

# Complex Variables Toolkit - Final Project -

## Nathan Yee

### About

This notebook serves as a group of utilities (functions) that allow a user to interact with and understand complex functions. In particular it provides functions that generate exponential and sphere (inverse stereograph) spaced points, point based circle generation, point based line and grid generation, complex image mapping utilities, color creation tools, stereographic projection utilities, 3D printing tools, and finally animation generators of numerical methods such as Newton's method. As an added benefit, every function has a usage line (Mathematica equivalent of docstring), and at least one example of how it is used.

The line below is intended for internal use. It tells Mathematica to save all initialization cells to a .m file. This is the file that will be imported that you import when you want to use the toolkit.

```
In[59]:= SetOptions[EvaluationNotebook[], AutoGeneratedPackage → Automatic]
```

### Functions

Provided below are the functions, documentation, and examples.

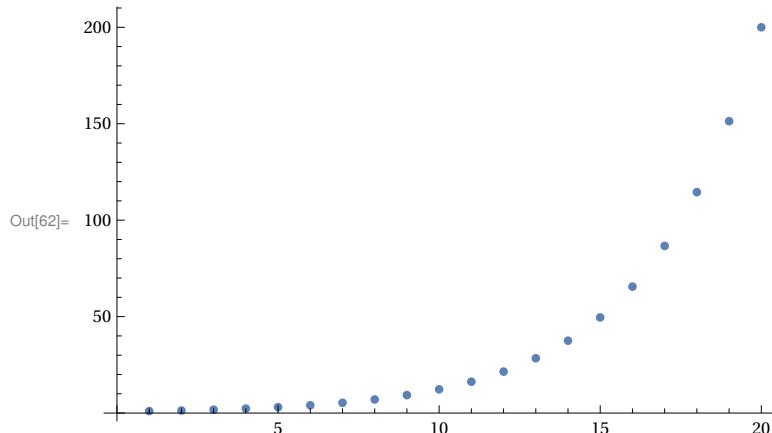
#### Exponential, and Sphere based Spacings

Various types of spacings serve as the foundation for circle, line, and grid point generator functions.

##### Inverse (default Log) spaced points

```
In[60]:= fSpace[min_, max_, numPts_, f_: Log] :=
Module[{ },
  N[InverseFunction[f] /@ Range[f@min, f@max, (f@max - f@min) / (numPts - 1)]]]
fSpace::usage =
"fSpace[min, max, steps, Log] gives inverse f (default Log) spaced
points from min to max over a given number of points";
```

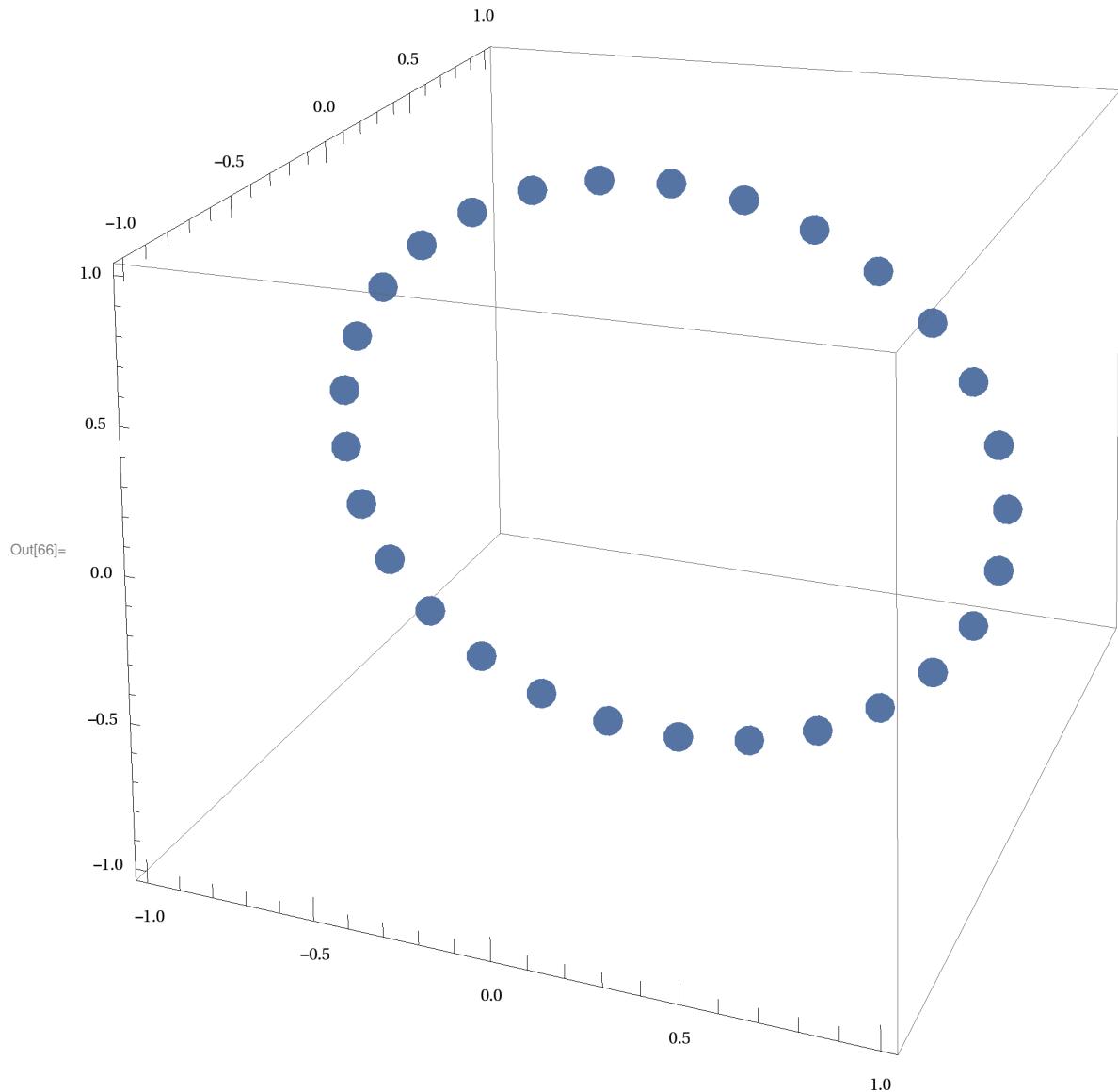
```
In[62]:= ListPlot[fSpace[1, 200, 20, Log]]
```



Sphere spaced points

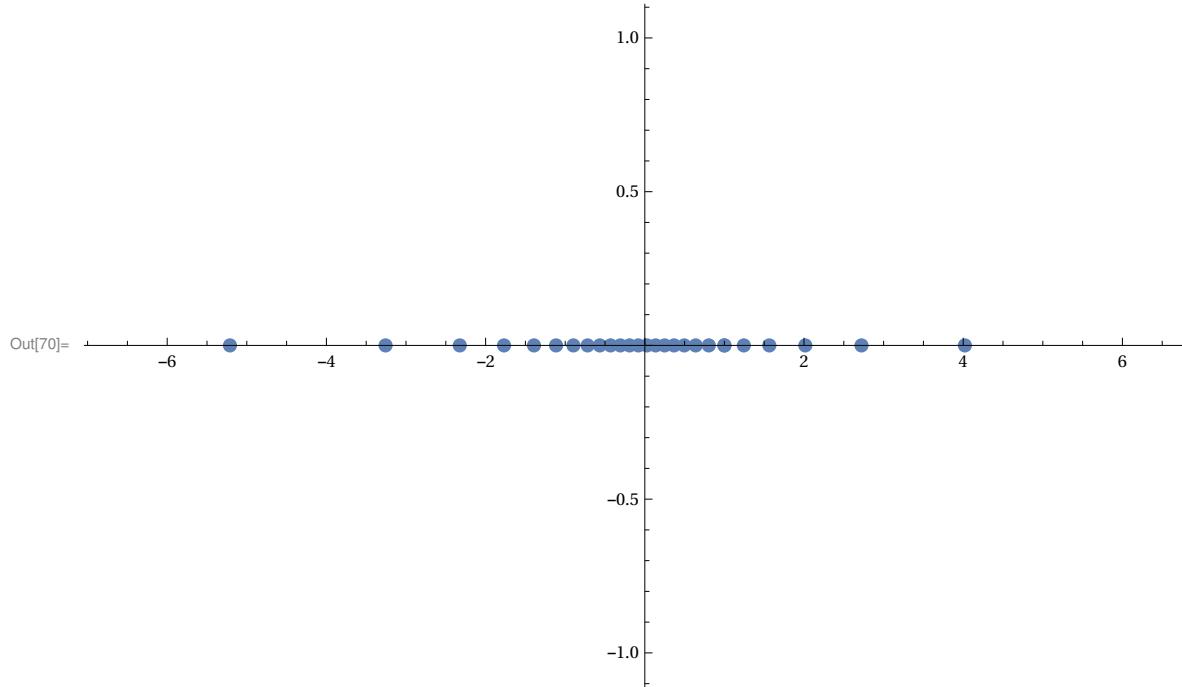
```
In[63]:= makeRiemannRealAxis[numPts_] := Module[
{angles, pts},
angles = N@Range[0 Pi, 2 Pi,  $\frac{2 \text{Pi} - 0 \text{Pi}}{\text{numPts} - 1}$ ];
pts = Table[{Cos[ang], 0, Sin[ang]}, {ang, angles}];
Return[pts]
]
makeRiemannRealAxis::usage =
"makeRiemannRealAxis[numPts_] makeRiemannRealAxis returns equally
spaced points around the positive real axis on the Riemann Sphere";
```

```
In[65]:= realAxis3D = makeRiemannRealAxis[30];
ListPointPlot3D[realAxis3D, AspectRatio -> 1, PlotStyle -> PointSize[.03],
ImageSize -> Large]
```



