme actually removes the unit from the constant but converts it to radians before doing so. If you don’t remember how to code units, first make sure the constant is an SGL type, right-click on the constant, select “Visible Items” and then “Unit Label” and enter “deg” into the box that appears to the right of the numeric constant. Next, get the “Convert Unit” icon from the Numeric/Conversion palette and enter “rad” into it.

12.2.3 Making Filled Shapes

Now we will look into filling our outline with a solid color. To make a filled shape, we need two plots, one of which will be filled to the other one. An easy way to create

this second plot is to simply reverse the order of the complex array defining the first plot. We need to then perform the usual conversion to the plot data type and then build the two plots into an array. Figure 14 shows the completed diagram.

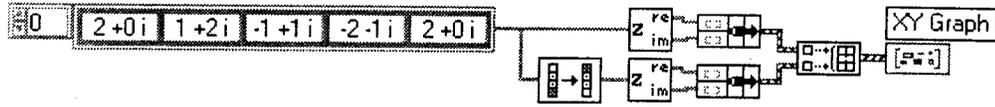


Figure 14: A simple way to fill a plot outline.

On the front panel, we need to drag down the Plot Legend to include two plots, click on Plot 1, go to Fill Base Line, and select Plot 0. You can then select a color for Plot 0 which will become the outline color and a different color for Plot 1 which will become the fill color. Figure 15 shows the resultant graph.

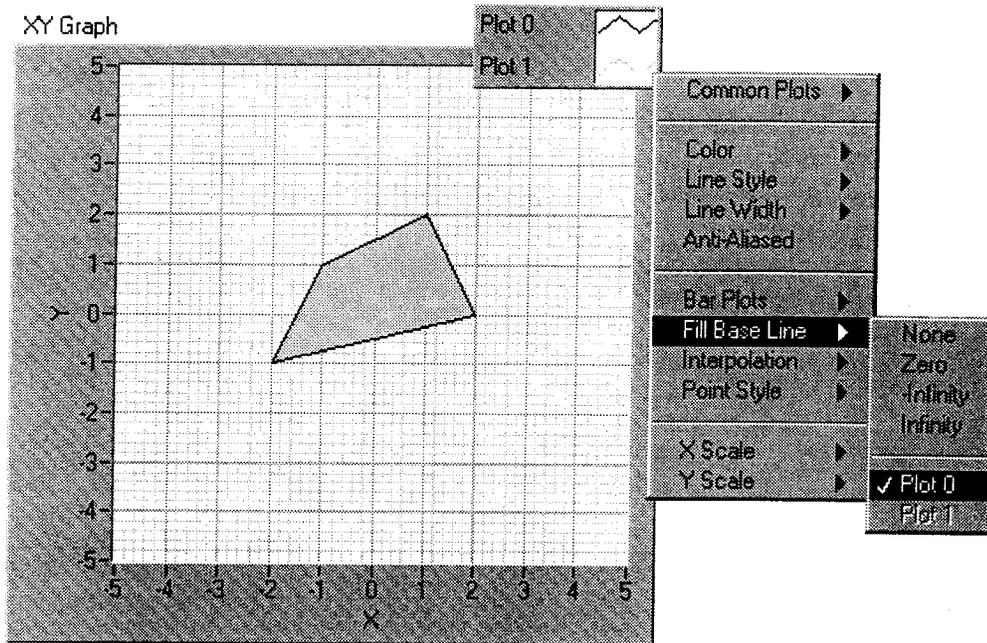


Figure 15: Filling Plot 1 to Plot 0 (see diagram in Figure 14).

Simply reversing the order of the original array to create the second array may not be the best scheme. A better approach is to break the initial array in half, replicate the last element of the first half as the first element of the second half, and then reverse the order of the second half. Not only does this procedure work better on different platforms and with different printers, it results in plots that are half as large. Figure 16 shows a subVI to perform this function and Figure 17 shows its use. This subVI is available in versions 5.0 and 6i of LabVIEW on the CD included with this book.

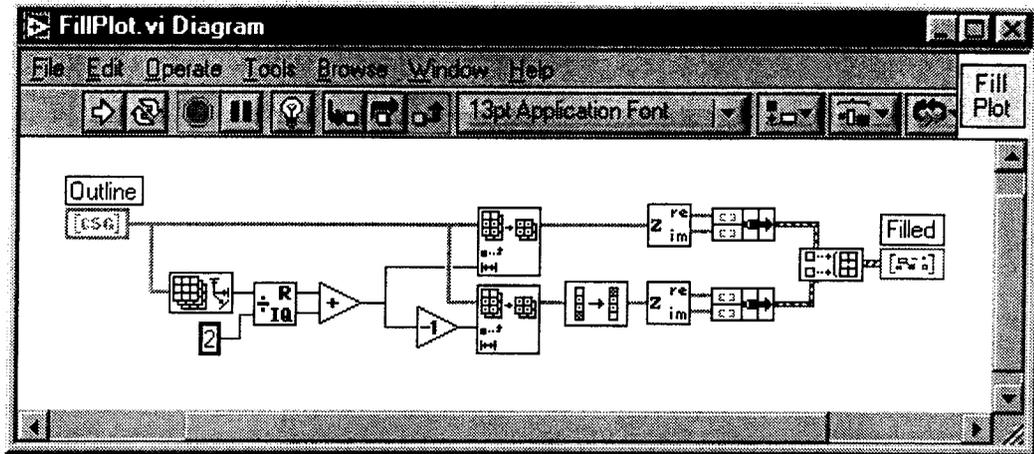


Figure 16: A subVI to more efficiently fill a plot outline.

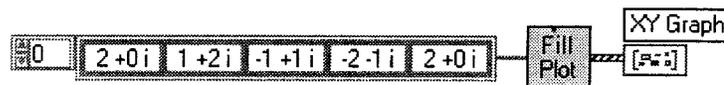


Figure 17: Use of the Fill Plot subVI (produces graph in Figure 15).

There is one additional step to make the graph work with an outline color different than the fill color when this subVI is used. You have to set two colors on the second plot by hitting the spacebar to select between them. The one on the right is the outline color and should be the same as the first plot and the one on the left is the fill color.

Instead of manually setting the plot colors and fill parameters, you can use property nodes to do the same thing. This is especially useful if you have a very large number of plots.

Be warned that the filling of convoluted outlines does not always work for all platforms and for all printers. It seems to work best under Windows with LabVIEW 6i, but you should always test results for your application. If you find a shape that does not work with the simple scheme of reversing the array or by using the Fill Plot VI, then you will have to break the image up into smaller overlapping pieces that mask the problem. We will not address this issue any more.

12.2.4 Animation

Now that we have covered the basics it's time to look at animation. We will use as a basis for learning about animation the robotic arm example, which you can find at [LabVIEW/examples/picture/robot.llb/robot.vi](#). (This example is not available in the Base LabVIEW package.) See Figure 18.

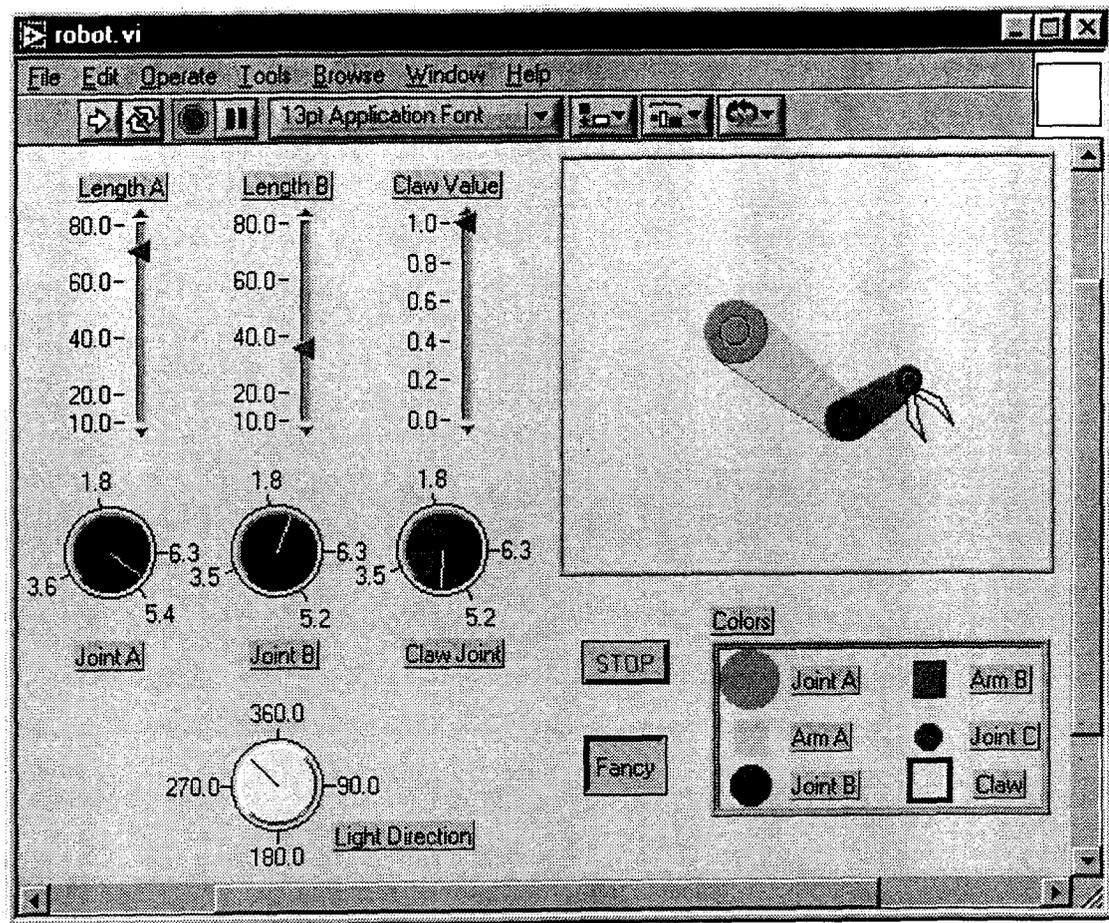


Figure 18: Front panel of robot example VI.

12.2.4.1 Robot Arm Example

The robot arm example has several parameters that can be adjusted by the user including three joint rotations, two arm lengths, and the claw opening. We will not use the Light Direction control but will instead use the Fancy style to add a black outline to all the objects just as shown in Figure 18 for the claw.

When developing the plots for a graph, we will always start with the most stationary object, in this case, joint A, and work our way to the most extreme object, in this case, the claw. However, the order that the plots are applied to the graph needs to be based on the priority of display. The Claw is always foremost on the graph, followed by its Joint, then Joint B, Arm B, Joint A and finally Arm A. These various objects will be created with their own subVI's. To make it more convenient to arrange these subVI's on the diagram, we will put the subVI for the most stationary object, Joint A at the bottom and work our way to the top.

Another important function of these subVI's is to provide offsets and rotations from one object to the next connected object. If an object can be rotated and causes the subsequent objects to rotate with it, then the subVI will add its rotation to subsequent

rotations. In a similar way, if an object provides an offset to subsequent objects, then the subVI will add its offset to subsequent offsets. We will place a dummy point in the shape-defining array so that we can calculate its new offset and then split the array and extract this dummy point from the points that are drawn on the plot so that we can add it to subsequent offsets.

