Two-Point Resistances of Symmetric Polyhedral Networks

by
Kevin E. Stern

A Thesis
Submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Physics

April 28, 2022

APPROVED:

_____

Professor Padmanabhan K. Aravind, Thesis Advisor

_____

Professor Brigitte Servatius, Committee Chair

_____

Professor Herman Servatius, Committee Member

Abstract

A random walk along the edges of a polyhedron consists of starting at a particular vertex, choosing one of its adjacent neighbors at random, stepping to that neighbor, and then repeating the process until a specified vertex is reached. The average number of steps it takes to get from a given vertex to another is called the hitting time. For polyhedra with some degree of symmetry, there are groups of vertices that have the same hitting time from a given starting vertex. These groups form symmetrically equivalent layers within the polyhedron. This thesis explores a number of novel approaches to calculating these layers based on Euclidean distance, minimum length paths, and the number of edges of each face of the polyhedron. After the hitting times have been found, the equivalent resistance between any two vertices of the network is easily calculated. The physical relevance of this problem is discussed, a comparison with other approaches to solving it will be made and some open questions are mentioned.

Acknowledgements

I would like to extend my gratitude to my advisor, Professor Padmanabhan K. Aravind, for his suggestions and guidance throughout this project, Professor Brigitte Servatius for her insight about Lovász's hitting time equation, and Professor Andrea Arnold, for introducing me to MATLAB and all of the wonderful things that we can accomplish with it.

Contents

List of Tables

List of Figures

1 Introduction

A polyhedron is a three-dimensional object with two-dimensional polygonal faces, straight edges, and point vertices. A random walk along the edges of a polyhedron is essentially a list of not necessarily unique vertices of the polyhedron. A random walk is formed by starting at one vertex, recording it, and then choosing one of its adjacent neighbors at random to step to. Once a neighbor has been chosen, it is also recorded, and then one of its adjacent neighbors is chosen at random to step to. This Markov chain continues until some stopping criterion is met. In this thesis, we will number the vertices and arbitrarily choose vertex 1 to be the ending vertex. That is, every random walk will end once it reaches vertex 1. Since polyhedra form connected networks of vertices, every random walk is guaranteed to have finite length [1]. We call the average number of steps it takes to get from vertex a to the ending vertex b the hitting time $H(a, b)$ [2], also called the first-hitting [3] or stopping time [4]. This average can be found by summing the products of the lengths of every possible random walk with the probabilities that that random walk actually happens. The hitting time of a vertex to itself, $H(b, b)$, is equal to zero, since no steps are required. Random walks have been studied since 1905 [5], and in 1982 a list of almost 300 references was compiled [6], with countless papers since then. Random walks on polygons and polyhedra are a newer topic still, and dozens of papers on the subject have been published in recent years [7][8][9][10].

For polyhedra with some degree of symmetry, such as the Platonic and Archimedean solids, there will be groups of vertices with identical hitting times [11][12]. We call these groups symmetrically equivalent layers, and our main goal was to develop a strategy to quickly and accurately determine these layers. Our secondary goal was to then calculate the hitting times for all vertices in each of the five Platonic solids and thirteen Archimedean solids. This problem is identical to the problem of finding two-point resistances between arbitrary pairs of vertices of these polyhedra, given that the edges are identical 1Ω resistors. Resistors networks, involving either a finite or infinite number of resistors, have been studied for years for their various applications [13]. Polyhedra also provide examples of such networks, when their edges are modeled by identical resistors. This problem was solved a while back in the case of the Platonic solids [14], but this thesis aims to find a new method and to use it to obtain similar results for all the Archimedean solids.

The Platonic solids are made of congruent regular polygons meeting at identical vertices [15]. For example, the cube is made of six squares, with three squares meeting at each of the eight vertices, and the icosahedron is made of twenty equilateral triangles, with five triangles meeting at each of the twelve vertices. There are five Platonic solids, the tetrahedron, cube, octahedron, dodecahedron, and icosahedron, and all are highly symmetric, due to their vertex-, edge-, and face-transitivity. Full results for the hitting times and two-point resistances of the Platonic solids are given in Appendix A.

The Archimedean solids are composed of regular polygons meeting at identical vertices. These are much more varied than the Platonic solids, and can have as few as 12 vertices or as many as 120. For example, the cuboctahedron is made from six squares and eight equilateral triangles, with two squares and two triangles meeting at each of the 12 vertices. There are thirteen Archimedean solids, the truncated tetrahedron, cuboctahedron, truncated cube, truncated octahedron, rhombicuboctahedron, truncated cuboctahedron, snub cube, icosidodecahedron, truncated dodecahedron, truncated icosahedron, rhombicosidodecahedron, truncated icosidodecahedron, and snub dodecahedron. The Archimedean solids are not edge- or face-transitive, but they are vertex-transitive. The vertex-transitivity of the Platonic and Archimedean solids means that the numerical results obtained will be the same regardless of the ending vertex chosen. Full results for the Archimedean solids have been recently presented [16], but this thesis obtains the results by a rather different method and summarizes them in Appendix B.

The motivation behind this research is multifaceted. From a purely theoretical standpoint, finding the hitting times and two-point resistances between every pair of vertices on a variety of polyhedra is a very interesting problem. It also has direct relationships to other random walk problems [17], which have applications from diffusion [18] to polymer and solid-state physics [19][20] to electrical networks [21]. Possibly the newest and most exciting application of random walks is their connection to network analysis [22], a subtopic of graph theory that ties into geographic information systems (GIS) [23] and social networks [24], among other topics.

The polyhedra studied in this thesis are interesting in their own right. Since they are some of the most symmetric three-dimensional objects possible, any research contributing to the wealth of knowledge about these objects is potentially useful. Many of the solids investigated show up in chemistry and geology, such as buckminsterfullerene [25], which is a truncated icosahedron, and faujasite [26], which is a truncated octahedron. Additionally, some of the many variations on the Rubik's cube take the forms of various Platonic and Archimedean solids, including the tetrahedron, octahedron, icosahedron, dodecahedron, truncated tetrahedron, and rhombicuboctahedron. Due to their face-transitivity, some of the Platonic and Catalan solids also appear as both common and rare dice shapes, such as the rhombic dodecahedron and rhombic triacontahedron used as 12 and 30-sided dice, respectively.

The organization of this thesis is as follows. Chapter 2 discusses the rules required to split the vertices of a polyhedron into symmetrically equivalent layers. Chapter 3 gives the equation for calculating hitting times and examines two examples. Chapter 4 extends these methods to two of the Catalan solids, the rhombic dodecahedron and rhombic triacontahedron. Chapter 5 discusses possible avenues of future research. Finally, Chapter 6 provides an explanation for the MATLAB code given in Appendix C.

2 Symmetrically Equivalent Layers

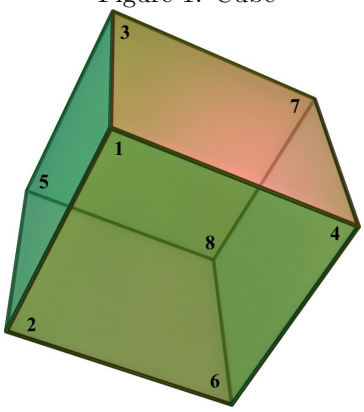
The symmetrically equivalent layers of a polyhedron are formed by groups of vertices that have identical hitting times relative the same ending vertex b. We denote these layers by a set $\{a_n\}$ such that $H(a_i, b)$ is the same for all $a_i \in \{a_n\}$. Vertices in the same layer must obey two rules. These rules have been developed experimentally throughout the course of this thesis. They accurately split the vertices of every Platonic and Archimedean solid into layers with identical hitting times. After transforming hitting times into two-point equivalent resistances, these results were corroborated with other sources [11][16]. Though these rules have only been confirmed for the Platonic and Archimedean solids, we conjecture that these rules would also accurately split the vertices of the Catalan and Johnson solids into layers, as discussed in Chapter 5. As far as we can tell, there are no other published results for the Catalan and Johnson solid to compare with, other than the rhombic dodecahedron and rhombic triacontahedron discussed in Chapter 4.

2.1 Rule 1

The first rule is that vertices in the same layer must be the same Euclidean distance from the ending vertex. The dot product between the coordinates of each pair of vertices can be used as a substitute for the distance as long as the vertices are all equidistant from the origin. In this case, the dot product will be a stand-in for the cosine of the angle between the vectors going from the origin to the two vertices in question. Since this angle must be between 0 and π, the cosine will be a strictly decreasing function, going from 1 for the dot product of a vertex with itself to -1 for the dot product of a vertex and its antipode (if the antipode exists). In this way, all of the vertices of a polyhedron may be arranged in order of distance from the ending vertex. This rule is not strictly necessary, as Rule 2 would split all of the vertices accurately on its own, but Euclidean distance provides a good initial splitting to get an idea of which vertices are closer or farther away.

This rule is enough to fully determine the layers of all five Platonic solids. If the coordinates of the cube are formed by the eight possible combinations of 1 and -1 for each of the three coordinates, then the dot products, hitting times, and two-point resistances are shown in Table 1. It is clear that there are four layers within the cube: the ending vertex, its neighbors, their neighbors, and the antipode of the ending vertex. These results can be confirmed by a random walk statistical model [27] or by a physical model with identical resistors [11][12]. For example, the nonzero equivalent resistances of the cube are $7/12\Omega$, $3/4\Omega$, and $5/6\Omega$.

