Implementing Stereographic Projection in Sage

MATH 480
Simon Spicer
Jeff Beorse
Kevin Lindeman

June 2, 2010

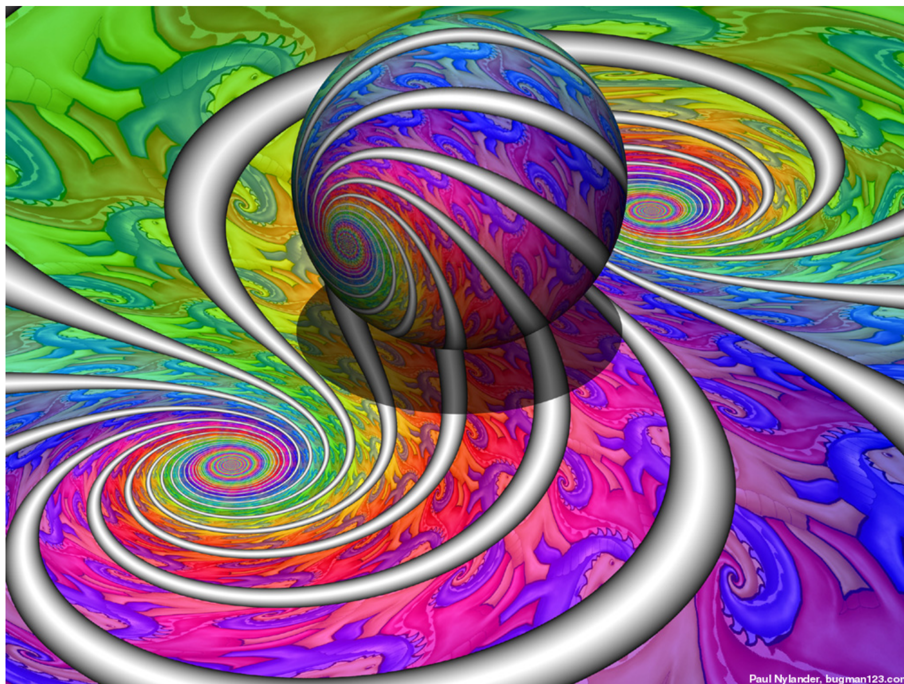

Figure 1: Woooo, trippy.

Abstract

The stereographic projection is a 1-1 mapping from the plane to the unit sphere and back again which has the special property of being *conformal*, or angle preserving. Because of this fact the projection is of interest to cartographers and mathematicians alike. In this project we explore implementing the stereographic projection in Sage, and discuss the challenges that arise in attempting to do so.

1 Introduction and Overview

This project arose from the following question: what would an elliptic curve over the reals look like if we wrapped it onto the sphere? In more generality, how would one go about projecting an arbitrary curve or image onto the sphere in a systematic way, and what do we get if we roll the sphere around a bit and then project back down?

The result is an interesting mix of mathematics and programming. Central to the project is the *stereographic projection*, a particularly well-behaved 1-1 mapping from the extended plane onto the unit sphere and back again.

Using this projection, an interact in Sage was created that takes an arbitrary user-defined symbolic real function or elliptic curve and wraps it onto the unit sphere. The user can then manipulate rotating the sphere about various axes and apply the inverse transformation, obtaining a mapping of the original curve back onto the plane.

This writeup gives some background to the mathematics that went into creating such code, as well as detailing the implementation itself and the challenges that came up in the process. Lastly, we discuss how this work could be extended and incorporated into Sage.

2 Theoretical Background

2.1 The Stereographic Projection

For the context of the project we will be working extensively with the following two spaces:

- The *extended real plane* $\mathbb{R}^2 \cup \{\infty\}$, where infinity is treated as a single point that can get mapped to (for example, 0 gets mapped to ∞ by the function $\frac{1}{(x^2+y^2)}$);
- The *unit sphere* $S = \{(x, y, z) \in \mathbb{R}^3 \quad s.t. \quad x^2 + y^2 + z^2 = 1\}$, the sphere of radius 1 centered at the origin.