12.2.4.2 Plot Circle

The diagram of our first subVI called Plot Circle is shown in Figure 19. It, as well as the remainder of the subVI's for this example, are in PlotRobot.llb for versions 5.0 and 6i of LabVIEW on the CD.

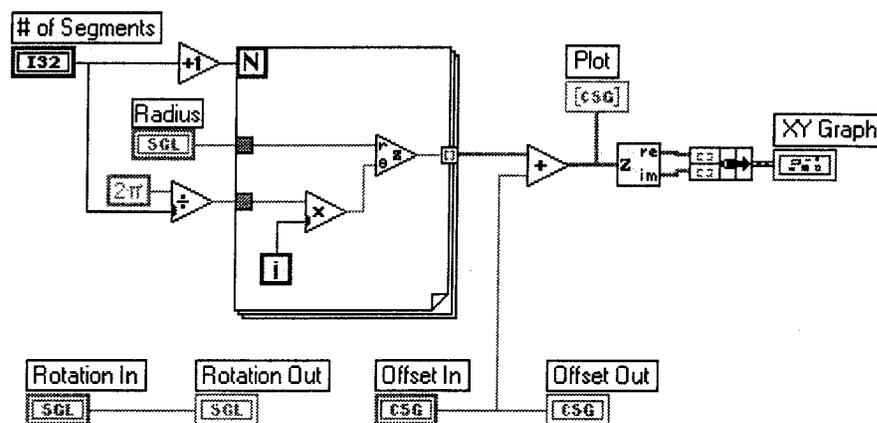


Figure 19: Diagram of Plot Circle VI.

We will use this subVI to draw the joints. Since plots cannot draw circles we will approximate them as regular polygons with a large number of sides or segments. There are two parameters passed to this subVI to define the circle, the Radius and the Number of Segments. We first divide the Number of Segments into 2π , a full circle, and use this value as an increment for the angle applied to a Polar To Complex converter. The Radius is also applied to this converter. The array coming out of the For Loop contains the complex numbers defining the polygon approximating our circle. Note that this array contains one more point than the number of segments in order to fulfill the requirement of the first point being replicated as the last. The array of complex values has its origin at zero but if the circle has an Offset In then it is added to the array before generating the Plot. Both the Rotation and Offset Inputs are brought out unchanged since the circle in this application does not affect the offset of objects connected to it. For convenience in testing and demonstrating the subVI, we also convert the complex array into the type required for an XY Graph and put the graph on the front panel. The front panel of the Plot Circle subVI is shown in Figure 20 and the connector pane is shown in Figure 21.

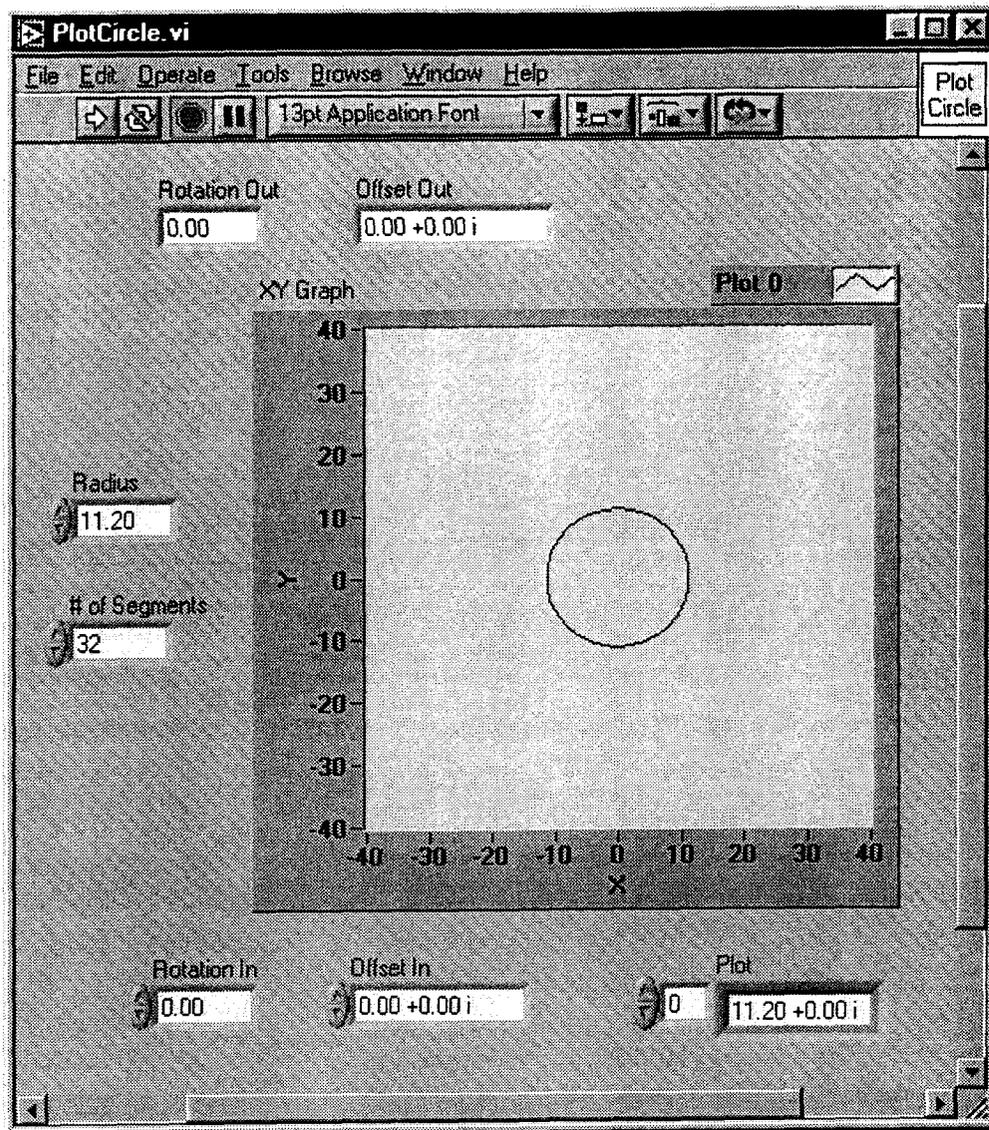


Figure 20: Front panel of Plot Circle VI.

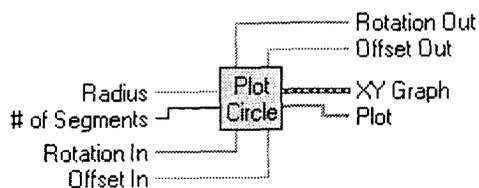


Figure 21: Connector pane of Plot Circle VI.

12.2.4.3 Plot Span

The next subVI, Plot Span is used to create the arms. It defines a trapezoid, which is a four-sided polygon similar to a rectangle, but with tapering sides. The input parameters allow us to specify the length and the two end widths which are the same

as the diameters of the two circles superimposed as joints on the ends of the arms but for convenience we will use the radius values of these two circles. The Joint input allows us to specify the angle of the arm.

Figure 22 shows the diagram of Plot Span. The first thing it does is build an array consisting of the points defining the trapezoid, plus (as the first element) one additional dummy point as mentioned earlier to provide the offset that is applied to the next objects attached to the arm. When we are defining these points, we assume that the arm is horizontal, extending from the origin to the right. This first element, the dummy point, is located at the center of the far end of the arm. Since the arm is horizontal, the coordinates of this point are merely the Length and zero. We apply these values to a Re/Im To Complex converter to get the first element of the array.

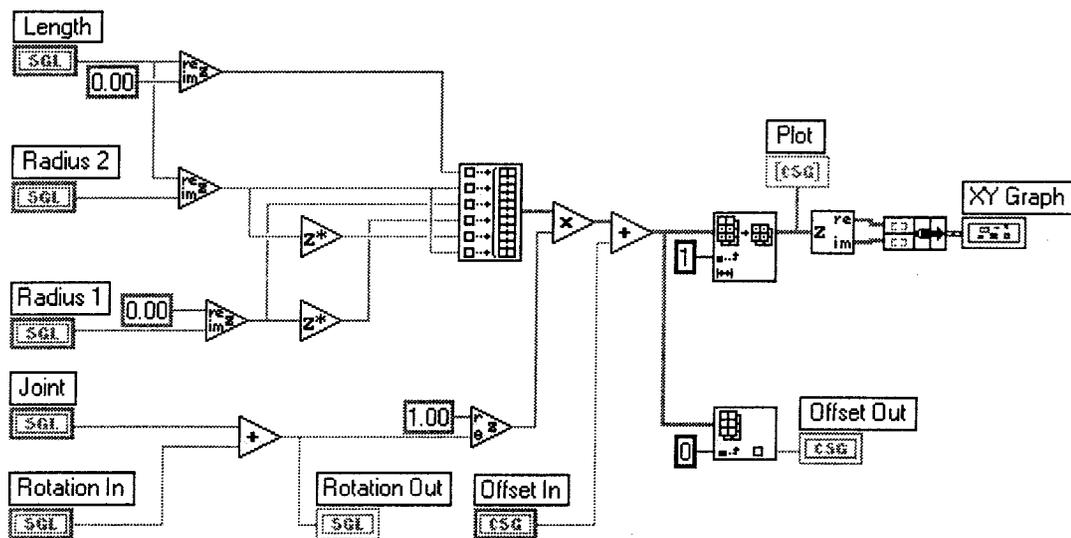


Figure 22: Diagram of Plot Span VI.

The next element is the first element of the polygon, which is located at the upper right corner of the horizontal arm. Its coordinates are the Length of the arm and the second Radius. Again, these two coordinates are converted to a complex number and applied to the Build Array function. This point is also applied to the last element of the array to fulfill the requirement to replicate the first point of the polygon as the last. The next point is in the upper left corner of the horizontal arm. The coordinates are zero and the first Radius.

The next two points are a repeat of the first two points of the polygon (in opposite order) except that the signs of the vertical components are changed. We could have changed the signs of Radius 1 and 2 and converted these to complex numbers but the Complex Conjugate function allows us to do it more conveniently by merely taking the complex conjugates of the previously defined points. (The symbol in the icon for the Complex Conjugate looks like x^* .)

Now that we have the array built, we need to apply a rotation. The rotation is a combination of any previous rotation plus the rotation defined by the Joint control. This sum is also brought out to apply to the more extreme objects. To apply it to this object, we connect it to the theta input of a Polar to Complex converter with the "r" input set to 1.00. This complex number is multiplied by the array. Remember that multiplying complex numbers together is the same as multiplying their magnitudes and adding their phases so this one simple operation takes the place of all the components in Figure 7 or 8.

After rotating we add the offset. Note that unlike the Rotation In and Out, which are simple numbers, the Offset In and Out are complex numbers. As mentioned earlier, the first element of the array is extracted and applied as the Offset Out. The remaining part of the array produces the Plot and again we convert it to the type required for an XY Graph for diagnostic purposes. Figure 23 shows the front panel of Plot Span and Figure 24 shows its connector pane.