Figure 1: Cube

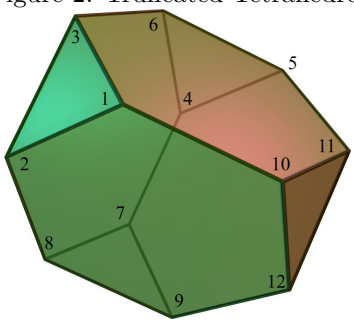


Vertex	Dot Product with Vertex 1	Hitting Time	Resistance (Ω)
1	3	0	0
2	1	7	7/12
3	1	7	7/12
4	1	7	7/12
5	-1	9	3/4
6	-1	9	3/4
7	-1	9	3/4
8	-3	10	5/6

Table 1: Layers of Cube by Dot Product

Just using the dot product, however, is not always enough to uncover all layers. For example, the truncated tetrahedron, shown in Figure 2, will not provide accurate layers with just the use of the dot product. For any vertex, its three neighbors are equal distances away, but are clearly not equivalent to each other. In the figure, vertex 1 is adjacent to vertices 2, 3, and 10, but the hitting time between either of vertices 2 and 3 and vertex 1 is 51/5 whereas the hitting time between vertices 10 and 1 is 63/5.

Figure 2: Truncated Tetrahedron



Using the dot product, the vertices would be sorted into the layers given in Table 2, whereas the vertices sorted by hitting time are shown in Table 3. It can clearly be seen that the dot product rule does not accurately split vertices 2, 3, and 10. Another rule must be added to avoid this undersplitting of layers.

Vertex	Dot Product with Vertex 1	Hitting Time	Resistance (Ω)
1	11	0	0
2	7	51/5	17/30
3	7	51/5	17/30
4	-9	99/5	11/10
5	-5	96/5	16/15
6	-1	87/5	29/30
7	-9	99/5	11/10
8	-1	87/5	29/30
9	-5	96/5	16/15
10	7	63/5	7/10
11	-1	87/5	29/30
12	-1	87/5	29/30

Table 2: Layers of Truncated Tetrahedron by Dot Product

Layer Number	Vertices in Layer	Hitting Time	Resistance (Ω)
1	1	0	0
2	2, 3	51/5	17/30
3	10	63/5	7/10
4	6, 8, 11, 12	87/5	29/30
5	5, 9	96/5	16/15
6	4, 7	99/5	11/10

Table 3: Hitting Times and Resistances of Truncated Tetrahedron

2.2 Rule 2

The second rule is that vertices in the same layer must have shortest paths of the same length to the ending vertex, and that those shortest paths must comprise the same types of edges. This might seem like two rules, but it is really the second part that separates the vertices into layers. The shortest path length, also known as geodesic distance, is the minimum number of steps it takes to get from a vertex to the ending vertex. For example, looking back at the cube in Figure 1, the shortest paths lengths are given in Table 4. As can be seen, this rule confirms what we already knew about the layers of the cube.

Vertex	Shortest Path Length	Possible Path
1	0	N/A
2	1	2→1
3	1	3→1
4	1	4→1
5	2	5→3→1
6	2	6→4→1
7	2	7→4→1
8	3	8→6→2→1

Table 4: Layers of Cube by Shortest Path Length

Edge types are found by looking at the sizes of the two polygonal faces that each edge separates. For the Platonic solids, edge types are not relevant because all of the faces are identical, and therefore all of the edges are the same type. For the Archimedean solids, however, the edge types do matter, since not all of the faces are identical. The truncated tetrahedron, for example, has both triangular and hexagonal faces, so we need to create a categorization of all 18 edges, as shown in Table 5. The edges are listed by lower vertex index and then higher vertex index, and the faces are listed by smaller face and then larger face. Of the 18 edges, 12 are between a triangle and a hexagon, and six are between two hexagons.

First Vertex	Second Vertex	First Face	Second Face
1	2	Triangle	Hexagon
1	3	Triangle	Hexagon
1	10	Hexagon	Hexagon
2	3	Triangle	Hexagon
2	8	Hexagon	Hexagon
3	6	Hexagon	Hexagon
4	5	Triangle	Hexagon
4	6	Triangle	Hexagon
4	7	Hexagon	Hexagon
5	6	Triangle	Hexagon
5	11	Hexagon	Hexagon
7	8	Triangle	Hexagon
7	9	Triangle	Hexagon
8	9	Triangle	Hexagon
9	12	Hexagon	Hexagon
10	11	Triangle	Hexagon
10	12	Triangle	Hexagon
11	12	Triangle	Hexagon

Table 5: Edge Types of Truncated Tetrahedron

We then need to consider all shortest paths from each vertex to the ending vertex, and look at the edge types that are used in each of those paths. This full accounting is shown in Table 6, and shows how vertices 2 and 3 are in a different layer than vertex 10. Each layer can be identified by matching up vertices with identical edge type paths. This task is trivial for most layers, but for Layer 5, which includes vertices 5 and 9, we must remember to account for both of the shortest paths from each vertex to the ending vertex. They both have a path that includes two (3,6) edges and one (6,6) edge and a path that includes one (3,6) edge and two (6,6) edges. Since the number and types of the edges are identical, we know that vertices 5 and 9 are in the same layer. The order that the edge types appear in has no effect on the layer splitting, which we can tell by looking at vertices 6, 8, 11, and 12. The paths from vertices 6 and 8 to vertex 1 take a (6,6) edge and then a (3,6) edge, but the paths from vertices 11 and 12 to vertex 1 take a (3,6) edge and then a (6,6) edge. Despite the edges appearing in a different order, all four vertices have the same hitting time and are in the same layer. This result is also not proven in general, but holds for all Platonic and Archimedean solids, and we conjecture that it also holds for the Catalan and Johnson solids.

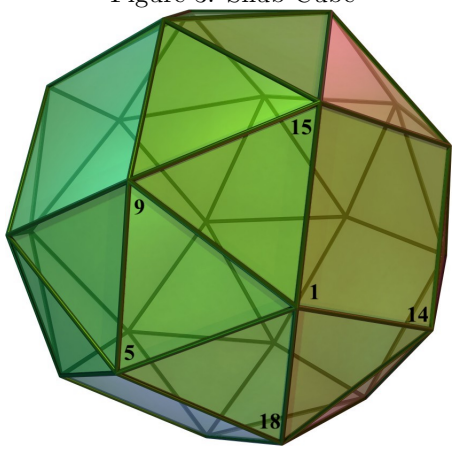
Vertex	Layer	Path Length	Paths	Edge Types
1	1	0	N/A	N/A
2	2	1	2→1	(3,6)
3	2	1	3→1	(3,6)
4	6	3	4→6→3→1	(3,6)→(6,6)→(3,6)
5	5	3	5→6→3→1 5→11→10→1	(3,6)→(6,6)→(3,6) (6,6)→(3,6)→(6,6)
6	4	2	6→3→1	(6,6)→(3,6)
7	6	3	7→8→2→1	(3,6)→(6,6)→(3,6)
8	4	2	8→2→1	(6,6)→(3,6)
9	5	3	9→8→2→1 9→12→10→1	(3,6)→(6,6)→(3,6) (6,6)→(3,6)→(6,6)
10	3	1	10→1	(6,6)
11	4	2	11→10→1	(3,6)→(6,6)
12	4	2	12→10→1	(3,6)→(6,6)

Table 6: Shortest Paths of Truncated Tetrahedron

2.3 Bordering Faces

The two rules described above are enough to properly split the vertices of every Platonic solid and 11 of the 13 Archimedean solids. However, to split the vertices of the snub cube and snub dodecahedron, there is one more wrinkle that must be considered. The snub cube, shown in Figure 3, has 32 triangular faces and six square faces. Eight of the triangles are like the face formed by vertices 1, 5, and 9, and are bordered by three other triangles, and 24 of the triangles are like the face formed by vertices 1, 9, and 15, and are bordered by two triangles and a square. This distinction between identically-sized faces with different bordering faces must be taken into account.

Figure 3: Snub Cube



Instead of just calling faces by the polygon that they are, we denote them by a decimal, where the whole number portion is the number of sides of the face, and each digit after the decimal point is the size of a bordering face. For example, there are three types of faces on the snub cube: eight triangles denoted by 3.333, 24 triangles denoted by 3.334, and six squares denoted by 4.3333. The face sizes after the decimal are arranged in increasing order for consistency. The five edges coming out of vertex 1 are then shown in Table 7. With just Rule 1, all five of the neighbors of vertex 1 would be in the same layer. After including Rule 2 but disregarding bordering faces, vertices 5, 9, and 18 would be in a layer (separated from vertex 1 by a triangle-triangle edge), and vertices 14 and 15 would be in a different layer (separated from vertex 1 by a triangle-square edge). Accounting for bordering faces, we can see that vertices 5 and 9 are in a different layer than vertex 18. Before accounting for bordering faces, the snub cube would appear to only have 15 layers, but this turns out to be an undersplitting, as there are actually 17 unique hitting times, including the layer with just vertex 1. Additionally, though the snub cube and snub dodecahedron come in two chiral forms (the mirror images of these polyhedra look different than their non-mirrored forms), the results are identical whether the "right-handed" or "left-handed" version is used.

First Vertex	Second Vertex	First Face	Second Face
1	5	3.333	3.334
1	9	3.333	3.334
1	14	3.334	4.3333
1	15	3.334	4.3333
1	18	3.334	3.334

Table 7: Selected Edge Types of Snub Cube

Once the vertices of a polyhedron have been split into the proper layers, the hitting times and two-point resistances can then be calculated much more efficiently, as shown in the next chapter.