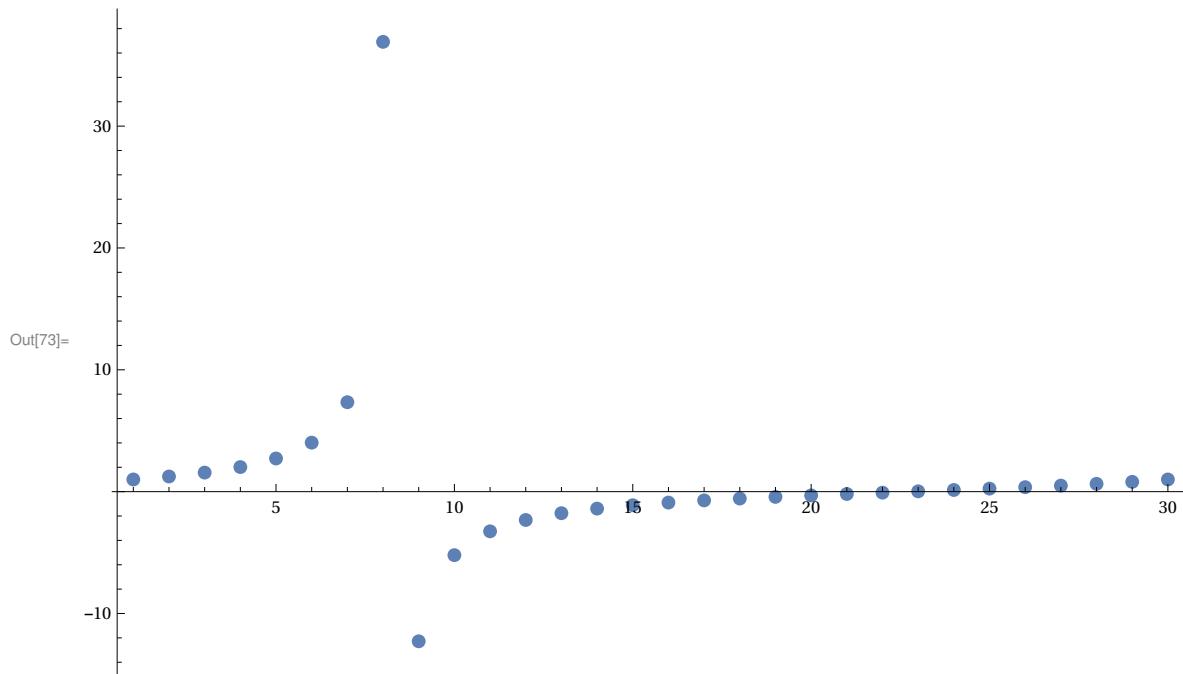
```
In[67]:= riemannPointToComplexPlane[{X_, Y_, Z_}] := Module[{sol, pair},
  sol = Solve[{x + i y == (X + i Y)/(1 - Z)}, {x ∈ Reals, y ∈ Reals}];
  pair = ({x, y} /. sol)[[1]];
  Return[pair]
]
riemannPointToComplexPlane::usage = "riemannPointToComplexPlane[{X_, Y_, Z_}]
riemannPointToComplexPlane returns the inverse stereographic projection of a point
from the Riemann sphere."
```

```
In[69]:= realAxis2D = riemannPointToComplexPlane[#] & /@ realAxis3D;
ListPlot[realAxis2D, ImageSize -> Large]
```



```
In[71]:= makeSphereSpacedPoints[numPts_] := Module[{pts, complexPts, realPts},
  pts = makeRiemannRealAxis[numPts];
  complexPts = riemannPointToComplexPlane[#] & /@ pts;
  realPts = complexPts[[All, 1]];
  Return[realPts]
]
makeSphereSpacedPoints::usage =
"makeSphereSpacedPoints[numPts_] makeSphereSpacedPoints returns
 the inverse stereographic projection of points around the
 positive real axis on the Riemann Sphere. It gets the positive
 real axis of the Riemann Sphere by calling makeRiemannRealAxis.
 It finds the inverse stereographic projection of the points
 by mapping riemannPointToComplexPlane onto each point";
```

```
In[73]:= ListPlot[makeSphereSpacedPoints[30], PlotRange -> All, ImageSize -> Large]
```



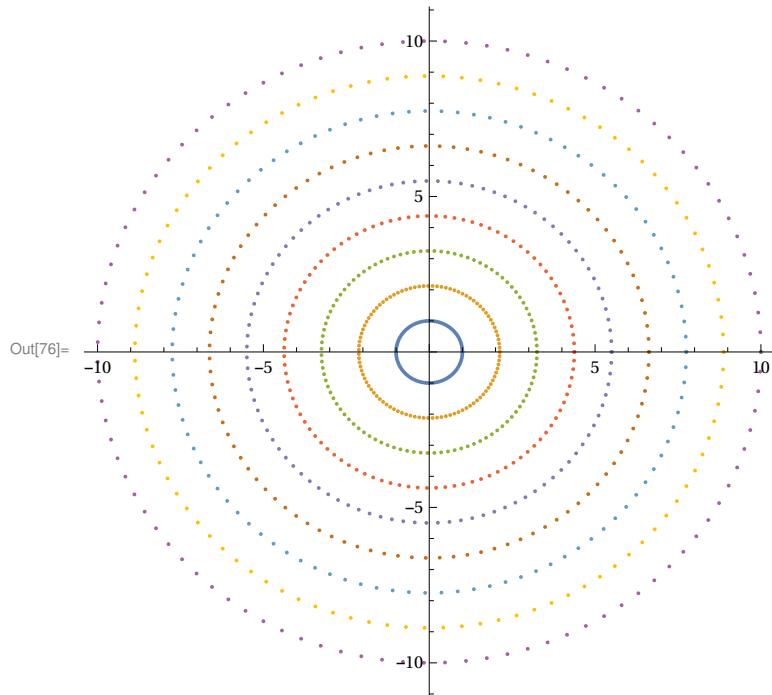
## Circle Creation

Point based circle generator functions are useful tool to visualize complex mappings.

Linear spaced circles

```
In[74]:= makeCirclePoints[smallRadius_,largeRadius_,numCircles_,ptsPerCircle_]:=Module[{ang,
ang=Range[0 Pi,2 Pi,2 \[Pi] (largeRadius-smallRadius)/numCircles];
lists=Table[{r Cos[ang],r Sin[ang]},{r,Range[smallRadius,largeRadius,(largeRadius-smallRadius)/numCircles]}];
pts=Transpose[#[]&/@lists];
Return[pts]
}
makeCirclePoints::usage="makeCirclePoints[smallRadius_,largeRadius_,numCircles_,ptsPerCircle_]=>
makeCirclePoints returns expanding circles given smallest radius, largest radius, number of circles and points per circle
The constant step size dictates that the radius of the circles increases by a constant amount each time
the angle increases by \[Pi]."
```

