CONSTRUCTION OF CONVEX IDEAL POLYHEDRA IN HYPERBOLIC 3-SPACE

ALLEGRA ALLGEIER

Abstract. The purpose of this project is to provide a method to construct a convex ideal polyhedron in hyperbolic 3-space given a set of internal dihedral angles that belong to a unique convex ideal polyhedron up to isometry. Thorough constructions are done for hyperbolic polyhedra that have the same combinatorial structure as a cube or an octahedron.

Contents

1. Acknowledgments

I would like to thank the NSF for funding the REU program at UC Berkeley. Moreover, I am extremely grateful to Dr. Franco Vargas Pallete for his guidance, kindness, and patience. I am also thankful to Dr. Michele Intermont for being my SIP advisor and guiding me through the quarter. Last but not least, many of the nice figures, other than the plots generated by Python, were made by using the Mathematics Online Editor, Mathcha[7].

2. Introduction: A brief history on hyperbolic geometry

It is known that the theory of *hyperbolic geometry* started from discarding one of Euclid's five postulates of plane geometry. Borrowing the words of Trudeau[11], we

list these five postulates below.

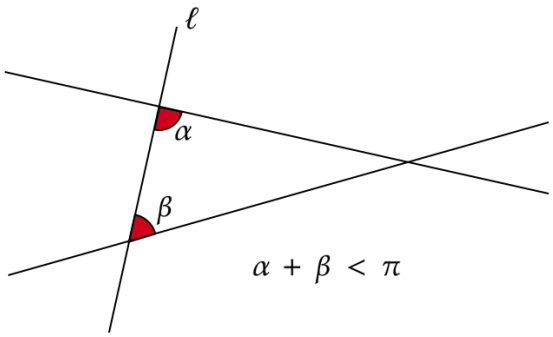


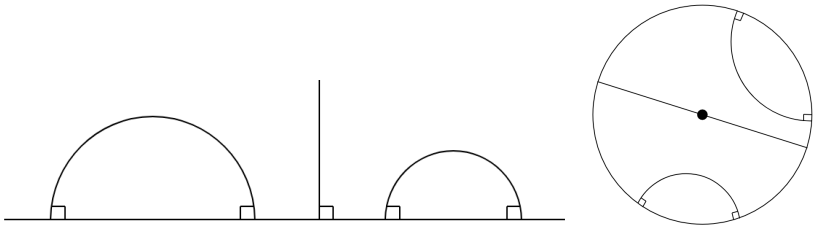
FIGURE 1. Fifth postulate

Postulate 1. There exists one and only one straight line that goes through two distinct points.
Postulate 2. Any finite straight line can be extended indefinitely from either end.
Postulate 3. There exists one and only one circle with a given center and a radius.
Postulate 4. All right angles are equal to one another.
Postulate 5. Suppose there is a straight line ℓ falling on two other straight lines. Suppose ℓ makes the sum of the interior angles on one of the sides less than π. Then if we extended the two straight lines indefinitely on that side, they will eventually meet. (Figure 1)

As one may think, the fifth postulate is not as simple as the others. In fact, it was thought to be the most unnatural among the five, and many mathematicians attempted to derive the fifth from the other four. The fifth postulate is also known as the *parallel postulate* as it was established that given the first four, the statement is equivalent to the following: Given a line and a point not on it, there is exactly one line going through the given point that is parallel to the given line[2]. However, after many unsuccessful attempts to derive the fifth postulate, mathematicians started to explore the theory that could be established by replacing it with its negation: Given a line and a point not on it, there is more than one line going through the given point that is parallel to the given line[2]. Such a change in paradigm was what lead to the discovery of hyperbolic geometry.

The remainder of our paper is organized as follows. In section 3, we will discuss some basic properties of hyperbolic space and its models. In sections 4 and 5, we will introduce and utilize a theorem by Rivin which provides us with a set of internal dihedral angles that belongs to a unique convex ideal polyhderon in $\mathbb{H}^3$(the definition of *convex* and *ideal* will be given in section 4). In sections 6 and 7, which are the main parts of this paper, we present a method to construct a convex ideal polyhedron given a set of dihedral angles that belongs to a unique convex ideal polyhdron. In section 8 we present our conclusions along with future research directions, and in section 9 we give further explanations to certain ideas discussed in the main part of the paper, and also present some additional work that has been done and also the code that was used throughout the project for obtaining the dihedral angles of convex ideal polyhedra and creating 3D images of the hyperbolic polyhedra.

3. Hyperbolic space and its models



(A) half-plane model (B) disk model

FIGURE 2

In order to have a more concrete understanding of hyperbolic geometry, we will start by exploring the hyperbolic plane $\mathbb{H}^2$. Although there are different models for $\mathbb{H}^2$, we will base our initial discussion on the *Poincaré half-plane model* defined on $\{(x, y) \in \mathbb{R}^2 : y > 0\}$ whose boundary is $\{(x, y) \in \mathbb{R}^2 : y = 0\}$ and the *Poincaré disk model* defined on $\{(x, y) \in \mathbb{R}^2 : x^2 + y^2 < 1\}$ whose boundary is $\{(x, y) \in \mathbb{R}^2 : x^2 + y^2 = 1\}$. Note that each of these boundaries are not a part of $\mathbb{H}^2$. In the upper half-plane model, the hyperbolic lines, or geodesics, are Euclidean semi-circles with their bases on the boundary and Euclidean vertical half-lines. In the disk model, the geodesics are Euclidean circular arcs orthogonal to the boundary and also the Euclidean diameters of the disk (see Figure 2). Furthermore, an angle between two curves is the same as the angle measured in Euclidean geometry in both models[3]. Thus, the circular arcs in Figure 2 (A) and the boundary, for example, in fact make a 90 degree angle in hyperbolic space.

We may now briefly come back to the fifth postulate. Given a hyperbolic line and a point P not on it, there can be more than one line that goes through P that is parallel to the given line as in Figure 3. Lines A and B are each parallel to the given line since they do not intersect with it in the upper half-plane. The given line and A are called *hyper-parallel* since they intersect at the boundary. The given line and B are called *ultra-parallel* since they do not intersect anywhere.

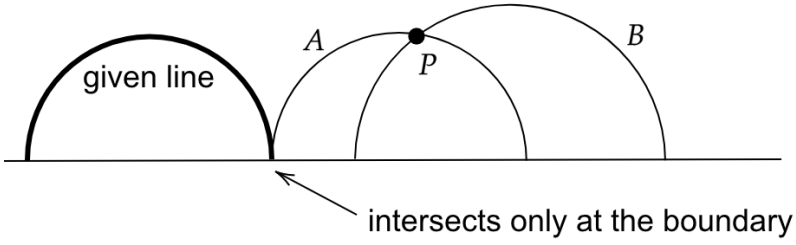


FIGURE 3. parallel lines in half-plane model

Now, we will go one step further and discuss the *metric* of $\mathbb{H}^2$. A metric is commonly called the "distance function" and gives the notion of distance to every pair of elements in a set. We will follow the definition of a *metric space* given in [4].

Definition 1. (Metric Space)
A metric space is a pair (X, d) consisting of a set X and a real function $d : X \times X \to \mathbb{R}$ (called the "metric"), such that:

(1) $d(x, y) \geq 0$ for all $x, y \in X$ and $d(x, y) = 0$ if and only if $x = y$.
(2) $d(x, y) = d(y, x)$ for all $x, y \in X$.
(3) $d(x, z) \leq d(x, y) + d(y, z)$ for all $x, y, z \in X$.

When discussing the metric of the half-plane model, its *Riemannian metric* expressed by the "differential line element"[8]

$$ds = \frac{\sqrt{dx^2 + dy^2}}{y}$$

is often presented instead of the metric function d. We have seen the symbol ds when calculating arc lengths in calculus classes. Let's try calculating the lengths of some curves in the upper half-plane to better understand the model.

Example 1. Length L_1 of the Euclidean line segment C_1 from (0, 1) to (0, 2). Since C_1 is a vertical line segment, $dx = 0$. Then

$$L_1 = \int_{C_1} ds = \int_1^2 \frac{\sqrt{dy^2}}{y} = \int_1^2 \frac{1}{y} dy = \ln 2.$$

Example 2. Length L_2 of the Euclidean line segment C_2 from (0, 3) to (0, 4).

$$L_2 = \int_{C_2} ds = \int_3^4 \frac{\sqrt{dy^2}}{y} = \int_3^4 \frac{1}{y} dy = \ln \frac{4}{3}.$$

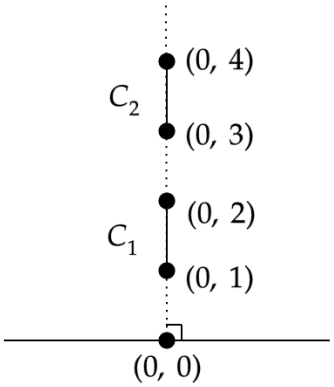


FIGURE 4. C_1 and C_2 in half-plane

We can see that although C_1 and C_2 have the same Euclidean length, C_1 has a greater hyperbolic length than C_2. More generally, the hyperbolic length L of a Euclidean line segment C from (x, a) to (x, b) is

$$L = \int_C ds = \int_a^b \frac{\sqrt{dy^2}}{y} = \ln \frac{b}{a}.$$

If $b = a + 1$, for example, we will have

$$L = \ln\left(1 + \frac{1}{a}\right).$$

Thus, although C has a Euclidean length of 1, the closer a is to the boundary, the greater its hyperbolic length L becomes. Although we only looked at a very particular type of curves in the half-plane, it is in fact true in general

that even if two points near the boundary looked close in the model, they may in fact be far apart. The disk model has similar properties and below is an image showing a tiling of the disk with congruent triangles of angles $\pi/2, \pi/3$, and $\pi/7$. The image is from [10].

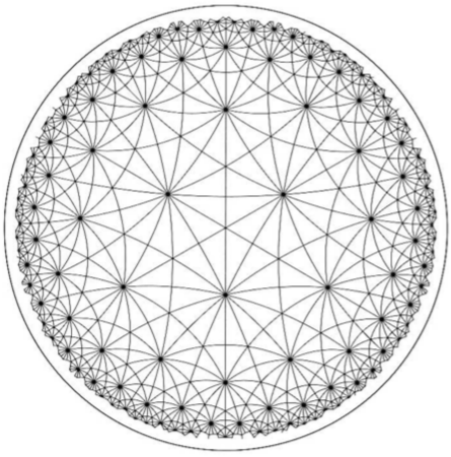


FIGURE 5. tiling of the disk [10]

As we move onto our main focus, which is the three-dimensional hyperbolic space $\mathbb{H}^3$, the models we introduce can be thought of as extensions of the ones for $\mathbb{H}^2$. We first have the *upper half-space model* defined on $\{(z,t) : z \in \mathbb{C}, t > 0\}$ with the boundary $\{(z,t) : z \in \mathbb{C}, t = 0\}$ and the *ball model* defined on $\{(x,y,z) \in \mathbb{R}^3 : x^2+y^2+z^2 < 1\}$ with the boundary $\{(x,y,z) \in \mathbb{R}^3 : x^2+y^2+z^2 = 1\}$. These models have analogous properties to the ones in the two-dimensional case. The following list is based on [5]. (1) The angle between two curves in the model is equal to the angle in Euclidean geometry; (2) In the upper half-space model, the hyperbolic planes are hemispheres with bases on the boundary and vertical half-planes. The lines are semicircles with bases on the boundary and vertical half-lines; (3) In the ball model, the hyperbolic planes are spherical caps orthogonal to the boundary and planes containing the center of the ball. The lines are circular arcs orthogonal to the boundary and spherical diameters.

Although we will not discuss the metrics of these models, the distance between a pair of points behaves in a similar way as in the two-dimensional case.

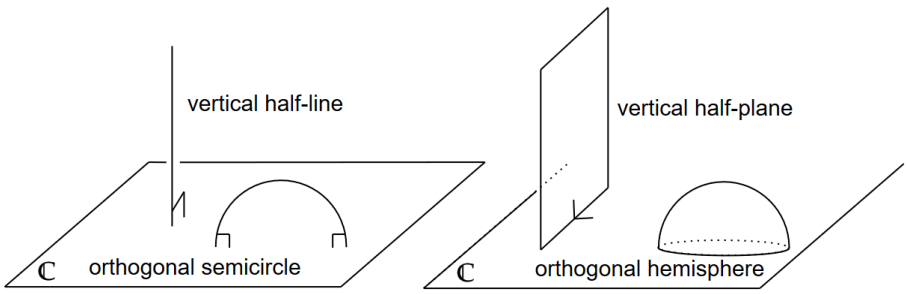


FIGURE 6. upper half-space model

4. Obtaining the dihedral angles of a convex ideal polyhedron

As discussed before, this section is for obtaining a set of internal dihedral angles that guarantees the existence and uniqueness of a convex ideal polyhedron in $\mathbb{H}^3$. Then in the sections that follow, we will present a method to actually construct a convex ideal polyhedron with such a set of dihedral angles.

