PharmaSUG 2018 - Paper DV-22

Adding another dimension to oncology graphs: 3-D Waterfall Plots in SAS

Elizabeth Thomas and Mark Woodruff, Epizyme, Inc.

ABSTRACT

Waterfall plots and swimmer's plots are almost ubiquitous for displaying the results of oncology trials. Both present compact patient-level summaries that can be quickly absorbed and offer more detail than a traditional table of best response, but they are typically presented as separate outputs with differing sort orders. This separation makes it difficult to match a patient's change in tumor burden with their duration of therapy. One proposed visualization displays the traditional waterfall plot on the facing X-Z plane and adds a swimmer's plot of duration of therapy on the orthogonal X-Y plane. This 3-D waterfall plot has been most commonly implemented in R, using a variety of 3-D projection libraries. We created a set of macros to dynamically normalize data to a unit cube, transform 3-D coordinates to a customizable 2-D projection, and used these projected coordinates with vector and polygon statements in PROC SGPLOT to construct a 3-D waterfall plot.

INTRODUCTION

Waterfall plots and swimmer's plots are almost ubiquitous for displaying the results of oncology trials. A waterfall plot (Figure 1) presents continuous data on maximal tumor shrinkage on study, while a swimmer's plot (Figure 2) places response milestones on patient-level timelines. Figure 3 is a 3-D Waterfall plot, proposed in a JCO article (Castanon Alvarez, Aspeslagh, & Soria, 2017), which combines some aspects of a swimmer's plot with a traditional waterfall plot to add information about the durability of a response to the magnitude of response. Because the 3-D Waterfall plot consists of two 2-D plots presented orthogonally to one another and is not truly 3 dimensional, use of PROC 3D is not recommended. We have created a SAS program based on multiple macro calls that constructs a clean 3-D Waterfall plot. A basic understanding of SAS macro coding and PROC SGPLOT is necessary to follow along, while a deeper understanding of matrix algebra is helpful to fully understand how the 3-D to 2-D projection works. All figures present randomly generated data with an assumed 75% correlation between time on study and maximum percent change in tumor size.

