Interactive Visualization of the Riemann Sphere

MATH 165B - Final Project

Ricardo J. Acuña

May 29, 2020

Abstract

While algebraic manipulation of complex valued functions yield direct formulas for geometric objects, it's often difficult to see what object an equation represents. With the advent of computers we're able to directly visualize complicated equations. In this paper, we'll provide methods to graph such complex valued equations both in the Complex Plane and in the Riemann Sphere.

1 Introduction

A function $f : \mathbb{C}^2 \circlearrowleft$ is initially hard to grasp geometrically. If we consider the analogue of a function $g : R \circlearrowleft$ we can consider it's graph $\{(x, y) \in \mathbb{R}^2 | y = g(x)\}$ which is a 2D object which we can grasp as a picture in the plane. However, the graph of $f = u + iv$ is $\{(z, w) \in \mathbb{C}^2 | w = g(z)\} \simeq \{(x, y, w_1, w_2) \in \mathbb{R}^4 | w_1 = u(x, y),$ and $w_2 = v(x, y)\}$ which is 4D, more dimensions than the ones we can experience.

So what we usually do is break up the domain and the codomain and graph them separately. And then we consider the function f as an action that transforms the x, y-plane into the u, v-plane.

However, initially there is no necessary geometric meaning of the set $\mathbb{C}$. But, we can give it such a meaning. We can see there is canonical isomorphism $\phi : \mathbb{C} \xrightarrow{\sim} \mathbb{R}^2$ between the set of complex numbers $\mathbb{C}$, and the set of pairs of real numbers $\mathbb{R}^2$, which gives us a coordinate system for the Euclidean plane. If we let $z \in \mathbb{C}$, then the isomorphism ϕ is given by,

$$\phi(z) = (\Re(z), \Im(z))$$

There are infinitely many ways we can embed $\mathbb{R}^2$ into $\mathbb{R}^3$. However if we let, $(x, y) \in \mathbb{R}^2$, it's natural to identify the plane with the inclusion map $\iota : \mathbb{R}^2 \hookrightarrow \mathbb{R}^3$

$$\iota(x, y) = (x, y, 0)$$

The Riemann Sphere is an isomorphism between the Extended Complex Plane $\mathbb{C} \cup \{\infty\}$, and the points the a unit sphere S^2 centered at the origin. The isomorphism given by a formula called stereographic projection.

In Visual Complex Analysis page 146 formula (20), Tristan Needham derives that formula, if a point on S^2 is given by the cartesian coordinates $(X, Y, Z) \in \mathbb{R}^3$, and a point in the extended Complex Plane$z = x + iy \in \mathbb{C} \cup \{\infty\}$ the stereographic projection is,

$$(X, Y, Z) \mapsto \frac{X}{1 - Z} + i\frac{Y}{1 - Z}$$

And the inverse stereographic projection is given by,

$$x + iy \mapsto \left(\frac{2x}{1 + x^2 + y^2}, \frac{2y}{1 + x^2 + y^2}, \frac{-1 + x^2 + y^2}{1 + x^2 + y^2}\right)$$

The derivation uses the plane $z = 0$, however notice that the formula for the inverse stereographic projection is independent of the lowercase z. That means, we can embed the plane into $\mathbb{R}^3$ however we like, we do so by the mapping ι_{-1} where,

$$\iota_{-1} : \mathbb{C} \hookrightarrow \mathbb{R}^3; x + iy \mapsto (x, y, -1)$$

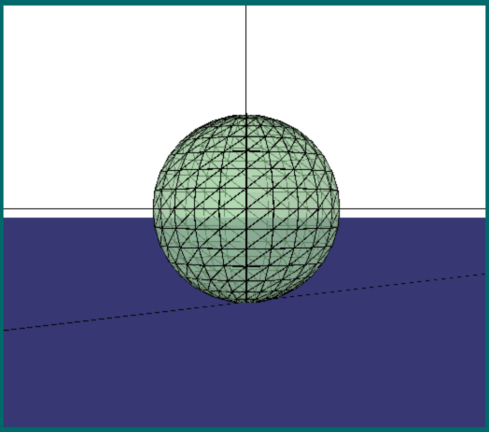
The reason we want to embed this way is because, while we could make a 3D rendering of S^2, and a 2D rendering of $\mathbb{C}$ side by side. It's much nicer to look at them together as a 3D graph. And in the derivation notice, that the plane $z = 0$, slices S^2 in the middle, so much of the detail of the stereographic projection would be muddled.