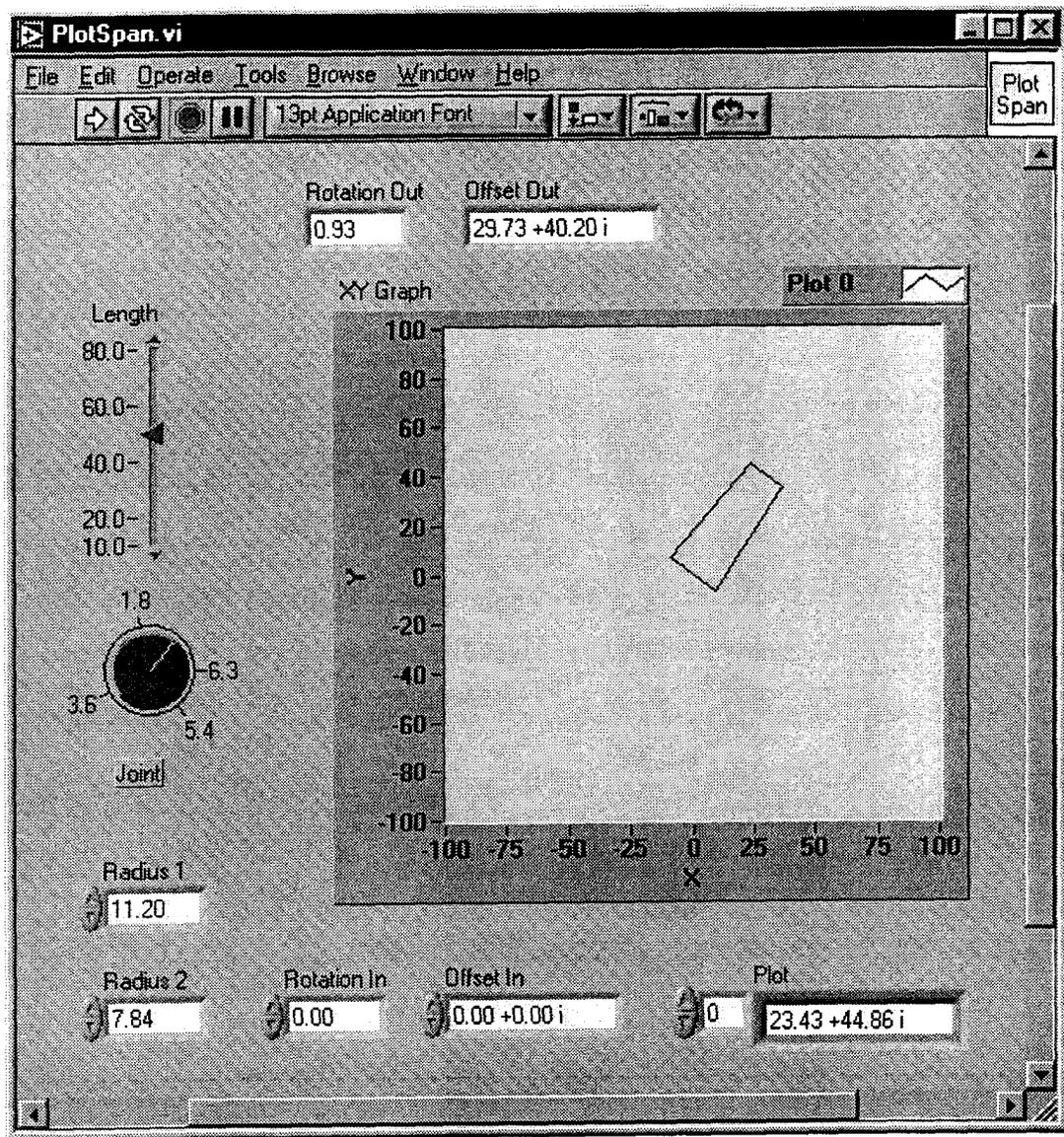


Figure 23: Front panel of Plot Span VI.

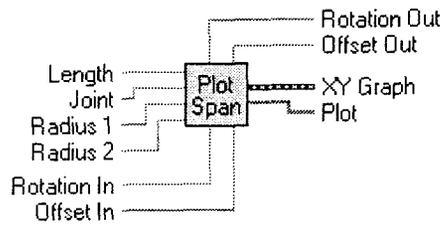


Figure 24: Connector pane of Plot Span VI.

12.2.4.4 Plot Claw

The final subVI is Plot Claw. Its diagram is shown in Figure 25. This subVI defines most of the points of the horizontal claw as fixed coordinates but the two end points are calculated as radius points from the center of the joint on which the claw is mounted. The normalized claw value is multiplied by an empirically derived factor (0.26) that converts it into radians and is applied to the theta input of a Polar To Complex converter. The “r” input is the distance from the end points to the center of the joint. This produces the coordinates of the end point of the horizontal upper claw. Again, we use the Complex Conjugate function to generate the coordinates of the lower claw from the upper claw.

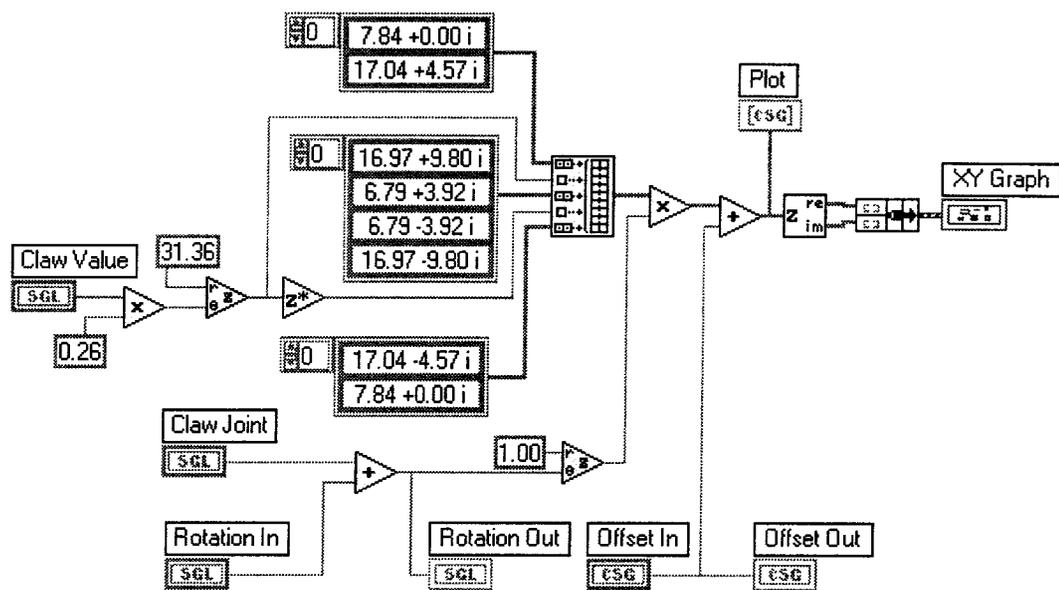


Figure 25: Diagram of Plot Claw VI.

Next we apply the rotation and offset factors just as we did for the span except that since there are no further objects attached to this one, we don't bother to calculate a new offset we just pass the input straight through.

The front panel of Plot Claw is shown in Figure 26 and the connector pane is shown in Figure 27.

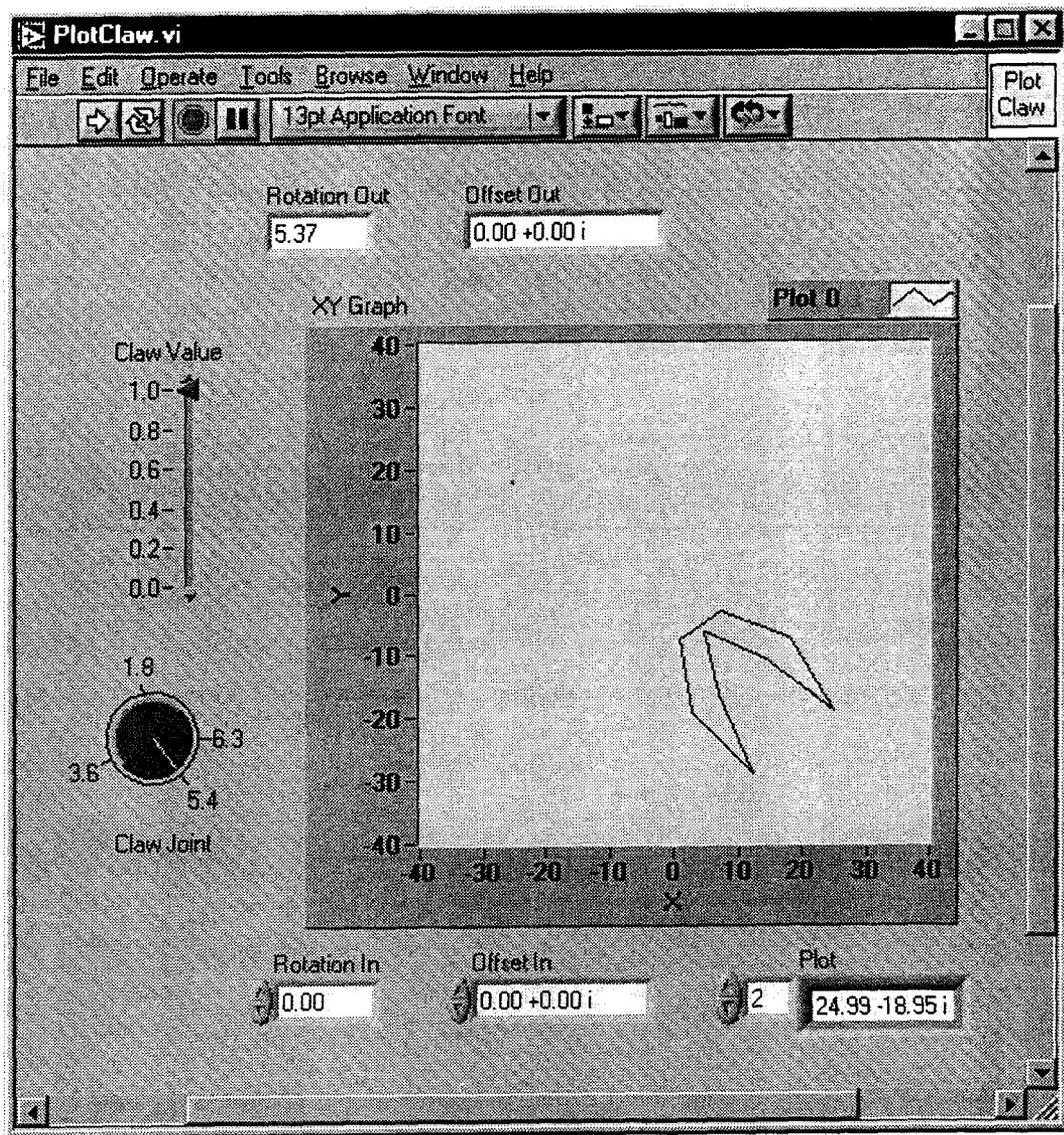


Figure 26: Front panel of Plot Claw VI.

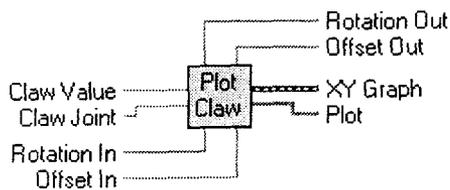


Figure 27: Connector pane of Plot Claw VI.

12.2.4.5 Plot Robot

Finally we come to the top level VI called Plot Robot that uses all these subVI's. Its diagram is shown in Figure 28. There are two functions inside the loop. The uppermost part calculates the plots of the different objects making up the robotic arm and the lower part applies the colors to the property nodes for the plots whenever one of them or the style changes. There are a total of six pairs of plots. Remember that each pair draws one object by filling from the second plot of the pair to the first one. The Fill Plot subVI expands the outline of each object into a pair of plots so that one can be filled to the other.