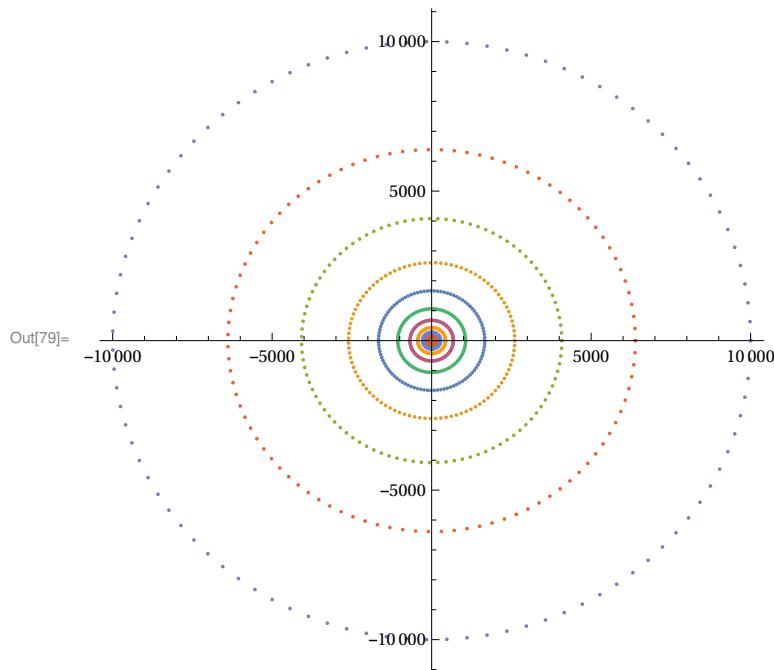
In[76]:= `ListPlot[makeCirclePoints[1, 10, 9, 100], AspectRatio -> 1]`



Exponentially spaced circles

```
In[77]:= makeExponentialSpacedCirclePoints[smallRadius_, largeRadius_, numCircles_,
  ptsPerCircle_] := Module[{ang, lists, pts},
  ang = Range[0 Pi, 2 Pi,  $\frac{2\pi}{ptsPerCircle - 1}$ ];
  lists = Table[{r Cos[ang], r Sin[ang]},
    {r, fSpace[smallRadius, largeRadius, numCircles]}];
  pts = Transpose[#[[;;]] & /@ lists];
  Return[pts]
]
makeExponentialSpacedCirclePoints::usage =
"makeExponentialSpacedCirclePoints[smallRadius_,largeRadius_,numCircles_,
  ptsPerCircle_] makeExponentialSpacedCirclePoints returns
  expanding circles given smallest radius, largest radius, number
  of circles, and number of points per circle. The number of
  circles dictates that the circles be spaced exponentially
  within the largestRadius - smallestRadius range";
```

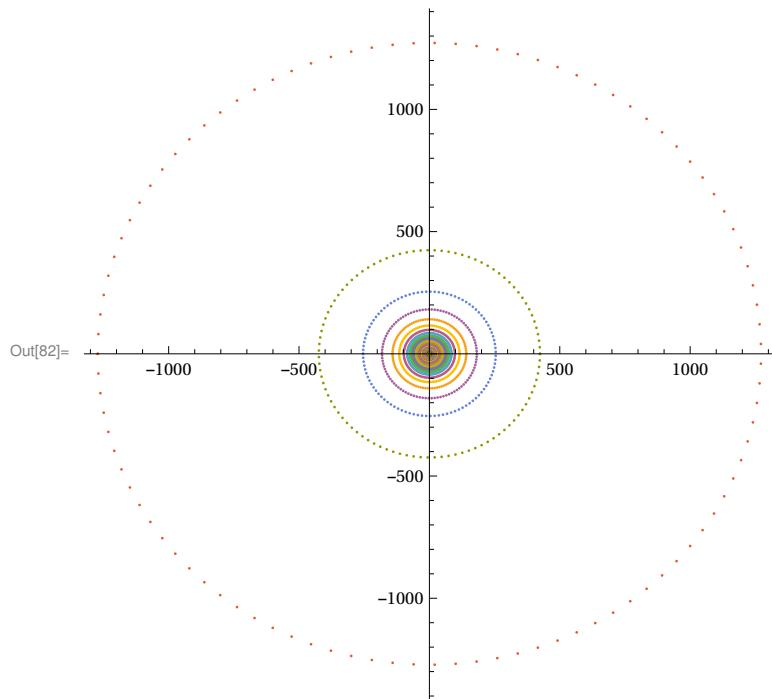
```
In[79]:= ListPlot[makeExponentialSpacedCirclePoints[2, 10 000, 20, 100], AspectRatio -> 1,
PlotRange -> Full]
```



“Sphere” spaced circles

```
In[80]:= makeSphereSpacedCirclePoints[numCircles_, ptsPerCircle_] :=
Module[{ang, lists, pts},
ang=Range[0 Pi,2 Pi,  $\frac{2\pi-\theta}{ptsPerCircle-1}$ ];
lists=Table[{r Cos[ang],r Sin[ang]},{r,makeSphereSpacedPoints[numCircles]}];
pts=Transpose[##]&@lists;
Return[pts]
]
makeSphereSpacedCirclePoints::usage =
"makeSphereSpacedCirclePoints[numCircles_, ptsPerCircle_]
makeSphereSpacedCirclePoints returns circles spaced by inverse
stereographic projection given a number of circles, and the
number of points per circle";
```

```
In[82]:= ListPlot[makeSphereSpacedCirclePoints[1000, 100], AspectRatio -> 1,
PlotRange -> Full]
```



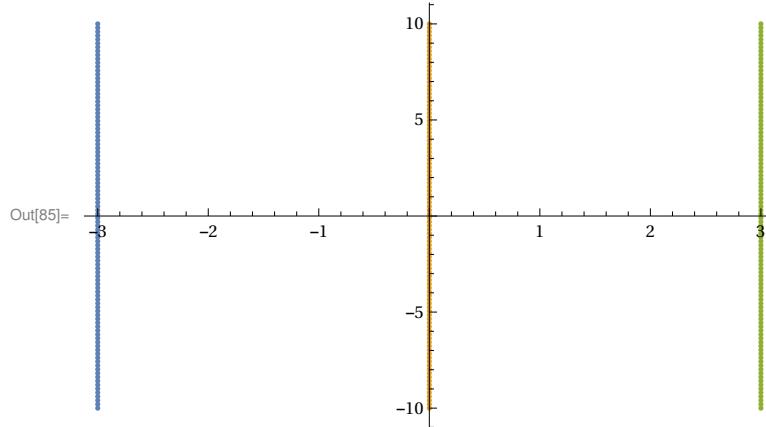
## Line and Grid creation

**Point based line and grid generator functions are useful tool to visualize complex mappings.**

### Linearly spaced lines and grids

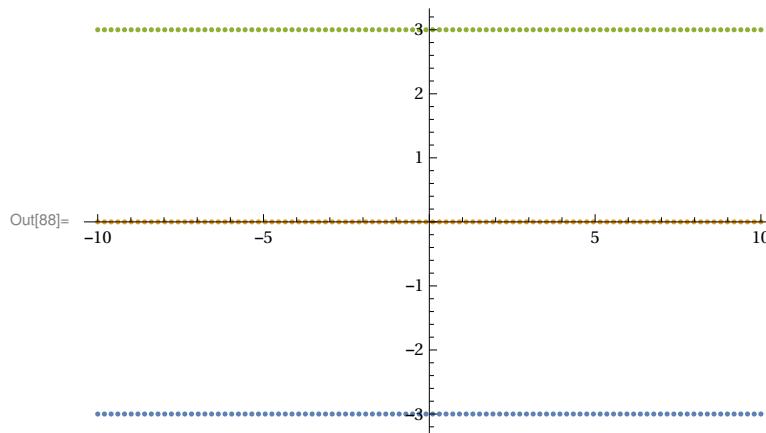
```
In[83]:= makeVerticalPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_] :=
Module[{horiPts},
horiPts=Table[Table[{x,y},{y,Range[minY,maxY, (maxY-minY)/(ptsPerLine-1)]}],
{x,minX,maxX, (maxX-minX)/(numLines-1)}];
Return[horiPts]
]
makeVerticalPts::usage =
"makeVerticalPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]
returns numLines number of vertical lines of equally spaced points
starting from minX to maxX with length maxY-minY with ptsPerLine
number of points per line";
```

```
In[85]:= ListPlot[makeVerticalPts[-3, 3, -10, 10, 100, 3], PlotRange → Full]
```