Mathematically, the stereographic projection is a bijection between the extended plane and the unit sphere; that is, every point on the plane (including infinity) corresponds to a point on the sphere and vice versa. If (X, Y) is a point on the plane and (x, y, z) a point on the sphere, the bijection is given by

$$
\begin{aligned}
(x, y, z) &= \left(\frac{2X}{X^2+Y^2+1}, \frac{2Y}{X^2+Y^2+1}, \frac{X^2+Y^2-1}{X^2+Y^2+1} \right) \\
(X, Y) &= \left(\frac{x}{1-z}, \frac{y}{1-z} \right)
\end{aligned}
$$

We shall define P to be the projection from the plane onto the sphere, and P^{-1} to be the inverse projection from the sphere back onto the plane (note that many other sources, Wikipedia included, define it the other way round). That is,

$$
\begin{aligned}
P((X, Y)) &= \left(\frac{2X}{X^2+Y^2+1}, \frac{2Y}{X^2+Y^2+1}, \frac{X^2+Y^2-1}{X^2+Y^2+1} \right) \\
P^{-1}((x, y, z)) &= \left(\frac{x}{1-z}, \frac{y}{1-z} \right)
\end{aligned}
$$

A little bit of arithmetic legwork will show that this map is indeed a bijection i.e. $P^{-1} \circ P((X, Y)) = (X, Y)$ and $P \circ P^{-1}((x, y, z)) = (x, y, z)$.

We can get a better intuitive sense of the projection as follows: Consider placing the unit sphere at the origin in 3-d space, so that the xy-plane slices the sphere in two. Pick a point (X, Y) on the plane, and draw a straight line through that point and the north pole on the sphere. The point on the sphere corresponding to (X, Y) is the point where the line intersects the sphere.

Again, a little checking shows that this correspondence is mathematically equivalent to the stereographic projection P defined above.

Note that as one gets further and further away from the origin on the plane, the closer the corresponding point on the sphere gets to the north pole. However, the north pole itself is

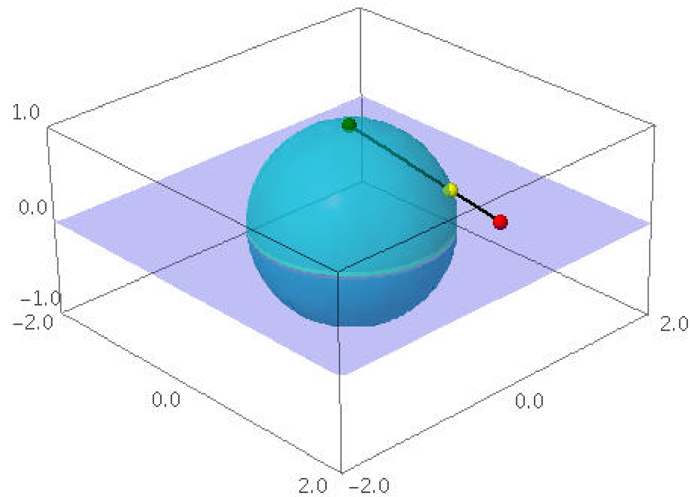

Figure 2: An example of the stereographic projection in action. The yellow point $\left(\frac{2}{3}, \frac{2}{3}, \frac{1}{3}\right)$ on the sphere corresponds to the red point $(2, 2, 0)$ on the plane (the green point is the north pole of the sphere).

not mapped to given either of the definitions above. To fix this we say that the north pole $(0, 0, 1)$ corresponds to ∞, our special point at infinity on the extended plane; this patch actually makes sense mathematically from a number of perspectives.

Applying P as defined above allows us to wrap any two-dimensional image onto the sphere. Clearly some distortion will be involved, but the stereographic projection has one important quality: it is *conformal* both ways, or angle preserving. That is, if two lines intersect at angle θ on the plane, then their images under P on the sphere will also intersect at angle θ, and vice versa.
The fact that the stereographic projection is conformal makes it of great use to cartographers, as it provides a way to project portions of the sphere down onto the plane in such a way that directional information is preserved. It thus one of the standard projections used to depict the earth - a sphere - on flat bits of paper.