2 Method

At first I didn't know how to approach the problem, so I did the naïve thing and made a couple of spheres, together with a plane.

The first one was made using clojurescript, and a rendering library called quil,

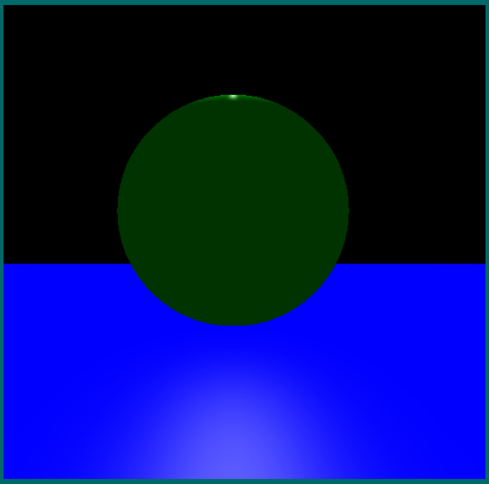
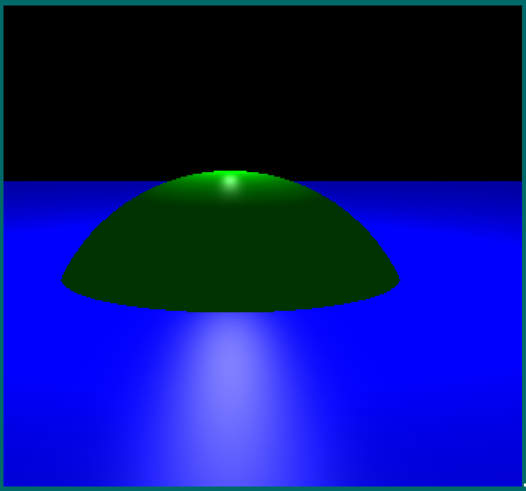
```
(ns sphere.core
  (:require [quil.core :as q :include-macros true]
            [quil.middleware :as m]))

(def width (. js/window -innerWidth))
(def height (. js/window -innerHeight))

(defn draw [state]
  (q/background 255)
  (q/lights)
  (q/fill 230 110 110)
  (q/with-translation [[(/ width 2) (/ height 2) 0]]
    (q/fill 255 0 0)
    (q/curve 0 0 0 0 0 0 0 0 0 (* 50 width) 0 0)
    (q/curve 0 0 0 0 0 0 0 0 0 (- (* 50 width)) 0 0)
    (q/fill 0 255 0)
    (q/curve 0 0 0 0 0 0 0 0 0 0 (* 50 width) 0)
    (q/curve 0 0 0 0 0 0 0 0 0 0 (- (* 50 width)) 0)
    (q/fill 0 0 255)
    (q/curve 0 0 0 0 0 0 0 0 0 0 0 (* 50 width))
    (q/curve 0 0 0 0 0 0 0 0 0 0 0 (- (* 50 width)))
    (q/fill 110 110 230)
    (q/with-translation [[0 100 0]]
      (q/rotate-x (/ 3.1415 2))
      (q/plane (* 50 width) (* 50 width)))
    (q/fill (q/color 110 230 110 120))
    (q/with-translation [[0 0 0]]
      (q/sphere 100)
      (q/with-translation [[0 -101 0]]
        (q/sphere 1)))))

(q/defsketch sphere
  :host "sphere"
  :draw draw
  :renderer :p3d
  :middleware [m/fun-mode m/navigation-3d]
  :size [width height])
```