In 1996, Igor Rivin characterized the dihedral angles of convex ideal polyhedra in $\mathbb{H}^3$. A *convex hyperbolic polyhedron* is a hyperbolic polyhedron such that the hyperbolic line connecting any pair of points in the polyhedron is contained in the polyhedron. An *ideal polyhedron* is a hyperblic polyhedron such that all of its vertices are on the boundary of the ball model or the upper half-space model.

Before we discuss Rivin's theorem in detail, there are a few definitions and concepts we state here for convenience. Let G be a planar graph. Then its *dual* graph G^* is defined as follows (we will follow the explanation in [1]). The vertices of G^* are centers, or any interior points, of the faces of G. Two vertices A and B of G^* are connected by k edges if and only if the corresponding faces in G had k edges in common (See Figure 7).

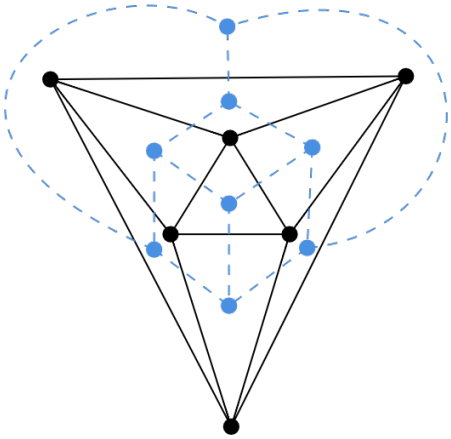


Figure 7. Octahedral graph G (black) and its dual cubical graph G^* (blue)

Continuing with our review of certain terms, a *walk* on a given graph is a sequence of vertices $v_1, v_2, ..., v_k$ such that (v_i, v_{i+1}) is an edge in the graph. A *closed walk* is a walk that starts and ends at the same vertex. Finally, a *simple circuit* is a closed walk that does not contain any repeated edges or vertices except the first and the last.

We now come back to Rivin's theorem. We will mostly quote from [3] but will paraphrase some parts while referring to [9] where Rivin presents the same theorem with slightly different terms.

Theorem 2 (Rivin). *Let P be a polyhedral graph with weights $w(e)$ assigned to the edges. Let P^* be the planar dual of P, where the edge e^* dual to e is assigned the dual weight $w^*(e^*)$. Then P can be realized as a convex ideal polyhedron in $\mathbb{H}^3$ with internal dihedral angle $w(e) = \pi - w^*(e^*)$ at every edge e if and only if the following conditions hold:*

Condition 1. $0 < w^*(e^*) < \pi$ for all edges e^* of P^*.

Condition 2. If the edges $e_1^*, e_2^*, ..., e_k^*$ form the boundary of a face of P^*, then $w^*(e_1^*) + w^*(e_2^*) + \cdots + w^*(e_k^*) = 2\pi$.

Condition 3. If $e_1^*, e_2^*, ..., e_k^*$ form a simple circuit which does not bound a face of P^*, then $w^*(e_1^*) + w^*(e_2^*) + \cdots + w^*(e_k^*) > 2\pi$.

Theorem 3 (**Rivin**). *A realization guranteed by Theorem 2 is unique up to isometries of* $\mathbb{H}^3$.

In the next section, we will show how we can find sets of edge weights $w^*(e^*)$ of an octahedral graph that satisfies the conditions 1-3 in Theorem 2.

5. Algorithm for finding Rivin's edge weights

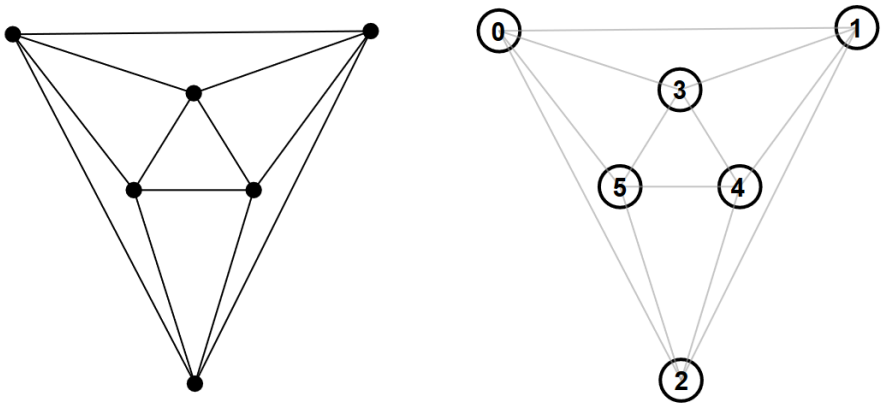


Figure 8. octahedral graph

We found an algorithm to count all of the simple circuits on the planar graphs of platonic solids. In short, the algorithm looks at all of the paths on the graph and then records the ones that can form simple circuits if we were to connect the endpoints. Let us explain in detail how the algorithm works for an octahedral graph. We begin by constructing an adjacency matrix of the vertices. Suppose the vertices are numbered as in Figure 8.

We start by looking at paths starting from vertex 0. Let `circuits` be a variable recording the simple circuits starting and ending at 0. Let `paths` be a variable recording the paths staring from 0.

First, find the vertices adjacent to 0 using the adjacency matrix. Update `paths` by recording all the distinct paths we get as we make the first step (at this point they are 0-1, 0-3, 0-5, 0-2). When we make the second step, we must avoid zero so we won't repeat an edge. Make the second step accordingly and update `paths`. For example, 0-1 will become 0-1-3, 0-1-4, or 0-1-2. From the third step onward, if any of the endpoints of the paths become adjacent to 0, we know we will get simple circuits in the next step. As we make that step, add those simple circuits to `circuits`. At the same time, update `paths` for the ones that didn't form simple circuits. For example, 0-1-3 will become 0-1-3-0, 0-1-3-4 or 0-1-3-5 in the next step. We add 0-1-3-0 to `circuits`, and update `paths` with 0-1-3-4 and 0-1-3-5. Since an octahedral graph has 6 vertices, we should have all simple circuits containing 0 after at most 6 steps in total. As we find all simple circuits containing 0, we must make sure we are not double counting. Since the algorithm is so far counting both a path and its

inverse, such as 0-1-3-4-5-0 and 0-5-4-3-1-0, we must remove one of them. After we do so, we will have all distinct simple circuits containing 0.

Let's move onto considering the simple circuits containing 1. The process is similar to the previous one. However, none of the paths starting from 1 can contain 0 since all of the simple circuits containing 0 have been counted. Thus, we want to count all the simple circuits containing 1 that do not contain 0. After that is done, we will count the simple circuits containing 2 that do not contain 0 or 1. Then we count the simple circuits containing 3 that do not contain 0, 1, or 2, and so on. In the end, we will have all distinct simple circuits on the octahedral graph. The table below shows the counts produced by our program based on the algorithm we presented.

Number of simple circuits				
tetrahedron	cube	octahedron	dodecahedron	icosahedron
7	28	63	1168	12878

The simple circuits we find will be used to build a linear system based on Rivin's theorem as we assign variables to the edges of the graph. In the end, we will obtain a set of external dihedral angles of a convex ideal polyhdron. Since solving such a system is a very hard problem in itself, we chose to find finitely many solutions to the system in order to complete the construction.

6. Constructing a convex ideal cube in $\mathbb{H}^3$

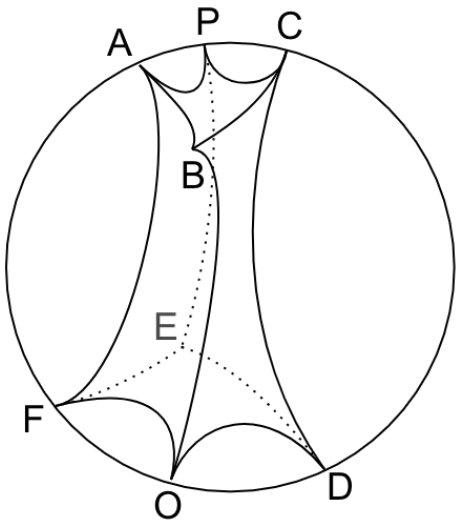


Figure 9. hyperbolic cube in the ball model

Let's now construct a convex ideal cube with prescribed internal dihedral angles. We would like to clarify that we'll be using the word cube to mean *cuboid* from now on. A cuboid is a polyhedron with the same combinatorial structure as a cube. A cuboid has six faces consisting of four edges, and each vertex is incident to three faces.

Now, we would like to complete the construction in the upper half-space model since the calculations seem to be simpler than in the ball model. In the ball model, a hyperbolic cube may present itself as in Figure 9. However, it is not clear how it

may be described in the upper half-space model. Luckily, considering a mapping that takes the ball model to the upper half-space model can help us. The composition of spherical inversion and planar reflection we are about to present is a common mapping used to go from the ball model to the upper half-space model[5]. This mapping is invertible, preserves hyperbolic planes and lines, and preserves angles. Note that inversion in two dimensions will be discussed in section 9.1 of this paper, and the properties can be extended to three dimensions.

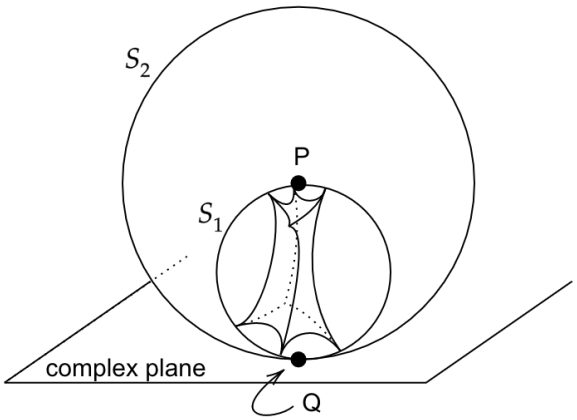


FIGURE 10. before inversion and reflection

We will call the spherical surface of the ball model S_1. Firstly, choose a vertex of the cube and draw a diameter of S_1. Let's call this chosen vertex P and the opposing point on the diameter Q. Then draw a sphere centered at P with radius PQ. We will call this sphere S_2. Place S_1 and S_2 on the complex plane so that P is in the upper half-space, Q is on the complex plane, and PQ is orthogonal to the complex plane (see Figure 10). Then invert the interior of the ball model with respect to S_2. Then reflect the image across the complex plane.

Although there are other variations, this procedure takes everything in the Poincaré ball model to the upper half-space model. We chose (0, 0, 0) to be Q and (0, 0, 2) to be P when plotting in $\mathbb{R}^3$.

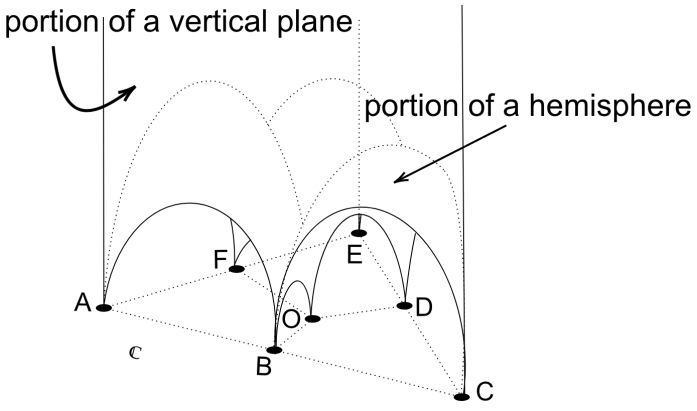


FIGURE 11. vertices on complex plane

Now, let f be the mapping described above. We can now consider how a cube in the upper half-space may present itself. Suppose we identified its vertices as in Figure 9 in the ball model. The mapping f takes a face not containing the center of

inversion P to a portion of a hemisphere in the upper half-space model. Moreover, a face containing P will become a portion of a vertical half-plane under f since it becomes a part of a half-plane after inversion and it must make a right angle with the boundary (because f preserves angles). Taking properties of inversion into account, we will see that the cube in Figure 9 will be expressed as in Figure 11 in the upper half-space model.

Figure 12 shows the vertices of the cube on the complex plane. Note that each of the edges AC (or ABC), CE (or CDE), and AE (or AFE) in Figure 12 is on the base of a vertical half-plane. The sets of four vertices $\{A, B, O, F\}$, $\{B, C, D, O\}$, and $\{D, E, F, O\}$ are each on the base of a hemisphere, and the circles drawn in Figure 12 represent those bases.

