PhUSE 2010

Paper TS09

Capturing Tabular Data from Graphical Output: Web and Windows-Based Tools

Brian Fairfield-Carter, ICON Clinical Research, Redwood City, USA
Stephen Hunt, ICON Clinical Research, Redwood City, USA

ABSTRACT

The pharmaceutical industry is a competitive arena where new drugs inevitably have to find a market among competitors. Comparisons against these competitors can be problematic, however, since proprietary and legacy data may not be available in analysis-ready format. How would you meet a sponsor's request to overlay a reference curve for a competitor drug on an efficacy plot where the data that produced the reference curve were not available? The most obvious but least appealing approach would probably be to estimate 'by eye' the X and Y coordinates for each data point on the reference curve.

This paper presents an accurate and labor-saving alternative: use a graphical API to capture screen coordinates directly from the graphical display, and translate the screen coordinates to plot coordinates in order to recover the original data values. Two simple, open-source applications (one Web-based (HTA) and the other WinAPI-based) are presented that implement this technique.

INTRODUCTION

Developing and marketing a new compound can not only involve demonstrating efficacy against placebo, but also against other compounds. In an industry typified by active research programs and a steady stream of new data, planned analyses are often supplemented by unplanned and ad hoc comparisons against other published information. Such comparisons may require the use of legacy and/or proprietary data that are not available in analysis-ready, tabular form.

To illustrate, suppose you have created a Kaplan-Meier survival curve (Figure 1), and a request is made to overlay a reference curve from a competitor drug (Figure 2):

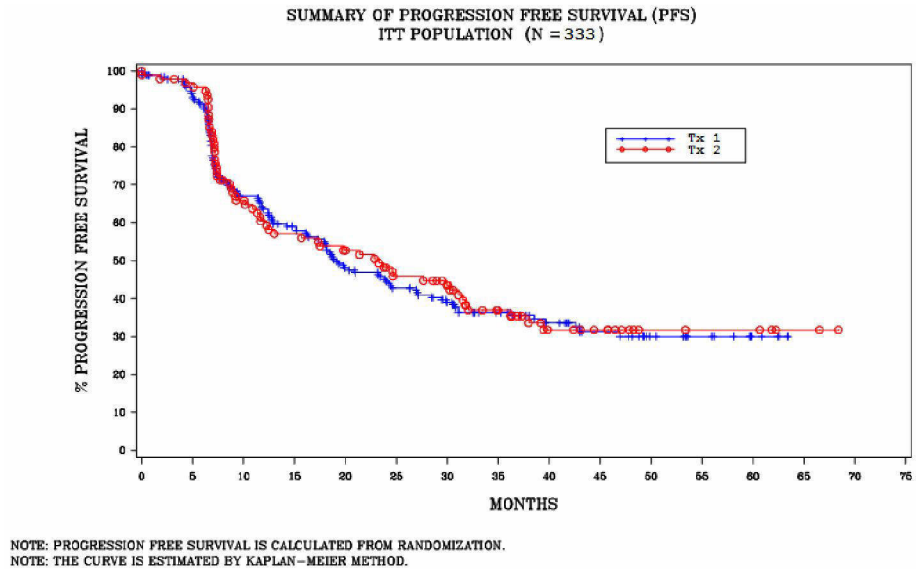

Figure 1. Survival curve to which reference curve is to be added

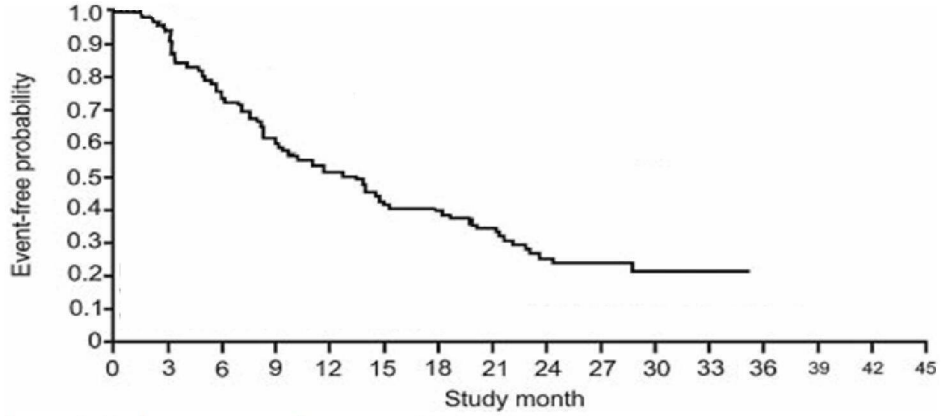

Figure 2. Reference curve for competitor

Raw tabular data for the reference curve are not available (the reference curve is only available as a .pdf image), but the curve somehow has to be represented in SAS/Graph output:

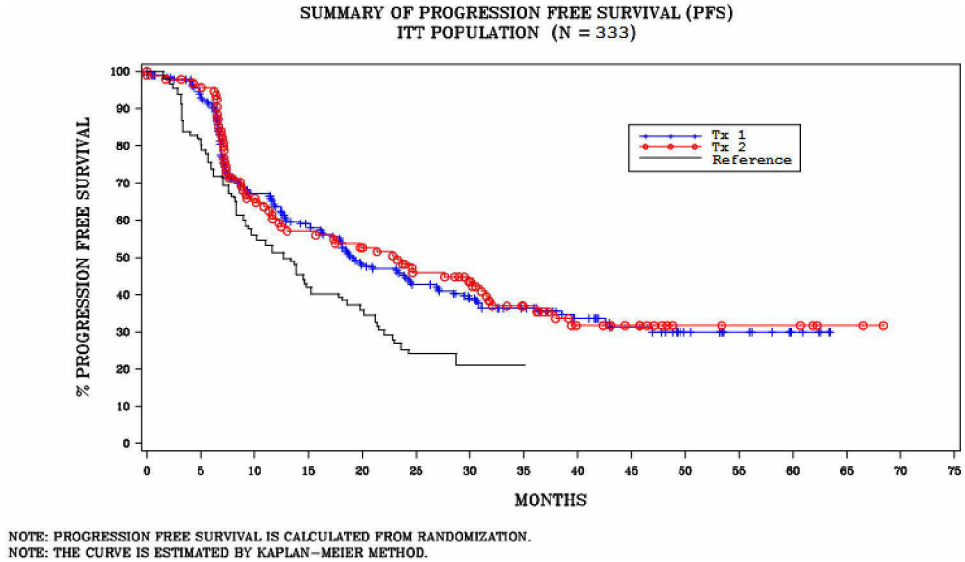

Figure 3. Objective: survival curve with reference curve added

Clearly, plotting the reference curve in SAS/Graph requires that data points be available in a SAS dataset, meaning some sort of 'reverse-engineering' has to take place. A manual approach, attempting to estimate data values from drop-down lines to the X and Y axes, is not only inaccurate, but is also very time-consuming.

A partial solution can be provided by displaying the image in Windows Paint (i.e. as a screen-capture), which displays the current position (in screen coordinates) of the mouse pointer, provided the 'status bar' is activated (select the View/Status Bar drop-down menu) (Figure 4). This means that screen coordinates for each target data point can be recorded with relative precision, and then some rudimentary transformation performed to derive (X,Y) values that are meaningful in the context of the original plot axis system. However, the process of manually recording screen coordinates is itself very labor-intensive, so while this improves accuracy it doesn't provide an entirely viable solution.

What is required is an application that tracks screen coordinates for the mouse pointer, and also automatically logs these coordinates to an output file. Since the concept of a 'graphical user interface' by and large involves tracking screen coordinates (for example, to keep track of cursor position and mouse pointer position, color attributes of individual pixels, etc.), its not surprising that an Applications Programming Interface (API) tends to give the programmer access to and control over the same coordinate system, meaning the requirements of a simple data-capture system are actually fairly easy to meet.

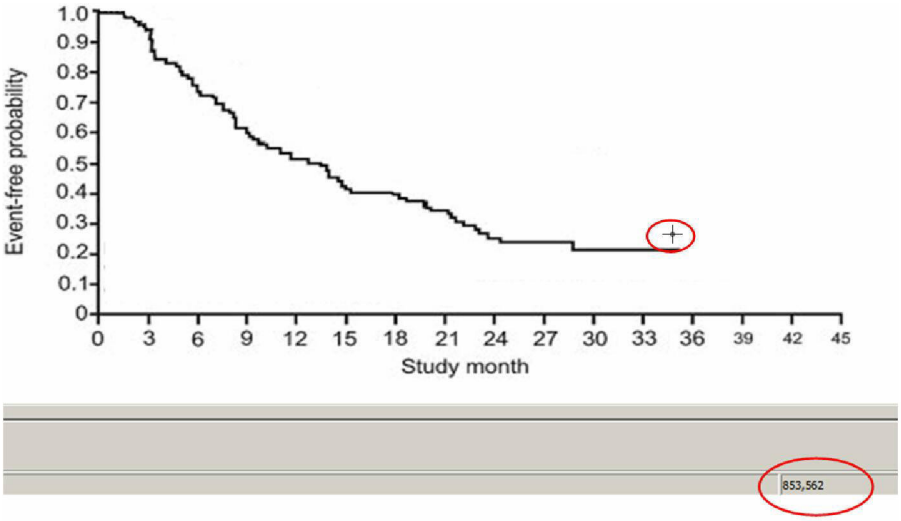

Figure 4. Image displayed in Windows Paint, showing screen coordinates for the mouse pointer