However, that package didn't allow enough control on the movement of the sphere. Next, I used the same computer language, but now with a library called threeagent. I was able to add controls such that the position of the sphere changed when you clicked the up and down arrows.

```
(ns sphere.app
  (:require ["three" :as three]
            [threeagent.alpha.core :as th]
            [reagent.core :as r]))

(def sphere-velocity 5.0)

(defonce state (th/atom {:sphere {:right  {:height 3.0
                                           :position [0 0 0]}}}))
(def world-scale 1.0)

(defn sphere [side]
  (let [[x-pos y-pos z-pos] @(th/cursor state [:sphere side :position])
        height @(th/cursor state [:sphere side :height])]
    [:object
     [:point-light {:intensity 1.8
                    :position [x-pos (+ y-pos 2.3) z-pos]}]
     [:sphere {:radius 2
               :position [x-pos y-pos z-pos]
               :width-segments 32
               :height-segments 32
               :material {:color "green"
                          :transparency 0.5
                          :transparent true}}]]))

(defn root []
  [:object
   [:hemisphere-light {:position [0 10 10]
                       :intensity 0.4}]
   [:object {:position [0 0 -10]
             :scale [world-scale world-scale 1]}
    [:box {:position [0 -2 0]
           :scale [20 0.1 20]
           :material {:color "blue"
                      :transparency 0.9
                      :transparent true}}]
    [sphere :right]]])

(defn- update-sphere! [delta-time]
  (doseq [p [:right]]
    (let [[px py pz] (get-in @state [:sphere p :position])
          velocity (get-in @state [:sphere p :velocity])
          new-py (+ py (* delta-time velocity))]
      (swap! state assoc-in [:sphere p :position] [px new-py pz]))))

(defn- tick [delta-time]
  (update-sphere! delta-time))

(defn on-keydown [evt]
  (case (.-code evt)
    "ArrowUp" (do
                (.preventDefault evt)
                (swap! state assoc-in [:sphere :right :velocity] sphere-velocity))
    "ArrowDown" (do
                  (.preventDefault evt)
                  (swap! state assoc-in [:sphere :right :velocity] (- sphere-velocity)))
    nil))

(defn- on-keyup [evt]
  (case (.-code evt)
    "ArrowUp" (swap! state assoc-in [:sphere :right :velocity] 0)
    "ArrowDown" (swap! state assoc-in [:sphere :right :velocity] 0)
    nil))

(defn init []
  (th/render [root]
             (.getElementById js/document "root")
             {:on-before-render tick})
  (.addEventListener js/window "keydown" on-keydown)
  (.addEventListener js/window "keyup" on-keyup))

(defn ^:dev/after-load reload []
  (th/render [root]
             (.getElementById js/document "root")
             {:on-before-render tick}))
```

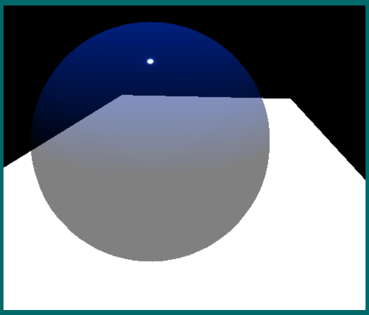
A light was placed atop the sphere in the hopes I could shine it through a texture on the sphere and project it's shadow onto the plane, such that as I moved the sphere, the corresponding shadow on the plane changed. The result is as follows,



.