3 Calculating Hitting Times

The method that this thesis will use to turn layers into hitting times is based on a generalization of a previously presented equation [2]. This equation makes intuitive sense, as the hitting time from a vertex a to another vertex b is equal to the average of the hitting times to b of the neighbors of a, plus the one step it takes to get from a to any of those neighbors v.

$$H(a,b) = 1 + \frac{1}{\deg(a)} \sum_{v \in N(a)} H(v,b) \tag{1}$$

This equation is very useful, and on its own would allow us to calculate the hitting times for every vertex in a polyhedron. To do so, simply find the adjacency matrix of the vertices and then solve the resulting system of equations, taking into account that $H(b,b) = 0$. However, if we want to take advantage of the layer splitting of the vertices, then we must generalize this equation. Equation 2, derived during the course of this thesis, has the same structure as Equation 1, but accounts for every vertex in a given layer. The hitting time from any one of the vertices a_i in the layer $\{a_n\}$ to a vertex b is equal to the average of the hitting times from all of the neighbors v_i of all of the vertices a_i in the set $\{a_n\}$ to b, plus the one step it takes to get from a_i to v_i. Since Equation 1 is valid for all connected graphs [2], we conjecture that Equation 2 is also valid for all connected graphs where a layer structure exists. It certainly is valid for all Platonic and Archimedean solids, and we conjecture that it is also valid for the Catalan and Johnson solids.

$$H(\{a_n\},b) = 1 + \frac{1}{\sum_{a_i \in \{a_n\}} \deg(a_i)} \sum_{a_i \in \{a_n\}} \sum_{v_i \in N(a_i)} H(v_i,b) \tag{2}$$

3.1 Example: Cube

Using Equation 1, the cube would have the following matrix equation for the hitting times of every vertex except for vertex 1 (which by definition has a hitting time of zero):

$$\begin{bmatrix} 1 & 0 & 0 & -1/3 & -1/3 & 0 & 0 \\ 0 & 1 & 0 & -1/3 & 0 & -1/3 & 0 \\ 0 & 0 & 1 & 0 & -1/3 & -1/3 & 0 \\ -1/3 & -1/3 & 0 & 1 & 0 & 0 & -1/3 \\ -1/3 & 0 & -1/3 & 0 & 1 & 0 & -1/3 \\ 0 & -1/3 & -1/3 & 0 & 0 & 1 & -1/3 \\ 0 & 0 & 0 & -1/3 & -1/3 & -1/3 & 1 \end{bmatrix} \begin{bmatrix} H(2,1) \\ H(3,1) \\ H(4,1) \\ H(5,1) \\ H(6,1) \\ H(7,1) \\ H(8,1) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Taking advantage of the layer structure seen in Tables 1 and 4, and Equation 2, we can simplify the calculation from a 7x7 system down to a 3x3 system, which is much faster to solve:

$$\begin{bmatrix} 1 & -2/3 & 0 \\ -2/3 & 1 & -1/3 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} H(\{2,3,4\},1) \\ H(\{5,6,7\},1) \\ H(8,1) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Both systems yield the same result, shown in Table 8 below. If we desired to find the equivalent resistance between any two vertices, modeling the edges as identical 1Ω resistors, we would divide the hitting times by the total number of edges. This is an experimental result that we discovered by comparing the calculated hitting times to published equivalent resistances. This equation holds for all Platonic and Archimedean solids, but must be modified for polyhedra with vertices of different degrees, as discussed in Chapter 4. We then find that the equivalent resistance on a cube between adjacent vertices is $7/12\Omega$, between vertices a face diagonal apart is $3/4\Omega$, and between antipodal vertices is $5/6\Omega$.

Layer	Vertices	Hitting Time	Resistance (Ω)
1	1	0	0
2	2,3,4	7	7/12
3	5,6,7	9	3/4
4	8	10	5/6

Table 8: Hitting Times and Resistances of Cube

There is another way to derive this matrix equation, stemming from Markov transition matrices. Specifically, the random walks that we are performing form absorbing Markov chains, where once a certain state is reached (the ending vertex), it is impossible to leave that state. The entries of the transition matrix, P_{ij}, give the probability of stepping from state i to state j. These probabilities are easy to find in our case, and the matrix is closely related to the layer matrix discussed in earlier research [7][11][12]. The entries in the layer matrix, L_{ij}, are the average number of vertices in layer j connected to each vertex in layer i. Due to how the layers are divided, this might not be a whole number. To find the transition matrix, each row of the layer matrix must be normalized relative to the 1-norm, and then the row and column containing the absorbing state must be deleted [28]. For example, the layer and transition matrices for the cube are shown below:

$$L = \begin{bmatrix} 0 & 3 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 3 & 0 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 2/3 & 0 \\ 2/3 & 0 & 1/3 \\ 0 & 1 & 0 \end{bmatrix}$$

To find the actual hitting times, we must then solve the matrix equation $(I - P)H = \mathbf{1}$, where I is the identity matrix, H is the column vector of hitting times, and $\mathbf{1}$ is the column vectors of ones [28]. The matrix equation for the cube is the same one that Equation 2 provides.

3.2 Example: Truncated Tetrahedron

We see a similar level of system simplification for the truncated tetrahedron. Using Equation 1, the matrix equation is:

$$\begin{bmatrix} 1 & -\frac{1}{3} & 0 & 0 & 0 & 0 & -\frac{1}{3} & 0 & 0 & 0 & 0 \\ -\frac{1}{3} & 1 & 0 & 0 & -\frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{3} & -\frac{1}{3} & -\frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{3} & 1 & -\frac{1}{3} & 0 & 0 & 0 & 0 & -\frac{1}{3} & 0 \\ 0 & -\frac{1}{3} & -\frac{1}{3} & -\frac{1}{3} & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{3} & 0 & 0 & 1 & -\frac{1}{3} & -\frac{1}{3} & 0 & 0 & 0 \\ -\frac{1}{3} & 0 & 0 & 0 & 0 & -\frac{1}{3} & 1 & -\frac{1}{3} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{3} & -\frac{1}{3} & 1 & 0 & 0 & -\frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -\frac{1}{3} & -\frac{1}{3} \\ 0 & 0 & 0 & -\frac{1}{3} & 0 & 0 & 0 & 0 & -\frac{1}{3} & 1 & -\frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{3} & -\frac{1}{3} & -\frac{1}{3} & 1 \end{bmatrix} \begin{bmatrix} H(2,1) \\ H(3,1) \\ H(4,1) \\ H(5,1) \\ H(6,1) \\ H(7,1) \\ H(8,1) \\ H(9,1) \\ H(10,1) \\ H(11,1) \\ H(12,1) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

However, taking advantage of the layer structure in Table 6, we can simplify from a 11x11 system to a 6x6 system:

$$\begin{bmatrix} 2/3 & 0 & -1/3 & 0 & 0 \\ 0 & 1 & -2/3 & 0 & 0 \\ -1/6 & -1/6 & 5/6 & -1/3 & -1/6 \\ 0 & 0 & -2/3 & 1 & -1/3 \\ 0 & 0 & -1/3 & -1/3 & 2/3 \end{bmatrix} \begin{bmatrix} H(\{2,3\},1) \\ H(10,1) \\ H(\{6,8,11,12\},1) \\ H(\{5,9\},1) \\ H(\{4,7\},1) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

We now see our first instance of diagonal entries not all being equal to one. This occurs when vertices in a layer are adjacent, which did not happen with the cube. The cube is actually the exception here, as it is one of only four Platonic and Archimedean solids where no vertices in a layer are ever adjacent. The other three are the truncated octahedron, the truncated cuboctahedron, and the truncated icosidodecahedron, all of which are Archimedean solids.

Regardless of which matrix equation we solve, we will obtain the same solution, which is shown in Table 9 below.

Layer Number	Vertices in Layer	Hitting Time	Resistance (Ω)
1	1	0	0
2	2, 3	51/5	17/30
3	10	63/5	7/10
4	6, 8, 11, 12	87/5	29/30
5	5, 9	96/5	16/15
6	4, 7	99/5	11/10

Table 9: Hitting Times and Resistances of Truncated Tetrahedron

For completeness, the layer and transition matrices for the truncated tetrahedron are shown below. The non-integer values of the layer matrix indicate that only some of the vertices in layer 4 are adjacent to the vertices in layers 2, 3, 4, and 6.

$$L = \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1/2 & 1/2 & 1/2 & 1 & 1/2 \\ 0 & 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 1/3 & 0 & 1/3 & 0 & 0 \\ 0 & 0 & 2/3 & 0 & 0 \\ 1/6 & 1/6 & 1/6 & 1/3 & 1/6 \\ 0 & 0 & 2/3 & 0 & 1/3 \\ 0 & 0 & 1/3 & 1/3 & 1/3 \end{bmatrix}$$

Using the layer structure to reduce the size of the system works wonders for all of the Platonic and Archimedean solids. It reduces the number of equations by at least half for every studied polyhedron except for five: the truncated octahedron, truncated cuboctahedron, truncated icosidododecahedron, and the snub cube and snub dodecahedron. Even for those five solids, the layer method still helps tremendously. The number of vertices and layers for each of the Platonic and Archimedean solids, as well as the percent reduction in the number of rows of the matrix equation, are given in Table 10. The percent reduction is calculated by (number of rows in original matrix - number of rows in reduced matrix) / (number of rows in original matrix). The truncated icosidodecahedron is by far the most complicated Archimedean solid. It has 120 vertices, double the number of any other Archimedean solid. Solving for the hitting times directly using Equation 1 would result in having to solve a 119x119 matrix equation, made even more difficult if exact arithmetic is used. Once the 76 layers have been found, however, Equation 2 can be used to set up the 75x75 matrix equation, a reduction in size of almost 37%. This reduction is even more impressive when considering that the best algorithms for matrix inversion have complexity $\mathcal{O}(n^{2.376})$ [29], so even a modest reduction in size allows us to reap large computational benefits.