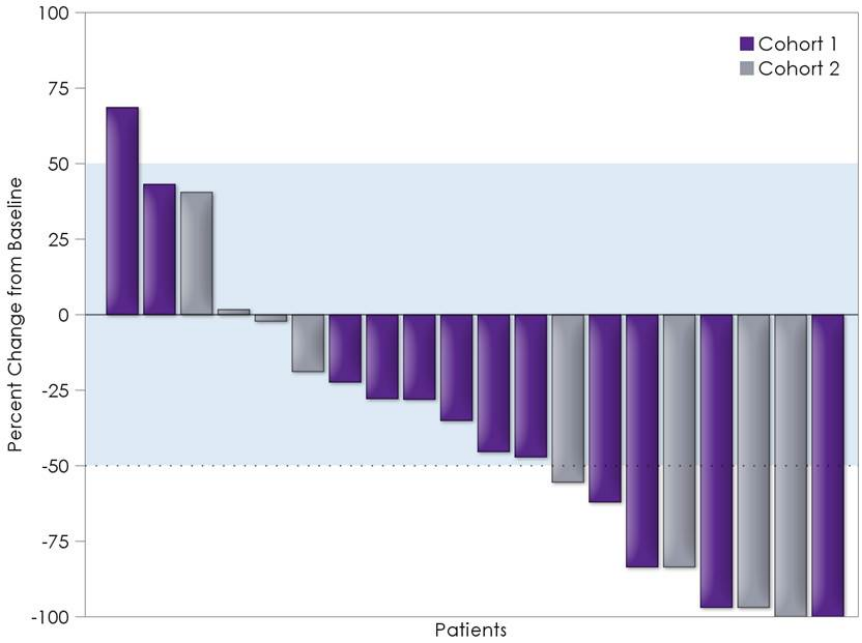

Figure 1. Classical Waterfall Plot

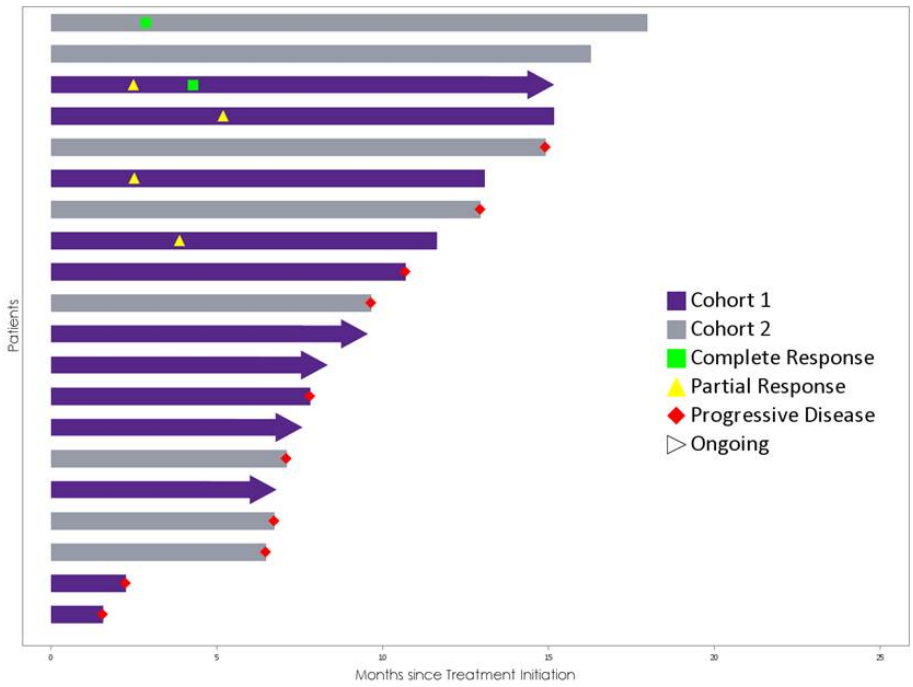

Figure 2. Swimmer's Plot

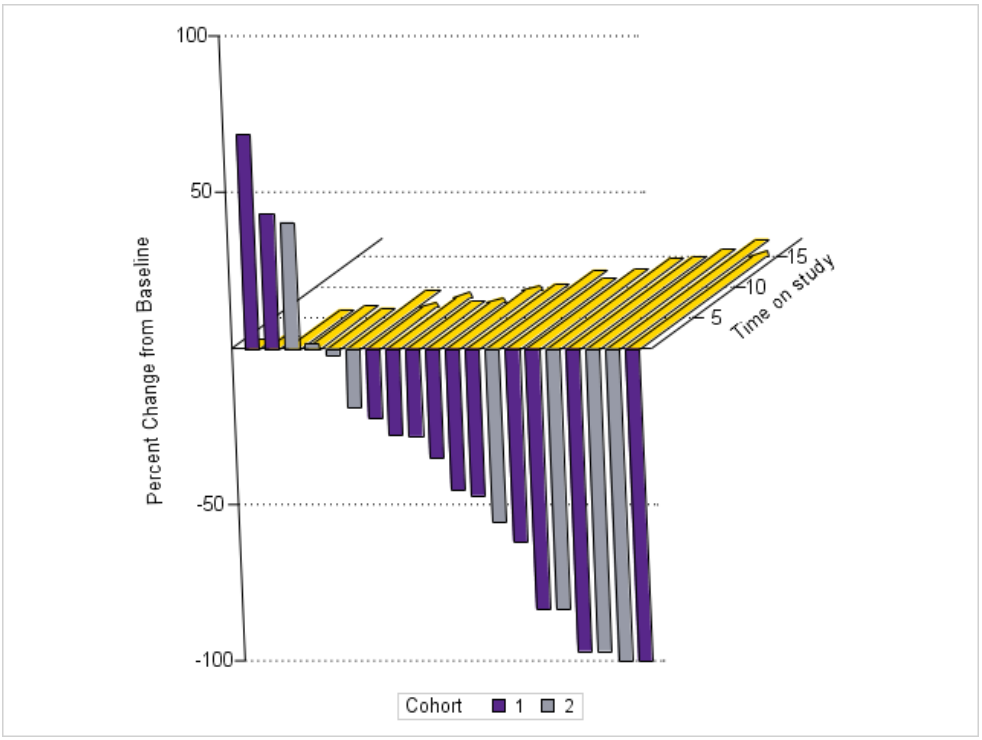

Figure 3. 3-D Waterfall Plot (Orthographic projection)