A WEB-BASED APPLICATION

Probably the most convenient applications-programming interface is provided by a web browser such as Internet Explorer, which not only renders HTML but also interprets the script (i.e. VBScript, JavaScript) components of 'dynamic' HTML. However, web browser security features often make it impractical to use dHTML, so the variant 'HTA', or 'HTML Application' makes more sense (HTA is interpreted under Windows by the 'HTML Application Host'). The essential difference between dHTML and HTA applications is that the former have to adhere to the security rules of the web browser, which assumes communication with remote servers, while the latter resides and is executed only on the local machine. This means that in HTA, the power of dHTML is available without having to negotiate security policies.

In order to capture screen coordinates for specific data points on a graph, the application must perform three essential functions:

1. Display the graphical image (for simplicity, we'll assume the image is a bit-map file, created as a screen-capture of the original figure)
2. Track mouse pointer position and determine screen coordinates at key 'events' (such as mouse-clicks)
3. Write screen coordinates to a file

The first two tasks are rudimentary to any graphical user interface, so it's not surprising that it really takes very little code to implement. The following example illustrates probably the simplest way of displaying an image, which is simply as the background to a window, and shows how 'events' can be handled by user-defined functions to capture and display screen coordinates for the mouse pointer. This example is pared down for illustrative purposes, but is nonetheless perfectly functional and usable.

```
<center><div id="divid"> TEST </div></center>

<body id="bodyid"
      onClick="capture(event)"
      onMousemove="getcoord(event)"
      onUnload="endcapture(event)"
      background="graph.bmp">
</body>

<head>

   <script type="text/javaScript">

      var fso=new ActiveXObject("Scripting.FileSystemObject");
      var wshell=new ActiveXObject("WScript.Shell");

      var mytext=fso.CreateTextFile("MyCoordinates.xml",true);
```

```
      mytext.Writeline("<?xml version='1.0'?>")
      mytext.Writeline("<?xml-stylesheet type='text/xsl'
href='coordinates.xsl'?>")
      mytext.Writeline("<catalog>")

      function getcoord(event)
      {
         var x=event.clientX;
         var y=event.clientY;
         document.getElementById("divid").innerText="COORDINATES:" + x + " & " +y;
      }

      function capture(event)
      {
         var x=event.clientX;
         var y=event.clientY;
         mytext.Writeline("<COORDINATES>");
         mytext.Writeline("<X_COORD>" + x + "</X_COORD>");
         mytext.Writeline("<Y_COORD>" + Math.round(screen.height-y) +
"</Y_COORD>");
         mytext.Writeline("</COORDINATES>");
      }
      function endcapture(event)
      {
         mytext.Writeline("</catalog>");
         mytext.Close();
      }

   </script>
</head>
```

SOURCE CODE EXPLAINED

In the 'body' of the HTA, three standard events ('onClick', referring to mouse clicks, 'onMouseMove', and 'onUnload', referring to the act of closing the application) are passed to user-defined functions declared in the script component of the application. This means that each time any of these events fires, the corresponding function is called. The image 'graph.bmp' is displayed as the background to the application window, and a region within this window is declared to display the current screen coordinates of the mouse pointer (continuously updated as the mouse pointer moves). There is an 'onLoad' event that takes place as the application launches, and by default the script is run as part of the loading process. This essentially 'loads' the user-defined functions, and also opens a text file ('MyCoordinates.xml') to receive captured screen coordinates. Note that the xml file includes a style-sheet reference 'coordinates.xsl' which allows the xml data to be displayed in tabular format in the web browser (this will be shown later).

To launch the application, simply save the source code above to a text file with '.hta' extension, and double-click on the file. If you check your PC's process list, you'll see something called 'mshta.exe', which is essentially the component of the Windows operating system that interprets the application. The application looks like this when launched:

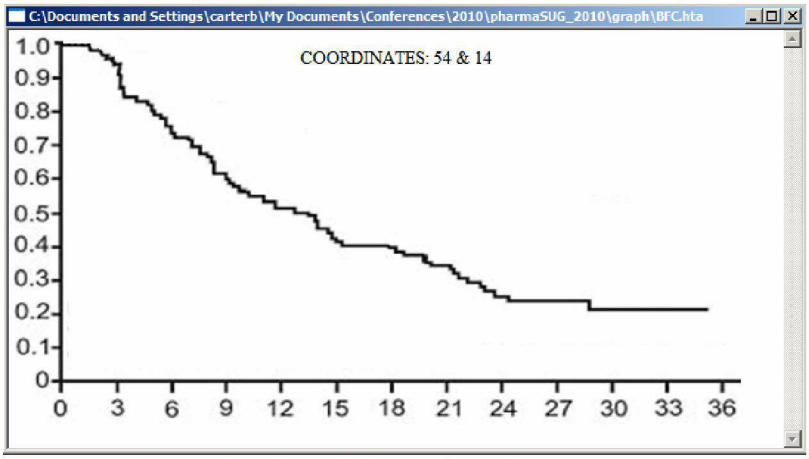

Figure 5. HTA Application Displaying Sample Figure

When the mouse pointer is moved over the window ('client' area), the 'onMousemove' event calls the 'getcoord' function, which captures coordinates as 'clientX' and 'clientY' attributes of the mouse event:

```
var x=event.clientX;
var y=event.clientY;
```

These coordinates are then displayed at the top of the window. Mouse click events call the 'capture' function, which also determines screen coordinates for the mouse pointer, and writes them to the xml file. In order to capture specific data points, simply position the mouse pointer over each desired data point and click. When the application is closed, the 'endcapture' function closes the xml file.

The resulting xml file is as follows:

```
<?xml version='1.0'?>
<?xml-stylesheet type='text/xsl' href='table.xsl'?>
<catalog>
<COORDINATES>
<X_COORD>72</X_COORD>
<Y_COORD>1005</Y_COORD>
</COORDINATES>

...

 <COORDINATES>
<X_COORD>176</X_COORD>
<Y_COORD>911</Y_COORD>
</COORDINATES>
</catalog>
```

A WINDOWS-BASED APPLICATION

For those willing to take on greater programming over-head, but with the reward of more control over the user interface and a richer set of interface components, the same application can be implemented on the Windows Application Programming Interface (API). The example shown here is implemented as a 'compiled' application written in C; while the source code is more involved, the three essential tasks of displaying the graphical image, tracking mouse pointer position and writing captured screen coordinates to a file are actually pretty recognizable. Complete source code is not provided here, but is available for download (along with the stylesheets discussed later on) from the 'Shellout' package on SourceForge at http://sourceforge.net/projects/shellout/. The application is built on MinGW (Minimalist GNU for Windows), a pared-down Windows version of GNU (GNU provides Unix functionality in an open-source distribution) providing header files, utilities, and a Windows version of the GNU Compiler Collection. For further details, and information on how to build the application from source, refer to Fairfield-Carter *et al* (2006).

To display a bitmap image in the application's graphics window:

```
hBmp = LoadImage(<instance>, <file>, IMAGE_BITMAP, 0, 0, <options>);
RedrawWindow(<bitmap window handle>,0,0,RDW_INVALIDATE);
```

Because the Windows API is fairly 'low-level', instructions have to be given both to load the image into memory, and then redrawn the bitmap window in order to display the image. While this low-level control is useful, it also means that you have to remember to re-draw the window after various events (for example, if a new bitmap file is opened, or if a dialog is launched in front of the bitmap window).

To track mouse pointer position and determine screen coordinates:

```
switch (message)
  {
     case WM_MOUSEMOVE:
       xPos = LOWORD(lParam);
       yPos = HIWORD(lParam);
```

The Windows API is a 'message-driven' interface, and when the mouse pointer moves, the application captures and deciphers the message to determine the x- and y-coordinates of the mouse pointer. The analogy to the 'onMousemove' event and associated function shown in the HTA example should be readily apparent, though again the code to trap and handle the event is a little more involved.

Finally, to log coordinates to an XML file:

```
bSaveFileName = GetSaveFileName(&sfn);
f=fopen(sfn.lpstrFile,"w");
fprintf(f,"%s","<?xml version='1.0'?>\n");
…(etc.)…
```

This application is a little 'fancier', since it calls the standard Windows 'save' dialog to allow the user to specify the file location and file name to save to; the file is opened for writing, and XML tags and x/y coordinates are written out to the file.