Solid Name	# Vertices	# Layers	Percent Reduction
Tetrahedron	4	2	67%
Cube	8	4	57%
Octahedron	6	3	60%
Dodecahedron	20	6	74%
Icosahedron	12	4	73%
Truncated Tetrahedron	12	6	55%
Cuboctahedron	12	5	64%
Truncated Cube	24	12	52%
Truncated Octahedron	24	13	48%
Rhombicuboctahedron	24	12	52%
Truncated Cuboctahedron	48	34	30%
Snub Cube	24	17	30%
Icosidodecahedron	30	9	72%
Truncated Dodecahedron	60	24	61%
Truncated Icosahedron	60	24	61%
Rhombicosidodecahedron	60	24	61%
Truncated Icosidodecahedron	120	76	37%
Snub Dodecahedron	60	38	37%

Table 10: Reduction in Size of Matrix Equations

4 Further Results

In addition to the Platonic and Archimedean solids, we can also apply the layer method to other partially symmetric polyhedra. The Catalan solids are the dual polyhedra of the Archimedean solids. Eleven of the 13 Catalan solids have edges that are not all the same length, but two of them are easy to treat. The rhombic dodecahedron and rhombic triacontahedron, the duals of the cuboctahedron and icosidodecahedron, are edge-transitive, so we can use the layer method on them. Though the edges do not present any problems, the vertices do. Since the cuboctahedron and icosidodecahedron have faces of various sizes, the vertices of their duals have vertices of different degrees. This means that we need to find two tables for each solid, one for each degree of vertex.

The Catalan solids also display a key feature of hitting times, which is that they are not always symmetric. In other words, $H(a, b)$ is not always equal to $H(b, a)$. This can be seen in Figure 4, where it takes 15 steps on average to get from a vertex of degree 4 to any of its degree 3 neighbors, and in Figure 5, where it takes 11 steps on average to get from a vertex of degree 3 to any of its degree 4 neighbors. The diagrams also show that it takes 21 steps to get from a degree 4 vertex to a degree 3 vertex that is three steps away, but that it takes 17 steps to accomplish the opposite feat. Because of this asymmetry, the hitting times between vertices of different degrees must be averaged to find the two-point resistances [11]. In equation form, this looks like

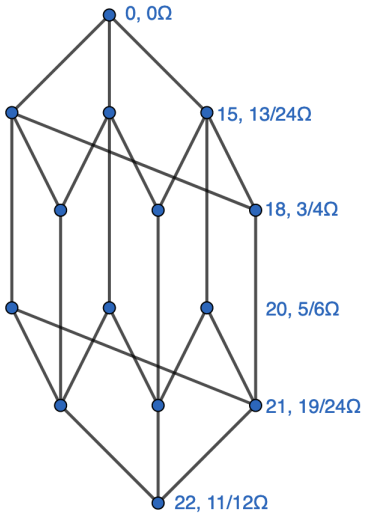
$$R(a, b) = R(b, a) = \frac{H(a, b) + H(b, a)}{2E} \tag{3}$$

In words, this means that the equivalent resistance between two vertices is equal to the average of the hitting times between the vertices divided by the total number of edges. This equation holds for every polyhedron studied in this thesis, and we conjecture that it continues to hold for the rest of the Catalan and Johnson solids. It is usually the case that $H(a, b) = H(b, a)$, causing the equivalent resistance between two vertices to be directly scaled from the hitting time.

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	3	15	13/24
3	3	18	3/4
4	3	20	5/6
5	3	21	19/24
6	1	22	11/12

Table 11: Rhombic Dodecahedron, Degree 3

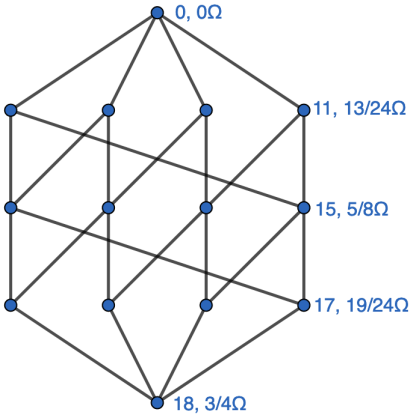
Figure 4: Rhombic Dodecahedron Layers, Degree 3



Layer	# Vertices	Hitting Time	Resistance (Ω)
2	4	11	13/24
3	4	15	5/8
4	4	17	19/24
5	1	18	3/4

Table 12: Rhombic Dodecahedron, Degree 4

Figure 5: Rhombic Dodecahedron Layers, Degree 4

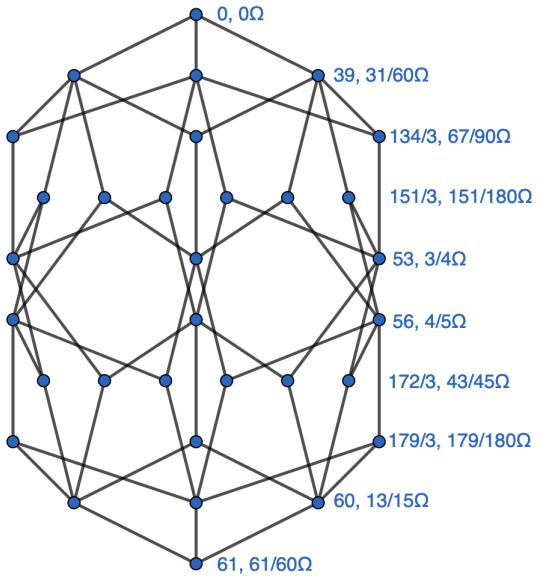


The hitting times and two-point resistances for the rhombic triacontahedron can be seen in Table 13 and Figure 6 for ending vertices of degree 3 and in Table 14 and Figure 7 for ending vertices of degree 5. Similar to the rhombic dodecahedron, the diagrams show that it takes 39 steps on average to get from a degree 5 vertex to a degree 3 vertex if they are neighboring, either 53 or 56 steps if they have a geodesic of 3, and 60 steps if they have a geodesic distance of 5. The reverse paths take 23, 37, 40, and 44 steps on average, respectively. There are two hitting times for vertices that are three steps apart because there are two different symmetries, as can be seen on a physical rhombic triacontahedron.

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	3	39	31/60
3	3	134/3	67/90
4	6	151/3	151/180
5	3	53	3/4
6	3	56	4/5
7	6	172/3	43/45
8	3	179/3	179/180
9	3	60	13/15
10	1	61	61/60

Table 13: Rhombic Triacontahedron, Degree 3

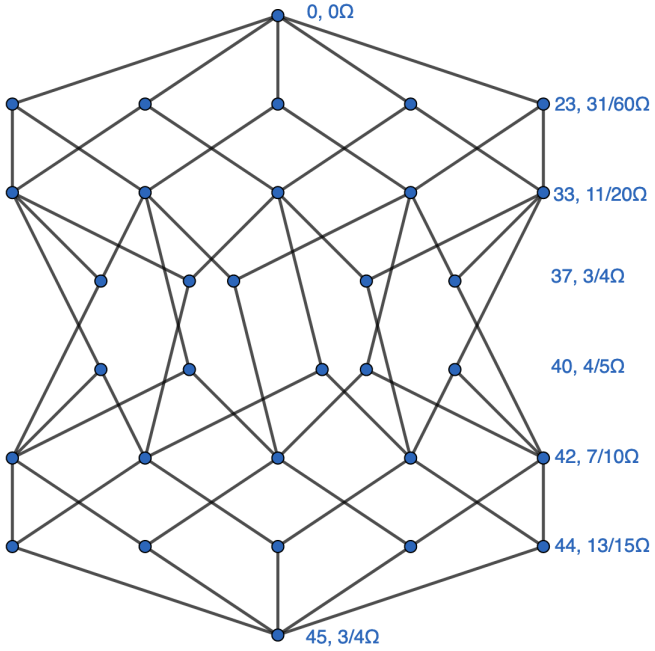
Figure 6: Rhombic Triacontahedron Layers, Degree 3



Layer	# Vertices	Hitting Time	Resistance (Ω)
2	5	23	31/60
3	5	33	11/20
4	5	37	3/4
5	5	40	4/5
6	5	42	7/10
7	5	44	13/15
8	1	45	3/4

Table 14: Rhombic Triacontahedron, Degree 5

Figure 7: Rhombic Triacontahedron Layers, Degree 5



5 Conclusion and Open Questions

The problem of finding the hitting times and two-point resistances between vertices of the Platonic and Archimedean solids has been fully solved. In addition to the methods presented in this thesis, there are at least two other methods to calculate the same results for these solids, and there may be more [7][11][12][16]. It is also possible that a better method for finding layers might exist.

A more fruitful area of future research might also be to extend the currently existing methods to the Johnson solids. The Johnson solids are a set of 92 convex polyhedra with regular polygons for faces. An added difficulty compared to the Platonic and Archimedean solids is that the vertices of the Johnson solids do not all have the same degree. This means that an arbitrary vertex cannot be chosen to be the universal ending vertex for every random walk, but that a representative vertex of each degree must be considered.

It would also be interesting, though much harder, to extend these methods to the rest of the Catalan solids. The hitting times between pairs of their vertices will be more difficult to calculate for two main reasons. The first reason is the same vertex degree issue as the Johnson solids. The second reason is that, for all of the Catalan solids other than the rhombic dodecahedron and rhombic triacontahedron, not all of the edges of each polyhedron are the same length. The way that the adjacency matrices of the polyhedra are calculated would have to be rewritten, as right now the code in Appendix C.1 relies on adjacent vertices being equidistant. Additionally, future researchers would have to consider whether to adjust the chances of a random walk stepping to any of the neighbors of the current vertex. For the equivalent resistance problem, this would correspond to scaling the 1Ω resistors according to the length of the edge. The easiest solution would be to keep all probabilities equal (and all resistors equal to 1Ω), but this is a choice that would have to be made. On the bright side, all the faces of each Catalan solid are congruent, so each edge (of the same length) will be the same type, since they all separate the same size faces.