The main steps to construct the 3-D waterfall plot are:

1. Sort data
2. Construct plot elements in 3-D coordinates
3. Normalize to unit cube
4. Project to 2-D coordinates
5. Plot from background to foreground

STEP 1: SORT THE DATA

Waterfall plots are arranged in descending order of tumor size change. Consolidate all relevant information, which consists at minimum of: Subject ID, tumor size change, time on study, and indicator of whether a subject is still under observation. Sort this data in descending order of the tumor size change variable, and add a new numeric subject ID assigned to the automatic variable _n_ to preserve this sort order.

STEP 2: CONSTRUCT PLOT DATASETS IN 3-D COORDINATES

There are three basic plot elements needed to construct the 3-D waterfall plot:

- Polygons (flat bars and arrows)
- Lines (axes, tickmarks, reference lines)
- Text (labels)

Start from a dataset with integer-valued subject IDs, continuous time on study, and continuous percent change from baseline, as well as a flag of whether the subject is continuing on study.

Determine plot limits: X-axis runs from 0 to the highest subject number; Y-axis runs from 0 to maximum time on study; Z-axis runs from -100 to a positive integer, usually 40 or 100, depending on response criteria and preference.

Polygon data is built using the X, Y, and Z variables, along with the indicator of continuing time. If the patient is still on study, an arrowhead will extend past the observed time. If the percent change in tumor burden is beyond the predefined plot area limits, an arrowhead will extend past the limit. Figure 4 illustrates how the coordinates for arrow and bars are parameterized.