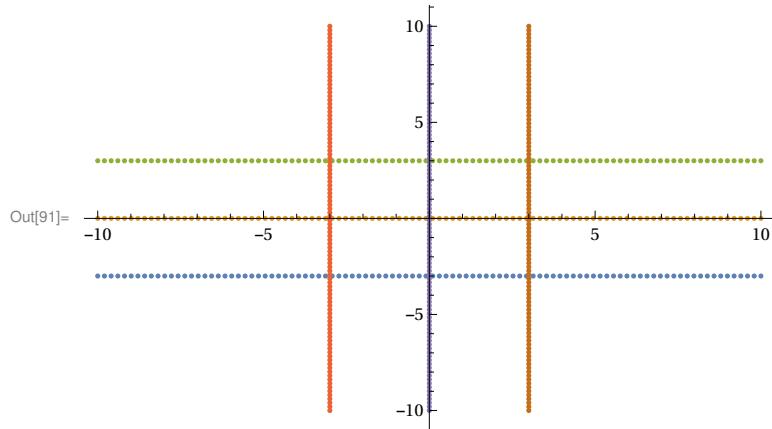
```
In[86]:= makeHorizontalPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_] :=
Module[{horiPts},
horiPts = Table[Table[{x, y}, {x, Range[minX, maxX, (maxX - minX)/(ptsPerLine - 1)]}],
{y, minY, maxY, (maxY - minY)/(numLines - 1)}];
Return[horiPts]
]
makeHorizontalPts::usage =
"makeHorizontalPts[minX_,maxX_,minY_,maxY_,ptsPerLine_,numLines_]
returns numLines number of horizontal lines of equally spaced points
starting from minY to maxY with length maxX-minX with ptsPerLine
number of points per line";
```

```
In[88]:= ListPlot[makeHorizontalPts[-10, 10, -3, 3, 100, 3], PlotRange → Full]
```



```
In[89]:= makeGridPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]:=Module[{vertPts,horiPts}
vertPts=makeVerticalPts[minY,maxY,minX,maxX,ptsPerLine,numLines];
horiPts=makeHorizontalPts[minX,maxX,minY,maxY,ptsPerLine,numLines];
pts=Join[horiPts,vertPts];
Return[pts]
]
makeGridPts::usage="makeGridPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]
returns numLines number of vertical of equally spaced points lines starting from min
length maxY-minY with ptsPerLine number of points per line and
returns numLines number of horizontal of equally spaced points lines starting from i
length maxX-minX with ptsPerLine number of points per line.
NOTE: enter args as if doing horizontal line";
```

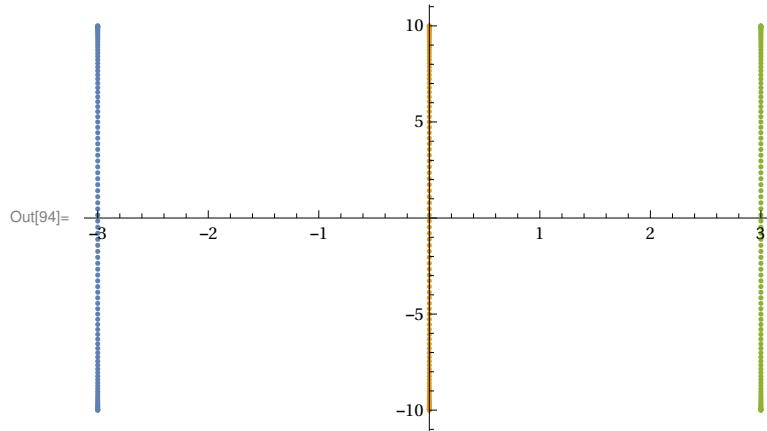
```
In[91]:= ListPlot[makeGridPts[-10, 10, -3, 3, 100, 3], PlotRange → Full]
```



### Exponentially spaced lines and grids

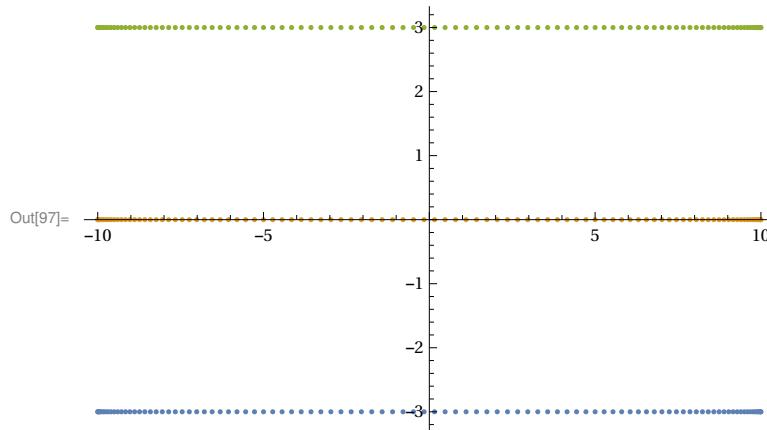
```
In[92]:= makeLogVerticalPts2[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]:=Module[{vertPts},
vertPts = Table[Table[{x, y}, {y, fSpace[minY, maxY, ptsPerLine]}],
{x, minX, maxX, (maxX - minX)/(numLines - 1)}];
Return[Re[vertPts]]
]
makeLogVerticalPts2::usage =
"makeLogVerticalPts2[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]
returns numLines number of vertical lines of exponentially spaced
points starting from minX to maxX with length maxY-minY with
ptsPerLine number of points per line.
NOTE: makeLogVerticalPts2 is deprecated due to not handling negative
minY to positive maxY ranges";
```

```
In[94]:= ListPlot[makeLogVerticalPts2[-3, 3, -10, 10, 100, 3], PlotRange -> Full]
```



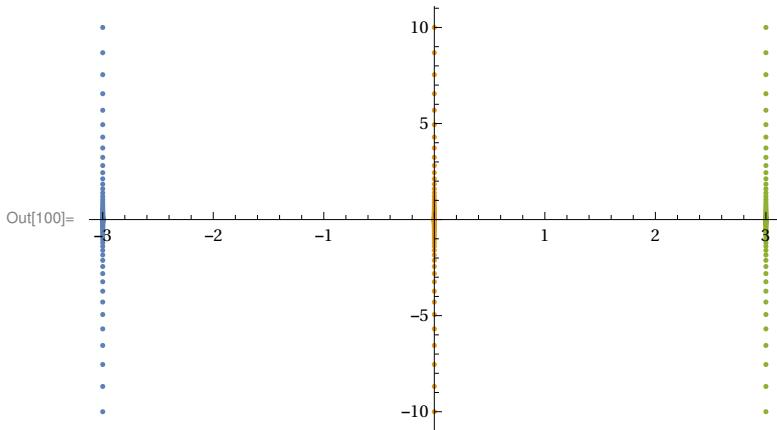
```
In[95]:= makeLogHorizontalPts2[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_] := Module[{horiP...  
horiPts = Table[Table[{x, y}, {x, fSpace[minX, maxX, ptsPerLine]}], {y, minY, maxY,  $\frac{maxY-minY}{numLines-1}$ }];  
Return[Re[horiPts]]  
]  
makeLogHorizontalPts2::usage = "makeLogHorizontalPts2[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]  
returns numLines number of horizontal lines of exponentially spaced points starting  
length maxX-minX with ptsPerLine number of points per line  
NOTE: makeLogHorizontalPts2 is deprecated due to not handling negative minX to pos...  
"
```

```
In[97]:= ListPlot[makeLogHorizontalPts2[-10, 10, -3, 3, 100, 3], PlotRange -> Full]
```



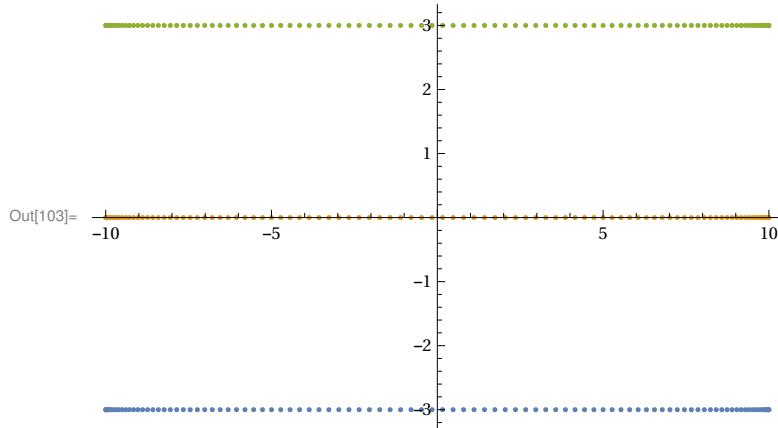
```
In[98]:= makeLogVerticalPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]:=Module[
{posVertPts,negVertPts,pts},
posVertPts=Table[Table[{x,y},{y,fSpace[.01,maxY,Floor[ptsPerLine/2]]}],{x,minX,maxX,(maxX-minX)/(numLines-1)}];
negVertPts=Table[Table[{x,y},{y,fSpace[minY,-.01,Floor[ptsPerLine/2]]}],{x,minX,maxX,(maxX-minX)/(numLines-1)}];
pts=MapThread[Join,{negVertPts,posVertPts}];
Return[Re[pts]]
]
makeLogVerticalPts::usage =
"makeLogVerticalPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]
returns numLines number of vertical lines of exponentially spaced
points starting from minX to maxX with length maxY-minY with
ptsPerLine number of points per line.";
```

```
In[100]:= ListPlot[makeLogVerticalPts[-3, 3, -10, 10, 100, 3], PlotRange → Full]
```