In terms of direct improvements to the work in this thesis, there are several possibilities. First, as discussed in Chapter 6, the facefinder.m MATLAB function relies on having the coordinates of the vertices to find faces and cycles. Additionally, the current implementation of Rule 1 also uses coordinates to find dot products. It would be useful to rewrite these functions to not require the use of coordinates. If this improvement is made, then the adjacency matrix of a graph is all that would be needed to find the hitting times and equivalent resistances. This would allow the methods developed in this thesis to be generalized beyond polyhedra with published coordinates. Additionally, several of the results presented in this thesis are purely experimental, such as the layer-based hitting time equation and the equation to convert hitting times to equivalent resistances (Equations 2 and 3, respectively). A future researcher might find it useful to prove these results directly.

6 Code Explanation

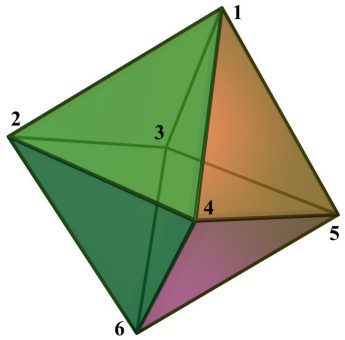
This section will provide an explanation of the MATLAB code contained within Appendix C. After generalizing the hitting time equation, writing this code was the most significant portion of this project, and it was the main tool used in obtaining the results. In addition to the raw code, there are descriptive comments on almost every line of the functions.

The first function, adjacency.m, is shown in Appendix C.1. The purpose of this function is to take a set of coordinates of a polyhedron and construct the adjacency matrix of the graph. It relies on the edges of the polyhedron being of equal length, with two vertices being adjacent if they are the smallest distance between any pair of vertices apart. The dot product for pairs of vertices is used as a more efficient substitute for calculating the Euclidean distance. This method relies on the vertices being equidistantly placed about the origin. In that case, the dot product between a vertex and itself will always be the greatest possible value, and the dot product between a vertex and one of its closest neighbors will be the second greatest possible value. A tolerance of 10^{-10} is used to account for input and rounding error. MATLAB keeps precision up to about 10^{-16}, so a tolerance of 10^{-10} is plenty to account for error without introducing extraneous adjacencies.

The next function, facefinder.m, is shown in Appendix C.2. The purpose of this function is to identify the faces of the given polyhedron by returning the indices of the vertices in each face. One of the inputs is nodedepth, a vector that is calculated in the findlayers.m function, where this function is called. Another optional input is Fs, a vector of the face sizes of the polyhedron. This input is only necessary for polyhedra with multiple face sizes, which is every polyhedron except the Platonic solids. Facefinder works by looping through the vertices until it finds a vertex that is at least as deep as two of its neighbors, where depth is defined by geodesic distance to the ending vertex, usually vertex 1. At that point, all other vertices are tested to see if they are in the same plane as those three vertices. If the number of vertices found in the plane is a face size of the polyhedron, then the vertices are put into cyclic order and added to the cycles output. The vertices are also sorted into ascending order and added to the faces output. This function uses the padarray function, which comes from the Image Processing Toolbox that MATLAB offers. The padarray function is not strictly necessary, and could be rewritten, but this was the most concise way to accomplish the task of padding arrays with zeros.

As an example of how facefinder.m works, we observe the octahedron, shown in Figure 8. The function starts by looking at vertex 1, but finds that it cannot be the deepest in a cycle since there are not at least two adjacent vertices that are not any deeper than vertex 1. Shifting focus to vertex 2 as the deepest, the function finds the faces containing vertices 1, 2, and 3 and vertices 1, 2, and 4. It then attempts to find a face using vertices 2, 3, and 4. Since the only possible face size for the octahedron is a triangle, the function tries to find the cycle containing those three vertices. Since they don't form a cycle (vertices 3 and 4 are not adjacent), the function discards this possible face. Vertex 6 is dismissed as a neighbor of vertex 2 since it is deeper, and would cause vertex 2 to no longer be the deepest vertex in a cycle. Moving to vertex 3, the function finds the face formed by vertices 1, 2, and 3 again, but does not add it because it has already been found. It does add the face containing vertices 1, 3, and 5. Using vertex 4 as the deepest vertex, the face formed by vertices 1, 4, and 5 is found and added. No new faces use vertex 5 as the deepest vertex, but vertex 6 is the deepest node of the remaining four faces.

Figure 8: Octahedron



The next function, vertpathfinder.m, is shown in Appendix C.3. The purpose of this function is to find the vertices along every minimum length path from a given "out" vertex to the "in" or ending vertex. It utilizes previously generated paths, which are stored in the Map object vpdict (vertpathdict in the main findlayers.m function), and just adds a single step to the front of each path, from the "out" vertex to any of its less deep neighbors. Due to how nodedepth is calculated within findlayers.m, any neighbors of the "out" vertex will have have a depth either one less than, equal to, or one more than the depth of the "out" vertex. Thus, mandating that the first step away from the "out" vertex must be to a neighbor with a smaller depth is the same as mandating that the depth must decrease by exactly one on every step. Despite that knowledge, it is easier to directly compare the nodedepth of the "out" vertex to the nodedepth of its neighbors than it is to check if their difference is equal to one, and in the correct order, so the first comparison is the one used in the code.

The next function, findlayers.m, is shown in Appendix C.4. The purpose of this function is to put the previous three together and split the vertices of a polyhedron into layers. The function starts by finding the depth of every vertex using a breadth-first search first coded by Moody in Java [11]. These depths are then used to identify the faces of the polyhedron (using facefinder.m). Next, the edges of the whole polyhedron and of each face are found. These edges are then used to find the sizes of all of the faces bordering each face, which is necessary to accurately split the snub cube and snub dodecahedron. A Map (a dictionary-like structure in MATLAB) is then created to classify the edges according to their bordering faces. A preliminary layer splitting is formed based on Euclidean distance from the ending vertex, and then the shortest paths from each vertex to the ending vertex are found. These shortest paths are then translated from vertices to edge types, and finally the edge types of the shortest paths are used to create the final, accurate, layer splitting of the polyhedron. Further explanation about each part of this function are given in the code in the appendix. The Image Processing Toolbox is also required for this function, as in facefinder.m.

The final function, findhits.m, is shown in Appendix C.5. The purpose of this function is to create and solve the matrix equation defined by the layer-based hitting time equation. Each entry A_{ij} of the coefficient matrix is equal to the negative of the number of (possibly repeated) vertices in layer $j+1$ adjacent to the vertices in layer $i+1$ divided by the sum of the degrees of the vertices in layer $i+1$. A one gets added to every diagonal entry because it is the hitting time of each layer that we are trying to calculate. The reason for the offset of one in the layers is due to the fact that $H(b,b)=0$, so the row and column corresponding to the first layer (the one with the ending vertex in it) are deleted from the matrix. The right hand side of the matrix equation is a column vector of ones, to account for the one step it takes to get from a vertex to any of its neighbors. After the matrix equation has been solved, the hitting times are sorted into ascending order, and the layers are reordered to match the new order. The functions splitting the vertices into layers do not have a way to know which layers will end up having a higher or lower hitting time, so the layers sometimes get generated out of order. After reordering the layers, the function displays the vertices in each ascending layer, and returns the fractional hitting times for each layer. To find the equivalent resistances, each hitting time must be divided by the total number of edges in the polyhedron. This function requires the use of the Symbolic Math Toolbox to return the exact fractional values of the hitting times rather than decimal approximations.

References

[1] A. R. D. van Slijpe, "Random walks on the triangular prism and other vertex-transitive graphs," *Journal of Computational and Applied Mathematics*, vol. 15, no. 3, pp. 383–394, 1986. DOI: https://doi.org/10.1016/0377-0427(86)90229-3.

[2] L. Lovász, *Graphs and Geometry*. Providence, RI: American Mathematical Society, 2019.

[3] V. V. Palyulin, G. Blackburn, M. A. Lomholt, *et al.*, "First passage and first hitting times of lévy flights and lévy walks," *New Journal of Physics*, vol. 21, no. 10, 2019. DOI: https://doi.org/10.1088/1367-2630/ab41bb.

[4] T. Fischer, "On simple representations of stopping times and stopping time sigma-algebras," *Statistics & Probability Letters*, vol. 83, no. 1, pp. 345–349, 2013. DOI: https://doi.org/10.1016/j.spl.2012.09.024.

[5] B. D. Hughes and B. W. Ninham, Eds., *The Mathematics and Physics of Disordered Media*. Berlin, Germany: Springer-Verlag, 1983.

[6] L. H. Liyanage, C. M. Gulati, and J. M. Hill, "A bibliography on applications of random walks in theoretical chemistry and physics," *Advances in Molecular Relaxation and Interaction Processes*, vol. 22, no. 1, pp. 53–72, 1982. DOI: https://doi.org/10.1016/0378-4487(82)80019-8.

[7] K. Stern, "Equivalent resistances of polytope networks," 2021. [Online]. Available: https://digital.wpi.edu/concern/student_works/th83m221c.

[8] S. I. Maiti and J. Sarkar, "Symmetric walks on paths and cycles," *Mathematics Magazine*, vol. 92, no. 4, pp. 252–268, 2019. DOI: https://doi.org/10.1080/0025570X.2019.1611166.