USER INTERFACE COMPONENTS

The interface is fairly simple to use, and consists of a small number of basic features. The 'File/Open' menu launches the standard Windows 'Open' dialog:

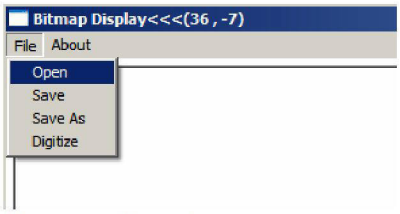

Figure 6. Drop-down Menus

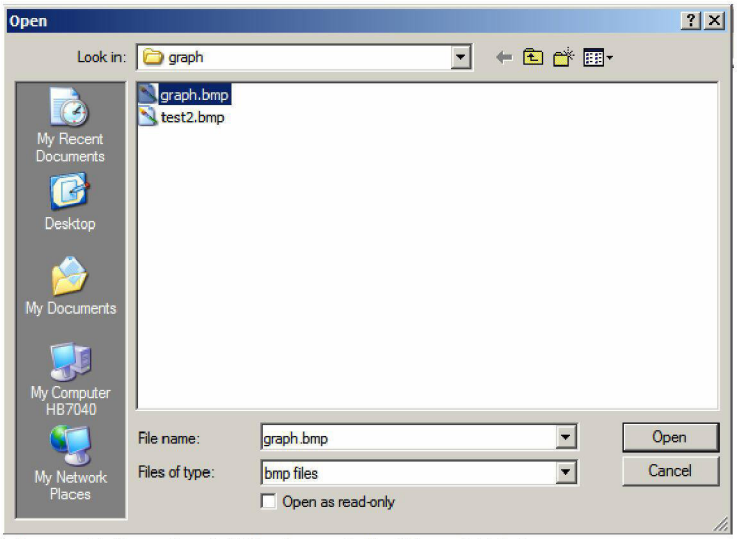

Figure 7. Standard Windows 'File/Open' Dialog

Only bitmap files can be opened by the application, and are displayed in a bitmap window which is automatically re-sized to the size of the image:

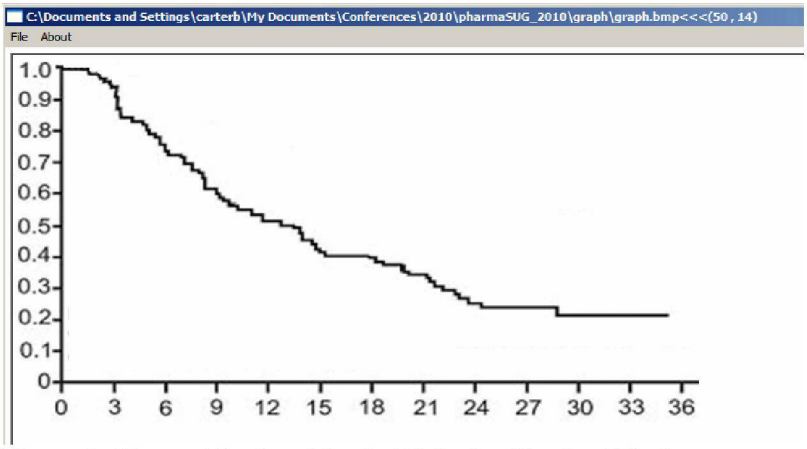

Figure 8. Bitmap Displayed in the Main Application Window

Similar to the HTA example, the title bar of the main window displays the path and file name of the currently opened bitmap, along with the screen coordinates of the mouse pointer (these coordinates are updated continually as the mouse pointer moves).

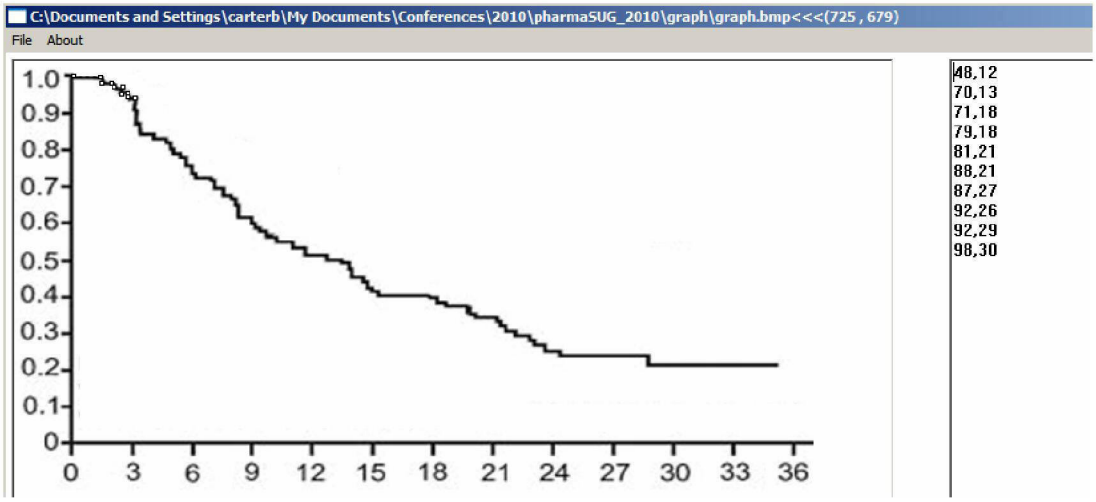

Figure 9. Target Data Points Highlighted on the Bitmap, with Screen Coordinates Logged in the Text Window