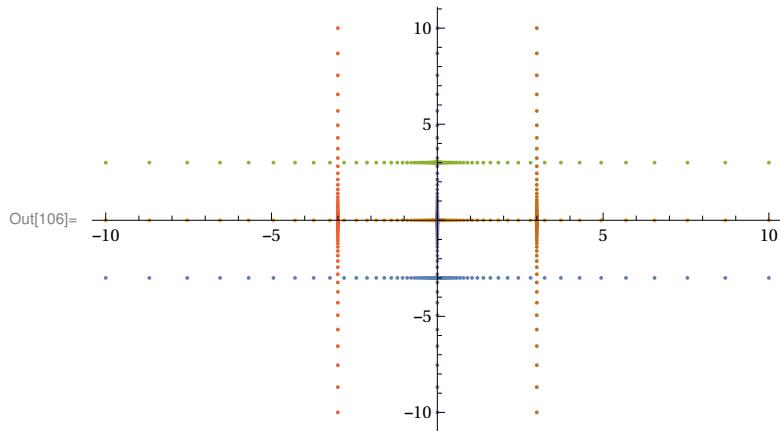
```
In[101]:= makeLogHorizontalPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]:=Module[
{posHoriPts,negHoriPts,pts},
posHoriPts=Table[Table[{x,y},{x,fSpace[.01,maxX,Floor[ptsPerLine/2]]}],{y,minY,maxY,(maxY-minY)/(numLines-1)}];
negHoriPts=Table[Table[{x,y},{x,fSpace[minX,-.01,maxX,Floor[ptsPerLine/2]]}],{y,minY,maxY,(maxY-minY)/(numLines-1)}];
pts=MapThread[Join,{negHoriPts,posHoriPts}];
Return[Re[pts]]
]
makeLogHorizontalPts::usage =
"makeLogHorizontalPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]
returns numLines number of horizontal lines of exponentially spaced
points starting from minY to maxY with length maxX-minX with
ptsPerLine number of points per line";
```

```
In[103]:= ListPlot[makeLogHorizontalPts2[-10, 10, -3, 3, 100, 3], PlotRange → Full]
```



```
In[104]:= makeLogGridPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_]:=Module[
{vertLogPts,horiLogPts,pts},
vertLogPts=makeLogVerticalPts[minY,maxY,minX,maxX,ptsPerLine,numLines];
horiLogPts=makeLogHorizontalPts[minX,maxX,minY,maxY,ptsPerLine,numLines];
pts=Join[horiLogPts,vertLogPts];
Return[pts]
]
makeLogGridPts::usage="makeLogGridPts[minX_, maxX_, minY_, maxY_, ptsPerLine_, numLines_
returns numLines number of vertical of exponentially spaced points lines starting f
with length maxY-minY with ptsPerLine number of points per line and
returns numLines number of horizontal of exponentially spaced points lines starting
with length maxX-minX with ptsPerLine number of points per line.
NOTE: enter args as if doing horizontal line";
```

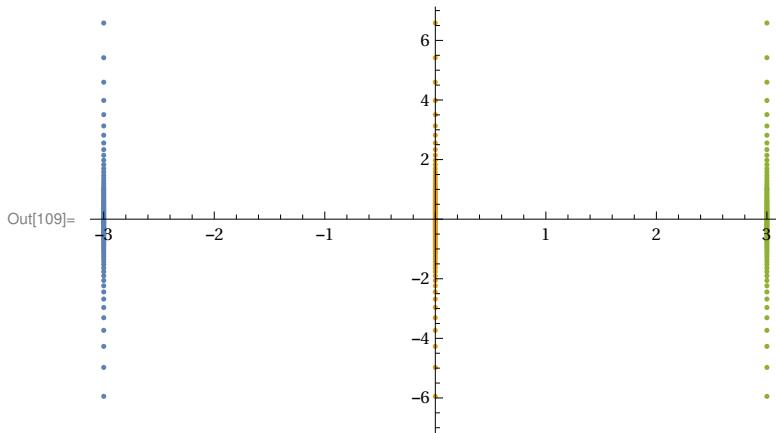
```
In[106]:= ListPlot[makeLogGridPts[-10, 10, -3, 3, 100, 3], PlotRange → Full]
```



### Sphere spaced lines and grids

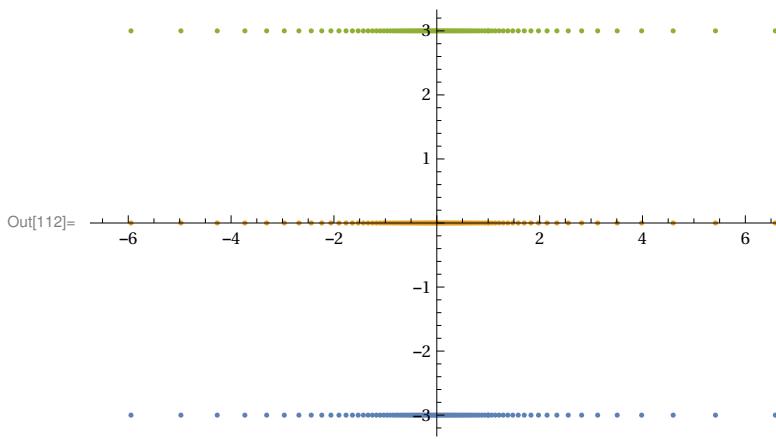
```
In[107]:= makeSphereVerticalLines[min_,max_,ptsPerLine_,numLines_]:=Module[{horiPts,spherePts
spherePts=makeSphereSpacedPoints[ptsPerLine];
horiPts=Table[Table[{x,y},{y,spherePts}],{x,min,max,frac{max-min}{numLines-1}}];
Return[horiPts]
]
makeSphereVerticalLines::usage=
"makeSphereVerticalLines[min_,max_,ptsPerLine_,numLines_] makes vertical lines
sphere spaced points given line bounds of min, max, ptsPerLine, and numLines";
```

```
In[109]:= ListPlot[makeSphereVerticalLines[-3, 3, 100, 3]]
```



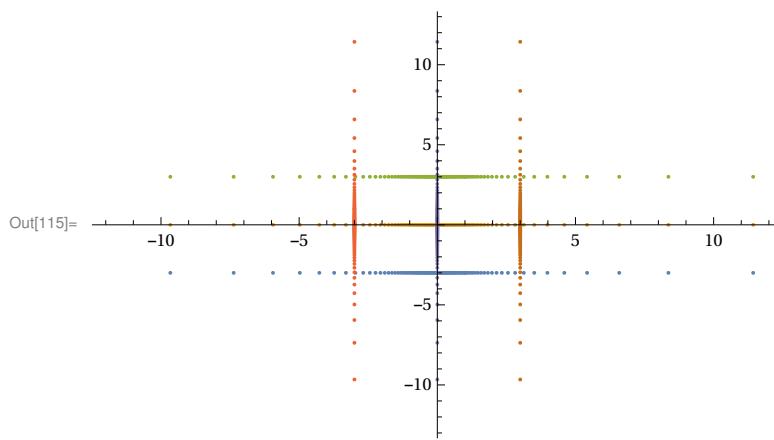
```
In[110]:= makeSphereHorizontalLines[min_, max_, ptsPerLine_, numLines_] :=
Module[{horiPts, spherePts},
spherePts = makeSphereSpacedPoints[ptsPerLine];
horiPts = Table[Table[{x, y}, {x, spherePts}], {y, min, max, (max - min) / numLines - 1}];
Return[horiPts]
]
makeSphereHorizontalLines::usage =
"makeSphereHorizontalLines[min_,max_,ptsPerLine_,numLines_] makes
horizontal lines of sphere spaced points given line bounds of
min, max, ptsPerLine, and numLines";
```

```
In[112]:= ListPlot[makeSphereHorizontalLines[-3, 3, 100, 3]]
```



```
In[113]:= makeSphereGrid[min_, max_, ptsPerLine_, numLines_] :=
Module[{vertPts, horiPts, pts},
  vertPts = makeSphereVerticalLines[min, max, ptsPerLine, numLines];
  horiPts = makeSphereHorizontalLines[min, max, ptsPerLine, numLines];
  pts = Join[horiPts, vertPts];
  Return[pts]
]
makeSphereGrid::usage =
"makeSphereGrid[min,max,ptsPerLine,numLines] makes a grid of sphere
spaced points given line bounds of min, max, ptsPerLine, and numLines";
```

```
In[115]:= ListPlot[makeSphereGrid[-3, 3, 100, 3]]
```



## Complex Plane to Image Plane

Here we make functions that let us deal with lists of points on the complex plane, a mapping expression, and the resulting image plane.