2.2 Linear Fractional Transformations

A natural question arises when playing with the stereographic projection: what if I project up onto the sphere, toss the sphere around a bit, and then project back down? More formally, what sort of map is $P^{-1}TP$, where T is some affine transformation of the sphere (i.e. rotation, translation and dilation)?

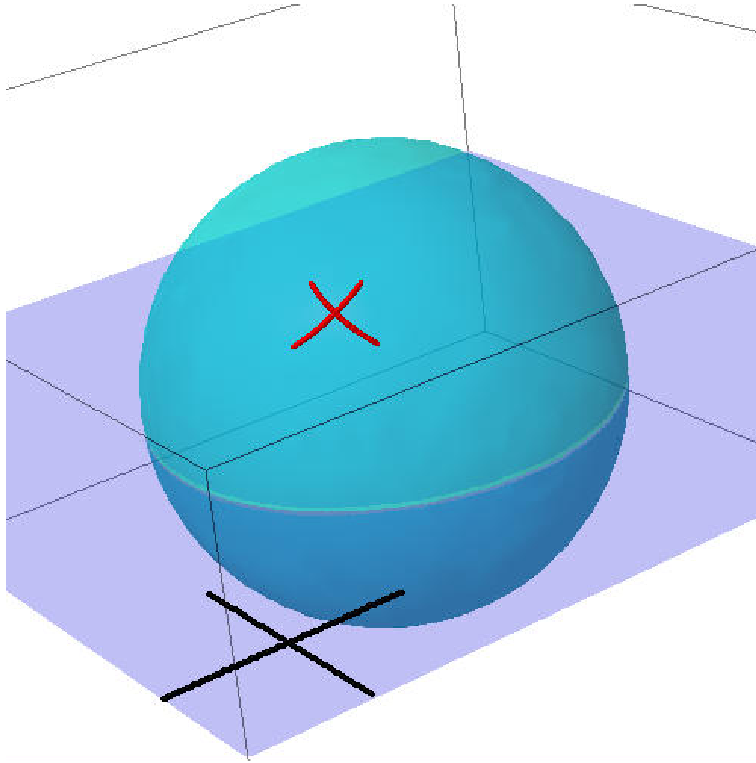

Figure 3: Two lines which intersect at right angles on the plane will still intersect at right angles under the stereographic projection.

The result is something called a *linear fractional transformation* (LFT) of the plane back onto itself. To better understand LFT's, we must venture into the realm of complex numbers $\mathbb{C}$.

Observe that any point (X, Y) in $(R)^2 \cup \{\infty\}$ can be identified with a point Z in the extended complex plane by $Z = X + iY$. Doing so, we can give an equivalent definition of the stereographic projection: one of a bijection between $\mathbb{C} \cup \{\infty\}$ and S given by

$$\begin{aligned} P(Z) &= \left(\frac{Z + \bar{Z}}{Z\bar{Z} + 1}, \frac{-i(Z - \bar{Z})}{Z\bar{Z} + 1}, \frac{Z\bar{Z} - 1}{Z\bar{Z} + 1} \right) \\ P^{-1}((x, y, z)) &= \left(\frac{x}{1 - z} \right) + i \left(\frac{y}{1 - z} \right), \end{aligned}$$

where $\bar{Z} = \overline{X + iY} = X - iY$.

Writing the stereographic projection thus allows us to give a very elegant formulation of the mapping $P^{-1} \circ T \circ P$ from the extended complex plane onto itself. It turns out that any such mapping is given by

$$P^{-1} \circ T \circ P(Z) = f(Z) = \frac{aZ + b}{cZ + d}$$

for some complex constants a, b, c and d. Maps of the form $\frac{aZ+b}{cZ+d}$ are known as *linear fractional transformations* or *Moëbius transformations*, and a rich theory surrounds them in complex analysis.
The question remains as to what the constants a, b, c and d are, given a certain affine translation of the sphere. For this purpose it should be noted that any LFT can be written (uniquely) in the form

$$f(Z) = \left(\frac{b-c}{b-a}\right) \cdot \left(\frac{Z-a}{Z-c}\right)$$

for distinct complex constants a, b and c. Doing so is advantageous, as it is easy to check that the points a, b and c are sent to $0, 1$ and ∞ respectively.