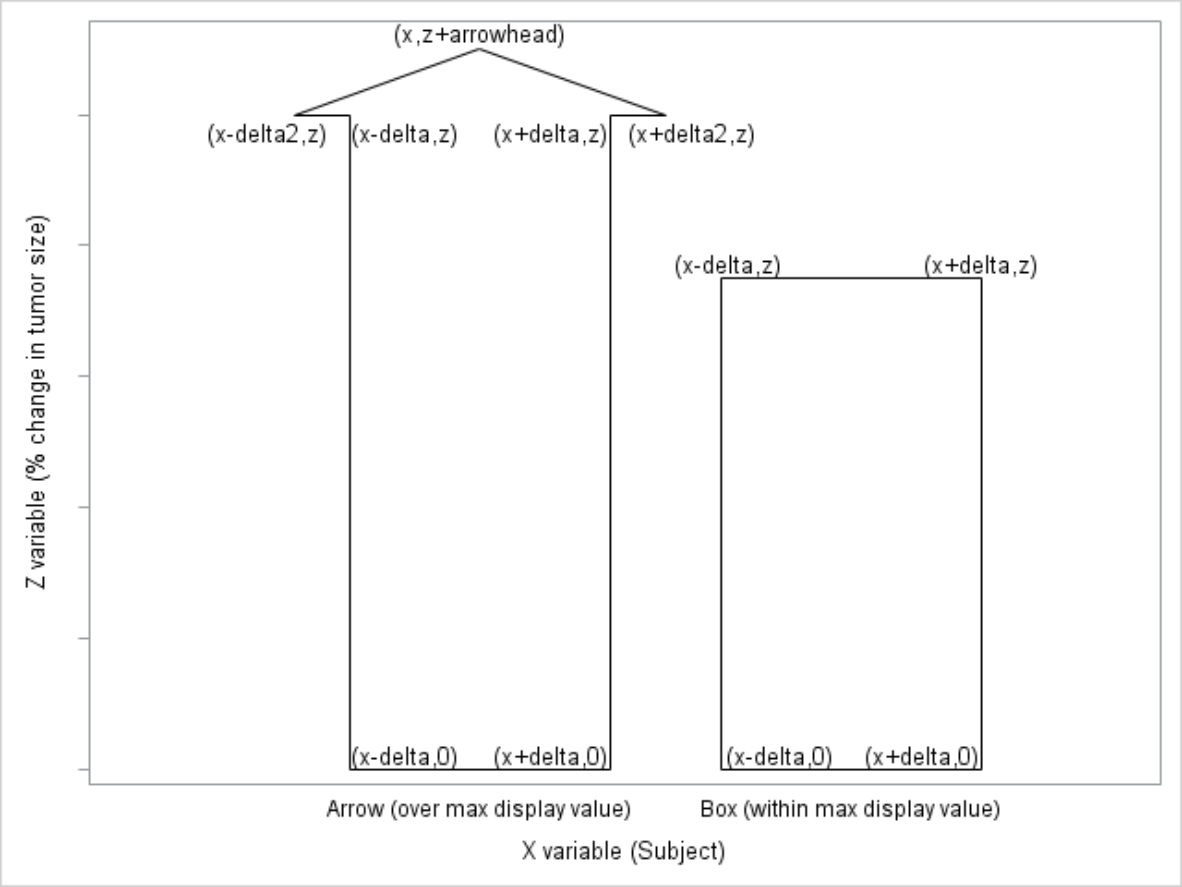

Figure 4. Arrow and Bar parameterization

Polygon data to construct the bar and arrows in the X-Y and X-Z plane uses some parameterization (delta is half the width of a bar (ranging from 0 to 0.5), wide represents the relative width of the arrowhead (1=take all the space, 0=same width as bar), yarrowmult and zarrowmult multiplicative factors representing the length of the arrowhead relative to ymax and zmax, respectively. The original X, Y, and Z variables are passed as macro variables defined outside of the data step:

```
data polydata(drop=delta2);
    set mydat;
    length grpid delta2 8;
    delta2=&wide.*(0.5) + (1-&wide.)*&delta.;
    grpid=1;
    x=&x.-&delta; y=0; z=0; output;
    y=&y.; output;
    if cont=1 then do;
          x=&x.-delta2; output;
          x=&x.; y=&y.+&yarrowmult.*&ymax.; output;
          x=&x.+delta2; y=&y.; output;
    end;
    x=&x.+&delta; output;
    y=0; output;
    x=&x.-&delta; output;
    grpid=2;
    x=&x.-&delta; y=0; z=0; output;
    z=min(&z.,&zmax.); output;
    if &z.>&zmax. then do;
          x=&x.-delta2; output;
```

```
            x=&x.; z=(1+&zarrowmult.)*&zmax.; output;
            x=&x.+delta2; z=&zmax.; output;
        end;
        x=&x.+&delta; output;
        z=0; output;
        x=&x.-&delta; output;
    run;
```

Axis data has a row for each starting coordinate and a row for each ending coordinate, along with an ID variable that matches up the start and end of each line:

```
    data axes;
        length idlab x y z 8 type $5;
        type="start"; idlab=1; x=&xmin.; y=0; z=0; output;
        idlab=2; x=0; y=&ymin.; z=0; output;
        idlab=3; x=0; y=0; z=&zmin.; output;
        idlab=4; x=&xmax.; y=0; z=0; output;
        type="end"; idlab=1; x=&xmax.; y=0; z=0; output;
        idlab=2; x=0; y=&ymax.; z=0; output;
        idlab=3; x=0; y=0; z=&zmax.; output;
        idlab=4; x=&xmax.; y=&ymax.; z=0; output;
    run;
```

In this example, idlab=1 constructs the x-axis, idlab=2 constructs the y-axis, idlab=3 constructs the z-axis, and idlab=4 adds a line parallel to the x-axis to the back wall.

Tickmark data can be constructed in a similar fashion.

STEP 3: RESCALE PLOT DATA TO UNIT CUBE

You can rescale data with or without recalculating the plot limits. This code constructs a range for each x, y, and z axis, and uses that range along with the minimum and maximum values to map the data so that the minimum equates to -0.99 and the maximum equates to 0.99. This allows a small gutter in each direction to give more space for annotations.