Get image of points

```
In[116]:= getImagePts[expr_, pts_] := Module[{imgPts},
  imgPts =
  {Re[expr /. z → #[[1]] + I#[[2]]], Im[expr /. z → #[[1]] + I#[[2]]]} & /@
  # & /@ pts;
  Return[imgPts]]
getImagePts::usage =
"getImagePts[expr[z],pts] returns the image points given takes in
an expression of z and a list of lists of lists of points";
```

```
In[118]:= getImagePts[z + 1 - I, {{1, 1}, {2, 2}}]
```

```
Out[118]= {{2, 0}, {3, 1}}
```

### Plot image of points

```
In[119]:= plotImage[pts_, expr_, pltRange1_: Automatic, PltRange2_: Automatic, colors_: 1] :=
Module[{},

  If[colors == 1, colors = RGBColor[1, 1, 1], colors = colors];
  {

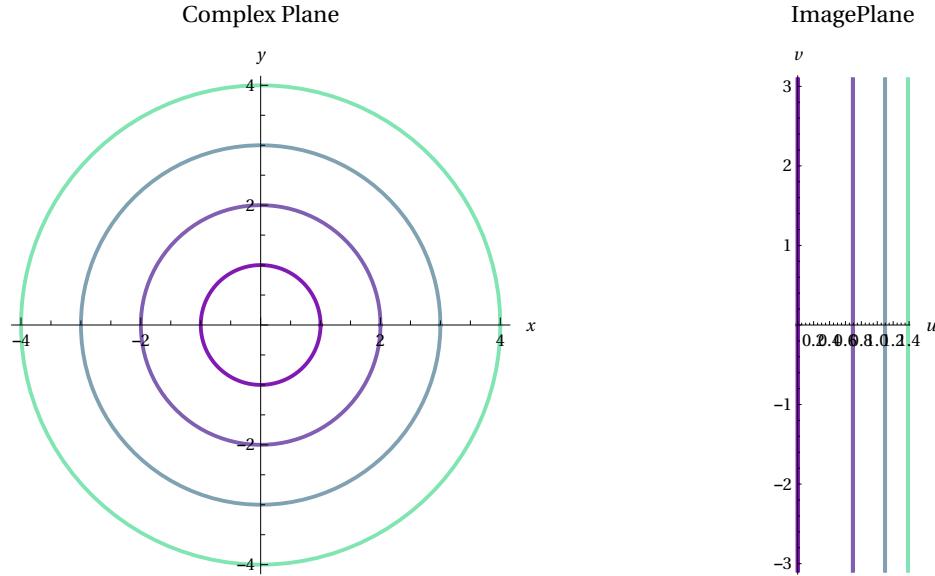
    Graphics[MapThread[{#1, Thick, Line[#2]} &, {colors, pts}],
      PlotRange -> pltRange1, Axes -> True, Background -> White, ImageSize -> {300, 300},
      AxesLabel -> {Style["x", Italic], Style["y", Italic]}, ImagePadding -> 20],
    Graphics[
      MapThread[
        {#1, Thick,
          Line[{Re[expr /. z -> #[[1]] + I #[[2]]], Im[expr /. z -> #[[1]] + I #[[2]]]} &/@#
          #2]} &, {colors, pts}], PlotRange -> PltRange2, Axes -> True,
      Background -> White, ImageSize -> {300, 300},
      AxesLabel -> {Style["u", Italic], Style["v", Italic]}, ImagePadding -> 20]
    ]
  }
]

plotImage::usage =
"plotImage[pts,expr,pltRange1:Automatic,PltRange2:Automatic,colors:1]
  returns a list containing the complex and image plots given a
  list of lists of lists of points, an expression of z, two plot
  ranges (can be Automatic), and a list of colors. Note that
  plotImage doesn't call getImagePts so we can MapThread with
  color gradients";
```

```
In[121]:= circles = Quiet[makeCirclePoints[1, 4, 4, 100]];
circleColors = makeColorGradient[circles]
Quiet@
  Grid[{{"Complex Plane", "ImagePlane"}, {plotImage[circles, Log[z], Automatic, Automatic, circleColors]}}, // TraditionalForm

Out[122]= {█, █, █, █}
```

Out[123]/TraditionalForm=



## Colors

This color section provides additional color generating functions to help the user understand what is going on in the complex plane to image plane or Riemann Sphere mappings

### Color functions

```
In[124]:= makeRandomColors[pts_]:=Module[{colors},
  colors=RandomColor[Length[pts]];
  Return[colors]]
makeRandomColors::usage="makeRandomColors[pts_] returns length of pts number of random colors"
```

In[126]:= makeRandomColors[{{0, 0}, {1, 1}, {2, 2}, {3, 3}, {4, 4}}]

Out[126]= {█, █, █, █, █}

```
In[127]:= makeColorGradient[pts_] := Module[{colors},
  colors = Table[RGBColor[.5, x, .7], {x, .1, .9,  $\frac{.9 - .1}{\text{Length}[pts] - 1}$ }];
  Return[colors]]
makeColorGradient::usage =
  "makeColorGradient[pts_] returns length of pts number of colors in
  a purple -> teal gradient";
```

```
In[129]:= makeColorGradient[{{0, 0}, {1, 1}, {2, 2}, {3, 3}, {4, 4}}]
```

```
Out[129]= {█, █, █, █, █}
```

## Riemann Sphere Stereographic Projection

**Functions that allow the projection of points from the complex plane to the Riemann Sphere through the use of stereographic projection**

Single point

```
In[130]:= complexTo3D[point_]:=Module[{x,y,z,mix,X,Y,Z},
  x=point[[1]];
  y=point[[2]];
  mix= $\frac{2x+i 2y}{1+x^2+y^2}$ ;
  X=Re[mix];
  Y=Im[mix];
  z=x+i y;
  Z= $\frac{\text{Abs}[z]^2-1}{\text{Abs}[z]^2+1}$ ;
  Return[{X,Y,Z}]
]
complexTo3D::usage="complexTo3D[point_] returns the stereographic projection of a p"
```

```
In[132]:= complexTo3D[{1, 1}]
```

```
Out[132]=  $\left\{ \frac{2}{3}, \frac{2}{3}, \frac{1}{3} \right\}$ 
```

### Multiple points

```
In[133]:= complexPtsTo3D[points_] := Module[{spherePts3D},  

spherePts3D = complexTo3D[#[#]&@@@points;  

Return[spherePts3D]  

]  

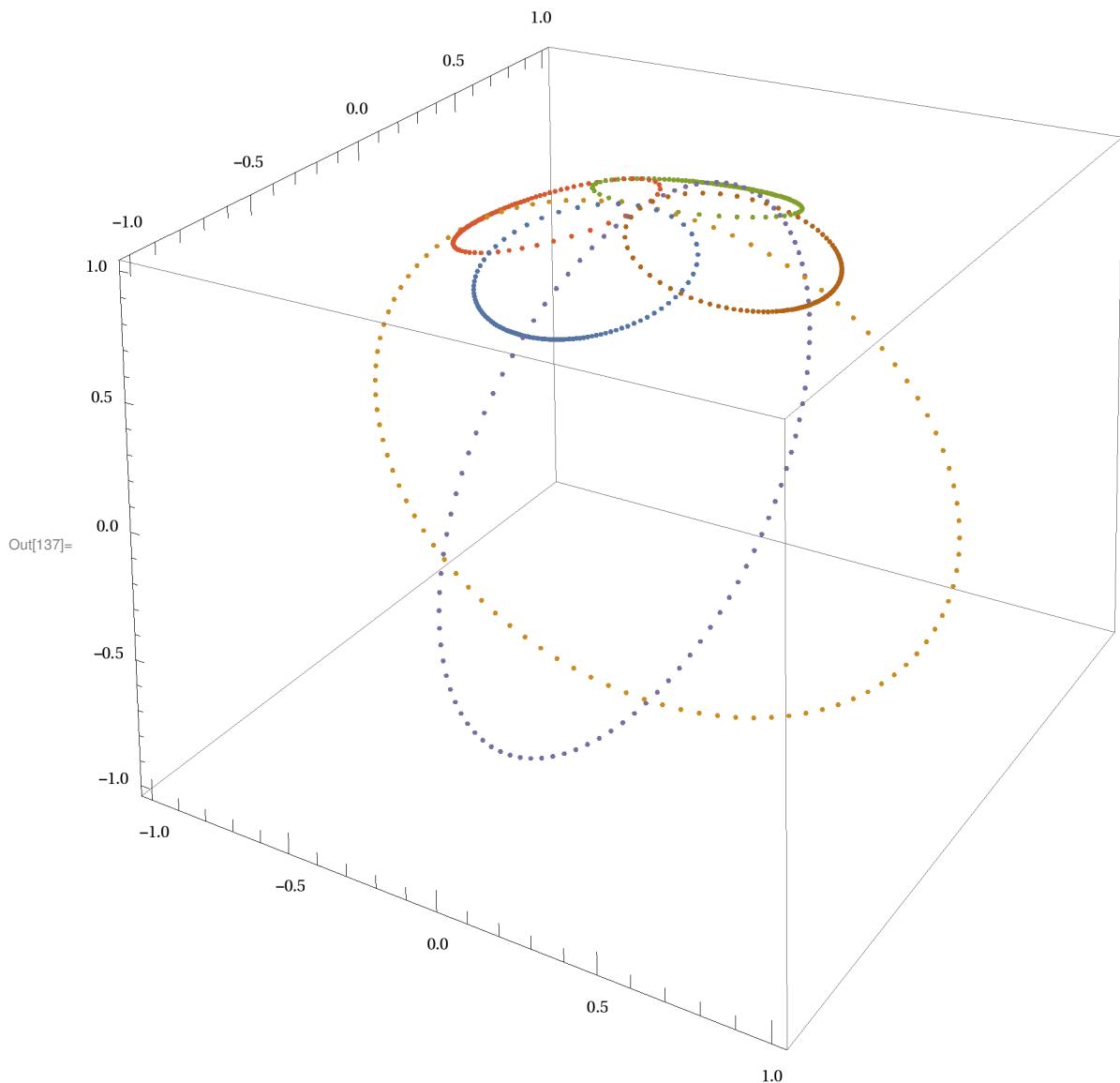
complexPtsTo3D::usage = "complexPtsTo3D[points_] returns the stereographic projection  

of points on the complex plane";  
  

In[135]:= gridPts2D = makeSphereGrid[-3, 3, 100, 3];  

gridPts3D = complexPtsTo3D[gridPts2D];  

ListPointPlot3D[gridPts3D, PlotRange -> All, AspectRatio -> 1, ImageSize -> Large]
```