Given a transformation T of the sphere, it is easy to determine these three constants; for example, c is just the point whose projection on the sphere is brought to the north pole. This provides us with a shortcut to evaluating $P^{-1} \circ T \circ P(Z)$ without going through the computationally intensive steps of projecting up, transforming, and projecting down again.

Lastly, note that since the composition of conformal maps is conformal, it follows that LFT's are also conformal: the image of two lines that intersect at angle θ will again intersect at angle θ. LFT's as a consequence are used in a number of mapping applications, from cartography to engineering and physics.

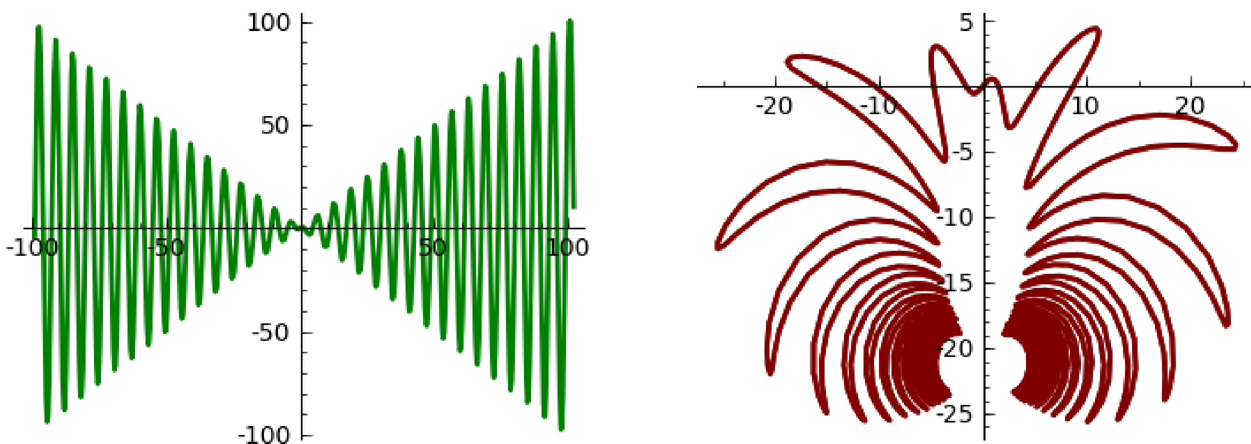

Figure 4: An example of an LFT in action. Here the graph of $y = x\cos(x)$ has been projected up onto the sphere, the sphere rotated about the x-axis by 0.03π, and then projected back down onto the plane. The point at infinity has been moved to $\left(0, \frac{-2}{0.03\pi}\right) \simeq (0, -21)$, which is why the arms of the graph on the right now converge to that point.

3 Algorithm Implementation and Development

We give a detailed breakdown below of the steps it took in order to get efficient functioning code as seen in the presentation.

3.1 Stage 1

Our first goal was to be able to wrap an explicit function `y = f(x)` onto the unit sphere. This proved relatively straightforward for explicit functions that had no disconnected parts of the graph: we were able to make the transform from 2D to 3D coordinates symbolically and pass it all off to `parametric_plot3d()`. Specifically, we transformed the function by making a list of each x and y coordinate by plugging in an x into the function, and then transforming it with `P(x,y)` above.

Now we want to visualize the 3D transformation atop a unit sphere. We were able to do this by adding two plots together, essentially taking the plot object obtained by `parametric_plot3d()` and then using the `+` operator to add a `sphere()` object.

There were some problems. A function `y` had to be explicitly defined in terms of x; we couldnt do plots of implicit functions. `y` had to also be defined for all of the range. The transformation and plot with `parametric_plot3d()` worked incorrectly if there were two parts of the plot that were disconnected. These two problems meant that we werent able to plot elliptic curves, which was one of the original goals of this project.

3.2 Stage 2

Since what we were really after was computing the projection of an elliptic curve, we needed to find some alternative to plotting using `parametric_plot3d()`. This is because elliptic curves (over the reals) can't be parameterized, and may have disconnected graph parts.