- &x-&xmin is the distance of x from the minimum.
- Dividing that by the range normalizes the xvalue to [0,1].
- Multiplying by 2 normalizes to [0,2]
- Subtracting 1 normalizes to [-1,1]
- Multiplying by 0.99 normalizes to [-0.99,0.99]

The macro code that normalizes the data is below:

```
    %macro normalize(dataset=, x=x, y=y, z=z);
    data &dataset._n(drop=xrange yrange zrange);
        set &dataset;
        xrange=&xmax-&xmin;
        yrange=&ymax-&ymin;
        zrange=&zmax-&zmin;
        x=0.99*(2*(&x-&xmin)/xrange-1);
        y=0.99*(2*(&y-&ymin)/yrange-1);
        z=0.99*(2*(&z-&zmin)/zrange-1);
    run;
    %mend;
```

STEP 4: PROJECT TO 2-D COORDINATES

Projection from 3-D to 2-D coordinates is a moderately complex problem. A thorough explanation for the interested reader is available (House, 2014).

The code approaches the projection process in several steps:

- Define rotation matrices
- Define translation matrix
- Define projection matrix
- Multiply matrices together to get a single 'world' matrix
- Set original data and express in 4-dimensional homogenous coordinates
- Apply the world matrix to the re-parameterized data
- Divide the first and second parameter of the resulting vector by the fourth parameter to obtain projected x and y coordinates, respectively.

ROTATION MATRICES

Rotation matrices twist the data around one of the three axes into a new set of coordinates. This allows the user to modify the angle of the view so that all data can be seen, as demonstrated in Figure 5:

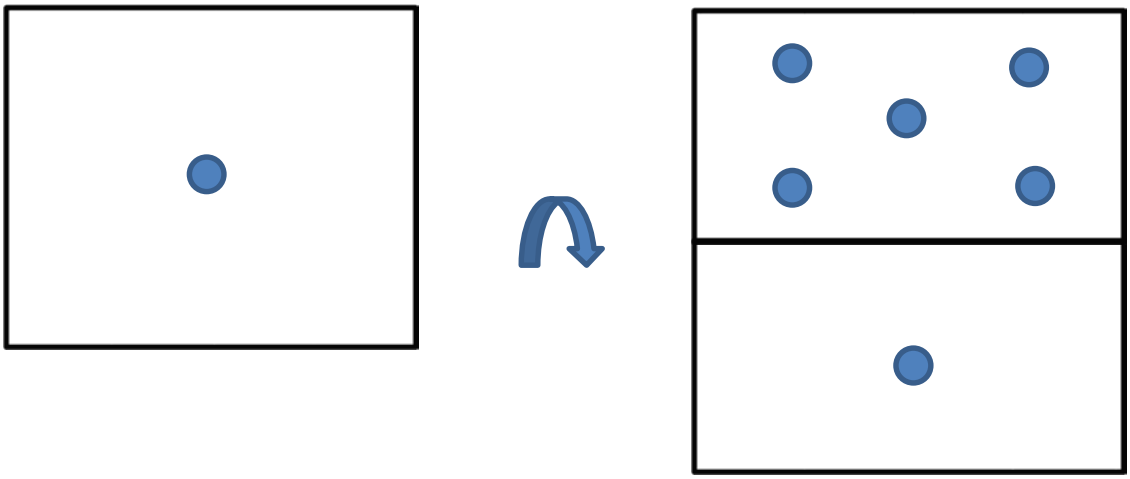

Figure 5. Rotating a die around the X-axis

Once you define a rotation around an axis in degrees (macro variables &rotx, &roty, and &rotz), the code transforms degrees into radians, and constructs an appropriate rotation matrix:

```
pi=constant("PI");
fac=pi/180;
A=&rotx.*fac;
/*--Set up X rotation matrix--*/
rx[1,1]=1;     rx[1,2]=0.0;     rx[1,3]=0.0;      rx[1,4]=0.0;
rx[2,1]=0.0;   rx[2,2]=cos(A);  rx[2,3]=-sin(A);  rx[2,4]=0.0;
rx[3,1]=0.0;   rx[3,2]=sin(A);  rx[3,3]=cos(A);   rx[3,4]=0.0;
rx[4,1]=0.0;   rx[4,2]=0.0;     rx[4,3]=0.0;      rx[4,4]=1.0;
```