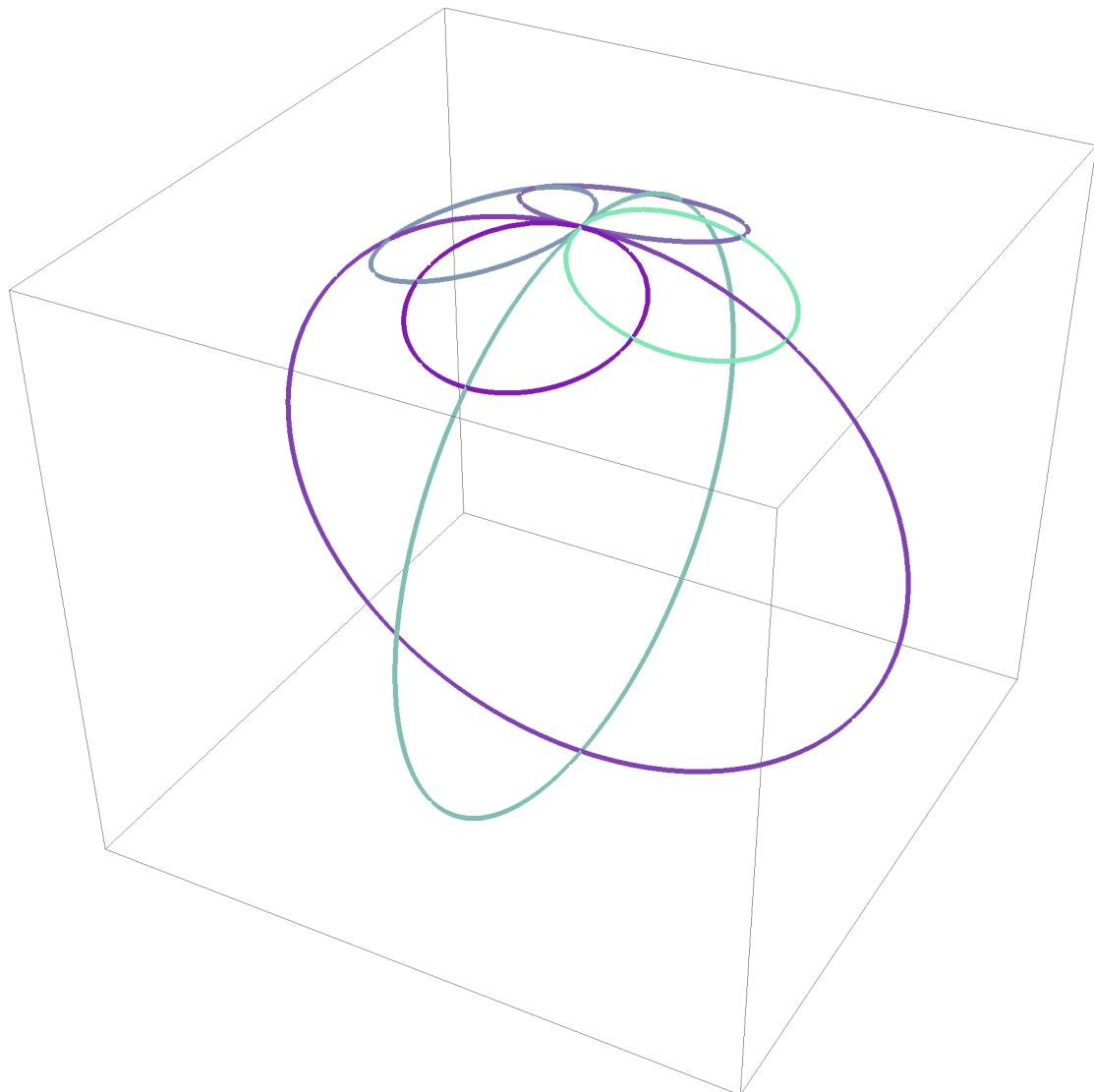


In[138]:=

### Color Graphics3D test

```
In[139]:= colors3D = makeColorGradient[gridPts3D]
Graphics3D[MapThread[{#1, Thick, Line[#2]} &, {colors3D, gridPts3D}],
AspectRatio -> 1, ImageSize -> Large]
Out[139]= {■, ■, ■, ■, ■, ■}
```

Out[140]=



## 3D printing

**3D printing allows us to take what we see in the computer and hold it in real life.**

### Connecting cylinders

```
In[141]:= takingPts[takes_,pts_]:=Module[{newPts},
  newPts=Take[pts, #]&/@takes;
  Return[newPts]]
takingPts::usage="takingPts[takes_,pts_] returns a list of taken points
given a list of takes and a list of points";
```

```
In[143]:= takingPts[{{1, 2}, {2, 3}, {3, 4}}, {2, 7, 4, 0}]
Out[143]= {{2, 7}, {7, 4}, {4, 0}}
```

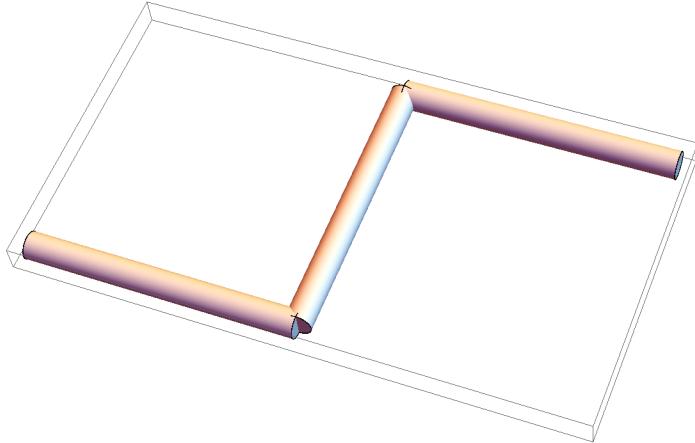
```
In[144]:= cylinderPts[pts_]:=Module[{takes},
  takes=Table[{x,x+1},{x,1,Dimensions[pts][[2]]-1}];
  Return[takingPts[takes, #]&/@pts]
]
cylinderPts::usage="cylinderPts[pts_] returns a list of taken points ready to be co
```

```
In[146]:= cylinderPts[{{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}}}]
Out[146]= {{{{1, 2, 3}, {4, 5, 6}}, {{4, 5, 6}, {7, 8, 9}}, {{7, 8, 9}, {10, 11, 12}}}}
```

```
In[147]:= cylinders[pts_, radius_]:=Module[{},
  Return[Cylinder[#, radius]&/@#&/@cylinderPts[pts]]]
cylinders::usage =
 "cylinders[pts_,radius_] returns cylinders linearly along the path
  of the list of points with specified radius";
```

```
In[149]:= Graphics3D[cylinders[{{{1, 2, 3}, {2, 2, 3}, {2, 3, 3}, {3, 3, 3}}}, .05]]
```