Screen coordinates for target data points are (as with the HTA example) captured by placing the mouse pointer over the desired data points, and clicking the left mouse button. However, to provide some feedback and the ability to review selected data points, when the mouse button is clicked, a small box is placed around the selected graph point, and the screen coordinates are logged in a small text window to the right of the bitmap window. Coordinates can also be typed and edited in the text window, with revisions reflected in the bitmap window each time it is refreshed. Data points in the bitmap window can also be selected and moved (simply click inside one of the boxes surrounding a given point, and then use the arrow keys on the keyboard: the box is re-positioned, and the coordinates displayed in the text window are indented so that they can be easily spotted, and are update to match the new position).

The 'File/Save' or 'File/Save As' menus launch the standard Windows 'Save/Save As' dialog, enabling the user to save the logged screen coordinates to an XML file:

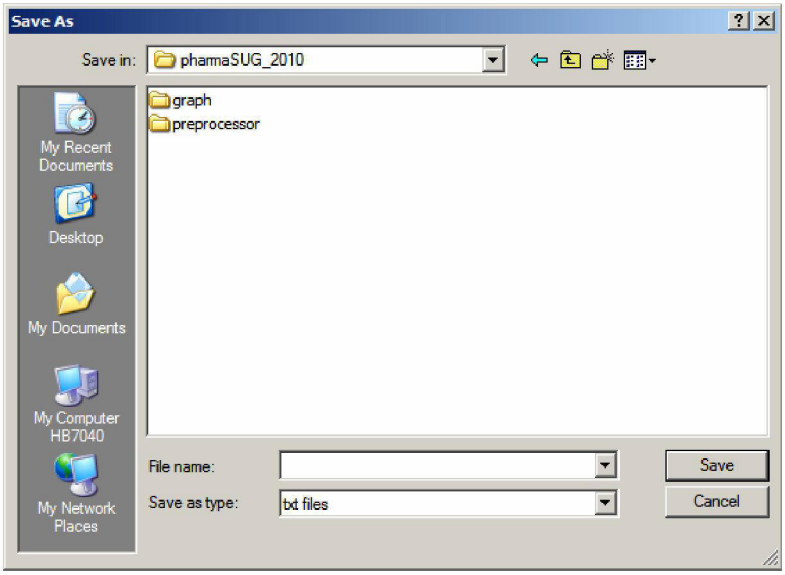

Figure 10. Standard Windows 'Save As' Dialog

Coordinates saved as an XML file:

```
<?xml version='1.0'?>
<?xml-stylesheet type='text/xsl' href='chart.xsl'?>
<catalog>
<COORDINATES>
  <X_COORD>48</X_COORD>
  <Y_COORD>12</Y_COORD>
</COORDINATES>
<COORDINATES>
  <X_COORD>70</X_COORD>
  <Y_COORD>13</Y_COORD>
</COORDINATES>
...(etc.)...
```

'COMPLETE' DATA CAPTURE

The interface provides one other experimental feature, accessed through the 'File/Digitize' menu:

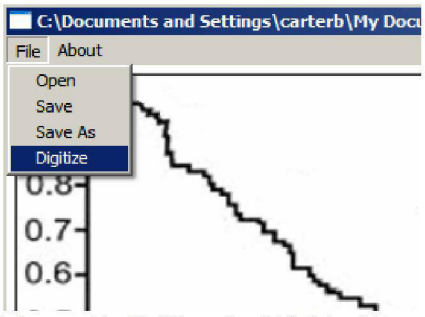

Figure 11. Calling the 'Digitize' Function

This menu selection calls a function which loops through the entire bitmap, pixel by pixel, and logs the screen coordinates of every non-white pixel. In most cases the use of this feature isn't really practical, since it captures far more data points than are necessary (usually the only data points of interest are those that fall at 'inflection points' along the curve, not at points on interpolated sections of the curve), and (so far) does not discriminate between 'useful' data points (those falling on the curve you're trying to capture) and extraneous elements, like axis lines and labels (this means that if you use this feature, you'll want to first edit the bitmap).

IMPORTING AND TRANSLATING SCREEN COORDINATES TO PLOT COORDINATES

Having captured screen coordinates for target data points, how do you get these coordinates into a SAS dataset, and how do you translate these to plot coordinates, which have meaning within the context of the plot coordinate system?