[9] J. Sarkar and S. I. Maiti, "Symmetric random walks on regular tetrahedra, octahedra, and hexahedra," *Calcutta Statistical Association Bulletin*, vol. 69, no. 1, pp. 110–128, 2017. DOI: https://doi.org/10.1177/0008068317695974.

[10] J. Sarkar, "Symmetric walk on a polygon," *Institute of Mathematical Statistics Lecture Notes - Monograph Series*, vol. 50, J. Sun, A. DasGupta, V. Melfi, and C. Page, Eds., pp. 31–43, 2006. DOI: https://doi.org/10.1214/074921706000000581.

[11] J. Moody, "Efficient methods for calculating equivalent resistance between nodes of a highly symmetric resistor network," 2013. [Online]. Available: https://digital.wpi.edu/concern/student_works/jq085m469.

[12] ——, "Resistor networks based on symmetrical polytopes," *Electronic Journal of Graph Theory and Applications*, vol. 3, no. 1, pp. 56–69, 2013. DOI: https://doi.org/10.5614/ejgta.2015.3.1.7.

[13] R. C. Mishra and H. Barman, "Effective resistances of two-dimensional resistor networks," *European Journal of Physics*, vol. 42, no. 1, 2021. DOI: https://doi.org/10.1088/1361-6404/abc526.

[14] F. J. van Steenwijk, "Equivalent resistors of polyhedral resistive structures," *American Journal of Physics*, vol. 66, no. 1, pp. 90–91, 1998. DOI: https://doi.org/10.1119/1.18820.

[15] H. S. M. Coxeter, *Regular Polytopes*. New York: Dover Publications, 1963.

[16] F. Perrier and F. Girault, "Two-point resistances in archimedean resistor networks," *Results in Physics*, vol. 36, 2022. DOI: https://doi.org/10.1016/j.rinp.2022.105443.

[17] D. Aldous and J. A. Fill, *Reversible markov chains and random walks on graphs*, Unfinished monograph, recompiled 2014, available at http://www.stat.berkeley.edu/$\sim$aldous/RWG/book.html, 2002.

[18] R. Gorenflo, F. Mainardi, D. Moretti, and P. Paradisi, "Time fractional diffusion: A discrete random walk approach," *Nonlinear Dynamics*, vol. 29, pp. 129–143, 2002. DOI: https://doi.org/10.1023/A:1016547232119.

[19] J. Rudnick and G. Gaspari, "The asphericity of random walks," *Journal of Physics A: Mathematics and General*, vol. 19, no. 4, pp. L191–L194, 1986. DOI: https://doi.org/10.1088/0305-4470/19/4/004.

[20] G. H. Weiss and R. J. Rubin, in *Advances in Chemical Physics*, I. Prigogine and S. A. Rice, Eds. New York: John Wiley & Sons, 1983, vol. LII, ch. Random Walks: Theory and Selected Applications.

[21] C. S. J. A. Nash-Williams, "Random walk and electric currents in networks," in *Mathematical Proceedings of the Cambridge Philosophical Society*, B. J. Green, Ed., vol. 55, Cambridge University Press, 2008, pp. 181–194. DOI: https://doi.org/10.1017/S0305004100033879.

[22] G. Golnari, Z. Zhang, and D. Boley, "Markov fundamental tensor and its applications to network analysis," *Linear Algebra and its Applications*, vol. 564, pp. 126–158, 2019. DOI: https://doi.org/10.1016/j.laa.2018.11.024.

[23] K. M. Curtin, in *Comprehensive Geographic Information Systems*, B. Huang, Ed. Elsevier, 2018, vol. 1, ch. 1.12 - Network Analysis. DOI: https://doi.org/10.1016/B978-0-12-409548-9.09599-3.

[24] P. V. Marsden, in *Encyclopedia of Social Measurement*, K. Kempf-Leonard, Ed. Elsevier, 2005, ch. Network Analysis. DOI: https://doi.org/10.1016/B0-12-369398-5/00409-6.

[25] H. W. Kroto, J. R. Heath, S. C. O'Brien, R. F. Curl, and R. E. Smalley, "C60: Buckminsterfullerene," *Nature*, vol. 318, no. 6042, pp. 162–163, 1985. DOI: https://doi.org/10.1038/318162a0.

[26] J. V. Smith, "Structural classification of zeolites," in *Papers and Proceedings of the Third General Meeting*, D. J. Fisher, A. J. Frueh, C. S. Hurlbut, and C. E. Tilley, Eds., Mineralogical Society of America, 1962, pp. 281–290.

[27] P. J. Nahin, *Mrs. Perkins's Electric Quilt*. Princeton, NJ: Princeton University Press, 2009.

[28] C. Grinstead and J. Snell, *Introduction to Probability*, 2nd. American Mathematical Society, 1997.

[29] M. Petković and P. Stanimirović, "Generalized matrix inversion is not harder than matrix multiplication," *Journal of Computational and Applied Mathematics*, vol. 230, no. 1, pp. 270–282, 2009. DOI: https://doi.org/10.1016/j.cam.2008.11.012.

Appendices

A Results of the Platonic Solids

This appendix includes the hitting times and equivalent resistances for the vertices of each of the five Platonic solids. Each table in this appendix, as well as in Appendix B, should include a row with the entries 1, 1, 0, 0, referring to the ending vertex, which is on a layer by itself with a hitting time and equivalent resistance of zero. This row has been omitted to save space. Additionally, diagrams for the cube and dodecahedron have been added to show how the tables can be applied to the polyhedra themselves. Diagrams for the other Platonic solids are omitted for space, and diagrams for the Archimedean solids are omitted due to the complexity of the graphs.

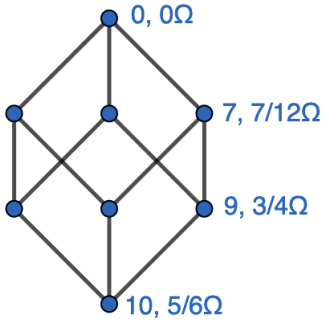
Layer	# Vertices	Hitting Time	Resistance (Ω)
2	3	3	1/2

Table 15: Tetrahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	3	7	7/12
3	3	9	3/4
4	1	10	5/6

Table 16: Cube

Figure 9: Cube Layers



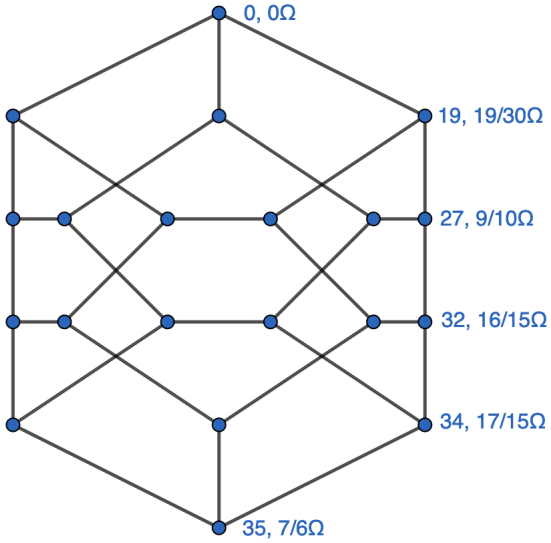
Layer	# Vertices	Hitting Time	Resistance (Ω)
2	4	5	5/12
3	1	6	1/2

Table 17: Octahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	3	19	19/30
3	6	27	9/10
4	6	32	16/15
5	3	34	17/15
6	1	35	7/6

Table 18: Dodecahedron

Figure 10: Dodecahedron Layers



Layer	# Vertices	Hitting Time	Resistance (Ω)
2	5	11	11/30
3	5	14	7/15
4	1	15	1/2

Table 19: Icosahedron

B Results of the Archimedean Solids

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	2	51/5	17/30
3	1	63/5	7/10
4	4	87/5	29/30
5	2	96/5	16/15
6	2	99/5	11/10

Table 20: Truncated Tetrahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	4	11	11/24
3	2	14	7/12
4	4	15	5/8
5	1	16	2/3

Table 21: Cuboctahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	2	21	7/12
3	1	27	3/4
4	4	39	13/12
5	2	231/5	77/60
6	2	234/5	13/10
7	2	48	4/3
8	2	249/5	83/60
9	4	261/5	29/20
10	1	273/5	91/60
11	2	279/5	31/20
12	1	282/5	47/30

Table 22: Truncated Cube

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	2	625/28	625/1008
3	1	341/14	341/504
4	1	405/14	45/56
5	4	981/28	109/112
6	2	1081/28	1081/1008
7	2	274/7	137/126
8	2	1153/27	1153/1008
9	4	171/4	19/16
10	1	621/14	69/56
11	1	629/14	629/504
12	2	1273/28	1273/1008
13	1	324/7	9/7

Table 23: Truncated Octahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	2	767/35	767/1680
3	2	843/35	281/560
4	2	1028/35	257/420
5	1	153/5	51/80
6	4	1133/35	1133/1680
7	4	1229/35	1229/1680
8	1	1263/35	421/560
9	2	1292/35	323/420
10	2	189/5	63/80
11	2	1343/35	1343/1680
12	1	1368/35	57/70