TRANSLATION MATRIX

Once all the rotations are done, the data is still centered at the origin, and may have both positive and negative Z-values. The origin in the step prior to the projection represents the location of the viewer's eye, so you will translate the data away from the origin by at least two units in the Z direction. This allows the full viewing area to be seen, as illustrated in Figure 6:

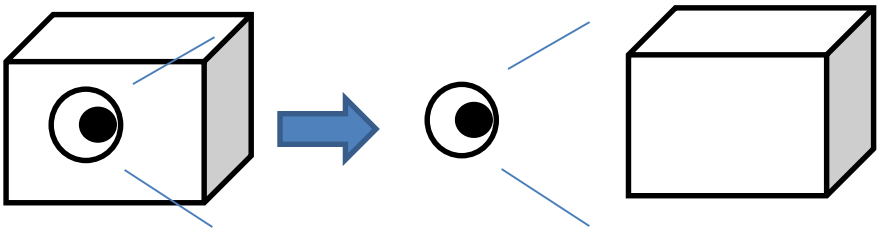

Figure 6. Viewpoints (inside and outside viewing area)

```
/*--Set up translation matrix--*/
tr[1,1]=1.0;   tr[1,2]=0.0;  tr[1,3]=0.0;      tr[1,4]=0.0;
tr[2,1]=0.0;   tr[2,2]=1.0;  tr[2,3]=0.0;      tr[2,4]=0.0;
tr[3,1]=0.0;   tr[3,2]=0.0;  tr[3,3]=1.0;      tr[3,4]=-2.0;
tr[4,1]=0.0;   tr[4,2]=0.0;  tr[4,3]=0.0;      tr[4,4]=1.0;
```

PROJECTION MATRIX

There are two commonly-used projection transformations: orthographic projection and perspective projection.

Orthographic projection preserves parallel lines, but can be misleading in depth.

Perspective projection does not preserve parallel lines, and gives a more natural expression of depth.

The textbook definitions of the orthographic and perspective transformation matrices in homogenous coordinates are:

$$P_{ortho} := \begin{pmatrix} \frac{1}{r} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}, P_{persp} := \begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

The parameters for these matrices are as follows:

- r: ½ the view plane width, which should be 1 since we normalized to the unit cube.
- t: ½ the view plane height, which is also 1.
- n is the location of the near plane (in Z-coordinates), which is set to -1
- f is the location of the far plane, which is set at -3

WORLD MATRIX

Sanjay Matange published several matrix multiplication functions to support the 3-D Scatter Plot Macro published on Graphically Speaking in 2015 (Matange, A 3D Scatter Plot Macro, 2015). Development of our code for the 3-D Waterfall Plot started as a modification of this published macro, and still uses the

MatMult function defined here:
https://blogs.sas.com/content/graphicallyspeaking/files/2015/03/Matrix_Functions.txt.

```
/*--Build transformation matrix--*/
call MatMult(ry, rx, u);  *Rotate in X direction first, then Y direction;
call MatMult(rz, u, uu);  *Rotate in Z direction;
call MatMult(tr, uu, v);  *Translate away from the Z-origin so viewpoint is
                           not within the field;
call MatMult(m, v, w);    *Apply the projection matrix;
```

ORIGINAL DATA IN HOMOGENOUS COORDINATES

A 3-D vector can be represented in homogenous coordinates by appending a parameter with the addition of a scale parameter w:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} xw \\ yw \\ zw \\ w \end{pmatrix}$$

The simplest assignment is w=1.