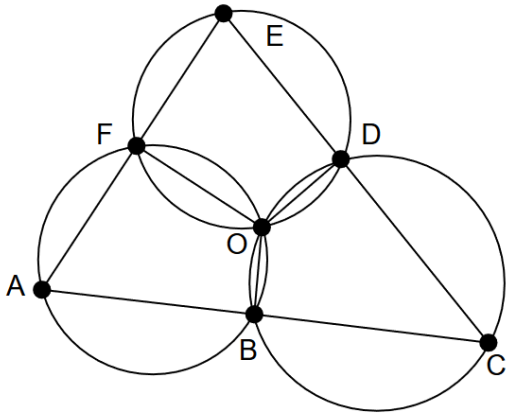
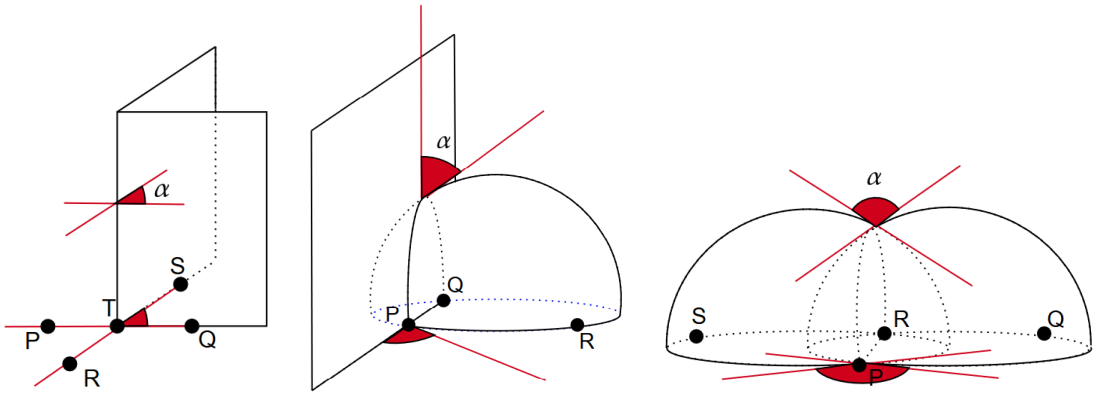


FIGURE 12. vertices on complex plane

Let's determine the positions of the vertices (up to isometry) with a given set of internal dihedral angles. We will need the following lemmas to translate the dihedral angles to planar angles that will appear in Figure 12. Note that in the following lemmas we are discussing objects in the upper half-space model of $\mathbb{H}^3$.



(A) plane-plane angle (B) plane-sphere angle (C) sphere-sphere angle

FIGURE 13

Lemma 4. *Let P, Q, R, S, T be distinct points on the complex plane. Suppose T is the intersection of the lines PQ and RS. Let α be a dihedral angle made by two*

orthogonal half-planes whose bases are the lines PQ and RS, respectively. Suppose $\angle STQ$ is in the same region as α (see Figure 13 (A)). Then $\alpha = \angle STQ$.

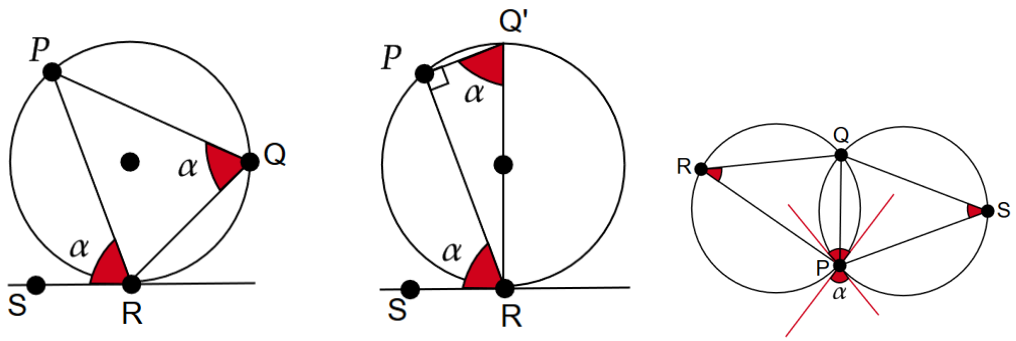
Proof. Euclidean translation preserves angles in the upper half-space model. Thus $\alpha = \angle STQ$. □

Lemma 5. *Let P, Q, R be distinct points on the complex plane. Let α be a dihedral angle made by an orthogonal plane whose base is the line PQ and a hemisphere whose base is the circle containing P, Q, R. Then α is equal to the angle made by the tangent of the circle at P and the line PQ on the same side as α (see Figure 13 (B)).*

Proof. Euclidean rotation preserves angles in the upper half-space model. □

Lemma 6. *Let P, Q, R, S be distinct points on the complex plane. Let α be a dihedral angle made by two intersecting hemispheres whose bases are circles containing P, R, S and P, Q, R, respectively. Then α is equal to the angle made by the tangents of these circles at P on the same side as α (Figure 13 (C)).*

Proof. Euclidean rotation preserves angles in the upper half-space model. □



(A) (B) (C)

FIGURE 14

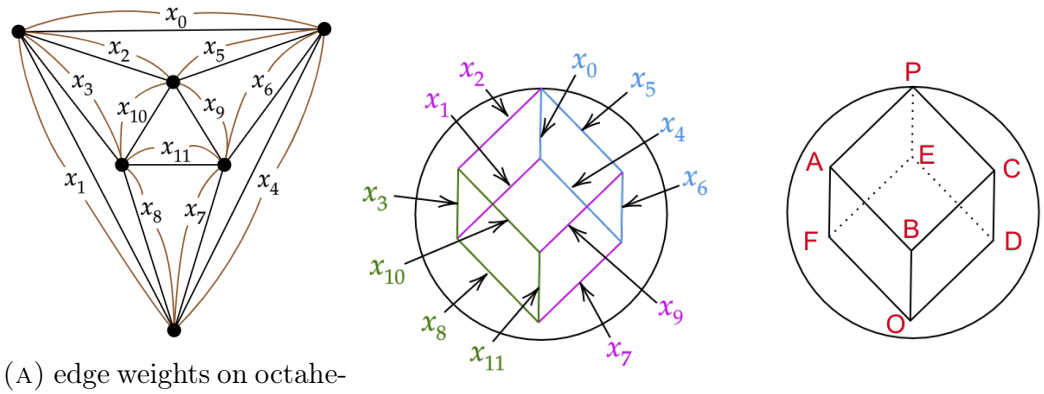
From now on, we will refer to the internal dihedral angle made by two orthogonal planes in the upper half space as plane-plane (P-P) angle, one made by an orthogonal plane and a hemisphere as plane-sphere (P-S) angle, and one made by two hemispheres as sphere-sphere (S-S) angle. Let's prove two more properties that will be used repeatedly.

Lemma 7. *Let a circle on the Euclidean plane be given. Let P, Q, R be points on the circle. Suppose this circle has a tangent at R (see Figure 14 (A)). Let S be on this tangent so that $\angle PRS$ does not overlap the edge QR. Then $\angle PQR = \angle PRS$.*

Proof. Fix PR and shift QR so it'll be a diameter (see Figure 14 (B)). Let Q' be the point we get after moving Q. Then $\angle PQR = \angle PQ'R$ because two inscribed angles with the same intercepted arc are equal. Moreover, $\angle Q'PR = \frac{\pi}{2}$. Then since $\angle Q'RS = \frac{\pi}{2}$, $\angle PQR = \angle PQ'R = \angle PRS$. □

Lemma 8. *Let P, Q, R, S be points on the Euclidean plane such that R and S are on different sides of the line PQ. Consider the circles containing P, Q, R and P, Q, S, respectively. Let α be the angle made by the tangents of the circles at P in the exterior of both circles. Then $\angle PRQ + \angle PSQ = \alpha$ (see Figure 14 (C)).*

Proof. The result follows from applying Lemma 5 to the tangents at P, $\triangle PQR$, and $\triangle PQS$. □



(A) edge weights on octahedron

(B) external dihedral angles

FIGURE 15

Now, suppose we assigned the variables $x_0, x_1, ..., x_{11}$ to the edges of an octahedral graph as in Figure 15 (A) and obtained weights $w^*(e^*)$ that satisfies Rivin's theorem. Then we can give corresponding edges of the hyperbolic cube the external dihedral angles $x_0, x_1, ..., x_{11}$ (Figure 15 (B)). Let $y_i = \pi - x_i, i = 0, 1, ..., 11$. Then y_i is the internal dihedral angle at each edge. Then these internal dihedral angles of the cube will be identified in the upper half-space model as in figure 16. Then by using the lemmas we proved, we obtain Figure 17.

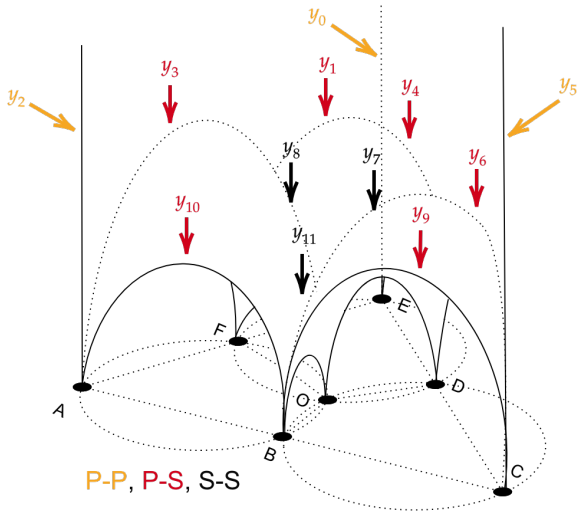


FIGURE 16. Internal dihedral angles identified

Here, y_0, y_2, y_5 are the plane-plane angles, and $y_1, y_3, y_4, y_6, y_9, y_{10}$ are the plane-sphere angles, and y_7, y_8, y_{11} are the sphere-sphere angles. Note that we added additional edges FB, BD, and DF represented by the dashed lines and assigned variables to the new angles as follows: $\angle OFB = a, \angle OBF = b, \angle OBD = c, \angle ODB = d, \angle ODF = e, \angle OFD = f$. Having these angles will help us locate the edges OB, OD, OF. As we employ Lemma 8 for the sphere-sphere dihedral angles, we obtain the following system.

$$(6.1)\qquad \text{P-P: } \begin{cases} a+b=y_2 \\ c+d=y_5 \\ e+f=y_0 \end{cases} \quad \text{S-S: } \begin{cases} a+d=y_{11} \\ c+f=y_7 \\ e+b=y_8 \end{cases} \quad \text{P-S: } \begin{cases} a+f+y_4+y_{10}=\pi \\ d+e+y_9+y_1=\pi \\ b+c+y_3+y_6=\pi. \end{cases}$$

This system of equations gives one free variable (we chose f). Then the solution in terms of f is

$$a = y_{11} - y_5 + y_7 - f,\ b = y_8 - y_0 + f,\ c = y_7 - f,\ d = y_5 - y_7 + f,\ e = y_0 - f.$$

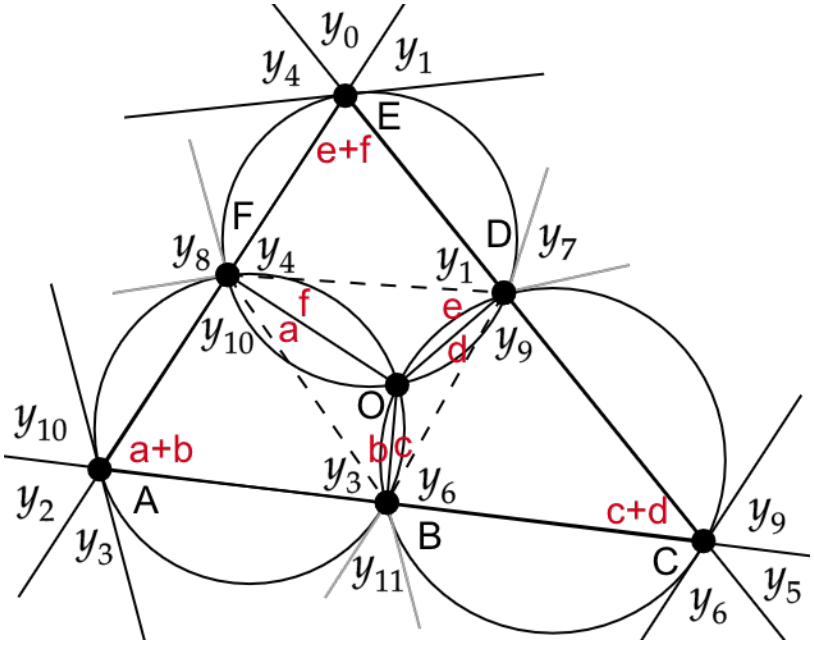


FIGURE 17. Vertices on complex plane with angle information

There must be a unique value for f since a convex polyhedron with prescribed dihedral angles obtained from Theorem 2 is unique as stated in Theorem 3. We also observed that $\triangle FBD$ was not always forming when f was left as a free variable. For example, a situation similar to Figure 18 was happening. Here, F' represents the point that was supposed to coincide with F.

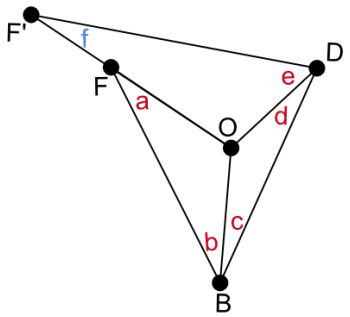


FIGURE 18. Triangle FBD not closing

Since we know $\triangle FBD$ must exist because vertices F, B, D are distinct, and since the edges FB, BD, DF are the edges of $\triangle$ FBD, we must find an f value that will

form $\triangle$ FBD. This will happen when $OF = OF'$. Since we have defined e to be the existing angle made by the edges OD and DF (O is in the interior of $\triangle$ FBD), if DF' does not coincide with DF, then e does not satisfy its definition. Now, referring to Figure 18, it follows that the relationship

$$OF' = OF \cdot \frac{\sin a \cdot \sin c \cdot \sin e}{\sin b \cdot \sin d \cdot \sin f}$$

must hold. Since we want $OF = OF'$, referring to the previous equation, it follows that we need

$$\sin f \cdot \sin d \cdot \sin b - \sin e \cdot \sin c \cdot \sin a = 0. \tag{6.2}$$

In the end, by solving 6.1 and 6.2 together, we are able to get the desired unique solution.