Table 24: Rhombicuboctahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	1	63859/1430	63859/102960
3	1	5059/110	5059/7920
4	1	36002/715	18001/25740
5	1	42144/715	1756/2145
6	2	7923/110	2641/2640
7	2	108723/1430	36241/34320
8	1	22751/286	22751/20592
9	2	59474/715	29737/25740
10	2	127093/1430	127093/102960
11	1	11729/130	11729/9360
12	1	130927/1430	130927/102960
13	2	11907/130	1323/1040
14	1	27351/286	3039/2288
15	2	137289/1430	45763/34320
16	1	70416/715	978/715
17	1	14260/143	3565/2574
18	2	11001/110	3667/2640
19	2	11233/110	11233/7920
20	2	147793/1430	147793/102960
21	2	30293/286	30293/20592
22	1	151627/1430	151627/102960
23	1	30699/286	3411/2288
24	2	154029/1430	51343/34320
25	1	154083/1430	51361/34320
26	2	77622/715	12937/8580
27	2	158539/1430	158539/102960
28	1	158787/1430	17643/11440
29	2	1121/10	1121/720
30	1	7372/65	1843/1170
31	1	81294/715	13549/8580
32	1	2511/22	279/176
33	1	163803/1430	54601/34320
34	1	6332/55	1583/990

Table 25: Truncated Cuboctahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	2	70685/3168	14137/38016
3	1	5965/264	1193/3168
4	2	75685/3168	15137/38016
5	2	94975/3168	18995/38016
6	1	8665/288	1733/3456
7	1	2005/66	401/792
8	2	100345/3168	20069/38016
9	2	100715/3168	20143/38016
10	2	108305/3168	21661/38016
11	1	109015/3168	21803/38016
12	2	3065/88	613/1056
13	1	10045/288	2009/3456
14	1	113455/3168	22691/38016
15	1	10465/288	2093/3456
16	1	115855/3168	23171/38016
17	1	9685/264	1937/3168

Table 26: Snub Cube

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	4	29	29/60
3	4	122/3	61/90
4	4	127/3	127/180
5	4	140/3	7/9
6	4	49	49/60
7	4	152/3	38/45
8	4	157/3	157/180
9	1	160/3	8/9

Table 27: Icosidodecahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	2	267/5	89/150
3	1	351/5	39/50
4	4	519/5	173/150
5	2	127	127/90
6	2	128	64/45
7	2	672/5	112/75
8	4	731/5	731/450
9	4	751/5	751/450
10	2	151	151/90
11	4	788/5	394/225
12	2	162	9/5
13	2	167	167/90
14	4	863/5	863/450
15	4	876/5	146/75
16	2	178	89/45
17	4	896/5	448/225
18	2	907/5	907/450
19	2	183	61/30
20	2	184	92/45
21	4	934/5	467/225
22	1	946/5	473/225
23	2	952/5	476/225
24	1	191	191/90

Table 28: Truncated Dodecahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	2	48819/836	16273/25080
3	1	25167/418	8389/12540
4	2	34851/418	11617/12540
5	4	74247/836	24749/25080
6	2	40911/418	13637/12540
7	4	8007/76	2669/2280
8	2	22398/209	3733/3135
9	4	23616/209	1312/1045
10	4	97557/836	32519/25080
11	2	99399/836	33133/25080
12	2	101505/836	6767/5016
13	2	103215/836	6881/5016
14	2	104529/836	34843/25080
15	4	106107/836	35369/25080
16	4	27036/209	1502/1045
17	2	27528/209	4588/3135
18	4	110307/836	36769/25080
19	2	56301/418	18767/12540
20	4	113577/836	37859/25080
21	2	57081/418	19027/12540
22	1	57657/418	19219/12540
23	2	115509/836	38503/25080
24	1	1530/11	17/11

Table 29: Truncated Icosahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	2	52543/957	52543/114840
3	2	60383/957	60383/114840
4	2	72548/957	18137/28710
5	4	81253/957	81253/114840
6	2	83903/957	83903/114840
7	1	3175/33	635/792
8	4	92185/957	18437/22968
9	4	31771/319	31771/38280
10	2	96068/957	24017/28710
11	4	33661/319	33661/38280
12	2	103003/957	103003/114840
13	2	104443/957	104443/114840
14	4	35341/319	35341/38280
15	2	108848/957	13606/14355
16	4	36571/319	36571/38280
17	1	110795/957	22159/22968
18	4	110905/957	22181/22968
19	2	113423/957	113423/114840
20	4	113653/957	113653/114840
21	2	115208/957	14401/14355
22	2	115823/957	115823/114840
23	2	116623/957	116623/114840
24	1	39060/319	651/638

Table 30: Rhombicosidodecahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	1	166172084/1486191	41543021/66878595
3	1	173751140/1486191	8687557/13375719
4	1	190646963/1486191	190646963/267514380
5	1	73895035/495397	14779007/17834292
6	2	90790858/495397	45395429/44585730
7	2	98369914/495397	49184957/44585730
8	1	17725804/87423	4431451/3934035
9	2	320673518/1486191	160336759/133757190
10	2	345148397/1486191	345148397/267514380
11	2	734601/3077	244867/184620
12	1	361971116/1486191	90492779/66878595
13	1	369550172/1486191	92387543/66878595
14	2	18145933/70771	18145933/12738780
15	1	130026555/495397	2889479/1981588
16	1	394156361/1486191	394156361/267514380
17	2	134600587/495397	134600587/89171460
18	2	135093480/495397	2251558/1486191
19	2	59070173/212313	59070173/38216340
20	1	417927248/1486191	104481812/66878595
21	2	423905327/1486191	423905327/267514380
22	1	430313930/1486191	43031393/26751438
23	2	431484383/1486191	431484383/267514380
24	2	431615693/1486191	431615693/267514380
25	1	8534332/29141	2133583/1311345
26	2	146254097/495397	146254097/89171460
27	2	147377878/495397	73688939/44585730
28	2	148650615/495397	3303347/1981588
29	2	149143508/495397	37285877/22292865
30	1	456438590/1486191	45643859/26751438
31	2	457489082/1486191	228744541/133757190
32	2	21817867/70771	21817867/12738780
33	2	462416669/1486191	462416669/267514380
34	1	154457356/495397	38614339/22292865
35	2	470296886/1486191	235148443/133757190
36	2	476034686/1486191	238017343/133757190
37	1	20731975/64617	4146395/2326212
38	1	9381265/29141	1876253/1049076
39	2	22793542/70771	11396771/6369390
40	2	69106436/212313	17276609/9554085
41	2	161951312/495397	40487828/22292865
42	2	162268632/495397	4507462/2476985
43	1	493083218/1486191	246541609/133757190

44	2	165702724/495397	41425681/22292865
45	2	165850193/495397	165850193/89171460
46	2	71294471/212313	71294471/38216340
47	1	167582480/495397	8379124/4458573
48	2	503089004/1486191	125772251/66878595
49	2	505386815/1486191	101077363/53502876
50	2	506941514/1486191	253470757/133757190
51	2	169773601/495397	169773601/89171460
52	1	511182242/1486191	255591121/133757190
53	2	513097181/1486191	513097181/267514380
54	1	171645620/495397	8582281/4458573
55	2	171858391/495397	171858391/89171460
56	2	172170119/495397	172170119/89171460
57	2	24621101/70771	24621101/12738780
58	1	173631457/495397	173631457/89171460
59	1	1333385/3801	266677/136836
60	1	173902406/495397	86951203/44585730
61	2	522228251/1486191	522228251/267514380
62	2	523782950/1486191	52378295/26751438
63	2	75004829/212313	75004829/38216340
64	1	176224234/495397	88112117/44585730
65	1	176535962/495397	88267981/44585730
66	2	75728819/212313	75728819/38216340
67	2	177238049/495397	177238049/89171460
68	2	177746036/495397	44436509/22292865
69	2	535548332/1486191	133887083/66878595
70	1	179029786/495397	89514893/44585730
71	2	538353884/1486191	134588471/66878595
72	1	540120215/1486191	108024043/53502876
73	1	180091871/495397	180091871/89171460
74	1	180288130/495397	18028813/8917146
75	1	180599858/495397	90299929/44585730
76	1	428150/1173	42815/21114

Table 31: Truncated Icosidodecahedron

Layer	# Vertices	Hitting Time	Resistance (Ω)
2	2	26954193/478108	8984731/23905400
3	1	6871376/119527	3435688/8964525
4	2	29823985/478108	5964797/14343240
5	2	37376431/478108	37376431/71716200
6	1	9556454/119527	4778227/8964525
7	2	40564297/478108	40564297/71716200
8	2	40882371/478108	13627457/23905400
9	2	2410887/28124	803629/1406200
10	1	44358325/478108	1774333/2868648
11	2	45182813/478108	45182813/71716200
12	2	11354346/119527	1892391/2988175
13	2	45660607/478108	45660607/71716200
14	2	45978681/478108	15326227/23905400
15	2	46559183/478108	46559183/71716200
16	1	48175213/478108	48175213/71716200
17	2	48958491/478108	16319497/23905400
18	2	49240567/478108	49240567/71716200
19	2	49914079/478108	49914079/71716200
20	1	12491079/119527	4163693/5976350
21	2	50687019/478108	16895673/23905400
22	1	50856597/478108	16952199/23905400
23	2	51341449/478108	51341449/71716200
24	2	52281493/478108	52281493/71716200
25	2	52553379/478108	17517793/23905400
26	1	52608385/478108	10521677/14343240
27	2	52759287/478108	17586429/23905400
28	1	776040/7031	25868/35155
29	2	53258901/478108	17752967/23905400
30	1	53486365/478108	10697273/14343240
31	1	54026481/478108	18008827/23905400
32	2	3190471/28124	3190471/4218600
33	1	54360689/478108	54360689/71716200
34	2	13634545/119527	2726909/3585810
35	1	55029105/478108	3668607/4781080
36	1	55182621/478108	18394207/23905400
37	1	55349725/478108	2213989/2868648
38	1	13842543/119527	4614181/5976350