TRANSFORM DATA

The code below assigns the data, creates a temporary vector d representing the data in homogenous coordinates, multiplies by the world matrix into another temporary vector p to complete rotation, translation, and projection, then divides the resultant xw and xy coordinates of the vector by the scale parameter w:

```
set &ds.;
/*--Transform data--*/
d[1,1]=&x; d[2,1]=&y; d[3,1]=&z; d[4,1]=1;
call MatMult(w, d, p);
x2=p[1,1]/p[4,1]; y2=p[2,1]/p[4,1];
```

FULL PROJECTION MACRO CODE

```
/* Projection macro -> translates &x, &y, &z into coordinates x2 and y2 */
%macro project(ds=, x=x, y=y, z=z, rotx=100, roty=198, rotz=180);
data &ds._p;
  array u[4,4] _temporary_;  /*--Intermediate Matrix--*/
  array v[4,4] _temporary_;  /*--Intermediate Matrix--*/
  array uu[4,4] _temporary_; /*--Intermediate Matrix--*/
  array w[4,4] _temporary_;  /*--Final Transformation Matrix--*/
  array m[4,4] _temporary_;  /*--Projection Matrix--*/
  array rx[4,4] _temporary_; /*--X rotation Matrix--*/
  array ry[4,4] _temporary_; /*--Y rotation Matrix--*/
  array rz[4,4] _temporary_; /*--Z rotation Matrix--*/
  array tr[4,4] _temporary_; /*--Translation Matrix--*/
  array d[4,1] _temporary_;  /*--World Data Array --*/
  array p[4,1] _temporary_;  /*--Projected Data Array --*/
  retain r t f n;
  r=1; t=1; f=-3; n=-1;
  pi=constant("PI");
  fac=pi/180;
  A=&rotx.*fac; B=&roty.*fac; C=&rotz.*fac;

  /*--Set up orthographic projection matrix--*/
  m[1,1]=1/r;   m[1,2]=0.0;  m[1,3]=0.0;      m[1,4]=0.0;
  m[2,1]=0.0;   m[2,2]=1/t;  m[2,3]=0.0;      m[2,4]=0.0;
```

```
  m[3,1]=0.0;   m[3,2]=0.0;  m[3,3]=-2/(f-n); m[3,4]=-(f+n)/(f-n);
  m[4,1]=0.0;   m[4,2]=0.0;  m[4,3]=0.0;      m[4,4]=1.0;

  /*--Set up translation matrix--*/
  tr[1,1]=1.0;   tr[1,2]=0.0;  tr[1,3]=0.0;      tr[1,4]=0.0;
  tr[2,1]=0.0;   tr[2,2]=1.0;  tr[2,3]=0.0;      tr[2,4]=0.0;
  tr[3,1]=0.0;   tr[3,2]=0.0;  tr[3,3]=1.0;      tr[3,4]=-2.0;
  tr[4,1]=0.0;   tr[4,2]=0.0;  tr[4,3]=0.0;      tr[4,4]=1.0;

  /*--Set up X rotation matrix--*/
  rx[1,1]=1;     rx[1,2]=0.0;     rx[1,3]=0.0;      rx[1,4]=0.0;
  rx[2,1]=0.0;   rx[2,2]=cos(A);  rx[2,3]=-sin(A);  rx[2,4]=0.0;
  rx[3,1]=0.0;   rx[3,2]=sin(A);  rx[3,3]=cos(A);   rx[3,4]=0.0;
  rx[4,1]=0.0;   rx[4,2]=0.0;     rx[4,3]=0.0;      rx[4,4]=1.0;

  /*--Set up Y rotation matrix--*/
  ry[1,1]=cos(B);  ry[1,2]=0.0;  ry[1,3]=sin(B);  ry[1,4]=0.0;
  ry[2,1]=0.0;     ry[2,2]=1.0;  ry[2,3]=0.0;     ry[2,4]=0.0;
  ry[3,1]=-sin(B); ry[3,2]=0.0;  ry[3,3]=cos(B);  ry[3,4]=0.0;
  ry[4,1]=0.0;     ry[4,2]=0.0;  ry[4,3]=0.0;     ry[4,4]=1.0;

  /*--Set up Z rotation matrix--*/
  rz[1,1]=cos(C);  rz[1,2]=-sin(C); rz[1,3]=0.0;  rz[1,4]=0.0;
  rz[2,1]=sin(C);  rz[2,2]=cos(C);  rz[2,3]=0.0;  rz[2,4]=0.0;
  rz[3,1]=0.0;     rz[3,2]=0.0;     rz[3,3]=1.0;  rz[3,4]=0.0;
  rz[4,1]=0.0;     rz[4,2]=0.0;     rz[4,3]=0.0;  rz[4,4]=1.0;

  /*--Build transformation matrix--*/
  call MatMult(ry, rx, u);  *Rotate in X direction first, then Y direction;
  call MatMult(rz, u, uu);  *Rotate in Z direction;
  call MatMult(tr, uu, v);  *Translate away from the Z-origin so viewpoint
                             is not within the field;
  call MatMult(m, v, w);    *Apply the projection matrix;

  set &ds.;

  /*--Transform data--*/
  d[1,1]=&x; d[2,1]=&y; d[3,1]=&z; d[4,1]=1;
  call MatMult(w, d, p);
  x2=p[1,1]/p[4,1]; y2=p[2,1]/p[4,1];
run;
%mend;
```