Now we can move onto actually drawing the cube. Since we only care about the shape up to isometry, we can choose an arbitrary length for one of the edges. We are placing A at the origin and B at (5, 0) for the example. Note that since the complex plane can be treated in the same way as $\mathbb{R}^2$, we are using $\mathbb{R}^2$ for the plotting. Starting from the edge AB, we obtain the rest of the edges and the vertices by the angles. For our demonstration, we are using the values below and got Figure 19 through plotting in Python.

$x_0 = 1.9748981268459183$, $x_1 = 2.7384076996659408$, $x_2 = 2.2979863709366652$,
$x_3 = 1.4516735513314263$, $x_4 = 1.5698794806677308$, $x_5 = 2.0103008093970063$,
$x_6 = 2.322152710378157$, $x_7 = 2.391153116133702$, $x_8 = 2.0931040561822227$,
$x_9 = 1.9507317874044263$, $x_{10} = 2.5335253849114983$, $x_{11} = 1.7989281348636652$

As we take the figure to three dimensions and add the spherical faces, we obtain Figure 20 (A). We can now complete the plot in the upper half-space by adding the orthogonal faces (Figure 20 (B)).

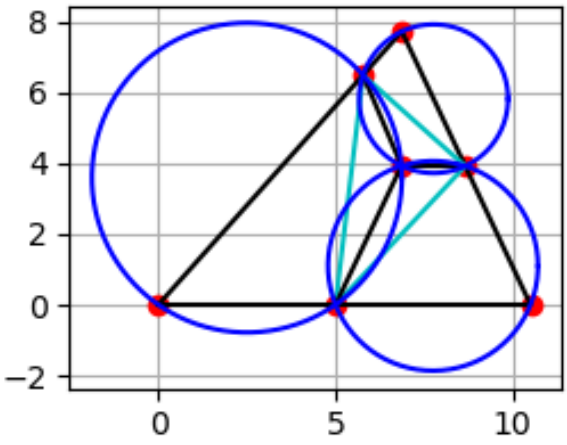
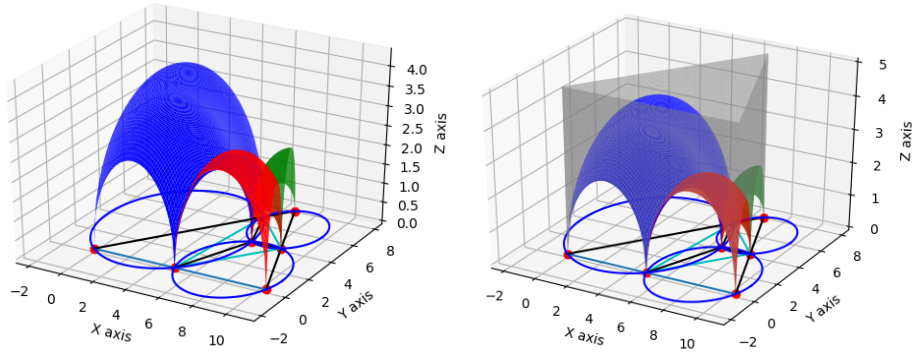


FIGURE 19. vertices on complex plane

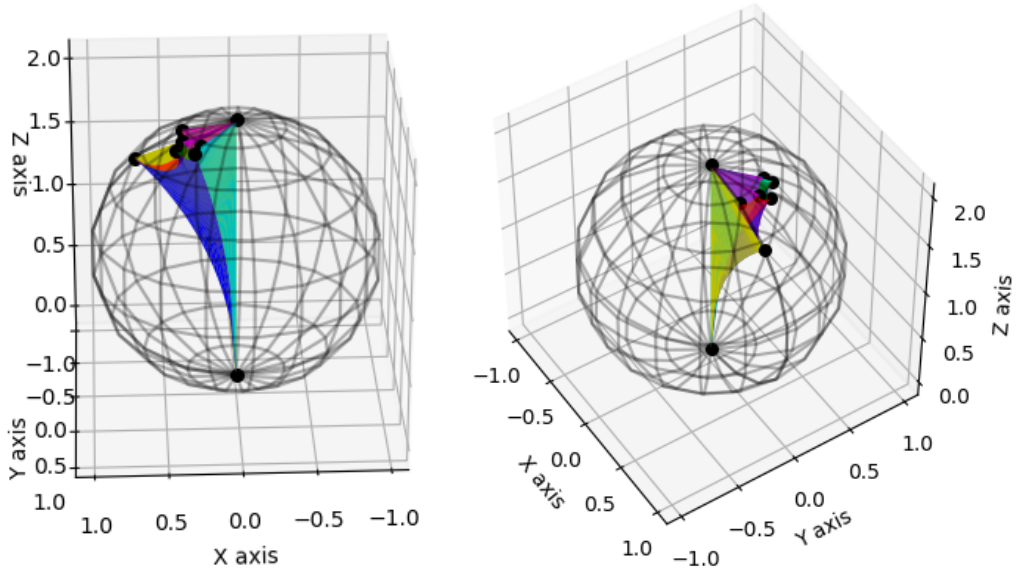
Finally, let's take the cube in the upper half-space model to the Poincaré ball model. First, reflect the cube in Figure 20 across the complex plane (or the xy plane)



(A) spherical faces (B) all faces

FIGURE 20. cube in the upper half-space model

and then conduct spherical inversion in terms of the sphere with radius 2 and center (0, 0, 2). Note that the ball model is a sphere of radius 1 and we are choosing to place its center at (0, 0, 1). In the end, we get figure 21 .



(A) (B) from a different angle

FIGURE 21. cube in the ball model

7. CONSTUCTING A CONVEX IDEAL OCTAHEDRON IN $\mathbb{H}^3$

We would like to clarify in this section as well that when we say “octahedron" we are referring to a polyhedron with the same combinatorial structure as an octahedron.

We can construct a convex ideal octahedron by following similar steps as in section 6. We put the details of the initial steps in the appendix(section 9.2) since it is similar to the case for constructing a convex ideal cube. Now, after we assign the variables as in the appendix, we get Figure 22. Each arrow in the figure indicate the internal dihedral angle between two spherical faces intersecting at a semi-circle whose base corresponds to the edge the arrow is pointing at.

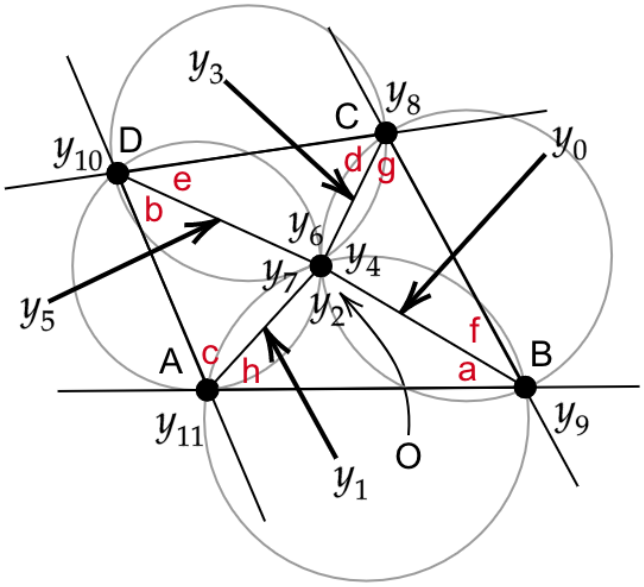


FIGURE 22. vertices on complex plane with angle information

Assigning angles $a, b, ..., h$ in a similar fashion as in the case for the cube, we obtain the following system.

$$\text{(7.1)} \qquad \text{P-P: } \begin{cases} c+h=y_{11} \\ b+e=y_{10} \\ d+g=y_8 \\ a+f=y_9 \end{cases} \quad \text{S-S: } \begin{cases} a+b=y_1 \\ c+d=y_5 \\ e+f=y_3 \\ g+h=y_0 \end{cases} \quad \text{P-S: } \begin{cases} e+d+y_6=\pi \\ b+c+y_7=\pi \\ a+h+y_2=\pi \\ f+g+y_4=\pi \end{cases}$$

Solving this system gives one free variable. As we choose h as the free variable, we get

$$a=\pi-y_2-h, \qquad b=y_1+y_2-\pi+h, \quad c=y_{11}-h, \quad d=y_5-y_{11}+h,$$
$$e=y_{10}-y_1-y_2+\pi-h, \quad f=\pi-y_0-y_4+h, \quad g=y_0-h.$$

Similar to the case with the hyperbolic cube, we know that the quadrilateral $ABCD$ must exist. Thus

$$\text{(7.2)} \qquad \sin b \cdot \sin d \cdot \sin f \cdot \sin h \cdot -\sin c \cdot \sin e \cdot \sin g \cdot \sin a = 0.$$

Solving this last equation gives us a unique h as we restrict h to be between 0 and π. As a side note, so far we have found one free variable in each of the linear systems we have built while constructing an octahedron and a cube (6.1 and 7.1). We will further discuss this matter of free variables in section 8.

Now, we can draw the octahedron. Below is the solution we are using for our example. Figure 23 shows where the vertices will be, and Figure 24 is the plot with the spherical faces. Figure 24 (C) is the complete plot with the orthogonal faces. By taking the object from upper half-space model to the ball model, we will get Figure 25.

$$x_0 = 1.4979245357535738, \quad x_1 = 1.6741479146682856, \quad x_2 = 1.562470918969697,$$
$$x_3 = 1.3755342183140784, \quad x_4 = 1.9821811657682555, \quad x_5 = 1.735578638443652,$$
$$x_6 = 0.7011902235351827, \quad x_7 = 2.0373429989064546, \quad x_8 = 2.2242796995620733,$$
$$x_9 = 1.2406086866880637, \quad x_{10} = 1.8090734462943001, \quad x_{11} = 1.0092234746351527.$$

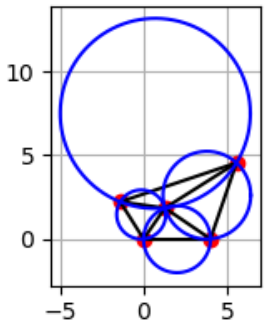
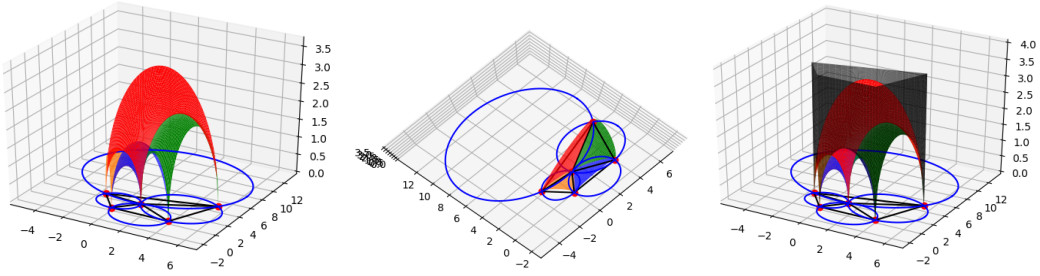
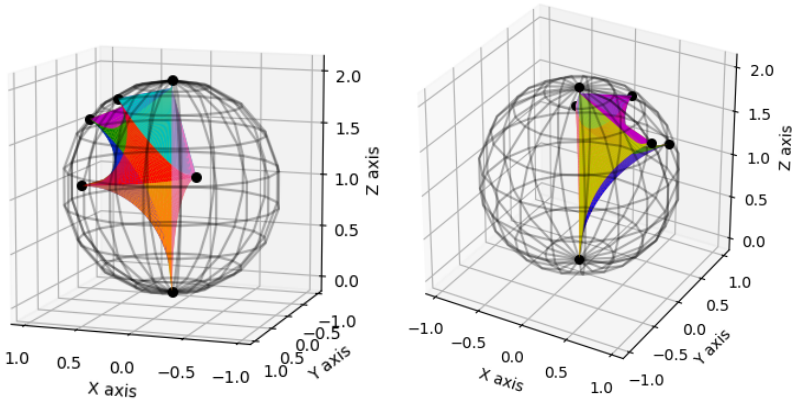


FIGURE 23. vertices on complex plane



(A) spherical faces (B) different angle (C) all faces

FIGURE 24. Octahedron in upper half-space



(A) (B) different angle

FIGURE 25. octahedron in ball model

8. Conclusions and future directions

We have presented a method to construct convex ideal cubes and octahedra given sets of internal dihedral angles. One interesting topic to explore in the future may be the number of free variables that arise from the linear systems such as 6.1 and 7.1. Knowing the number of free variables in a linear system may help us predict how many nonlinear equations we may need in the end as we try to find the necessary planar angles. As we will discuss in section 9.3, the linear system we obtain when constructing a dodecahedron has ten free variables, which matches with the number of vertices in the interior of the outermost triangle OPS as seen in Figure 29. Considering we have one free variable in both 6.1 and 7.1 and the interiors of the outermost triangle ACE in Figure 17 and the quadrilateral $ABCD$ in Figure 22 each containing one vertex, we may informally state the following conjecture.