The XML file can be read into a SAS dataset very simply using the XML libname:

```
filename myxml 'graph.xml';
filename sxlemap 'graph.map';

libname myxml xml xmlmap=sxlemap;

data graph;
  set myxml.coordinates;
run;
```

Well, fairly simply; SAS requires an XML 'map' file (in this case called 'graph.map') to specify how the XML file is to be parsed:

```
<SXLEMAP VERSION='1.1' NAME='COORDINATES'>
  <TABLE NAME='COORDINATES'>
    <TABLE-PATH>/catalog/COORDINATES</TABLE-PATH>
      <COLUMN NAME='X_COORD'>
        <PATH>/catalog/COORDINATES/X_COORD</PATH>
        <TYPE>NUMERIC</TYPE>
        <DATATYPE>INTEGER</DATATYPE>
        <LABEL>X Coordinate</LABEL>
        <LENGTH>8</LENGTH>
      </COLUMN>
      <COLUMN NAME='Y_COORD'>
        <PATH>/catalog/COORDINATES/Y_COORD</PATH>
        <TYPE>NUMERIC</TYPE>
        <DATATYPE>INTEGER</DATATYPE>
        <LABEL>Y Coordinate</LABEL>
        <LENGTH>8</LENGTH>
      </COLUMN>
   </TABLE>
</SXLEMAP>
```

This effectively instructs SAS that the table 'COORDINATES' contained in the XML file consists of two columns (both of numeric/integer data type), indicated by the tags 'X_COORD' and 'Y_COORD'.

Now that we have the data in a SAS dataset, we need to translate screen coordinates to plot coordinates. For the X axis, we need, for each data point, the ratio of the distance from the SCREEN x-axis origin divided by the total length of the SCREEN x-axis, multiplied by the PLOT x-axis range. Similarly, for the Y-axis, we need the ratio of the distance from the SCREEN y-axis origin divided by the total length of the SCREEN y-axis, multiplied by the PLOT y-axis range. In a SAS datastep, this is implemented as:

```
data graph;
  set graph;
  plot_x=((screen_x-screen_x_minimum)/(screen_x_maximum-
          screen_x_minimum))*plot_x_range;
  plot_y=((screen_y_maximum-screen_y)/(screen_y_maximum-
          screen_y_mimum))*plot_y_range;
run;
```

The 'plot_x' and 'plot_y' values can now be used by PROC GPLOT to reproduce the original graph within the context provided by its original coordinate system. By assigning an arbitrary treatment code and concatenating the 'graph' dataset to that producing the survival curve (Figure 1), and passing the resulting dataset to PROC GPLOT, we can produce the figure with reference curve as shown in Figure 3:

```
data graph;
```

```
  set graph;
  trt=3;
run;

data final;
  set final graph;
run;

proc format;
  value trtcode_ 1="Tx 1"
                 2="Tx 2"
                 3="Reference";
run;

proc gplot data=final;
  plot plot_y*plot_x = trt;
    / vaxis=axis1 haxis=axis2 ;
run;
```

RENDERING XML USING XSL AND VML

Prior to importing XML data into SAS, it's useful to be able to view the data, and even visualize how it will look when plotted. There is a large and bewildering array of tools and 'meta-languages' for parsing, transforming and rendering XML, and we'll just touch on some basics here. XSL stands for 'EXtensible Stylesheet Language', and essentially describes how the XML file should be displayed (for instance, in a web browser). XML tags have no meaning to a web browser, so the XML document needs to be 'transformed' into HTML (XHTML); XSL provides a mechanism for doing this (actually, more specifically, 'XSLT' for 'XML Transformation').

From our simple XML file containing x- and y-coordinates:

```
<?xml version='1.0'?>
<?xml-stylesheet type='text/xsl' href='coordinates.xsl'?>
<catalog>
<COORDINATES>
<X_COORD>72</X_COORD>
<Y_COORD>1005</Y_COORD>
</COORDINATES>
```

, notice that there is a stylsheet 'coordinates.xsl' specified. This particular stylesheet transforms the XML file into an HTML table:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>

<body>

<table valign="top" align="left" width="20%" border="1" id="myTable" style="font-
family:Arial Narrow;font-size:12px">
  <tr>
   <thead>
     <th width="100">X-Coordinate</th>
     <th width="100">Y-Coordinate</th>
   </thead>
  </tr>

  <xsl:for-each select="catalog/*">
   <tr align="left">
     <td valign="top" align="left"><xsl:value-of select="X_COORD"/>      <span
cols="18"></span></td>
     <td valign="top" align="left"><xsl:value-of select="Y_COORD"/>      <span
cols="18"></span></td>
```

```
    </tr>
  </xsl:for-each>

 </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

When rendered in a web browser, this HTML table looks like this:

X-Coordinate	Y-Coordinate
72	1005
103	979
136	947
176	911

Figure 12. XML Rendered as Tabular Data

What if you wanted to render the XML file as a graph, to get a preview of what it might look like when plotted in SAS/Graph? The process is more involved, but essentially follows the same pattern: the XML must be transformed to something interpretable to the web browser, but instead of displaying tabular data, the web browser must draw various graphics components to produce a plot. Vector Markup Language (VML), an XML language used to produce vector graphics, is especially suited to this task.