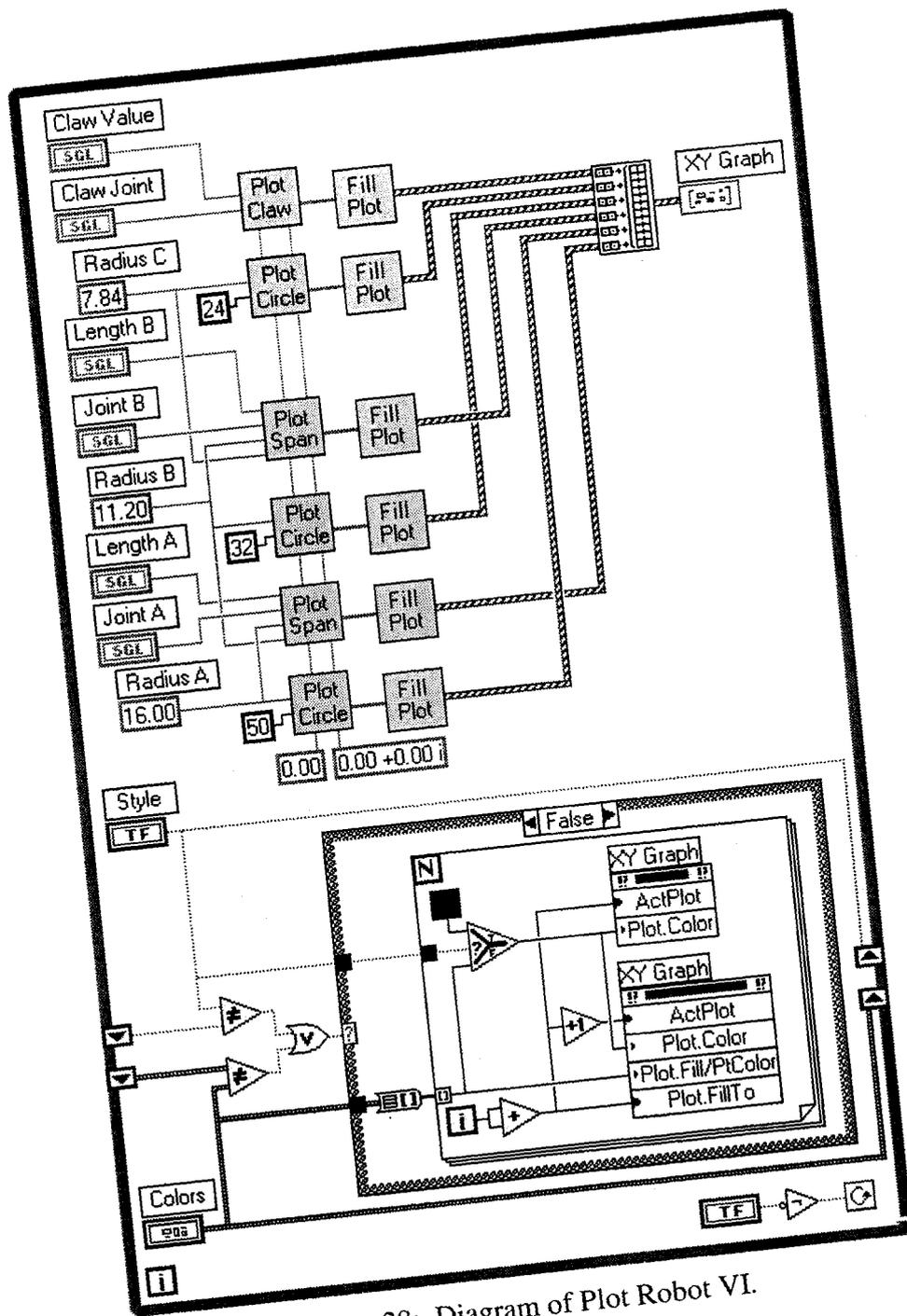


Figure 28: Diagram of Plot Robot VI.

The first object is Joint A, which is a circle generated by Plot Circle located at the origin with a radius of 16.00 and is made up of 50 segments. After that is Span A, which is an arm tapering from Radius A to Radius B and with Length A. Joint A is the user-controlled rotation of Span A. The Rotation Out and Offset Out from the Plot Circle subVI immediately below are applied to the Plot Span subVI and similarly all the way up to the top subVI, Plot Claw.

Arm B is made up of a circle of radius 11.20 with 32 segments and a span similar to Arm A but with a different set of parameters.

Finally, at the top we have the claw, which is made up of a circle of radius 7.84 with 24 segments and the Plot Claw subVI.

The front panel of Plot Robot is shown in Figure 29. It looks very much like the Robot example VI (Figure 18) with the exception of the Light Direction control and it behaves in exactly the same way.

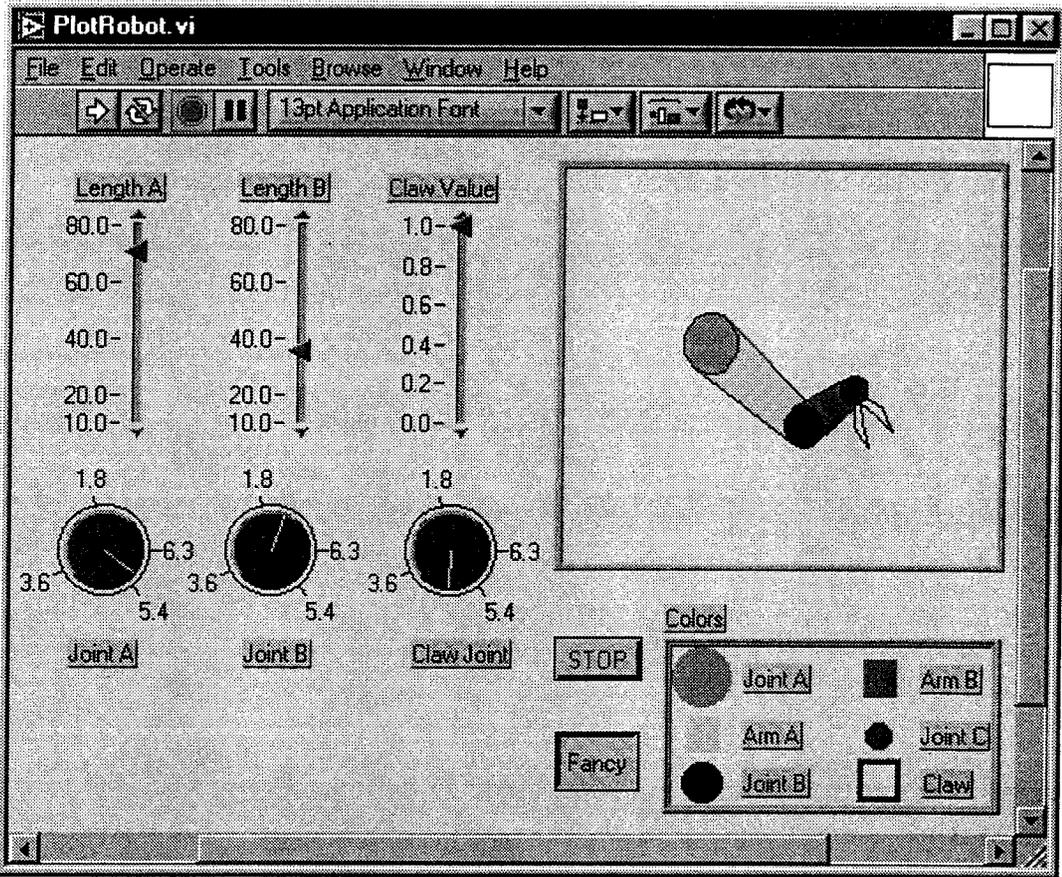


Figure 29: Front panel of Plot Robot VI.

12.2.5 Segmented Plots

The next section deals with a subject that often comes up when making graphs. It conveniently solves the problem of drawing a large number of disconnected lines by using a single plot instead of having a separate plot for each one. The trick here is to use NaN to separate the objects. When the graph drawing routine comes to an X-Y pair of NaN's it stops drawing a continuous line, in effect lifting the pen, and beginning again with the next element in the arrays. Figure 30 illustrates an example diagram to draw a bunch of random irregular objects scattered on the graph. Note that when the array is built to add another object to the final array, a NaN complex value

is added after each one. There is only one plot on the graph shown in Figure 31, even though it looks like there are twenty. This example shows closed polygons but the technique can also be used to show arrowheads or even letters of the alphabet. This subVI is available on the CD.

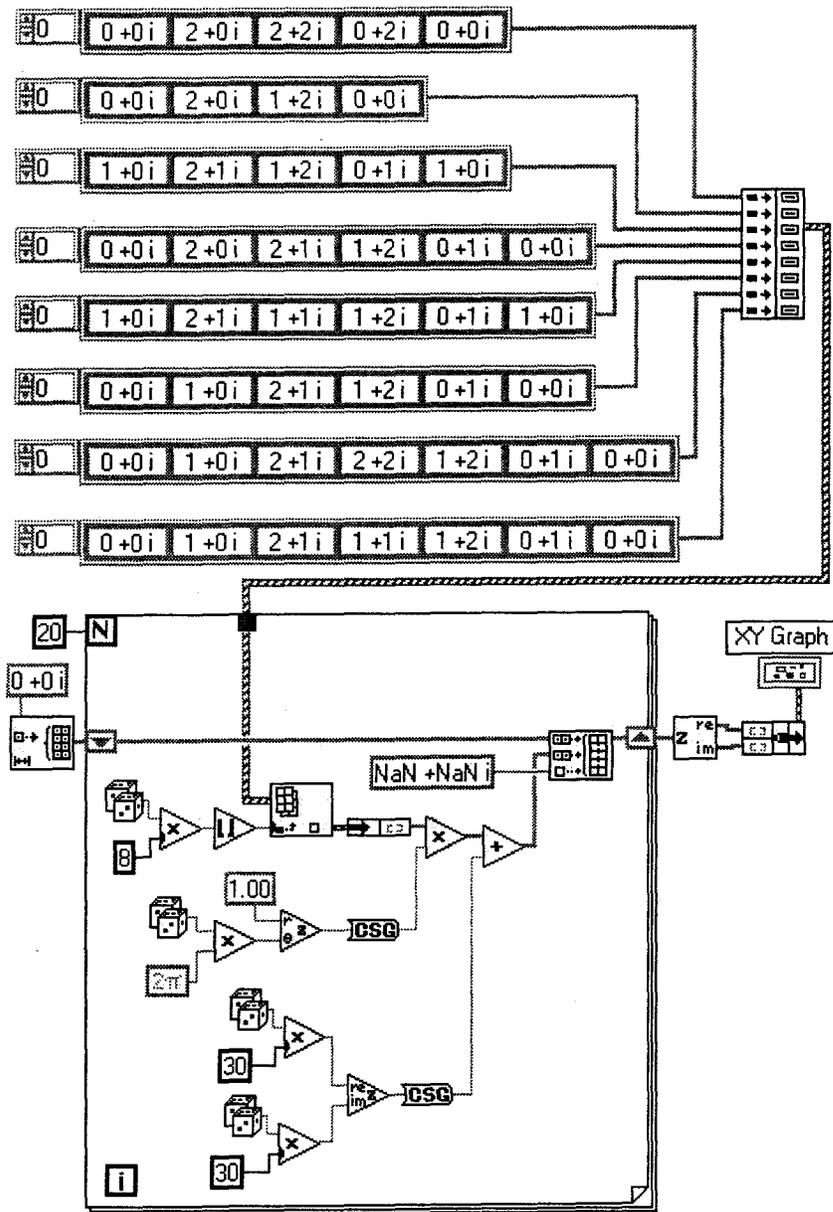


Figure 30: Diagram of Plot Multi Outlines VI.