> Conjecture: The number of free variables of a linear system constructed from the planar angle variables (such as a, b, c, d, e, f in Figure 17) is equal to the number of vertices in the interior of the outermost polygon created by the edges that connect the vertices of the convex ideal polyhedron in the upper half-space model when one of the vertices is at infinity.

Moreover, another question to be answered is whether there is a systematic way to know from where the nonlinear equations will arise. For the cube and the octahedron, we found the nonlinear equations by considering the forming of triangle FBD (Figure 17) and quadrilateral $OBCD$ (Figure 22,), respectively. These conditions were easy to find since we only needed one more equation aside form the linear system we had. However, what would we do if we had ten free variables as in the case for the dodecahedron (section 9.3)? It may be useful to investigate this problem in the future.

9. Appendix

9.1. **Inversion.** We will discuss the two dimensional case based on [10]. In the Poincaré disk model, a hyperbolic reflection about the diameter is equivalent to conducting a Euclidean reflection across the same diameter. However, hyperbolic reflections about other lines (circular arcs orthogonal to the boundary of the disk) must be dealt with separately. It turns out that these hyperbolic reflections are equivalent to a transformation called *inversions* in the Euclidean plane.

Definition. (image of a point under Euclidean inversion)
Consider a circle with center O and radius r. If a point P is not O, the image of P under inversion with respect to the given circle is the point P' lying on the ray OP such that $OP \cdot OP' = r^2$.

Inversions have many nice properties.

Proposition.
Let C be a circle in the Euclidean plane. Let i_C be the inversion with respect to C. Then i_C is conformal*, that is, it preserves local angles. Moreover, i_C maps circles not containing the center of C to circles, circles containing the center to lines, lines not containing the center to circles containing the center, and lines containing the center to themselves.*

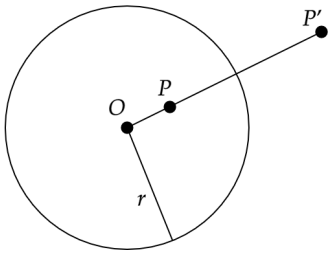
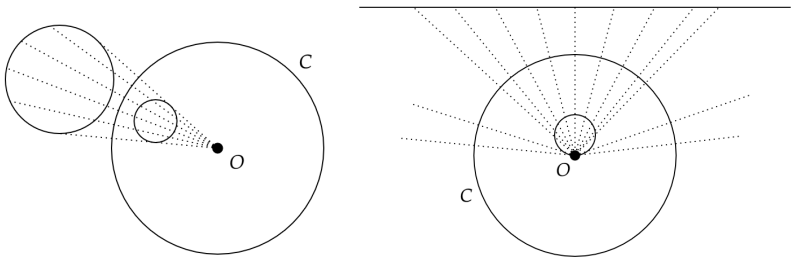


FIGURE 26. Inversion

Proof. See Thurston[10] for a proof. □

We will explain what a conformal mapping is by following the definition and explanation given in [6]. Recall that an angle between curves is defined as the angle made by their tangents at the point of intersection. Suppose we have two curves S_1 and S_2 intersecting at a point p. Assuming these curves are smooth at p, let T_1 and T_2 be their tangents at p. Define θ to be the acute angle from T_1 to T_2. It follows that this angle will have a sign attached to it since we may go clockwise or anti-clockwise from T_1 to T_2. Under a conformal mapping, the images of S_1 and S_2 will still make an angle of θ at the image of p, and this will be true for any pair of curves that go through any point p in the domain.

We provide some visual aid below to understand how hyperbolic lines may be preserved (Figure 27).



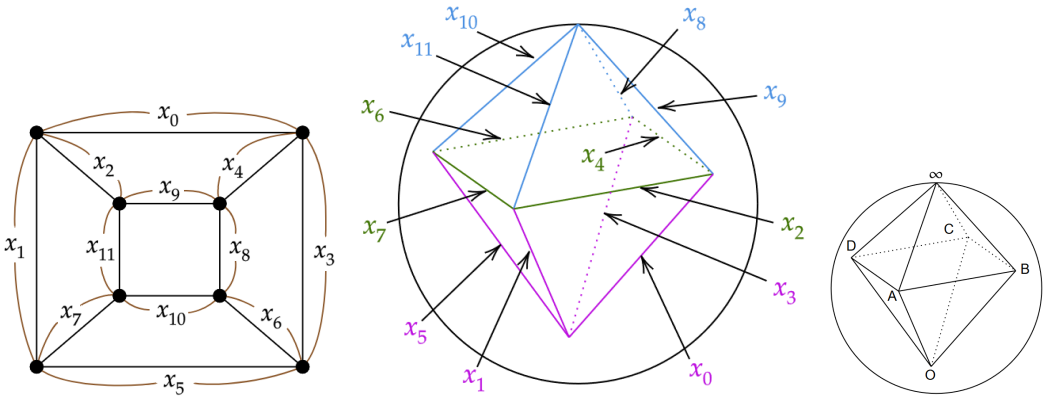
(A) circle not containing center to circle

(B) line not containing center to circle containing center

FIGURE 27

9.2. **Octahedron: variable assignment.** We present in detail the initial steps for constructing a convex ideal octahedron as described in section 7. We assign variables $x_0, x_1, ..., x_{11}$ to the edges of a a cubical graph as in Figure 28 (A) and obtain weights satisfying Rivin's theorem through solving a system of linear inequalities and equalities. Then we give corresponding edges of the hyperbolic octahedron external dihedral angles $x_0, x_1, ..., x_{11}$ (Figure 28 (B)). Let $y_i = \pi - x_i, i = 1, 2, ..., 11$, and y_i is the internal dihedral angle at each edge. By using Lemmas 2, 3, 4, 6, we get Figure 22.

9.3. **Dodecahedron: variable assignment and equations.** We can build a system of linear equations for the dodecahedron as well. Again, to clarify, when we say dodecahedron we are referring to a polyhedron with the same combinatorial structure as a dodecahedron. In Figure 29, the y's are the internal dihedral angles, and how



(A) edge weights on cube (B) external dihedral angles (C) vertices

FIGURE 28

they are situated is based on how we assign the edge weight variables to its planar dual, which we won't present here. The a's are the variables we assign to the angles in the figure similar to how we assign $a, b, ..., f$ to the cube. In this case, we assign thirty-six variables $a_0, a_1, ..., a_{35}$ and get a system of thirty-six equations from the $P-P$, $S-S$, and $P-S$ angles along with the conditions to be satisfied by the internal angles of triangles and so forth. However, we end up having ten free variables which curiously corresponds to the number of vertices inside the outermost triangle OSP.

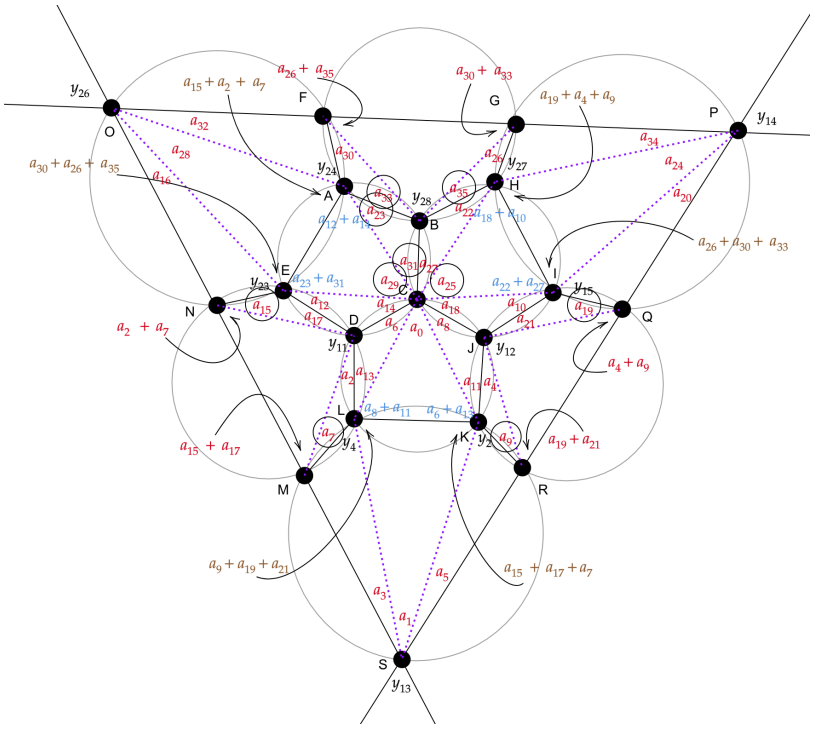


FIGURE 29. dodecahedron

9.4. **Icosahedron: variable assignment and equations.** Again, we use the word icosahedron to refer to a polyhedron with the same combinatorial structure as an

icosahedron. In Figure 30, we assigned forty angle variables $a_0, a_1, ..., a_{39}$ and found forty equations. We did not proceed any further since we expected many free variables similar to the case with the dodecahedron.