However, there were problems with the library's control of the transparency. Next, I decided to switch to Javascript and use the library three.js directly, as three.js was the backend for threeagent. What I lost was the ability to control the sphere, but I gained control over the transparency of the sphere.

```
import * as THREE from 'https://threejsfundamentals.org/threejs/resources/threejs/r115/build/three.m
import {OrbitControls} from 'https://threejsfundamentals.org/threejs/resources/threejs/r115/examples

function main() {
  const canvas = document.querySelector('#c');
  const renderer = new THREE.WebGLRenderer({canvas});

  const fov = 75;
  const aspect = 2;  // the canvas default
  const near = 0.1;
  const far = 25;
  const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
  camera.position.z = 4;
  const controls = new OrbitControls(camera, canvas);
  controls.target.set(0, 0, 0);
  controls.update();

  const scene = new THREE.Scene();
  scene.background = new THREE.Color('black');

  function addLight(...pos) {
    const color = 0xFFFFFF;
    const intensity = 1;
    const light = new THREE.DirectionalLight(color, intensity);
    light.position.set(...pos);
    scene.add(light);
  }
// addLight(-1, 2, 4);
  // addLight( 1, -1, -2);

  const radius = 1;
  const widthSegments = 32;
const heightSegments = 32;
  const sphereGeometry = new THREE.SphereBufferGeometry(radius, widthSegments,heightSegments);
const planeGeometry = new THREE.PlaneGeometry( 5, 20, 32 );

function hsl(h, s, l) {
    return (new THREE.Color()).setHSL(h, s, l);
  }

  function makeSphere(geometry, color, x, y, z) {
    const material = new THREE.MeshPhysicalMaterial({
      color: color,
metalness: 0,
alphaTest: 0.5,
roughness: 0,
depthWrite: false,
transparency: 0.5,
      opacity: 1,
      transparent: true,
    });

    const sphere = new THREE.Mesh(geometry, material);
    scene.add(sphere);
addLight(x,y+1,z)
    sphere.position.set(x, y, z);

    return sphere;
  }


function makePlane(geometry, color, x, y, z) {
const material = new THREE.MeshBasicMaterial( {color: color, side: THREE.DoubleSide} );

const plane = new THREE.Mesh(geometry, material );
scene.add( plane );


plane.position.set(x, y-1, z);
plane.rotateX(3.14/2)
return plane;
}

makeSphere(sphereGeometry,hsl(5/8,1,.5),0,0,0)
makePlane(planeGeometry,"white",0,0,0)

  function resizeRendererToDisplaySize(renderer) {
    const canvas = renderer.domElement;
    const width = canvas.clientWidth;
    const height = canvas.clientHeight;
    const needResize = canvas.width !== width || canvas.height !== height;
    if (needResize) {
      renderer.setSize(width, height, false);
    }
    return needResize;
  }

  let renderRequested = false;

  function render() {
    renderRequested = undefined;

    if (resizeRendererToDisplaySize(renderer)) {
      const canvas = renderer.domElement;
      camera.aspect = canvas.clientWidth / canvas.clientHeight;
      camera.updateProjectionMatrix();
    }

    renderer.render(scene, camera);
  }
  render();

  function requestRenderIfNotRequested() {
    if (!renderRequested) {
      renderRequested = true;
      requestAnimationFrame(render);
    }
  }

  controls.addEventListener('change', requestRenderIfNotRequested);
  window.addEventListener('resize', requestRenderIfNotRequested);
}

main();
```

As you can see its quite a bit more verbose, the resulting sphere is this,



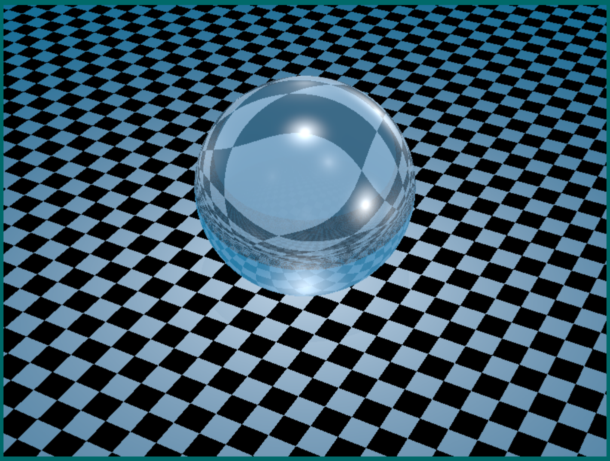
Notice, it doesn't cast shadows. Actually casting shadows is a rather expensive in terms of computation. Its done by a technique called ray casting.

In the paper Moebïus Transformations revealed Douglas N. Arnold and Jonathan Rogness reveal that through that technique they made the animation found in the popular YouTube video of the same name. This project started as an attempt to reproduce the visual effects found in that video. Since, the previous attempts didn't yield satisfactory graphics, I decided to study their methods. However, in the paper they don't give any code whatsoever, they just say it was easy to do with the package POVRayRender with some help of Mathematica.