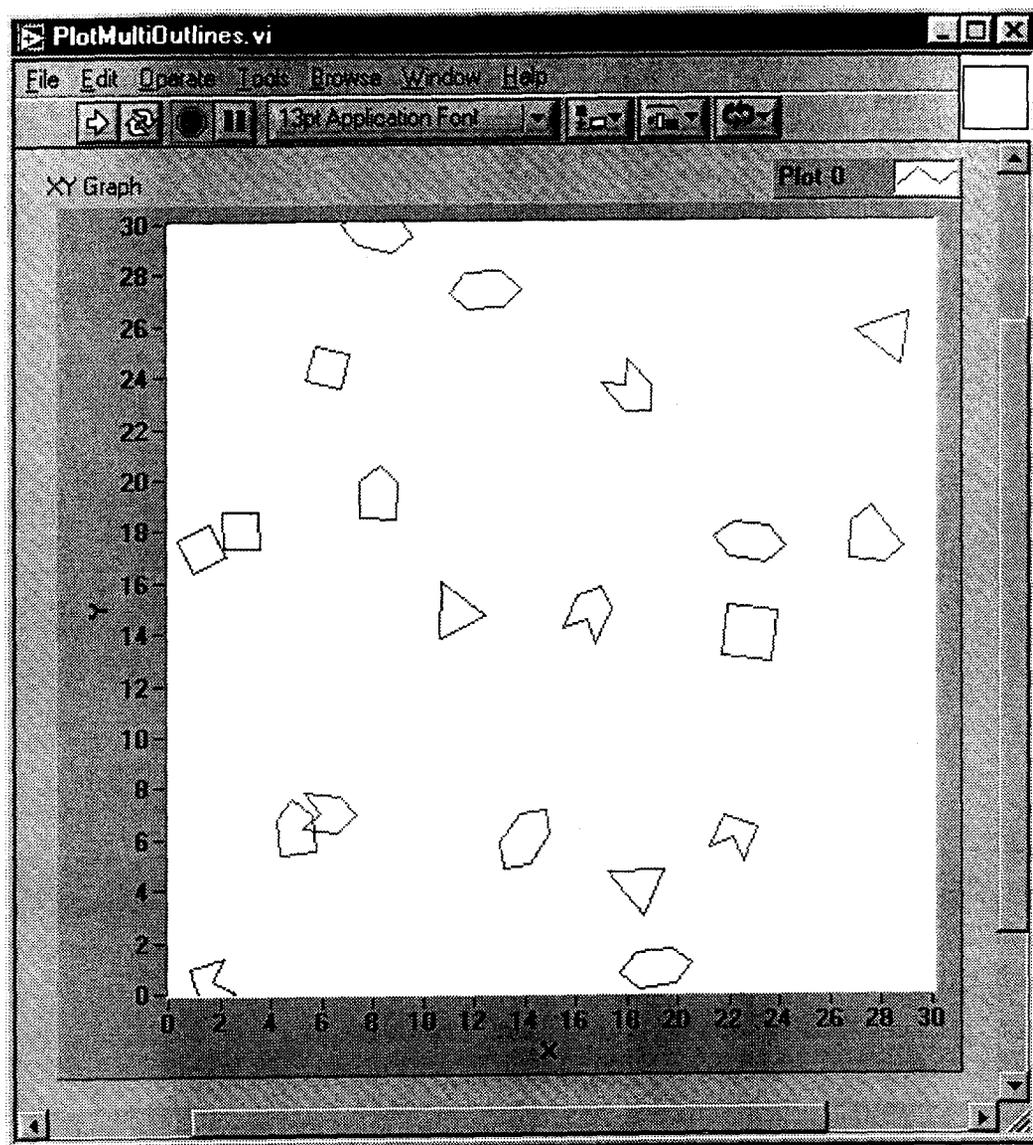


Figure 31: Front panel of Plot Multi Outlines VI.

12.2.6 Simulating an Intensity Graph Using an XY Graph

Now we come to a way to simulate approximately what an intensity graph does but on an XY Graph. This technique provides a way to arbitrarily color various areas of a graph. This example shows intensity as a function of the distance from the origin.

The diagram in Figure 32 contains four For Loops. The one at the top is an initialization sequence, setting up the properties of the graph so that each plot has a point style with the largest solid square point available. There are eight plots, each with a different shade varying from light to dark. The For Loop also creates empty arrays of complex numbers bundled one for each plot that will eventually contain coordinates for each color. In a real application, this For Loop would only be executed once.

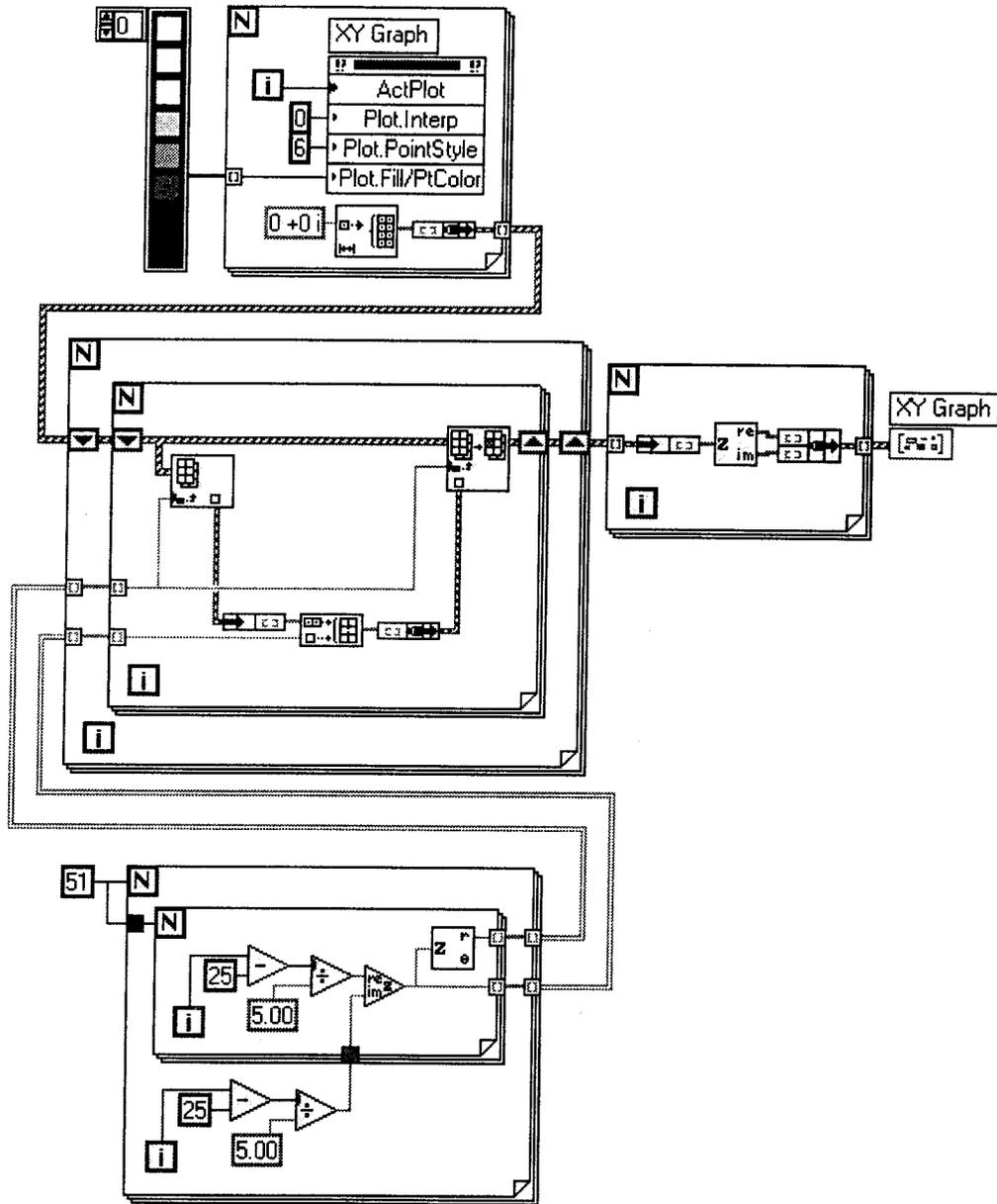


Figure 32: Diagram of Intensity VI.

At the bottom of the diagram is a pair of nested For Loops, which generate the data that are to be displayed. The data consist of two arrays, the lower one for the coordinates of a point, and upper one for the intensity of the point. In this example, there is one point for every position on the graph, but in a real application, the pair of arrays could be generated by any means and doesn't even have to cover the entire graph.

The working part of the diagram is the For Loops in the middle. The nested For Loops on the left sort through all the elements of the arrays coming in from the bottom and adds each point to one of the eight plots depending on its intensity value. Note that it is the coordinates of the point that are added to one of the plot arrays. After the nested loops finish running, another loop is used to convert the complex number type to the type required by the plot. Figure 33 shows the resulting graph. This subVI is available on the CD.

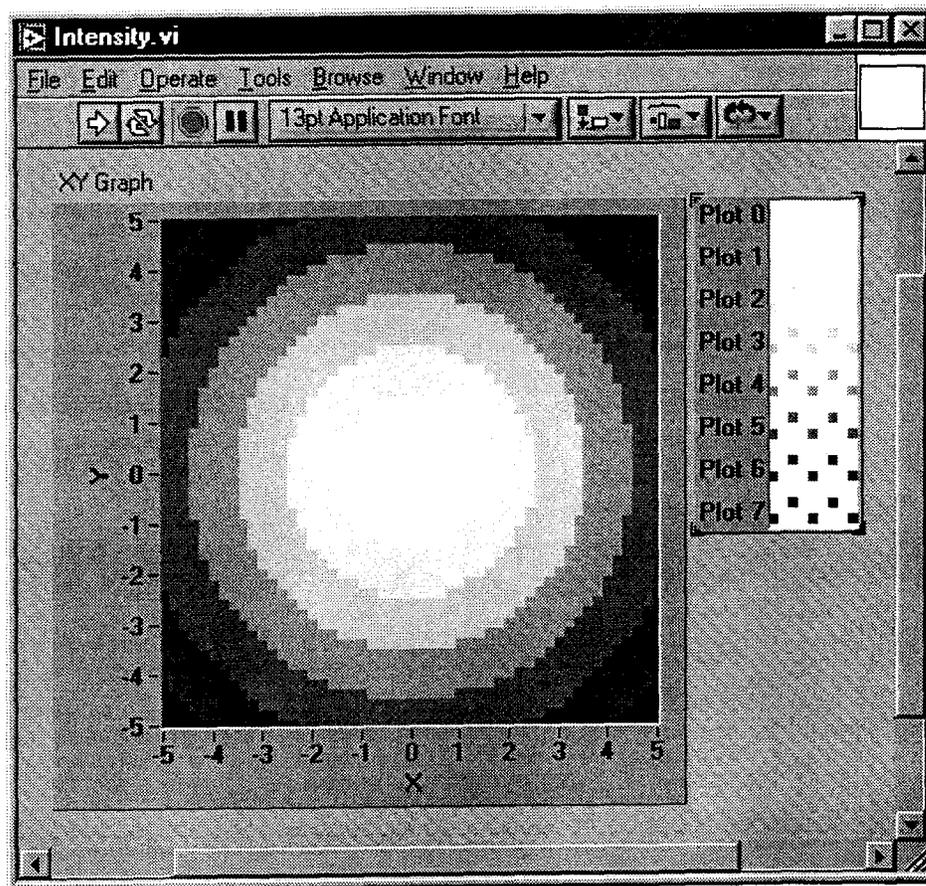


Figure 33: Front panel of Intensity VI.