STEP 5: PLOT BACKGROUND TO FOREGROUND

You will start by reshaping the line data (axes, tickmarks) from long data (with one row for start and another row for end, and a key variable defined in Step 2) to wide data, with xs, ys as the projected coordinates of the start and xe, ye as the projected coordinates of the line end. If you want to plot different line styles (e.g. gridlines), either create a variable with the line style or use distinct start and end coordinate names. Append to this dataset the polygon data, split into the foreground waterfall plot polygons using coordinates xz, yz and the swimmer's plot polygons in the background, using coordinates xy and yy.

First we plot the solid lines, then dotted lines, then the filled swimmer's plot polygon, the outline of the swimmer's plot polygon, the filled waterfall polygons, and finally the outline of the waterfall polygon:

```
proc sgplot data=combined nowall noborder aspect=1 noautolegend
sganno=anno;
    vector x=xe y=ye / xorigin=xs yorigin=ys noarrowheads
     lineattrs=(color=black) attrid=AxisTick;
    vector x=xe2 y=ye2 / xorigin=xs2 yorigin=ys2 noarrowheads
     lineattrs=(pattern=dot color=black) attrid=RefLine;
    polygon id=&x. x=xy y=yy / fill fillattrs=(color=gold);
    polygon id=&x. x=xy y=yy / lineattrs=(color=black pattern=solid);
    polygon id=&x. x=xz y=yz / fill group=&group. name="dose";
    polygon id=&x. x=xz y=yz / lineattrs=(color=black pattern=solid);
    keylegend "dose" / title="Cohort";
    xaxis display=none;
    yaxis display=none;
run;
```

CONCLUSION

There is an ongoing discussion about whether the 3-D Waterfall Plot is an appropriate way to display oncology efficacy and exposure data together. Despite the valid criticisms of this data presentation style, it has been increasingly requested by clinicians and investigators. This paper walks through one way to construct this plot using SAS, but it is involved and requires a lot of intermediate steps. At the time of the writing of this paper, we learned that Sanjay Matange is developing a macro using orthographic projection to create a similar plot (Matange, A 3D waterfall chart, 2018). We look forward to seeing whether 3-D Waterfall Plots are adopted and welcome your thoughts and opinions.

REFERENCES

Castanon Alvarez, E., Aspeslagh, S., & Soria, J.-C. (2017). 3D waterfall plots: a better graphical representation of tumor response in oncology. *Annals of Oncology*, 454-456.

House, D. (2014). *Orthographic and Perspective Projection*. Retrieved from CPSC 405 Course Notes: https://people.cs.clemson.edu/~dhouse/courses/405/notes/projections.pdf

Matange, S. (2015). *A 3D Scatter Plot Macro*. Retrieved from Graphically Speaking: https://blogs.sas.com/content/graphicallyspeaking/2015/03/10/a-3d-scatter-plot-macro/

Matange, S. (2018). *A 3D waterfall chart*. Retrieved from Graphically Speaking: https://blogs.sas.com/content/graphicallyspeaking/2018/01/11/3d-waterfall-chart/

CONTACT INFORMATION

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

Liz Thomas
ethomas@epizyme.com