In any case I set out to reproduce their methods, I found that an extension of Mathematica allows us to output POVrayrender images out of Mathematica graphs. While it was rather technically difficult to install, because there were some issues with Linux compatibility, I managed to make the following sphere. First by making the graphic of a sphere with a plane in Mathematica, and then modifying the source code of some intermediate temporary POVRayrender file to change the transparency of the material and the pattern in the plane. The code is as follows,

```
#include "colors.inc"

global_settings {
  max_trace_level 5
  assumed_gamma 1.0
  radiosity {
    pretrace_start 0.08
    pretrace_end   0.01
    count 35
    nearest_count 5
    error_bound 1.8
    recursion_limit 2
    low_error_factor .5
    gray_threshold 0.0
    minimum_reuse 0.015
    brightness 1
    adc_bailout 0.01/2
  }
}

#default {
  texture {
    pigment {rgb 1}
    finish {
      ambient 0.0
      diffuse 0.6
      specular 0.6 roughness 0.001
      reflection { 0.0 1.0 fresnel on }
      conserve_energy
    }
  }
}

light_source {
    <0, 0, 1>
    color White
    cylinder
    radius 15
    falloff 20
    tightness 10
    point_at <0, 0, 0>
}
```

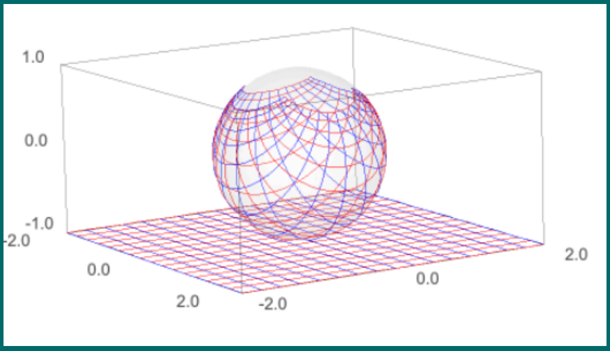
```
light_source {
    <0, 1, 0>
    color White
    cylinder
    radius 15
    falloff 20
    tightness 10
    point_at <0, 0, 0>
}

light_source {
    <1, 0, 0>
    color White
    cylinder
    radius 15
    falloff 20
    tightness 10
    point_at <0, 0, 0>
}

background { rgb <0,.25,.5> }

camera {
location <3.9000000000000004, -7.199999999999999, 5.5>
direction <-3.9000000000000004, 7.199999999999999, -6.>
up <0, 0, 4.242640687119286>
right <5.64271211386865, 0, 0>
sky <0., 0., 1.>
look_at <0., 0., -0.5>
}

sphere {
<0., 0., 0.>,
1

pigment  { rgbf <1,1,1,0.95> }
finish   { phong 0.9 phong_size 40
 // A highlight
           reflection 0.2
 // Glass reflects a bit
 }
interior { ior 1.5}
 // Glass refraction

}

plane { z,-1
texture{
pigment{ checker rgb<1,1,1> rgb<0,0,0>}
scale 0.25
}
}
```



Which is really nice, notice that due to the transparency, the upper hemisphere of the sphere is a conformal mapping. Which means that if you look at the image of the plane in the sphere, the lines of the checkered pattern remain locally at right angles.

Some further research lead me to the code by Xah Le who created a Mathematica package for Möbius transformations, that included a way to use Mathematica to make a stereographic projection. He sells the library that he used to make his examples for $15, I bought it. However, he took a whole day to email me the code. So, by then I had figured out how to do it on my own in the Sage programming environment. This is my first prototype

```
from sage.plot.plot3d.shapes2 import line3d
from sage.plot.plot3d.shapes import Sphere

import numpy as np
interval = np.arange(-2,2.25,0.25)
g = Sphere(1, color = "white", alpha = 0.1)
def isp (p):
    x,y,z = p
    r = 1 + x^2 +y^2
    return (2*x/r,2*y/r,(r-2)/r)

def iz (x,y):
    return (x,y,-1)

for j in interval:
    g += line3d([iz(i,j) for i in interval],color = "blue")
    g += line3d([isp(iz(i,j)) for i in interval], color = "blue")
    g += line3d([iz(j,i) for i in interval], color = "red")
    g += line3d([isp(iz(j,i)) for i in interval], color ="red")

g.show(aspect_ratio=[1,1,1])
```



Notice, the function iz is the embedding ι_{-1}, and isp is the inverse stereographic projection. The prototype takes lines on the plane to lines on S^3.