The graph has to be manually resized to eliminate any gaps between the points. In a real application, the graph could actually be resized even smaller in which case the points will appear smaller because they overlap with priority given to the lower numbered plots.

12.2.7 Simulating a Waveform Chart Using an XY Graph

Sometimes you would like to take advantage of the convenience of a Waveform Chart because it automatically keeps a history of old points and keeps track of the X coordinate. However, if you need to display real time along the X axis and especially if your data points are not evenly spaced in time or if you have multiple plots which

may not all update at the same time, you must use an XY Graph instead. In this section, we will cover a few tricks that allow you to overcome some problems that might otherwise make this an unpleasant task.

The first problem has to do with handling the arrays that are updated each time new data points are added to the graph. If you do the obvious thing and start with empty arrays, you will create for yourself a quagmire of trying to keep track of the pointers that are used to delete the oldest array elements. Keep in mind that the arrays start with the oldest data and end with the newest. In general, we will want to append new data points on the end of the arrays and delete the first elements but this will prevent us from getting started because we will end up deleting the points just added unless we take special precautions.

The trick is to initialize the arrays with a fixed number of NaN's and always keep the size of the arrays constant. Remember from our previous discussion that points with NaN in them will not display at all. Now it is a simple matter of appending new points to the end and deleting the first points. The only thing we have to watch out for is that we provide enough elements in the arrays to handle the maximum range that we will want to display. If the points are added at a fixed rate, this will be easy to calculate. If they can be added at a variable rate, we will take the maximum rate and multiply this by the range of time the graph will cover (plus a few more to be on the safe side). An easy way to determine when it is necessary to perform the initialization is to check the size of any one of the arrays and initialize all of them if it is zero.

There are two ways to store the arrays. Back in the early days of LabVIEW, we had just the first way, which was to use shift registers on loops that executed once per update. Now we have a second way, which is to use a local variable reference to the graph itself. We use the second method in this example.

We can also use either of the two graph data types to process the arrays (see Figures 1 and 2) but unlike our previous skills involving complex numbers which used graph data type 2, waveforms are better suited to graph data type 1.

The second problem that we have to deal with when using an XY Graph to simulate a Waveform Chart is ranging. There are several problems with allowing the graph to autoscale, not the least of which is that it will leave blank space on both edges of the graph (see Figure 34). Instead we want to give the illusion that the waveform is scrolling off the left edge of the graph.

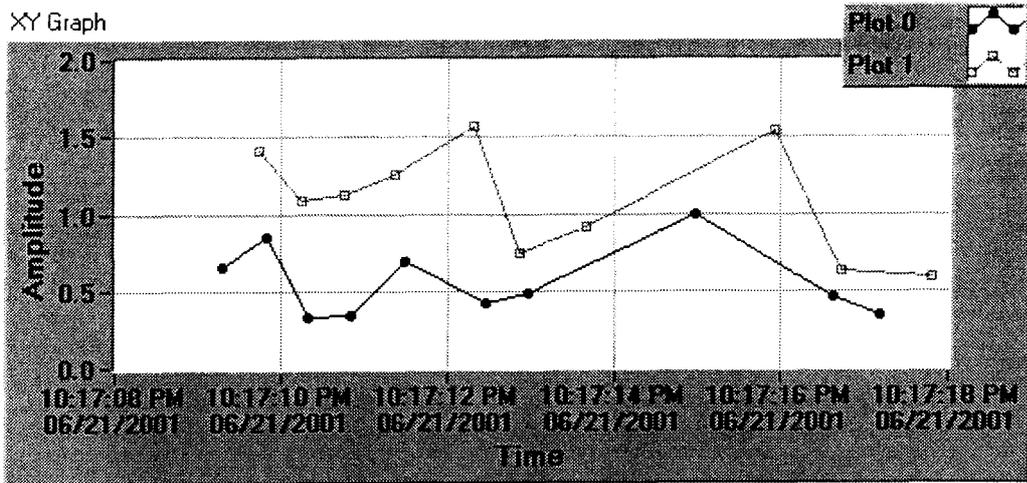


Figure 34: Autoscaling a Simulated Waveform Chart.

We want to simulate the strip chart update mode. To do this, we could take the most recent time and plug this into an XScale.Maximum property node and subtract from this the range of time we want the graph to cover and plug this into an XScale.Minimum property node, but this has the following disadvantages.

First, the graph will put gridlines at intervals that will not generally coincide with the maximum (most recent) time or the minimum (oldest) time. What usually happens is that a vertical gridline appears so close to the maximum time that it does not have room for a display, which can be confusing to the user because it looks like the time of the gridline is the maximum time. See Figure 35.

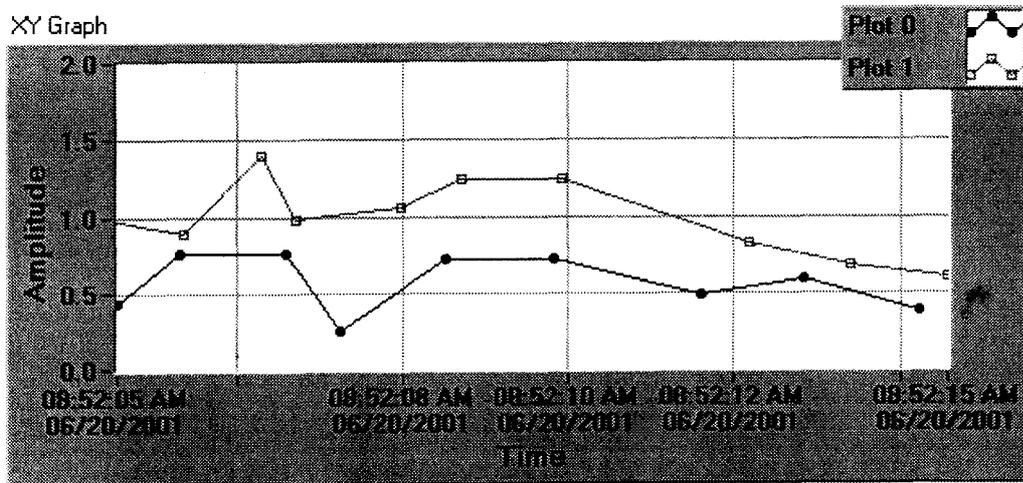


Figure 35: Using the Most Recent Time as the Maximum.

Second, the entire waveform will scroll on every update and this can be quite annoying to the user.

The solution to these problems is to perform a special kind of roundup. First, we have to decide how far apart we want the gridlines to be. They have to be at least far enough apart so that there is room to display a time for every gridline and this will be dependent on the text font, the date/time format, and the physical size of the graph. You have to test this by trial and error. Once you have decided on a spacing (in seconds), you will apply this to an XScale.Increment property node and use this value in a calculation which will be applied to the XScale.Maximum property node. This calculation involves dividing the most recent time by the increment, rounding up and then multiplying by the increment. Finally, we subtract the range from the result and apply it to the XScale.Minimum property node. Figure 36 shows the final result of applying all the tricks.

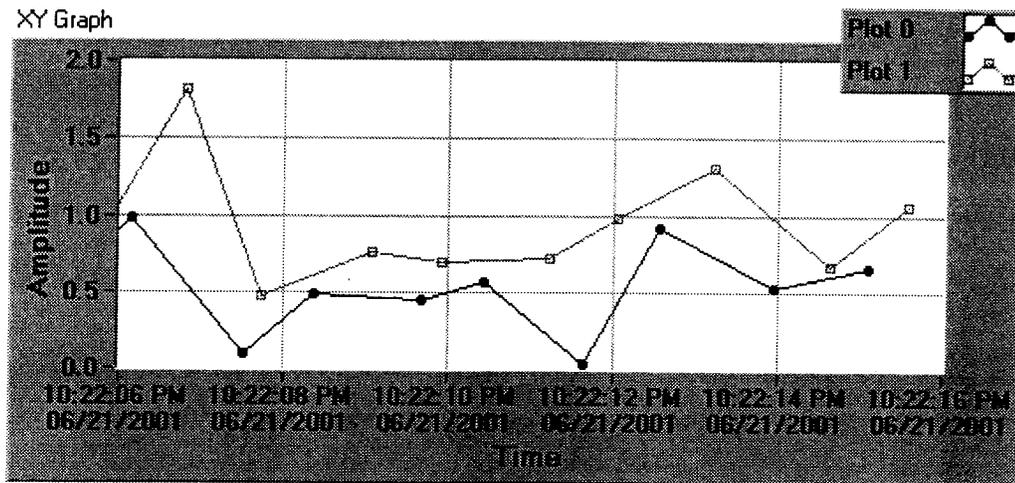


Figure 36: Special Roundup Trick to Calculate Maximum Time.

In our example shown in Figure 36, the increment is 2 and the range is 10. As the graph updates in real time, whenever the last increment overflows, the entire graph scrolls to the left by one increment, partially filling the new last increment, which then continues to fill until it overflows and the process repeats. It presents a very friendly interface to the user. The diagram for the last example is shown in Figure 37. All three examples are available on the CD in Waveform.llb.

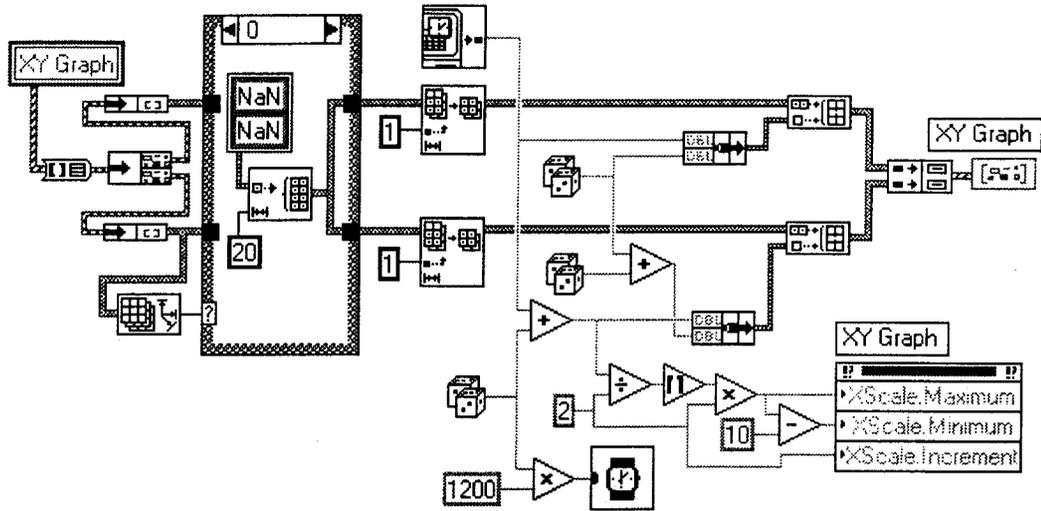


Figure 37: Diagram for Simulated Waveform Chart.