VML consists of elements (rectangles, circles, lines, etc.) and attributes (color, etc.); for example, here is an HTML document with embedded VML drawing an oval and a line:

```
<html>
 <body>

  <!-- Include the VML behavior -->
  <style>v\: * { behavior:url(#default#VML); display:inline-block }</style>
  <!-- Declare the VML namespace -->
  <xml:namespace ns="urn:schemas-microsoft-com:vml" prefix="v" />

  <v:oval style="width:100pt;height:50pt" fillcolor="red">
  </v:oval>
  <v:line from="0,10" to="50,50">
  </v:line>

</body>
</html>
```

In Internet Explorer, this is displayed as:

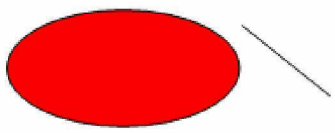

Figure 13. Shapes Drawn using VML

To recap, in order to display our 'coordinates' xml data as a graph, as opposed to a table, we need to transform the XML data into HTML with embedded VML elements (dots and connecting lines); as with HTML table, this transformation would be specified in a stylesheet:

```
<?xml-stylesheet type='text/xsl' href='chart.xsl'?>
```

Using this stylesheet, the XML is displayed in the web browser as:

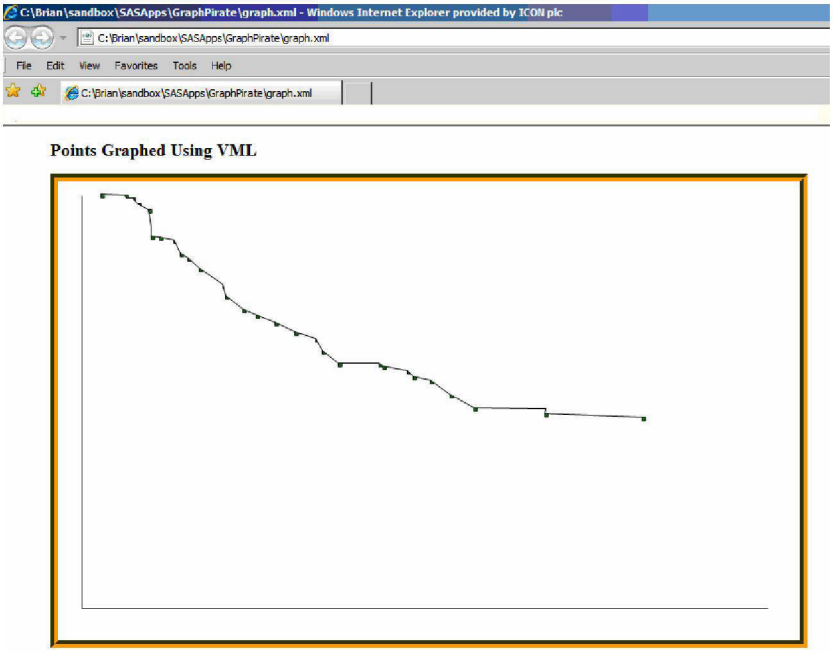

Figure 14. XML Rendered Graphically using VML

While this example isn't very sophisticated, it does give some idea of the potential offered by VML.

CONCLUSION

As a manual process, reverse-engineering data from graphical output is a tedious and error-prone task, and this provides considerable incentive for exploring applications-programming interfaces to automate the task. This paper presented simple web and windows-based applications, usable in their own right, but also serving as starting points for readers interested in furthering their own applications-programming skills.

REFERENCES

SAS, GNU & Open Source: MinGW Development Tools and Sample Applications. Brian Fairfield-Carter, Stephen Hunt, and Tracy Sherman. Proceedings of the 2006 Pharmaceutical Industry SAS Users Group Conference.

ACKNOWLEDGMENTS

We would like to thank ICON Clinical Research for consistently encouraging and supporting conference participation, and all our friends at ICON for their great ideas, enthusiasm and support, in particular Syamala Schoemperlen and Jackie Lane. We would also very much like to thank John Moone for making our participation in this year's conference possible.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Brian Fairfield-Carter, Stephen Hunt
ICON Clinical Research
Email: fairfieldcarterbrian@gmail.com