However, I wanted a little more control on the transparency, so I made the following Sphere.

```
from sage.plot.plot3d.shapes2 import polygon3d
import numpy as np

def isp (p):
    x,y,z = p
    r = 1 + x^2 +y^2
    return (2*x/r,2*y/r,(r-2)/r)

def ix_3 (z):
    x,y = z
    return (x,y,-1)

def make_square (pt,l,c):
    x,y = pt
    s = [(x,y),(x+l,y),(x+l,y+l),(x,y+l)]
    return polygon3d([ix_3(z) for z in s],color = c)

def make_sector (pt,l,c):
    x,y = pt
    s = [(x,y),(x+l,y),(x+l,y+l),(x,y+l)]
    return polygon3d([isp(ix_3(z)) for z in s],color = c,alpha = 0.75)

def make_instance(f):
    instance = Graphics()
    step = 0.25
    interval = np.arange(-2,2+step,step)
    for j in interval:
        for i in interval:
            if (i*4)%2 == 0 and (j*4)%2 != 0 :
                instance += f((0+i,j),step,"red");
            elif (i*4)%2 == 0 and (j*4)%2 == 0 :
                instance += f((0+i,j),step,"green");
            elif (i*4)%2 != 0 and (j*4)%2 != 0 :
                instance += f((0+i,j),step,"yellow");
            else:
                instance += f((0+i,j),step,"blue");
    return instance

g = make_instance(make_square) + make_instance(make_sector)
g.show(viewer='threejs',aspect_ratio=[1,1,1],axes=False,frame=False,online=True)
```

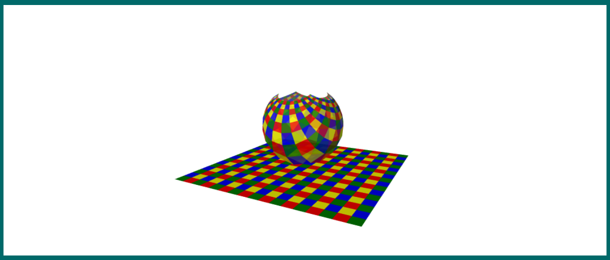
The function ix_3 is the same as iz but it acts on points in $\mathbb{R}^2$, instead of having arity 2 with arguments x, and y, it has only the single argument z which must be pair.

The function make_square makes a square, but also it can be extended to a constructor that maps the image of a square under a complex function by applying a map before doing the inclusion on the edges.

The function make_sector makes the image of a square under the inverse stereographic projection, but again it can be extended to the image of a complex map on S^2 by applying a map before doing the inclusion.

The key step for the conclusion of this project is in the last line of the code. The keyword online=True, gives the compiler the instruction to include the sources of all the required dependencies in the HTML file of the Sage Notebook where this runs. On the Sage Notebook, I downloaded a HTML version of the .ipynb source file. Which then I managed to modify by deleting all the elements that weren't the visualization.

So, I got a standalone 3D visualization that can be zoomed in and zoomed out and rotated.



And actually the visualization can be found in the url, http://sphere.drakezhard.org

3 Conclusions — Future Work

While the project was quite ambitious, it lent itself quite well to computation. I've shown in the previous section that the code is easily extensible to arbitrary complex mappings.

However, a major thing about the YouTube video Möbius Transformations Revealed was that the shadow of a pattern on S^2 corresponds to a transformation of the plane as the S^2 gets translated and rotated.

So a natural direction of future work, would be to combine the ideas about tiling the plane and inverse projecting onto the sphere, and the code where I managed to make the sphere move. The challenge would be to rework the code into the other language.

4 References

Needham, T. Visual Complex Analysis. Oxford University Press, 1997.

Arnold, D; Rogness, J. Möbius Transformations Revealed. Notices of The AMS, Vol. 55, Num. 10.

Lee, X. Stereographic Projection. Online, 2016.
URL http://xahlee.info/MathGraphicsGallery_dir/sphere_projection/index.html