12.2.8 A Warning about Time

There is one more very important trick that we need to know to make graphs work right and look good in LabVIEW. In the previous example, the range and increment are very small numbers. But if we had a graph with a range of several hours or days we might find that LabVIEW would set the gridlines at oddball times, say at 5:00 P.M. instead of at midnight, even though there is only one gridline per day. This is caused by the way that LabVIEW displays a number as a time. Remember that LabVIEW does not have a special data type for time, it merely uses an ordinary double-precision number that has a special meaning. According to the documentation, this meaning is: “a time-zone-independent number that contains the number of seconds that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time”.

However, when LabVIEW displays this number formatted as Time & Date, it is no longer independent of the time zone, nor of the daylight savings state. The significance of this is that LabVIEW decides where to place the gridlines based on Universal Time but marks them with local time. Daily gridlines on a graph at 5:00 PM (Pacific Daylight Time) correspond to midnight Universal Time. Similarly, if you had a graph covering several hours, LabVIEW might place the gridlines at odd numbered hours local time because they correspond to even numbered hours Universal Time.

The solution to this problem (and others that we will discuss soon) is to set the time zone on your computer to GMT. But don't use the one for Greenwich Mean Time because that one permits daylight savings, instead use the one for Casablanca, Monrovia because it never goes on daylight savings.

Here is a very important experiment you should try. Place a digital display on a front panel, keeping its value set to zero. Right-click on the display and select Format & Precision and change the Format to Time & Date. Enlarge the display to show all the digits. Unless your time zone is set to GMT, you will see something other than 12:00:00 AM, 01/01/1904. Now use the Date/Time control panel to change to a different time zone but make sure the control panel is not covering up the display. You will not see the display change, as it should, but if you cover the display with another window such as the control panel and uncover it, it will show the correct change. The moral of the story, don't change time zones in the middle of a LabVIEW session unless you remember to update the displays by covering and uncovering them with another window. This applies to digital controls, indicators, graphs, and constants.

The new waveform data type in LabVIEW 6i is a special cluster that includes t0, dt, and Y. If you wire this directly to a waveform graph, it ignores the value of t0 (the real time of the first data point) and substitutes a value of zero so that the graph displays offset time in seconds starting from zero. If you want to display real time along the x-axis, you need to right-click on the center portion of the graph and uncheck the last item, "Ignore Timestamp". Don't make the mistake of right-clicking on the x-scale markers because you won't see this new menu item. You can do this operation for data as it is acquired in real time, as well as for data that has been previously saved to disk using the Write Waveform to File vi and later retrieved using the Read Waveform from File vi.

Now consider this disconcerting situation. If you have collected data which is explicitly time-tagged using Get Date/Time In Seconds or implicitly time-tagged using the waveform data type (as described in the previous paragraph) and you examine the data after the day light savings state has changed, you will find that the time tags are shifted by one hour. The shift could be even greater if the time zone is different between the computer writing the data and the computer reading the data. This could be an embarrassing situation for you if you are trying to correlate or locate an event in your data file based on a manually generated log of when that event occurred.

All of the foregoing leads to the strong conviction that you should set the time zone for any computer using LabVIEW to GMT (Casablanca, Monrovia) as mentioned earlier.

12.2.9 Using an Intensity Graph to Display a Bitmap Image

The last technique we will cover allows us to precisely display bitmaps in an intensity graph. This is actually fairly well covered in the LabVIEW documentation but there are a few tricks not covered that inspire this discussion. Two of the features of the intensity graphs are that they can automatically interpolate the colors between those specified and they will automatically interpolate the pixels between those specified. In other words, it is possible to under specify the colors and still get acceptable results

and you can resize the graph and still get acceptable results for most applications. But for those applications where you want a one-to-one mapping between the elements in the defining array and the pixels on the graph and where you want precise control of the colors, these tricks will help.

The intensity graph can display a palette of 256 colors and there are two ways to specify these colors, both of which require the use of property nodes. The first allows for interpolation and will not be discussed here. The second uses an array of 256 elements of type U32 integers, which represent the 256 colors in the palette. The meaning of the values of these numbers can best be understood by thinking of the 32-bit integer as divided into four bytes. The first byte is a flag to specify transparent but since this is illegal in an intensity graph, it should always be zero. The second byte is the intensity of the red (R) component of the color, the third byte specifies the green (G), and the fourth byte defines the blue (B) component. In this example, we will specify a palette with 256 shades of gray, so all three of the R-G-B components will be the same within each U32 value. Figure 38 shows the diagram of the example we will use. This example is included on the CD as BitImage.vi.

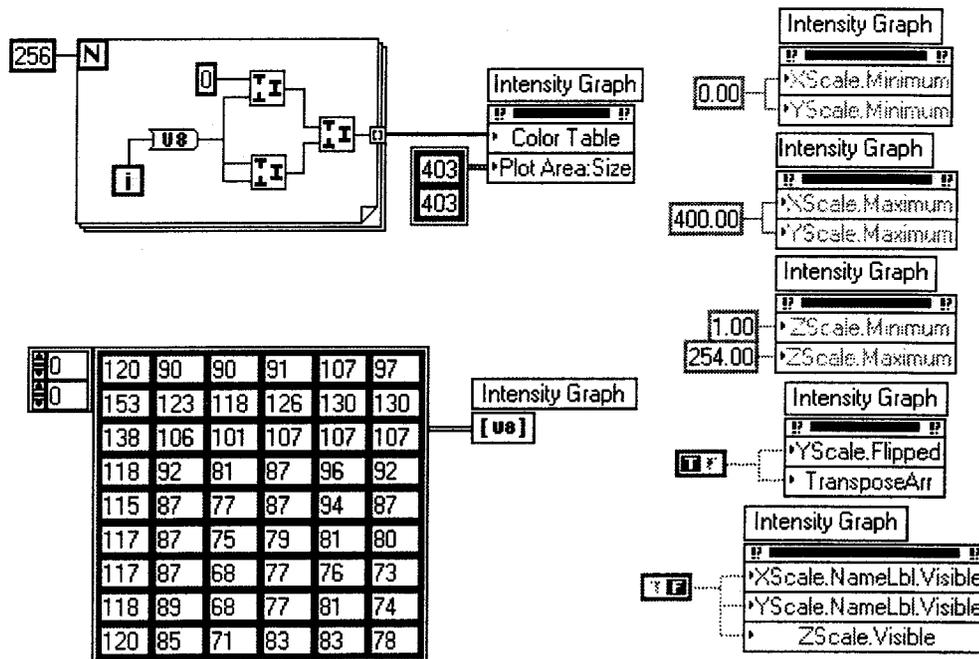


Figure 38: Diagram of Bit Image VI.

The For Loop generates the palette of 256 colors using several Join Numbers functions to first combine pairs of bytes into 16-bit words and then to combine them into 32-bit words. The first word is all zeroes, specifying black. The second word is hex 00010101 specifying the darkest shade of gray, followed by hex 00020202, and continuing all the way up to hex 00FFFFFF which is white. This array of 256 words

is applied to a property node created from the terminal for the Intensity Graph and selected for Color Table.

A second property is opened below the Color Table for the Plot Area Size and a constant is created for it with both values set to 403. This is because the actual area of the image for this example is 400 by 400 and we need to set the size parameters three greater. Why three? It's a trick. Your guess is as good as mine. By making the X and Y maximums 400 (instead of the actual 399), there is a black edge of pixels along the bottom and right hand edges of the graph. If you find this objectionable, change the maximums to 399 and use 402 for the sizes of the plot area. Of course, the image doesn't have to be square, you can use any size you want. Just make sure that the numbers you use for the Plot Area Size are three greater than the corresponding X and Y Scale Maximums.

There are several other property nodes along the right hand edge of the diagram. All of these properties could have been entered manually from the graph on the front panel but for the sake of clarity, we show them explicitly in the diagram.

There is one other important trick embedded in one of these property nodes and that is the Z Scale Minimum and Maximum. These come up as 0 and 100 when you place a new intensity graph on a front panel so you might be inclined to change the maximum to 255 thinking that this will cover the full range of possible intensities, but you would be wrong. There are actually two other colors, one above the maximum and one below the minimum. These handle intensities that are outside the specified range. The first value in the Color Array defines the color used by intensities below the minimum (in this case zero, making the minimum one) and the last value in the Color Array defines the color used by intensities above the maximum (in this case 255, making the maximum 254).

The image that is displayed by this example VI is defined by a two-dimensional U8 array constant. In order to show the array in the same orientation as the graph, we transpose the array and flip the Y Scale using another property node.

Finally, we use one last property node to turn off the scale labels and ramp since these are meaningless for this application.

12.3 Real-World Applications

Armed with all these skills, it is time to see what they can do. These applications are taken from several projects where LabVIEW is used at NASA's Jet Propulsion Laboratory. Instead of calling them real-world applications, it might be more appropriate to call them out-of-this-world applications.

12.3.1 Galileo

The image that is produced by the example in section 12.2.9 is shown in Figure 39. It is a picture taken by the Galileo spacecraft of lava flows on one of Jupiter's moons Io that had been corrupted but fixed using a LabVIEW program. You can see the original and read about the problem and fix at:

<http://photojournal.jpl.nasa.gov/cgi-bin/PIAGenCatalogPage.pl?PIA02517>

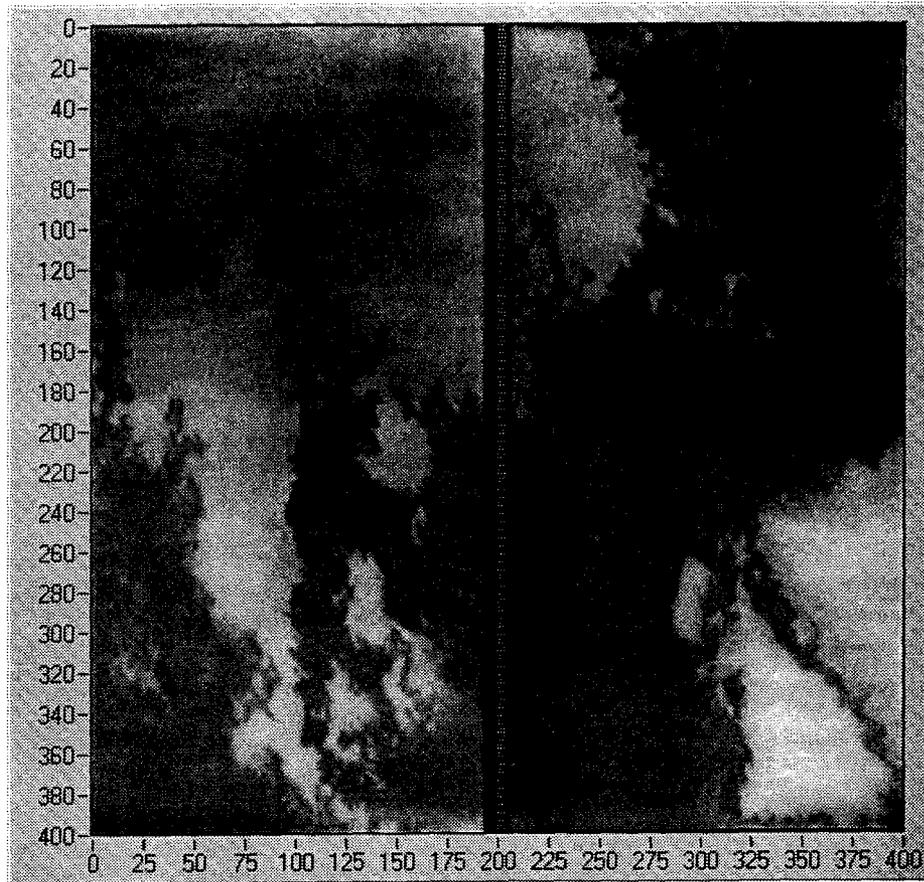


Figure 39: Reconstructed image of lava flows on Jupiter's moon Io taken by Galileo spacecraft (using diagram of Figure 38).