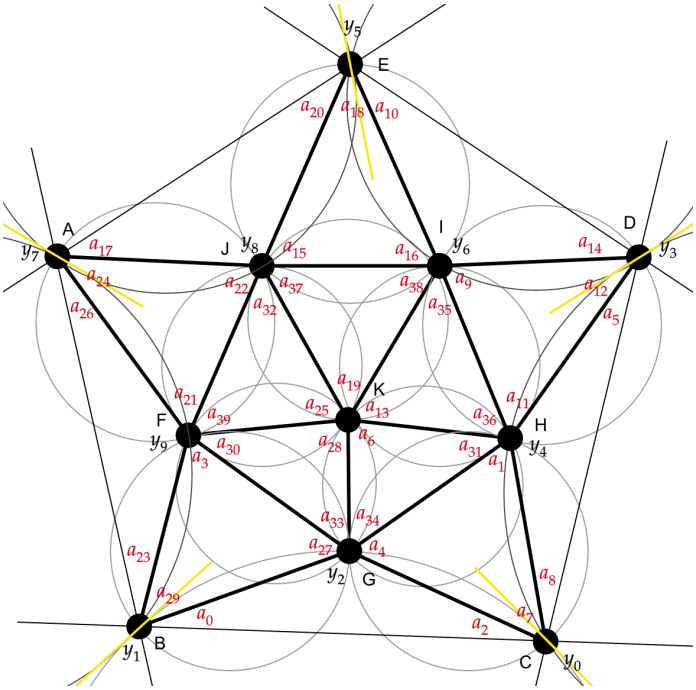


FIGURE 30. icosahedron

9.5. **Python code for counting simple cycles.** This is the code we used to count the simple cycles on a planar graph of a platonic solid. (Python 3.6)

```
import numpy as np
"""
This function returns the adjacency matrix of the vertices of an octahedron.
"""
def createMatrix():
    #returns the adjacency matrix of the vertices of an octahedronn
    return  [[0, 1, 1, 1, 0, 1],[1, 0, 1, 1, 1, 0],[1, 1, 0, 0, 1, 1],
            [1, 1, 0, 0, 1, 1],[0, 1, 1, 1, 0, 1],[1, 0, 1, 1, 1, 0]]

#w is the number of vertices that bounds a single face
w = int(input("Number of vertices bounding a face? "))
#v is the number of all the vertices on the polyhedra
v = len((createMatrix()[0]))

"""
This function returns the edges of a graph.
Each edge is represented as a set the contains two vertices.

parameters:
    n - number of vertices on the polyhedra
"""
def getEdges(n):
    adjacency = createMatrix()
    edges = []
    for i in range(len(adjacency)):
        for j in range(i+1, n):
            if adjacency[i][j] == 1:
                edges.append(set([i, j]))
    return edges
```

```
print("-Number of edges:")
print(len(getEdges(v)))
print("-Edges:")
print(getEdges(v))

"""
This function returns the list of vertices that are adjacent to the chosen vertex.

parameters:
    n - number of all the vertices on the polyhedra
    vertex - vertex of our choice
return:
    list of vertices
"""
def adjacentVertices(n, vertex):
    adjacency = createMatrix()
    adjacentVertices = []
    for i in range(n):
        if adjacency[vertex][i] == 1:
            adjacentVertices.append(i)
    return adjacentVertices

"""
This function returns all the simple cycles that begin and end with a vertex of our choice
while avoiding specified vertices.

parameters:
    n - number of vertices
    startingVertex - vertex to start our cycle
    avoid - list of vertices we want to avoid (an empty list will
    give all the cycles that start and end at startingVertex)

return:
    list of lists of vertices that make up simple cycles
"""
def fixedVertexCycles(n, startingVertex, avoid):
    #completecycles records cycles that start and end at startingVertex that don't contain verticies in avoid.
    completecycles = []
    #path starts from startingVertex
    #go to vertices adjacent to startingVertex that's not in avoid
    adjacent = adjacentVertices(n, startingVertex)
    #We want to make the first step outside of the loop because we need
    #a list of lists. We don't want a list of integers.
    afterFirstStep = []
    for i in adjacent:
        if i not in avoid:
            afterFirstStep.append([startingVertex, i])
    #We want to make the first step outside of the loop in order to avoid
    #a "cycle" with 3 vertices that go 0 -> adjacent vertex -> 0 (a digon like situation).
    afterSecondStep = []
    for i in range(len(afterFirstStep)):
        endpt = afterFirstStep[i][len(afterFirstStep[i])-1]
        adjacent = adjacentVertices(n, endpt)
        for j in adjacent:
            if j != startingVertex and j not in avoid:
                afterSecondStep.append(afterFirstStep[i]+[j])
    #Change variable name
    previouspaths = afterSecondStep
    #We choose n for the range below because we already made 2 steps and
    #a walk cannot take steps longer than the number of all vertices.
    for i in range(n):
        #latestpaths will record the updates of the paths
        latestpaths = []
        for j in range(len(previouspaths)):
            #We will check that adjacent vertices of the endpoints of the paths
            endpoint = previouspaths[j][len(previouspaths[j])-1]
            adjacent = adjacentVertices(n, endpoint)
```

```
            for k in adjacent:
                if k not in avoid:
                    if k not in previouspaths[j]:
                        latestpaths.append(previouspaths[j]+[k])
                    elif k == startingVertex:
                        completecycles.append(previouspaths[j]+[k])
        previouspaths = latestpaths.copy()
    #at this point, completecycles contain paths that are merely the reversed version of another cycle.
    #We must remove them.
    indicies = []
    for i in range(len(completecycles)):
        reversedCycle = completecycles[i][::-1]
        for j in range(i+1,len(completecycles)):
            if completecycles[j] == reversedCycle:
                indicies.append(i)
    uniquecycles = list(np.array(completecycles)[indicies])
    #make sure all elements are lists
    uniquecycles = [list(x) for x in uniquecycles]
    return uniquecycles

"""
This function returns the list of all distinct simple cycles on the 1-skeleton of the given polyhedron.
Parameters:
    n - number of vertices in a graph
#Explained to Rainie
#explained to Franco
"""
def allSimpleCycles(n):
    allCycles = []
    avoid = []
    for startingVertex in range(n):
        allCycles = allCycles + fixedVertexCycles(n, startingVertex, avoid)
        avoid.append(startingVertex)
    return allCycles
print("-Number of simple cycles (total):")
print(len(allSimpleCycles(v)))
print("-List of simple cycles (total):")
print(allSimpleCycles(v))

"""
This function assigns variables to each face.
"""
def makeVariables(n):
    edges = getEdges(n)
    variables = []
    for i in range(len(edges)):
        variables.append("x"+str(i))
    return variables

"""
 !!! Caution !!!
 !!!This only works for the platonic solids!!!
 This function returns all distinct cycles that bound a single face.

 parameters:
    n - number of vertices
    numVertexForFace - the number of vertices needed to bound a single face

 *Note: For numVertexForFace, actually put the number of vertices around a face.
 The path will contain an additional vertex
 (hence numVertexForFace+1 in the code).
"""
def boundingFaces(n, numVertexForFace):
    allCycles = allSimpleCycles(n)
    boundingFacelist = []
    for i in range(len(allCycles)):
        if len(allCycles[i]) == numVertexForFace+1:
            boundingFacelist.append(allCycles[i])
```

```
    return boundingFacelist

#print("Number of cycles bounding a face:")
#print(len(boundingFaces(v, w)))

print("-Number of simple cycles bounding a face:")
print(len(boundingFaces(v, w)))

"""
This function returns the list of simple cycles that don't bound a single face.
"""
def notBoundingFace(n, numVertexForFace):
    allSimpleCycleslist = allSimpleCycles(n)
    notBoundingFacelist = []
    for i in range(len(allSimpleCycleslist)):
        if len(allSimpleCycleslist[i]) > numVertexForFace+1:
            notBoundingFacelist.append(allSimpleCycleslist[i])
    return notBoundingFacelist

print("-Number of simple cycles not bounding a face:")
print(len(notBoundingFace(v, w)))

print("-Number of inequalities (including edge weight restrictions:)")
print(len(notBoundingFace(v, w))+len(getEdges(v)))


"""
This function prints equations coming from the bounding face condition.
"""
def getEquationsBoundingFace(n, numVertexForFace):
    edges = getEdges(n)
    variables = makeVariables(n)
    cycles = []
    boundingFace = boundingFaces(n, numVertexForFace)
    #[[0]*4 for x in range(4)]
    matrix = [[0]*len(edges) for x in range(len(boundingFace))]
    for i in range(len(boundingFace)):
        cycle = []
        for j in range(len(boundingFace[i]) - 1):
            edge = set([boundingFace[i][j], boundingFace[i][j + 1]])
            for k in range(len(edges)):
                if edge == edges[k]:
                    cycle.append(variables[k])
                    matrix[i][k] = 1
        cycles.append(sorted(cycle))
    for i in range(len(cycles)):
        print('+ '.join(cycles[i]) + " == 2*np.pi:")
    print("-Left side of equations to matrix:")
    print(matrix)
print("-Equations and inequalities:")
getEquationsBoundingFace(v, w)

"""
This function prints the inequalities for
simple circuits not bounding a face.
"""
def getInequalitiesNotBoundingFace(n, numVertexForFace):
    cycles = []
    edges = getEdges(n)
    variables = makeVariables(n)
    nboundingFace = notBoundingFace(n, numVertexForFace)
    for i in range(len(nboundingFace)):
        cycle = []
        for j in range(len(nboundingFace[i]) - 1):
            edge = set([nboundingFace[i][j], nboundingFace[i][j + 1]])
            for k in range(len(edges)):
                if edge == edges[k]:
                    cycle.append(variables[k])
```

```
        cycles.append(sorted(cycle))
    for i in range(len(cycles)):
        print("if " + '+ '.join(cycles[i]) + " > 2*np.pi:")
        print("    count += 1")

getInequalitiesNotBoundingFace(v, w)

"""
This function prints the inequalities satisfied
by each edge weight.
"""
def getedgeWeightRange(n):
    variables = makeVariables(n)
    for i in variables:
        if i != variables[len(variables)-1]:
            print("if " + "0 < "+str(i)+" and " + str(i) + " < np.pi:")
            print("    count += 1")

        else:
            print("if " + "0 < "+str(i)+" and " + str(i) + " < np.pi:")
            print("    count += 1")
            print("{", ', '.join(makeVariables(v)), "}")

getedgeWeightRange(v)
```