This is when we discovered the `EllipticCurve` object in Sage, and it helped immensely. We passed off the `EllipticCurve` object to the `plot()` function – we thought that maybe we could get individual points that is used to plot the regular 2D version of the curve, and transform those individually. It turns out that when plotting an `EllipticCurve`, the plot object returned is actually a list of `line` objects - if you have a plot `p`, you can get to them with `p[0]`, `p[1]` and so on. Moreover, there is a `line` object for each discontinuous chunk of the elliptic curve. Once we got the `line` object for each `EllipticCurve`, we can get the `x` and `y` data with the `.xdata` and `.ydata` properties of the line object.

With could then transform these individual points as above when we had an explicit function. Obviously, the more points we have the better the transformed plot. Because the stereographic projection has a marked effect on the spatial distribution of points on a line object, we often had to go to quite high resolution to have a sufficiently smooth transformed

plot.

3.3 Stage 3

We needed to be able to get this code to work as an interact within Sage. This required a small iteration to explicitly call `show()` on the plot objects we were creating. This was a strict requirement, though, as lots of the functionality of a stereographic projection is obtained by being able to interact with a graph. The transformations can be done by rotating the projected up curve about the 3 primary axes – implemented using a different slider for each axis with values between -pi and pi – or dilating the sphere (this feature was not implemented, since it only results in a scaling of the original graph).

The end result is projecting functionality for explicit plots and elliptic curves, but not generic implicit functions. We wanted to be able to get the points from an `implicit_plot` object just like we do for a regular plot object above for an elliptic curve. However, due to the different nature of how implicit plots are implemented in Sage – essentially, the function is evaluated on a grid over the whole range, and the points close enough to zero colored in – we have so far been unsuccessful. Asking questions on the sage mailing list has also so far proven fruitless unhelpful.

3.4 Stage 4

After implementing the full functionality of the stereographic projection we thought it would be an interesting exercise to apply this mapping to an image file, projecting the indices of individual pixels up to the sphere and back. While this does not have any direct mathematical usage, it can be helpful in visualizing the effects of rotating about the projection on the sphere about the x, y, and z axes. The user can observe the bending and stretching after a rotation of the y axis versus the uniform rotation in the xy plane after a rotation of the z axis.

To implement this functionality, the individual pixels are correlated to the xy plane with the top left corner of the image serving as the origin. This is achieved using pylab's `imread` function, which returns a two dimensional array of RGB values. The existing projection code, however, takes a pair of one dimensional arrays, one for `X` values and one for `Y` values. Therefor the image's indices are translated to this form by reading them in, row by row, into one dimensional arrays. Note the the RGB values are not used in the transformation, just their indices.

From this point the existing stereographic projection code is used to perform the mapping of the individual points. After this is complete the image must be reconstructed. A new two dimensional array is constructed with the width and height determined by the new maxima of x and y, and the entire image is shifted to eliminate any negative values. The transformation maintains the order of the points, so the RGB values were correlated to their

appropriate `(X,Y)` pair and the mapping is complete.

This achieves the original intent of performing a stereographic projection of the pixels of an image, but as the rotations in the x and y planes grow, the points become farther apart and the image quality degrades significantly. The space between pixels grows and must by filled in with white space.

We attempted to fix this problem by making squares of pixels into `Polygon` objects which would stretch as the vertices were mapped and eliminate the white space problem. However, this turned out to be very slow, and our code could only realistically manage small images (about 100x100 pixels or less). Due to time constraints we were not able to fix these issues in time for the project deadline.

4 Future Work and Summary

As mentioned in the development section above, the two natural areas where functionality could be developed is with the manipulation of arbitrary implicit plots and with transforming images. We feel that we could have made good progress with both of these given a bit more time.

It would be quite fun to have some for of dynamic functionality in the interact i.e. have the transformed image update as the sphere is manipulated by the user, instead of having the redraw occuring post-slider change. However, given the current limitations of the Sage interact, this would require a considerable amount of digging and working directly with JMOL/matplotlib elements, and thus a heavy time investment.

In terms of optimization, it would be more mathematically elegant (and faster for large `(X,Y)` arrays) to calculate the end product LFT (of the form $f(Z) = \frac{aZ+b}{cZ+d}$) first and then apply it to the data, as opposed to applying a series of transformations in succession. This would also allow us to engage in some reverse engineering: compute the transformation on the sphere needed for a given outcome on the plane.