12.3.2 Mars Pathfinder Rover Mission

Next we have a pair of pictures used during the Mars Pathfinder Rover mission in 1997.

12.3.2.1 Kinematics

Figure 40 was used to show the orientation of the rover and its wheels as it navigated the Martian terrain. The VI that generated this graph made heavy use of the skills described in sections 12.2.2 through 12.2.4. Each of the wheels on the rover are independently suspended through the use of the bogeys. The front of the rover is to

the right in the side views. At the rear of the rover is the Alpha-Proton-X-ray-Spectrometer (APXS) instrument, which can be lowered to contact the ground or the face of a rock. The radial lines coming out the front and rear indicate the field of view of the three cameras. The small squares are contact sensors that tell the rover when it has hit an obstacle.

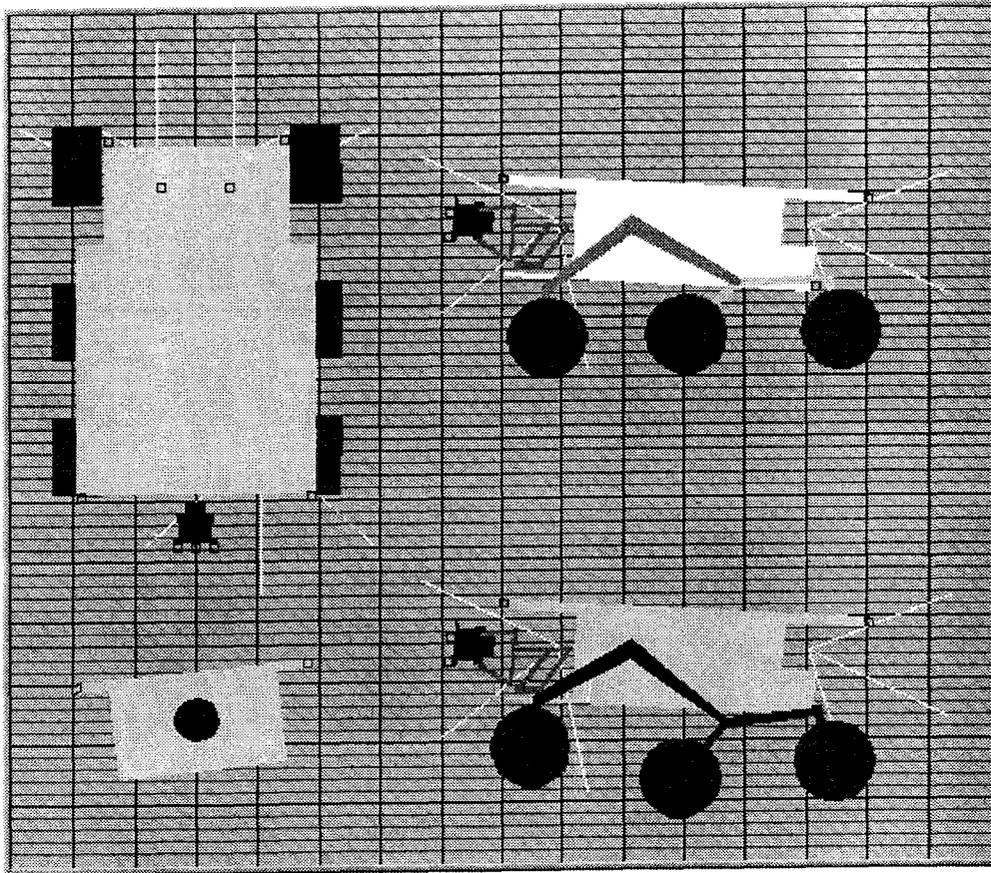


Figure 40: Graph orientation of rover used during Mars Pathfinder Mission in 1997.

12.3.2.2 Navigation

Figure 41 was used to show the location and path of the rover with respect to the lander and the major rocks (shown as rectangles). This image shows the progress of the rover on the third day. It backed down the ramp and measured the soil with its APXS instrument. The grid lines show distances in meters. This graph also used the techniques described in sections 12.2.2 through 12.2.4. You can learn more about the Mars Pathfinder Mission at:

<http://www.jpl.nasa.gov/missions/past/marspathfinder.html>

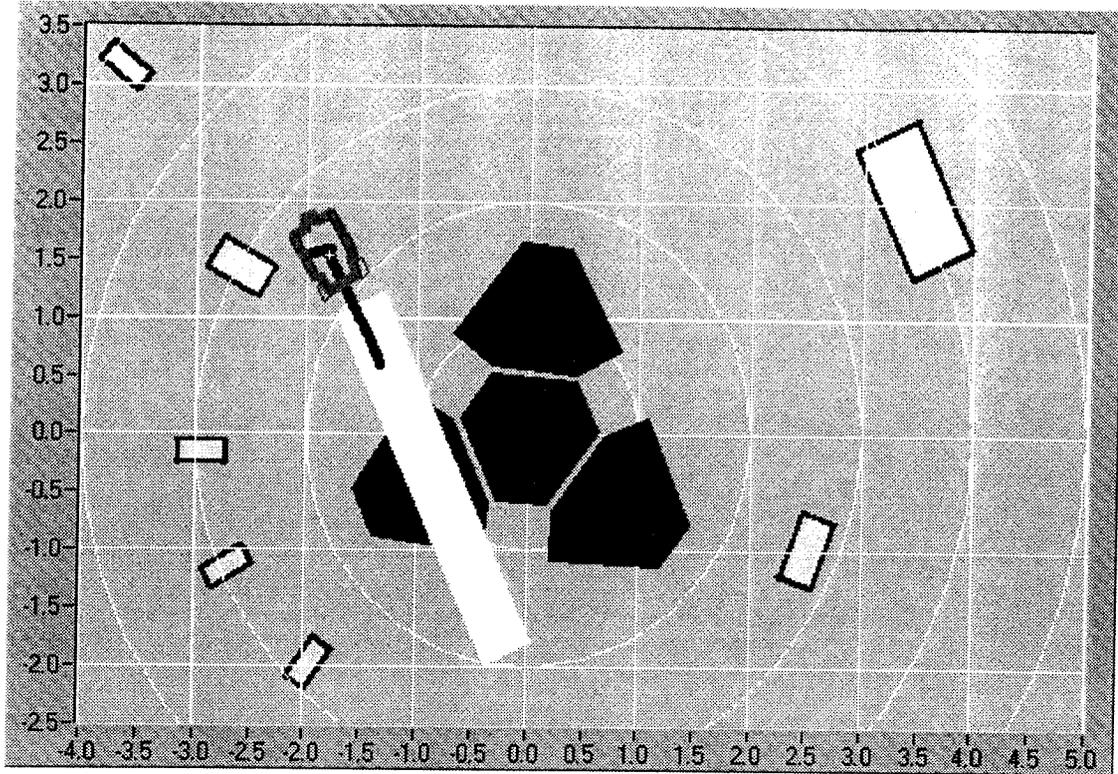


Figure 41: Graph showing path of rover used during Mars Pathfinder Mission in 1997.

12.3.3 SeaWinds

The SeaWinds instrument is a rotating dish radar antenna that beams down at the surface of the ocean and senses the backscatter off the ocean surface to measure the wind. The data consist of thousands of individual wind vectors consisting of a latitude, longitude, magnitude and direction, far too much data to present on a graph or for a human to comprehend. The LabVIEW VI that generates the graph in Figure 42 averages vectors within each 1-degree latitude by 1-degree longitude cell and then creates streamlines that attempt to connect adjacent cells that exhibit average wind directions that are more or less in the same direction. The streamlines are all one plot as described in section 12.2.5. The magnitudes of the average winds are shown with different colors using the technique from section 12.2.6.

You can learn more about this SeaWinds instrument at:

<http://www.jpl.nasa.gov/missions/current/quikscat.html>

Jet Propulsion Laboratory
Ten-day Temperature Charts
Measurement Technology Center/Instrument Services
Pasadena, California, U.S.A.

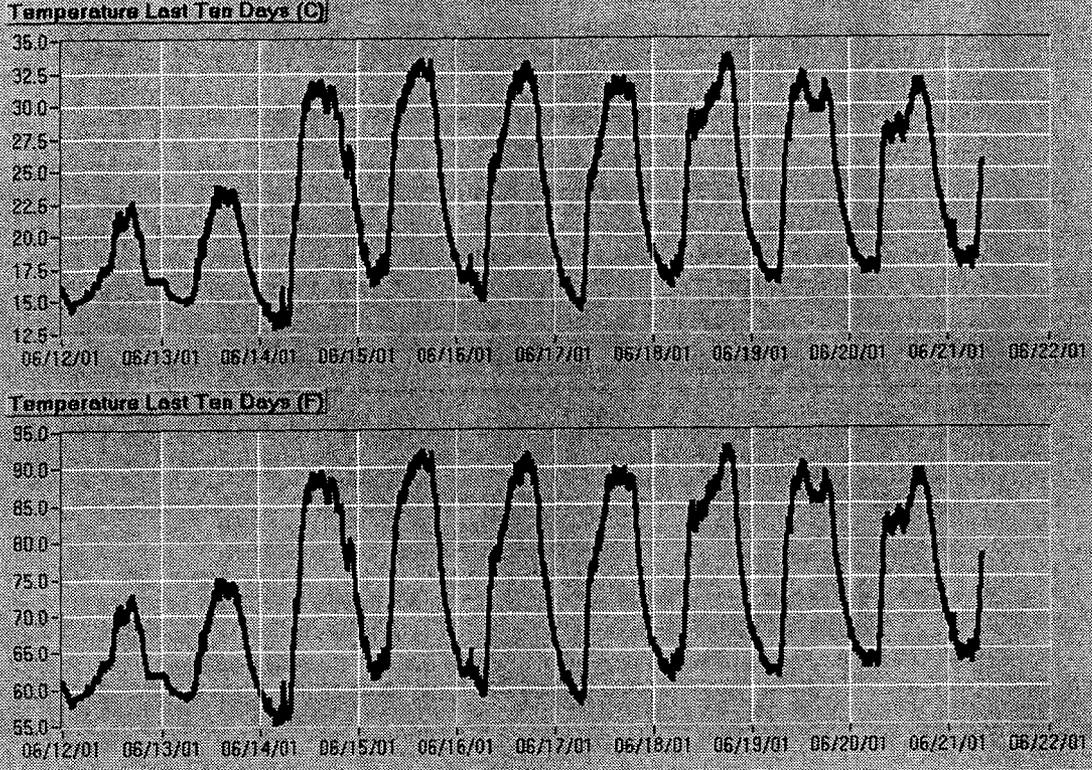


Figure 43: Temperature Graphs from the JPL weather station.