9.6. **Python code for visualizing convex ideal cube in the upper-half space model and the ball model.** (Python 3.6)

```
import numpy as np
import math
from sympy import Symbol, solve
from sympy import Matrix, S, linsolve, symbols
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
from mpl_toolkits.mplot3d import Axes3D
from sympy import Matrix, S, linsolve, symbols
import random
"""
These codes are for visualizing a convex ideal CUBE in HYPERBOLIC 3-space in the upper half-space model
and the ball model given a set
of external dihedral angles obtained from Rivin's theorem.
"""
externalAngles = [1.9748981268459183, 2.7384076996659408, 2.2979863709366652, 1.4516735513314263, 1.5698794806677308,
 2.0103008093970063, 2.322152710378157, 2.391153116133702, 2.0931040561822227, 1.9507317874044263, 2.5335253849114983,
  1.7989281348636652]

"""
This function returns the internal dihedral angles given the external dihedral angles.
"""
def get_internal_angles(extAngles):
    intAngles = [np.pi - extAngle for extAngle in extAngles]
    return intAngles

"""
Function representing the left side of equation 6.2
"""
def F(f):
    return np.sin(f)*np.sin(y[5]-y[7]+f)*np.sin(y[8]-y[0]+f) - np.sin(y[0]-f)*np.sin(y[7]-f)*np.sin(y[11]-y[5]+y[7]-f)

"""
Get possible solutions to equation F(f) = 0.
Since the solving function is unstable(nonlinear), we must take this step.
The solution function will take initial value in range(-10, 10).
Although it is not 100% guaranteed that it will always give the f we want, it is reasonable to assume
it will do so most of the time.
"""
def get_possible_fValues():
```

```
    possible_f_sols = []
    for i in range(-10, 10):
        possible_f_sols.append(list(fsolve(F, i)))
    return possible_f_sols

"""
The f values obtained from the previous function may not be between
0 and pi. This function makes sure f is in that range.
The solving function itself may give multiple f values. From my tests
they seem to be almost the same values and can be ignored.
"""
def check_f_appropriate():
    possible_f_sols = get_possible_fValues()
    appropriate_f_sols = []
    for i in range(len(possible_f_sols)):
        if 0 < possible_f_sols[i][0] and possible_f_sols[i][0] < np.pi:
            appropriate_f_sols.append(possible_f_sols[i][0])
    return appropriate_f_sols

"""
Give a, b, c, d, e, f corresponding to the f value obtained from solving the equation.
a, b, c, d, e, f, must be between 0 and pi.
"""
def check_abc():
    appropriate_f_sols = check_f_appropriate()
    fGivesabc = []
    for i in range(len(appropriate_f_sols)):
        f = appropriate_f_sols[i]
        a = y[11] - y[5] + y[7] - f
        b = y[8] - y[0] + f
        c = y[7] - f
        d = y[5] - y[7] + f
        e = y[0] - f
        count = 0
        if 0 < a and a < np.pi:
            count += 1
        if 0 < b and b < np.pi:
            count += 1
        if 0 < c and c < np.pi:
            count += 1
        if 0 < d and d < np.pi:
            count += 1
        if 0 < e and e < np.pi:
            count += 1

        if count == 5:
            fGivesabc.append([a, b, c, d, e, f])
    return fGivesabc

#Get the internal dihedral angles
y =  get_internal_angles(externalAngles)
#Get only the first one because the distinct solution sets are practically the same
abc =  check_abc()[0]

a = abc[0]
b = abc[1]
c = abc[2]
d = abc[3]
e = abc[4]
f = abc[5]

################################################################################
#VERTICES ON COMPLEX PLANE IN UPPER HALF_SPACE
################################################################################

"""
This function rotates a vector v = [v1, v2] with an angle of k (anticlockwise is positive).
"""
```

```
def rotate_vector(v1, v2, k):
    return np.array([math.cos(k)*v1 - math.sin(k)*v2, math.sin(k)*v1 + math.cos(k)*v2])

"""
Set triangle A, B, F.
"""
def get_A():
    return np.array([0, 0])
def get_B():
    return np.array([5, 0])

#line containing A, B. This will be the x-axis. This lines is parametrized.
def f1(t):
    return np.array([t, 0])

#line containing A, F. This lines is parametrized.
def f3(t):
    A = get_A()
    vecAF = rotate_vector(1, 0, y[2])
    return np.array([A[0] + t*vecAF[0], A[1] + t*vecAF[1]])

#line containing B, F. This lines is parametrized.
def g1(t):
    B = get_B()
    vecBF = rotate_vector(-1, 0, -y[3])
    return np.array([B[0] + t * vecBF[0], B[1] + t * vecBF[1]])

#Get the coordinates of F.
def get_F():
    s = Symbol("s")
    t = Symbol("t")
    sol = list(solve([f3(s)[0]-g1(t)[0], f3(s)[1]-g1(t)[1]], s, t, set=True)[1])[0]
    s = sol[0]
    t = sol[1]
    return [f3(s), s, t]


"""
Get triangle O, F, B.
"""
#line containing F, O. This lines is parametrized.
def h3(t):
    B = get_B()
    F = get_F()[0]
    vecFB = [B[0]-F[0], B[1]-F[1]]
    vecFO = rotate_vector(vecFB[0], vecFB[1], a)
    return np.array([F[0] + t*vecFO[0], F[1] + t*vecFO[1]])

#line containing B,O. This lines is parametrized.
def h1(t):
    A = get_A()
    B = get_B()
    vecBA = [A[0]-B[0], A[1]-B[1]]
    vecBO = rotate_vector(vecBA[0], vecBA[1], -(y[3]+b))
    return np.array([B[0] + t*vecBO[0], B[1] + t*vecBO[1]])

def get_O():
    s = Symbol("s")
    t = Symbol("t")
    sol = list(solve([h1(s)[0]-h3(t)[0], h1(s)[1]-h3(t)[1]], s, t, set=True)[1])[0]
    s = sol[0]
    t = sol[1]
    return [h1(s), s, t]


"""
Get triangle O, F, D.
```

```
"""
#line containing F, D. This lines is parametrized.
def g3(t):
    F = get_F()[0]
    O = get_O()[0]
    vecFO = [O[0]-F[0], O[1]-F[1]]
    vecFD = rotate_vector(vecFO[0], vecFO[1], f)
    return np.array([F[0] + t*vecFD[0], F[1] + t*vecFD[1]])

#line containing B, D. This lines is parametrized.
def g2(t):
    A = get_A()
    B = get_B()
    vecBA = [A[0]-B[0], A[1]-B[1]]
    vecBD = rotate_vector(vecBA[0], vecBA[1], -(y[3]+b+c))
    return np.array([B[0]+t*vecBD[0], B[1]+t*vecBD[1]])

def get_D():
    s = Symbol("s")
    t = Symbol("t")
    sol = list(solve([g3(s)[0]-g2(t)[0], g3(s)[1]-g2(t)[1]], s, t, set=True)[1])[0]
    s = sol[0]
    t = sol[1]
    return [g3(s), s, t]


"""
Get h2.
"""
#line containing O, D. This lines is parametrized.
def h2(t):
    O = get_O()[0]
    F = get_F()[0]
    vecOF = [F[0]-O[0], F[1]-O[1]]
    vecOD = rotate_vector(vecOF[0], vecOF[1], -(np.pi-e-f))
    return np.array([O[0] + t*vecOD[0], O[1] + t*vecOD[1]])

def get_t_for_h2_O_to_D():
    D_x = get_D()[0][0]
    t = Symbol("t")
    return solve(h2(t)[0]-D_x, t)[0]

"""
Complete the big triangle
"""
#line containing E, C. This lines is parametrized.
def f2(t):
    A = get_A()
    F = get_F()[0]
    D = get_D()[0]
    #Note that E, F, A are on the same line.
    vecFA = [A[0]-F[0], A[1]-F[1]]
    vecEC = rotate_vector(vecFA[0], vecFA[1], y[0])
    return np.array([D[0] + t*vecEC[0], D[1] + t*vecEC[1]])

def get_E():
    s = Symbol("s")
    t = Symbol("t")
    sol = list(solve([f3(s)[0]-f2(t)[0], f3(s)[1]-f2(t)[1]], s, t, set=True)[1])[0]
    s = sol[0]
    t = sol[1]
    return [f3(s), s, t]

def get_C():
    s = Symbol("s")
    t = Symbol("t")
    sol = list(solve([f1(s)[0]-f2(t)[0], f1(s)[1]-f2(t)[1]], s, t, set=True)[1])[0]
    s = sol[0]
```

```
    t = sol[1]
    return [f1(s), s, t]

"""

This function calculates the distance between two points.

"""
def distance(P, Q):
    return math.sqrt((P[0]-Q[0])**2 + (P[1]-Q[1])**2)

"""
This function returns the center of the circle that goes through the given three points.
DETAILS:
Two unknons t and s (parameters).
line1 = t*vecNorm1 + midpt1
line2 = s*vecNorm2 + midpt2
#x coordinate equal:
t*vecNorm1[0] + midpt1[0] = s*vecNorm2[0] + midpt2[0]
#y coordinates equal:
t*vecNorm1[1] + midpt1[1] = s*vecNorm2[1] + midpt2[1]
#So the system of equations to solve is:
t*vecNorm1[0] - s*vecNorm2[0] = midpt2[0] - midpt1[0]
t*vecNorm1[1] - s*vecNorm2[1] = midpt2[1] - midpt1[1]
"""
def get_Circle_center(P, Q, R):
    vecPQ = [Q[0]-P[0], Q[1]-P[1]]
    vecQR = [R[0]-Q[0], R[1]-Q[1]]
    #vector normal to vecPQ
    vecNorm1 = rotate_vector(vecPQ[0], vecPQ[1], np.pi/2)
    #vector normal to vecQR
    vecNorm2 = rotate_vector(vecQR[0], vecQR[1], np.pi/2)
    #midpoint  of PQ
    midpt1 = [(P[0] + Q[0]) / 2, (P[1] + Q[1]) / 2]
    #midpoint of QR
    midpt2 = [(Q[0] + R[0]) / 2, (Q[1] + R[1]) / 2]
    #line1 = t*vecNorm1 + midpt1
    #line2 = s*vecNorm2 + midpt2
    t, s = symbols("t, s")
    #Solve system of equations.
    A = Matrix([[vecNorm1[0], -vecNorm2[0]],
                [vecNorm1[1], -vecNorm2[1]]])
    b = Matrix([midpt2[0] - midpt1[0], midpt2[1] - midpt1[1]])
    sols = list(list(linsolve((A, b), [t, s]))[0])
    #put t = sols[0] into line1
    #Get center
    return [sols[0]*vecNorm1[0] + midpt1[0], sols[0]*vecNorm1[1] + midpt1[1]]

"""
This function will return the parametric function of the unique circle
that goes through points P, Q, R.
"""
def circle_function(t, P, Q, R):
    center = get_Circle_center(P, Q, R)
    radius = distance(P, center)
    return radius*np.cos(t)+center[0], radius*np.sin(t)+center[1]

"""
Below is the function that plots the vertices of the cube on the complex plane.
"""
def vertices_on_complex_plane():
    A = get_A()
    B = get_B()
    C = get_C()[0]
    D = get_D()[0]
    E = get_E()[0]
    F = get_F()[0]
    O = get_O()[0]
```

```
    #since circle is in parametric form, we want the input to be from 0 to 2pi.
    t_circ = np.arange(0, 2 * np.pi, 0.02)
    #Appropriate input value to show each line segment.
    t_f1 = np.arange(0, get_C()[1], 0.02)
    t_f3 = np.arange(0, get_E()[1], 0.02)
    t_g1 = np.arange(0, get_F()[2], 0.02)
    t_h1 = np.arange(0, get_O()[1], 0.02)
    t_h3 = np.arange(0, get_O()[2], 0.02)
    t_g3 = np.arange(0, get_D()[1], 0.02)
    t_g2 = np.arange(0, get_D()[2], 0.02)
    t_h2 = np.arange(0, get_t_for_h2_O_to_D(), 0.02)
    t_f2 = np.arange(get_E()[2],get_C()[2], 0.02)

    plt.figure(1)
    plt.subplot(211)
    plt.plot(A[0], A[1], 'ro')
    plt.plot(B[0], B[1], 'ro')
    plt.plot(F[0], F[1], 'ro')
    plt.plot(O[0], O[1], 'ro')
    plt.plot(D[0], D[1], 'ro')
    plt.plot(E[0], E[1], 'ro')
    plt.plot(C[0], C[1], 'ro')

    plt.plot(t_f1, [0]*len(t_f1), "k") #f1 is defined to be on the x-axis.
    plt.plot(f2(t_f2)[0], f2(t_f2)[1], "k")
    plt.plot(f3(t_f3)[0], f3(t_f3)[1], "k")
    plt.plot(g1(t_g1)[0], g1(t_g1)[1], 'c')
    plt.plot(g2(t_g2)[0], g2(t_g2)[1], 'c')
    plt.plot(g3(t_g3)[0], g3(t_g3)[1], 'c')
    plt.plot(h1(t_h1)[0], h1(t_h1)[1], "k")
    plt.plot(h2(t_h2)[0], h2(t_h2)[1], "k")
    plt.plot(h3(t_h3)[0], h3(t_h3)[1], "k")

    circle1 = circle_function(t_circ, A, B, O)
    circle2 = circle_function(t_circ, D, C, B)
    circle3= circle_function(t_circ,  F, E, O)
    plt.plot(circle1[0], circle1[1], "b")
    plt.plot(circle2[0], circle2[1], "b")
    plt.plot(circle3[0], circle3[1], "b")

    plt.gca().set_aspect('equal')
    plt.grid(True)

    #plt.legend()
    plt.show()

#vertices_on_complex_plane()

################################################################################
#UPPER HALF SPACE MODEL
################################################################################

"""
This function will give the parametric function for a spherical triangle.
IMPORTANT DETAILS:
Given 3 points T, U, V onj the complex plane, there is a unique circle that goes through
the three points and thus a unique half-sphere that has this circle as the base. Draw 3
planes that are orthogonal to the complex plane which base lines are line TU, UV, and VT.
The spherical triangle we want is the regioun surrounded by these 3 planes.
parameters:
    *Note: P is a point on the boundary or the interior of the triangle TUV
    mesh - the number of columns in a mesh
    points_on_TU - Number of points to have on line segment TU to make a mesh
    points_on_UP - Number of points to have on line segment UP to make a mesh
"""
def spherical_triangle(T, U, V):
    points_on_TU = 50
```

```
    points_on_UP = 50
    #We are building a mesh here. tt and ss will be the mesh.
    #interval is the numbers in [0, 1] that t will take.
    interval = np.linspace(0, 1, points_on_TU)
    tt = []
    ss = []
    for i in range(len(interval)):
        #By the property of similar triangles, s can take values 0 <= s <= t with 0 <= t <= 1.
        #Here, we are fixing t and obtaining points on the base of the triangle similar to TUV, which
        #side is length t*TU. Then we have UP = s*UV.
        s = list(np.linspace(0, interval[i], points_on_UP))
        t = list(np.array([interval[i]]*points_on_UP))
        tt.append(t)
        ss.append(s)
    #converting to arrays
    tt = np.array([np.array(x) for x in tt])
    ss = np.array([np.array(x) for x in ss])
    #P is an arbitrary point in the interior or the boundary of triangle TUV.
    #vecTP = t*vecTU + s *vecTV, 0<=t<=1, 0<=s<=t
    #Let O(0, 0, 0) and then
    #vecOP = t*vecTU + s *vecTV + vecOT, 0<=t<=1, 0<=s<=t
    vecTU = [U[0]-T[0], U[1]-T[1]]
    vecUV = [V[0] - U[0], V[1] - U[1]]
    #vecOP = [x, y, z].
    #The x, y, values P will take
    x = tt * vecTU[0] + ss * vecUV[0] + T[0]
    y = tt * vecTU[1] + ss * vecUV[1] + T[1]
    #Now, we will get the point on the sphere which has the same x, y coordinates as P.

    #Below is for obtaining the equation of the sphere which base is the circle going through T, U, V.
    #center of the circle containing T, U, V
    circleCenter = get_Circle_center(T, U, V)
    r = distance(T, circleCenter)

    #This is for obtaining the point on the sphere that has the same x, y coordinates as P.
    #!!!!!ATTENTION!!!!!+0.00001 is not suppose to be there but otherwise we were getting a complex number.
    #THUS, MAY NOT BE 100% ACCURATE.
    z = (r**2 - (x - circleCenter[0])**2 - (y - circleCenter[1])**2+0.00000001)**(1/2)
    #!!!!!!!ATTENTION!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    return [x.astype(float), y.astype(float), z.astype(float)]

"""
Funtion for cut plane (portion of vertical half plane consisting a face of the cube).
P, Q are the line segment from which the orthogonal cut plane extends.
By cut plane we meen not plotting the circle which is made by the sphere
and the plane intersecting.
"""
def orthogonal_CutPlane(P, Q, TrueIfUpperHalf):
    if (TrueIfUpperHalf == True):
        points_on_PQ = 20
        points_in_z_direction  = 20
        vecPQ = np.array([Q[0] - P[0], Q[1] - P[1]])
        t = np.linspace(0, 1, points_on_PQ)
        tt = np.array([np.array([t[i]] * points_in_z_direction) for i in range(len(t))])
        xx = vecPQ[0] * tt + P[0]
        yy = vecPQ[1] * tt + P[1]


        midpt = [(P[0] + Q[0]) / 2, (P[1] + Q[1]) / 2]
        r = distance(P, midpt)
        zz = []
        # z = midpt[2]+math.sqrt(r**2 - (xx-midpt[0])**2 - (yy-midpt[1])**2)
        for i in range(len(xx)):
            # !!!!!ATTENTION!!!!!+0.00001 is not suppose to be there but otherwise we were getting
            # a complex number(must be a computer issue about floats).
            #  This expect that this manipulation number would not affect the plot.
            zvalue = (r ** 2 - (xx[i][0] - midpt[0]) ** 2 - (yy[i][0] - midpt[1]) ** 2 + 0.00000000001) ** (1 / 2)
            # !!!!!!!ATTENTION!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
            zrange = np.linspace(float(zvalue), 5, points_in_z_direction)
            zz.append(zrange)
        zz = np.array(zz)
        #print(zz)
        return [xx.astype(float), yy.astype(float), zz.astype(float)]
    else:
        points_on_PQ = 50
        points_in_z_direction = 50
        vecPQ = np.array([Q[0] - P[0], Q[1] - P[1]])
        t = np.linspace(0, 1, points_on_PQ)
        tt = np.array([np.array([t[i]] * points_in_z_direction) for i in range(len(t))])
        xx = vecPQ[0] * tt + P[0]
        yy = vecPQ[1] * tt + P[1]

        midpt = [(P[0] + Q[0]) / 2, (P[1] + Q[1]) / 2]
        r = distance(P, midpt)
        zz = []
        # z = midpt[2]+math.sqrt(r**2 - (xx-midpt[0])**2 - (yy-midpt[1])**2)
        for i in range(len(xx)):
            # !!!!!ATTENTION!!!!!+0.00001 is not suppose to be there but otherwise we were getting a complex number.
            #THUS, MAY NOT BE 100% ACCURATE.
            zvalue = (r ** 2 - (xx[i][0] - midpt[0]) ** 2 - (yy[i][0] - midpt[1]) ** 2 + 0.000000000001) ** (1 / 2)
            # !!!!!!!ATTENTION!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            zrange = np.linspace(float(zvalue), 10000, points_in_z_direction)
            zz.append(zrange)
        zz = np.array(zz)
        return [xx.astype(float), yy.astype(float), zz.astype(float)]

"""
This function plots a convex ideal cube in the upper half-space.
"""
def plot_upper_half():
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    A = get_A()
    B = get_B()
    C = get_C()[0]
    D = get_D()[0]
    E = get_E()[0]
    F = get_F()[0]
    O = get_O()[0]

    #since circle is in parametric form, we want the input to be from 0 to 2pi.
    t_circ = np.arange(0, 2 * np.pi, 0.02)
    #Appropriate input value to show each line segment.
    t_f1 = np.arange(0, get_C()[1], 0.02)
    t_f3 = np.arange(0, get_E()[1], 0.02)
    t_g1 = np.arange(0, get_F()[2], 0.02)
    t_h1 = np.arange(0, get_O()[1], 0.02)
    t_h3 = np.arange(0, get_O()[2], 0.02)
    t_g3 = np.arange(0, get_D()[1], 0.02)
    t_g2 = np.arange(0, get_D()[2], 0.02)
    t_h2 = np.arange(0, get_t_for_h2_O_to_D(), 0.02)
    t_f2 = np.arange(get_E()[2],get_C()[2], 0.02)

    ax.plot([A[0]], [A[1]], [0],'ro', marker='o')
    ax.plot([B[0]], [B[1]], [0],'ro', marker='o')
    ax.plot([F[0]], [F[1]], [0],'ro', marker='o')
    ax.plot([O[0]], [O[1]], [0],'ro', marker='o')
    ax.plot([D[0]], [D[1]], [0],'ro', marker='o')
    ax.plot([E[0]], [E[1]], [0],'ro', marker='o')
    ax.plot([C[0]], [C[1]], [0],'ro', marker='o')

    ax.plot(f1(t_f1)[0], [0] * len(t_f1),[0]*len(t_f1))
    ax.plot(f2(t_f2)[0], f2(t_f2)[1], [0]*len(t_f2), "k")
    ax.plot(f3(t_f3)[0], f3(t_f3)[1], [0]*len(t_f3), "k")
    ax.plot(g1(t_g1)[0], g1(t_g1)[1], [0]*len(t_g1), 'c')
```

```
    ax.plot(g2(t_g2)[0], g2(t_g2)[1], [0]*len(t_g2), 'c')
    ax.plot(g3(t_g3)[0], g3(t_g3)[1], [0]*len(t_g3), 'c')
    ax.plot(h1(t_h1)[0], h1(t_h1)[1], [0]*len(t_h1), "k")
    ax.plot(h2(t_h2)[0], h2(t_h2)[1], [0]*len(t_h2), "k")
    ax.plot(h3(t_h3)[0], h3(t_h3)[1], [0]*len(t_h3), "k")

    circle1 = circle_function(t_circ, O, A, B )
    circle2 = circle_function(t_circ, D, C, B)
    circle3 = circle_function(t_circ, F, E, O)
    ax.plot(circle1[0], circle1[1], [0] * len(t_circ), "b")
    ax.plot(circle2[0], circle2[1], [0] * len(t_circ), "b")
    ax.plot(circle3[0], circle3[1], [0] * len(t_circ), "b")

    #planes
    planeAB = orthogonal_CutPlane(A, B, True)
    planeBC = orthogonal_CutPlane(B, C, True)
    planeCD = orthogonal_CutPlane(C, D, True)
    planeDE = orthogonal_CutPlane(D, E, True)
    planeEF = orthogonal_CutPlane(E, F, True)
    planeFA = orthogonal_CutPlane(F, A, True)

    ax.plot_surface(planeAB[0], planeAB[1], planeAB[2], color='tab:gray', alpha=0.5, shade=False)
    ax.plot_surface(planeBC[0], planeBC[1], planeBC[2], color='tab:gray', alpha=0.5, shade=False)

    ax.plot_surface(planeCD[0], planeCD[1], planeCD[2], color='tab:gray', alpha=0.5, shade=False)
    ax.plot_surface(planeDE[0], planeDE[1], planeDE[2], color='tab:gray', alpha=0.5, shade=False)

    ax.plot_surface(planeEF[0], planeEF[1], planeEF[2], color='tab:gray', alpha=0.5, shade=False)
    ax.plot_surface(planeFA[0], planeFA[1], planeFA[2], color='tab:gray', alpha=0.5, shade=False)

    #spheres
    testing = spherical_triangle(O, B, A)
    ax.plot_surface(testing[0], testing[1], testing[2], color="b", shade=False)

    testing = spherical_triangle(O, F, A)
    ax.plot_surface(testing[0], testing[1], testing[2], color="b", shade=False)

    testing = spherical_triangle(F, E, D)
    ax.plot_surface(testing[0], testing[1], testing[2], color="g", shade=False)

    testing = spherical_triangle(F, O, D)
    ax.plot_surface(testing[0], testing[1], testing[2], color="g", shade=False)

    testing = spherical_triangle(D, C, B)
    ax.plot_surface(testing[0], testing[1], testing[2], color="r", shade=False)

    testing = spherical_triangle(D, O, B)
    ax.plot_surface(testing[0], testing[1], testing[2], color="r", shade=False)

    ax.set_xlabel('X axis')
    ax.set_ylabel('Y axis')
    ax.set_zlabel('Z axis')
    plt.show()

#plot_upper_half()

################################################################################
#BALL MODEL
################################################################################

"""
This function will calcule the distance between points in 3D.
"""
def ThreeD_distance(P, Q):
    return ((P[0] - Q[0]) ** 2 + (P[1] - Q[1]) ** 2 + (P[2] - Q[2]) ** 2) ** (1 / 2)


"""
```

```
This function reflects a point across the complex plane.
Returns the coordinates of the image in a list
"""
def reflection_complex_plane(P):
    return [P[0], P[1], -P[2]]

"""
This function inverts a point with respect to the sphere with center (0, 0, 2) and radius 2.
Returns the point as a list.
"""
def inversion(P):
    center_of_inversion = np.array([0, 0, 2])
    vec_center_P = np.array(P) - center_of_inversion
    Q = vec_center_P * 4 / ((ThreeD_distance(center_of_inversion, P)) ** 2) + center_of_inversion
    return Q

"""
This function takes a point in the upper half-space model to the
ball model.
"""
def upper_half_to_ball_point(P):
    return inversion(reflection_complex_plane(P))

"""
This function takes a surface in the upper half-space model to the ball model
"""
def upper_half_to_ball_surface(mesh):
    mesh_listx = mesh[0]
    mesh_listy = mesh[1]
    mesh_listz = mesh[2]

    new_mesh_listx = []
    new_mesh_listy = []
    new_mesh_listz = []
    for i in range(len(mesh_listx)):
        listx = []
        listy = []
        listz = []
        for j in range(len(mesh_listx[i])):
            x = mesh_listx[i][j]
            y = mesh_listy[i][j]
            z = mesh_listz[i][j]
            oldpoint = [x, y, z]
            # reflect and invert
            new_point = upper_half_to_ball_point(oldpoint)
            listx.append(new_point[0])
            listy.append(new_point[1])
            listz.append(new_point[2])
        new_mesh_listx.append(np.array(listx))
        new_mesh_listy.append(np.array(listy))
        new_mesh_listz.append(np.array(listz))
    new_mesh_listx = np.array(new_mesh_listx)
    new_mesh_listy = np.array(new_mesh_listy)
    new_mesh_listz = np.array(new_mesh_listz)

    return [new_mesh_listx, new_mesh_listy, new_mesh_listz]

"""
This is a plot that brings the figure to the Poincare ball model.
"""
def ball_model():
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    A = get_A()
    B = get_B()
    C = get_C()[0]
    D = get_D()[0]
    E = get_E()[0]
```

```
F = get_F()[0]
O = get_O()[0]

# turn A, B, C into 3D coordinates since they only have x, y coordinates
O_3D = [O[0], O[1], 0]
A_3D = [A[0], A[1], 0]
B_3D = [B[0], B[1], 0]
C_3D = [C[0], C[1], 0]
D_3D = [D[0], D[1], 0]
E_3D = [E[0], E[1], 0]
F_3D = [F[0], F[1], 0]

# sphere of inversion
# u, v = np.mgrid[0:2 * np.pi:20j, 0:np.pi:10j]
# x = np.cos(u) * np.sin(v)
# y = np.sin(u) * np.sin(v)
# z = np.cos(v) + 1
# ax.plot_wireframe(x, y, z, color="k", alpha=0.3)

# planes
planeAB = upper_half_to_ball_surface(orthogonal_CutPlane(A, B, False))
planeBC = upper_half_to_ball_surface(orthogonal_CutPlane(B, C, False))
planeCD = upper_half_to_ball_surface(orthogonal_CutPlane(C, D, False))
planeDE = upper_half_to_ball_surface(orthogonal_CutPlane(D, E, False))
planeEF = upper_half_to_ball_surface(orthogonal_CutPlane(E, F, False))
planeFA = upper_half_to_ball_surface(orthogonal_CutPlane(F, A, False))

ax.plot_surface(planeAB[0], planeAB[1], planeAB[2], color='y', shade=False)
ax.plot_surface(planeBC[0], planeBC[1], planeBC[2], color='y', shade=False)

ax.plot_surface(planeCD[0], planeCD[1], planeCD[2], color='m', shade=False)
ax.plot_surface(planeDE[0], planeDE[1], planeDE[2], color='m', shade=False)

ax.plot_surface(planeEF[0], planeEF[1], planeEF[2], color='c', shade=False)
ax.plot_surface(planeFA[0], planeFA[1], planeFA[2], color='c', shade=False)

# spheres
face1 = upper_half_to_ball_surface(spherical_triangle(O, B, A))
ax.plot_surface(face1[0], face1[1], face1[2], color="b", shade=False)
face2 = upper_half_to_ball_surface(spherical_triangle(O, F, A))
ax.plot_surface(face2[0], face2[1], face2[2], color="b", shade=False)

face3 = upper_half_to_ball_surface(spherical_triangle(F, E, D))
ax.plot_surface(face3[0], face3[1], face3[2], color="g", shade=False)
face4 = upper_half_to_ball_surface(spherical_triangle(F, O, D))
ax.plot_surface(face4[0], face4[1], face4[2], color="g", shade=False)

face5 = upper_half_to_ball_surface(spherical_triangle(D, C, B))
ax.plot_surface(face5[0], face5[1], face5[2], color="r", shade=False)
face6 = upper_half_to_ball_surface(spherical_triangle(D, O, B))
ax.plot_surface(face6[0], face6[1], face6[2], color="r", shade=False)

# vertices after inversion
ball_O = upper_half_to_ball_point(O_3D)
ball_A = upper_half_to_ball_point(A_3D)
ball_B = upper_half_to_ball_point(B_3D)
ball_C = upper_half_to_ball_point(C_3D)
ball_D = upper_half_to_ball_point(D_3D)
ball_E = upper_half_to_ball_point(E_3D)
ball_F = upper_half_to_ball_point(F_3D)
ax.scatter([ball_A[0]], [ball_A[1]], [ball_A[2]], s=30, c='k', marker='o')
ax.scatter([ball_B[0]], [ball_B[1]], [ball_B[2]], s=30, c='k', marker='o')
ax.scatter([ball_F[0]], [ball_F[1]], [ball_F[2]], s=30, c='k', marker='o')
ax.scatter([ball_O[0]], [ball_O[1]], [ball_O[2]], s=30, c='k', marker='o')
ax.scatter([ball_D[0]], [ball_D[1]], [ball_D[2]], s=30, c='k', marker='o')
ax.scatter([ball_E[0]], [ball_E[1]], [ball_E[2]], s=30, c='k', marker='o')
ax.scatter([ball_C[0]], [ball_C[1]], [ball_C[2]], s=30, c='k', marker='o')
# inversion center
```

```
    ax.scatter([0], [0], [2], 'k', s=30, c='k', marker='o')

    ax.set_xlabel('X axis')
    ax.set_ylabel('Y axis')
    ax.set_zlabel('Z axis')
    ax.set_aspect('equal')
    plt.show()
\part{title}
ball_model()
```