Also mentioned was the fact that we limited the types of transformations that could be applied to the sphere to rotation about the three major axes. Extending the code to handle translation and dilation of the sphere would allow for arbitrary linear fractional transformations from the plane onto itself.

One could also extent the code to deal with more general conformal maps of the complex plane. A long-term goal might be to create code that would take an arbitrary region of the plane as input, and compute the series of comformal transformations that would map that region to, say, the unit disc.

However, given the timescale and scope of this project we feel that what has been accomplished is a good first foray into the world of the the stereographic projection and LFT's. What's more, we got to produce and play with some very pretty pictures.

APPENDIX - SAGE CODE

Below is the final version of the Sage interact code:

```
import numpy as np
import pylab
var('x y')

# Rotates a given numpy array of points on the unit sphere about one of the
# three principal axes
def rotate_about_axis(B,theta,axis=2):

    # Create the rotation matrix. Rotation about the x-axis corresponds to
    # axis==0, y-axis to axis==1 and z-axis to axis==2
    M = np.array([])
    if axis==0:
        M = np.array([[1,0,0],[0,cos(theta),-sin(theta)], \
        [0,sin(theta),cos(theta)]],dtype='float128')
    elif axis==1:
        M = np.array([[cos(theta),0,-sin(theta)],[0,1,0], \
        [sin(theta),0,cos(theta)]],dtype='float128')
    elif axis==2:
        M = np.array([[cos(theta),-sin(theta),0],[sin(theta),cos(theta),0], \
        [0,0,1]],dtype='float128')

    # Numpy makes large floating point matrix manipulations easy
    return np.dot(M,B)

# Projects a numpy array of plane points up onto the sphere
def project_up(A):
    V = 1 + A[0]^2 + A[1]^2
    return np.array([2*A[0]/V, 2*A[1]/V, (V-2)/V])

# Projects a numpy array of sphere points down onto the plane
def project_down(B):
    return np.array([B[0]/(1-B[2]), B[1]/(1-B[2])])

# Projects the numpy array of plate points up onto the sphere,
# rotates the sphere, and projects them back down to the plane
def project(A, x_axis_rot, y_axis_rot, z_axis_rot):
    H = project_up(A)
    H = rotate_about_axis(H,x_axis_rot,0)
    H = rotate_about_axis(H,y_axis_rot,1)
    H = rotate_about_axis(H,z_axis_rot,2)
```

```
    return project_down(H), H

# Extracts the x,y data from the image file and puts it into one dimensional
# arrays that can be used with the projection functions
def get_image_data(name):
    image = pylab.imread(DATA + name)
    y_max, x_max, pixSize = image.shape

    # Create the empty one dimensional arrays with length enough to
    # hold all the pixels
    x_data = np.zeros(x_max*y_max)
    y_data = np.zeros(x_max*y_max)

    # Fill in the indices of the pixels, row by row
    for i in range(y_max):
        for j in range(x_max):
            x_data[i*x_max+j] = j
            y_data[i*x_max+j] = i

    # Scale the indices down to 0 to 1 range to make the transformations
    # look better
    x_data /= x_max
    y_data /= y_max

    return [x_data, y_data, image]

# Wraps the projected x,y data back up into the pixel matrix so it
# can be displayed as a graphics object
def build_image(x_data, y_data, name):
    image = pylab.imread(DATA + name)
    old_y_max, old_x_max, pixSize = image.shape

    # Scale the indices back up to their original sizes
    x_data *= old_x_max
    y_data *= old_y_max

    # Find the maximum and minimum transforme indices so that they
    # can be shifted back towards the origin
    new_x_max = int(max(x_data))+1
    new_x_min = int(min(x_data))
    new_y_max = int(max(y_data))+1
    new_y_min = int(min(y_data))

    width = new_x_max - new_x_min
    height = new_y_max - new_y_min
```

```
    new_image = np.zeros((height, width, pixSize))

    # Iterate through all the indices, filling in the original pixels at the
    # corresponding transformed indices
    for i in range(0, old_y_max):
        for j in range(0, old_x_max):