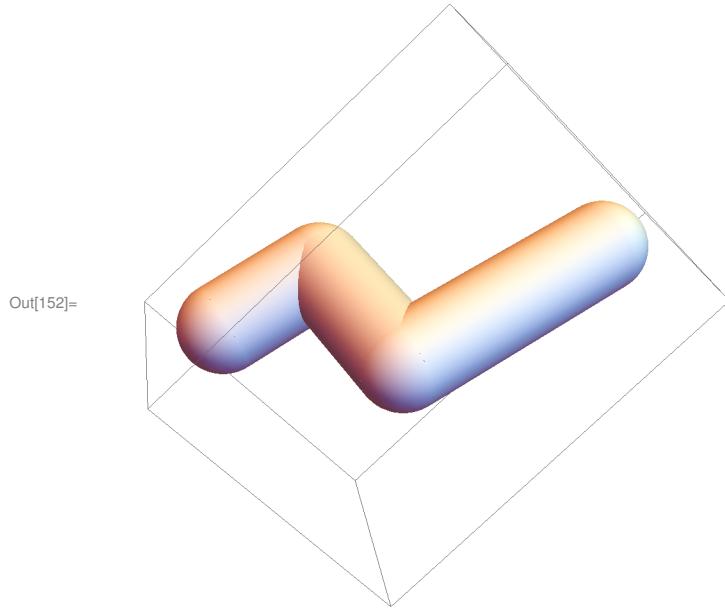
Out[149]=



### Connecting tubes

```
In[150]:= tubes[pts_, radius_] := Module[{ }, Tube[#, radius] & /@ complexPtsTo3D[pts]]
tubes::usage = "tubes[pts_,radius_] returns tubes linearly
along the path of the list of points with specified radius. This is
sometimes used as it gives a better result than cylinders but is less
reliable";
```

```
In[152]:= Graphics3D[tubes[{{{1, 2, 3}, {2, 2, 3}, {2, 3, 3}, {3, 3, 3}}}, .05]]
```



### Exporting STL's examples

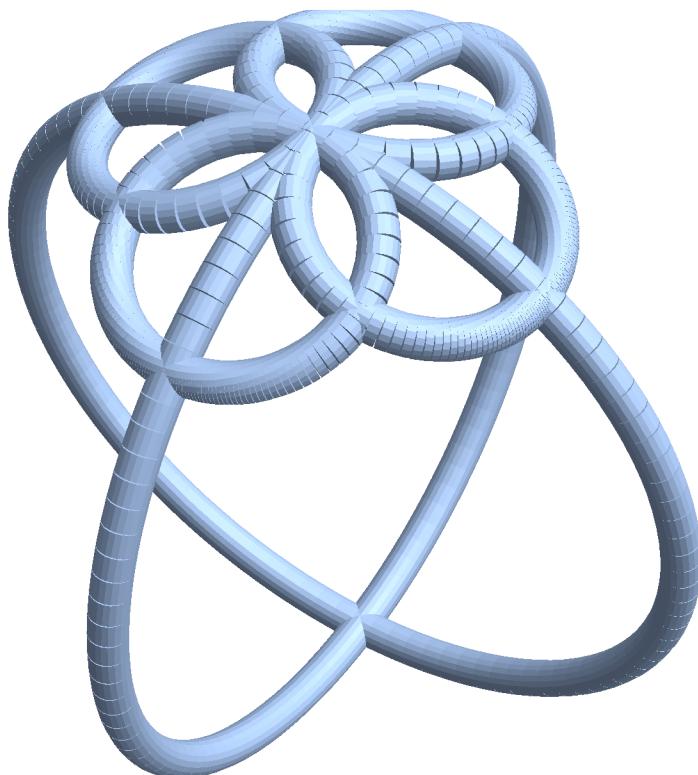
```
In[153]:= gridPts2D = makeSphereGrid[-3, 3, 100, 3];
gridPts3D = complexPtsTo3D[gridPts2D];

In[155]:= Export["grid2.stl", Graphics3D[cylinders[gridPts3D, .05]], {"STL", "BinaryFormat" \rightarrow True}]

Out[155]= grid2.stl
```

```
In[156]:= Import["grid2.stl"]
```

Out[156]=

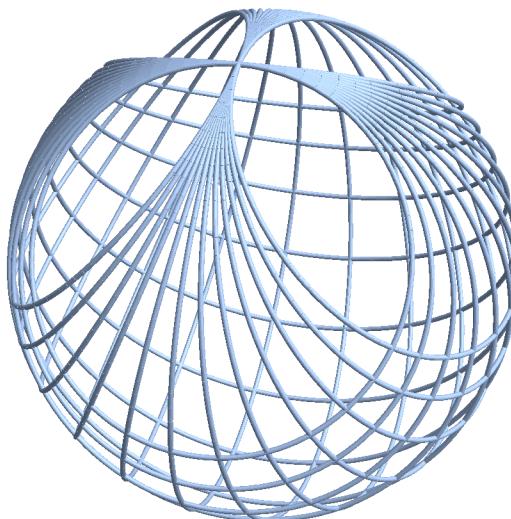


```
In[157]:= Export["grid_10_7.stl",
  Graphics3D[cylinders[complexPtsTo3D[makeSphereGrid[-1, 1, 100, 12]], .01]],
  {"STL", "BinaryFormat" → True}]
```

Out[157]= grid\_10\_7.stl

```
In[158]:= grid107 = Import["grid_10_7.stl"]
```

Out[158]=



## Newtons Method

**Newton's Method illustrates how visualizations of complex mappings helps us understand numerical methods.**

### Animation Function

```
In[159]:= makeNewtonMethodAnimation[plotRange_, map_, depth_, points_] :=
Module[{localPts, plots},
localPts = points;
plots = Flatten[{{
ListPlot[localPts,
PlotRange -> {{-plotRange, plotRange}, {-plotRange, plotRange}},
PlotStyle -> PointSize[.02], AspectRatio -> 1, ImageSize -> Large]}},
Table[
(*Apply map to grid pts*) localPts = getImagePts[map, localPts];
ListPlot[localPts, PlotRange -> {{-plotRange, plotRange},
{-plotRange, plotRange}}, PlotStyle -> PointSize[.02],
AspectRatio -> 1, ImageSize -> Large]
, {n, 1, depth}]]];
Return[Manipulate[plots[[n]], {n, 1, Length[plots], 1}]]];
makeNewtonMethodAnimation::usage =
"makeNewtonMethodAnimation[plotRange, map, depth, points] makes
animation of successive mappings";
```

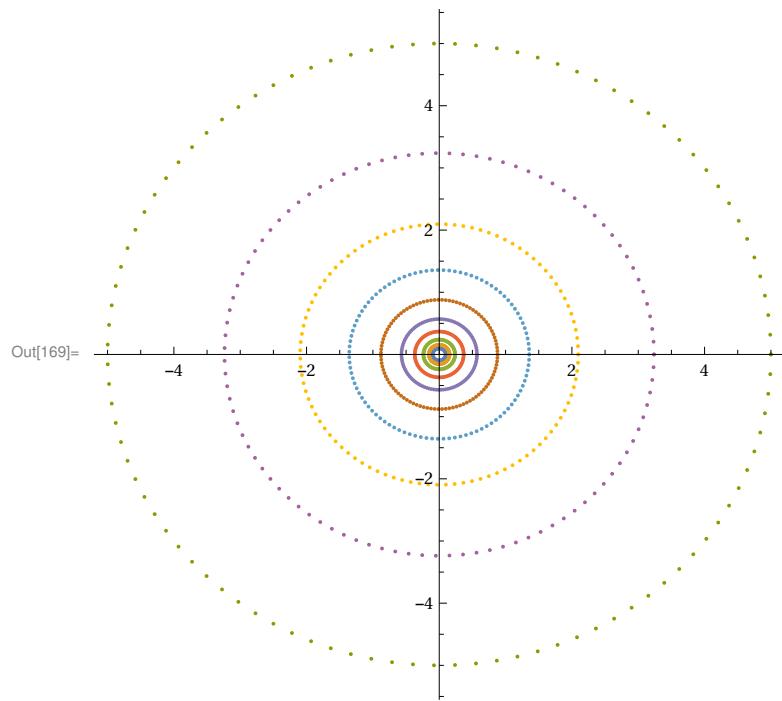
In[161]:=  $f[z_] := z^3 - 1$

Use definition of Newton's Method to derive mapping function

In[162]:=  $\text{map} = \left( z - \frac{f[z]}{f'[z]} \right);$

Make a circular grid:

```
In[163]:= smallestRadius=.1;
largestRadius=5;
numberOfCircles=10;
pointsPerCircle=100;
circleGrid = makeExponentialSpacedCirclePoints[smallestRadius,largestRadius,numberOfCircles,pointsPerCircle];
circleGridCopy1=circleGrid;
ListPlot[circleGridCopy1,AspectRatio→1,PlotRange→Full]
```



```
In[170]:= makeNewtonMethodAnimation[5, map, 35, circleGrid]
```