References

[1] Bońa, M. (2017). *A walk through combinatorics an introduction to enumeration and graph theory*. World Scientific.

[2] Cannon, J. W., Floyd, W. J., Kenyon, R., & Parry, W. R. (1997). Hyperbolic geometry. *Flavors of Geometry*, *31*, 59–115.

[3] Hodgson, C. D., Rivin, I., & Smith, W. D. (1992). A characterization of convex hyperbolic polyhedra and of convex polyhedra inscribed in the sphere. *Bulletin of the American Mathematical Society*, *27*, 246–251.

[4] Jänich, K. (1984). *Topology*. Springer-Verlag New York Inc.

[5] Marden, A. (2007). *Outer circles: An introduction to hyperbolic 3-manifolds*. Cambridge: Cambridge University Press.

[6] Needham, T. (2012). *Visual complex analysis*. Clarendon Press.

[7] Nha, B. D., & Nhat, P. T. M. (2018). Mathematics online editor. Retrieved from https://www.mathcha.io/

[8] Ramsay, A., & Richtmyer, R. D. (1995). *Introduction to hyperbolic geometry*. Springer Science+Business Media New York.

[9] Rivin, I. (1996). A characterization of ideal polyhedra in hyperbolic 3-space. *The Annals of Mathematics*, *143*, 51–70.

[10] Thurston, W. P., & Levy, S. (1997). *Three-dimensional geometry and topology*. Princeton University Press.

[11] Trudeau, R. J. (1987). *The non-euclidean revolution*. Birkhäuser Boston.