            xInd = int(x_data[i*old_x_max+j]) - new_x_min
            yInd = int(y_data[i*old_x_max+j]) - new_y_min
            new_image[yInd][xInd] = image[i][j]

    return new_image


@interact
def plotSterographicProjection(curve=input_box(x*cos(x), 'Function'), \
        zoom=slider(1, 100, 1, 30, 'Zoom'), \
        plot_resolution=slider(1, 10000, 1, 1000, 'Plot Resolution'), \
        x_axis_rot=slider(-pi, pi, pi/100, 0, 'Rotate Sphere About x-axis'), \
        y_axis_rot=slider(-pi, pi, pi/100, 0, 'Rotate Sphere About y-axis'), \
        z_axis_rot=slider(-pi, pi, pi/100, 0, 'Rotate Sphere About z-axis'), \
        show_3d_plot=checkbox(default=False, label="Show 3d Projection")):
    """
    This interact plots a stereographic projection of an explicit function,
    elliptic curve, or image provided by the user.

    Inputs
        Function
            A string that specifies the explicit function, elliptic curve,
            or image.
            Elliptic curves and images must be wrapped in quotes
            Images must be uploaded to the worksheet before they can be used

        Zoom
            A slider that specifies the zoom level of the plot
            This input has no function for images

        Plot Resolution
            A slider that specifies the number of points to plot
            Increasing this will make the projection calculate slower,
            but improve the accuracy of the plot
            This input has no function for images

        Rotate Sphere About x-axis, y-axis, z-axis
            Sliders that specify the number of radians to rotate the
```

```
        unit sphere about the specified axis

    Show 3-d projection
        A check box that specifies whether or not to show the projection
        of the plot onto the unit sphere
        This input has no function for images

Output
    Graphics showing the original function and its
    stereographic projection. Optional 3-d graphics showing the
    original function projected onto the unit sphere

EXAMPLES
    Explicit function: x*cos(x)
    Elliptic curve: '389a'
    Image: 'lion.png'

"""

# Figure out what object we're dealing with
try:
    X, Y, image = get_image_data(curve);
    P = None
except:
    try:
        E = EllipticCurve(curve)
        P = plot(E, plot_points=plot_resolution,xmin=-zoom,xmax=zoom)
    except:
        try:
            P = plot(curve, (x,-zoom,zoom),plot_points=plot_resolution)
        except:
            print "Please provide one of the following:"
            print "\tElliptic Curve\n\tExplicit function in terms of x"
            print "\tImage file (note that zoom and plot resolution \
             do not affect images)"
            return

# R, S and T are empty graphics objects that we will fill with the
# line data for the original, transformed and 3d graphs respectively
R,S,T = Graphics(), Graphics(), Graphics()

if P == None:
    # Pull the X and Y data from the image, project it
    A = np.array([X, Y])
    B, H = project(A, x_axis_rot, y_axis_rot, z_axis_rot)
```

```
    # Reconstruct the image matrix
    new_image = build_image(B[0], B[1], curve)

    # Plot
    g1 = graphics_array([matrix_plot(image)])
    g2 = graphics_array([matrix_plot(new_image)])
    show(g1, axes=False, figsize=(8,3))
    show(g2, axes=False, figsize=(8,3))
else:

    # Iterate over the plot objects in P (elliptic curves can have
    # two branches)
    for p in P:
        # Pull the X and Y data from P, project it
        A = np.array([p.xdata,p.ydata])
        B, H = project(A, x_axis_rot, y_axis_rot, z_axis_rot)

        # Create line objects from the 3 sets of data
        R += line(zip(A[0],A[1]),rgbcolor=(0,1/2,0),thickness=2)
        S += line(zip(B[0],B[1]),rgbcolor=(1/2,0,0),thickness=2)
        if show_3d_plot: T += line(zip(H[0],H[1],H[2]), \
         rgbcolor=(1,0,0),thickness=3)

    #Plot
    G = graphics_array([R,S])
    show(G, figsize=(9,3))
    if show_3d_plot:
        T += sphere(center=(0,0,0),size=1,opacity=0.4)
        T.show(aspect_ratio=[1,1,1])
```