Table 32: Snub Dodecahedron

C MATLAB Code

C.1 Adjacency

```
function adj = adjacency(coords)
% adjacency.m
% adj = adjacency(coords)
% Author: Kevin Stern
%
% Function to find adjacency matrix for given polyhedron.
%
% Inputs:   coords      = n-by-3 matrix, rows give coordinates of vertices
%
% Output:   adj         = n-by-n matrix, adj(i,j) = 0 if i and j are not
%                           adjacent and 1 if they are

%% Initialization
n   = size(coords,1);                       % Number of vertices

%% Find dot products
dots = zeros(n,1);                          % Initialize
for j = 1:n                                 % Loop through vertices
    dots(j) = dot(coords(1,:), coords(j,:));
                                            % Find dot product of vertex
                                                % and node 1
end
dots = unique(dots);                        % Find unique dot products
dots = sort(dots,'descend');                % Sort dot products in
                                                % descending order

%% Create adjacency matrix
adj = zeros(n);                             % Initialize
for i = 1:n                                 % Loop through vertices
    for j = 1:n                             % Loop through vertices
        if abs(dot(coords(i,:), coords(j,:)) - dots(2)) < 10^-10
                                            % If the dot product between
                                                % two vertices is the
                                                % second-largest product...
            adj(i,j) = 1;                       % then the two vertices
        end                                     % are adjacent
    end
end
end
```

C.2 Facefinder

```
function [faces, cycles] = facefinder(coords, nodedepth, Fs)
% facefinder.m
% [faces, cycles] = facefinder(coords, nodedepth, Fs)
% Author: Kevin Stern
%
% Function to find the faces of a polyhedron. Only works for a three-
% dimensional object.
%
% Inputs:   coords      = n-by-3 matrix, rows give coordinates of vertices
%           nodedepth   = 1-by-n vector, each entry is the minimum number
%                           of steps it takes to get from each vertex to
%                           the in vertex
%           Fs          = 1-by-x vector, face size(s) (optional)
%
% Outputs:  faces       = f-by-max(Fs) matrix, rows give each face of
%                           polyhedron
%           cycles      = f-by-max(Fs) matrix, rows give vertices of each
%                           face in connected cycle

%% Initialization
adj = adjacency(coords);
n   = size(adj,1);              % Number of vertices

%% Find E, F, and Fs
E   = sum(sum(adj))/2;          % Number of edges
F   = E-n+2;                    % Number of faces
if nargin == 2
    Fs  = 2*E/F;                % Average face size
end

%% Find faces
numface = 0;                    % Initialize count of found faces
faces = zeros(F, max(Fs));      % Initialize
cycles = zeros(F, max(Fs));     % Initialize

for j = 1:n
    depth = nodedepth(j);       % Depth of node j
    neighbors = find(adj(j,:)); % Find neighbors of node j
    neighbors(nodedepth(neighbors) > depth) = [];
                                % Remove deeper neighbors
    if length(neighbors) < 2; continue; end
                                % If node j has fewer than two neighbors on
                                % same or lower nodedepth, it can't be the
                                % deepest in a cycle
```

```
for k = 1:length(neighbors)             % Loop through neighbors
    knode = neighbors(k);               % Identify neighbor
    for m = k+1:length(neighbors)       % Loop through later neighbors
        mnode = neighbors(m);           % Identify second neighbor
        % Once we have three vertices (j, knode, and mnode), there is
            % at most one face that contains those three vertices
        if adj(knode,mnode) == 1 && any(Fs == 3)
                                        % If j, knode, and mnode all
                                            % touch, then we have a
                                            % triangle face as long as
                                            % that is a possible face
                                            % size
            row = sort([j knode mnode]);
                                        % Create row of j, knode, mnode
            row = padarray(row, [0 max(Fs)-3], Inf, 'post');
                                        % Pad row to length of largest
                                            % face size
            if ismember(row, faces, 'rows')
                                        % If the row is already in
                                            % the faces...
                continue                % skip it
            end                         % Otherwise...
            numface = numface + 1;      % Increment the count of faces
            cycles(numface,:) = row;    % Add the row to cycles
            faces(numface,:) = row;     % Add the row to faces
            continue                    % We have found the face using
                                            % j, knode, and mnode, so
                                            % move to next mnode
        end
        v1 = coords(j,:) - coords(knode,:);
                                        % Find vector from knode to j
        v2 = coords(j,:) - coords(mnode,:);
                                        % Find vector from mnode to j
        n = cross(v1, v2);              % Find cross product
        vertices = Inf*ones(1, max(Fs));
                                        % Initialize array of vertices
        vertices(1:3) = [j knode mnode];
                                        % Input first three vertices
        ind = 3;                        % Set index of vertices
        for node = 1:size(coords,1)   % Loop through nodes
            if ind == max(Fs); break; end
                                        % If vertices is full, break
            plane = dot(coords(node,:)-coords(j,:),n);
                                        % Test if node is in plane
                                            % defined by j, knode,
                                            % mnode
```

```
    if abs(plane) <= 1e-10 && ~ismember(node, vertices)
                                % If node is in plane and
                                    % hasn't been added to
                                    % vertices...
        ind = ind + 1;          % Increment index
        vertices(ind) = node;   % Add node to vertices
        if ind == max(Fs); break; end
                                % If vertices is full, break
    end
end
if ~any(sum(isfinite(vertices)) == Fs); continue; end
                                % If the number of vertices is
                                    % not a possible face size,
                                    % then we have not found a
                                    % face. j, knode, and mnode
                                    % are just coplanar
cycle = Inf*ones(size(vertices));
                                % Initialize cycle
cycle(1:2) = [j knode];         % Input first two vertices
for ind = 3:sum(isfinite(vertices))
                                % Loop to fill out cycle
    for vertex = vertices       % Loop through vertices
        if isinf(vertex); break; end
                                % If vertex is infinite, then
                                    % we have not found a face
        if adj(vertex, cycle(ind-1)) && ~ismember(vertex, cycle)
                                % If vertex is adjacent to
                                    % previous vertex in cycle,
            cycle(ind) = vertex;    % add it to the cycle
            break
        end
    end
    if isinf(cycle(ind)); break; end
                                % If we looped through all
                                    % vertices without adding
                                    % one, then we have not
                                    % found a face
end
if sum(isfinite(vertices)) ~= sum(isfinite(cycle))
                                % If number of vertices in
                                    % cycle is different then
                                    % number of vertices in
                                    % plane, then we have not
                                    % found a face
    continue
end
```

```
            vertices = sort(vertices);      % Sort vertices
            if ismember(vertices, faces, 'rows'); continue; end
                                            % If vertices is already in
                                                % faces, then we have found
                                                % a repeat face
            numface = numface + 1;          % Increment count of faces
            faces(numface,:) = vertices;    % Add vertices to faces
            cycles(numface,:) = cycle;      % Add cycle to cycles
            continue                        % We have found the face using
                                                % j, knode, and mnode, so
                                                % move to next mnode
        end
    end
end
if numface == size(faces,1)                 % If we have all the faces...
    if any(faces == 0)                      % If there are zeros in faces
        error('zeros in faces')                 % raise an error
    end
    [faces, I] = sortrows(faces);           % Sort the faces
    cycles = cycles(I,:);                   % Sort the cycles
    % The code below before the return statement isn't
        % necessary, but it can be used to check the number
        % of each face size
    facecounter = zeros(size(Fs));
                                            % Initialize count of number
                                                % of each face size
    for i = 1:size(faces,1)                 % Loop through faces
        fs = sum(isfinite(faces(i,:)));     % Find size of face
        facecounter(Fs == fs) = facecounter(Fs == fs) + 1;
                                            % Increment count of number
                                                % of each face size
    end
    return
end
error('end of loops')                       % If we didn't find enough
                                                % faces to return earlier,
                                                % then raise an error
end
```

C.3 Vertpathfinder

```
function vertpath = vertpathfinder(adj, nodedepth, vpdict, in, out)
% vertpathfinder.m
% vertpath = vertpathfinder(adj, nodedepth, vpdict, in, out)
% Author: Kevin Stern
%
% Function to find all possible minimum-length paths from the out vertex to
% the in vertex. In general, in = 1.
%
% Inputs:   adj         = n-by-n adjacency matrix
%           nodedepth   = 1-by-n vector, each entry is the minimum number
%                         of steps it takes to get from each vertex to
%                         the in vertex
%           vpdict      = Map object that stores previously generated
%                         vertpaths for vertices closer to in
%           in          = vertex that we are finding the path to
%           out         = vertex that we are finding the path from
%
% Output:   vertpath    = x-by-v vector, each row gives a sequence of
%                         vertices to take to get from the out vertex to
%                         the in vertex

%% Initialization
if in == out; vertpath = in; return; end
vertpath    = [];                           % Initialize

%% Find vertpath
possiblesteps = find(adj(out,:));           % Find all neighboring vertices
for step = possiblesteps                    % Loop through neighbors
    if nodedepth(step) >= nodedepth(out)    % If neighbor is at least as
                                                % deep as the out vertex...
        continue                                % skip it
    end
    numpaths = size(vpdict(step),1);        % Find number of paths to in
    tempstep = [out*ones(numpaths,1) vpdict(step)];
                                            % Find full paths using
                                                % current neighbor
    vertpath = [vertpath;tempstep]; %#ok<AGROW>
                                            % Add full paths to vertpath
end
end
```

C.4 Findlayers

```
function vertlayers = findlayers(coords, in)
% findlayers.m
% vertlayers = findlayers(coords, in)
% Author: Kevin Stern
%
% Function to find the layer of each node for a given polyhedron. Vertices
% on the same layer should 1) be the same distance from the in node, 2a) be
% the same number of steps from the in node. When each edge is categorized
% based on the size of the faces that it separates, vertices on the same
% layer should also have the quality that 2b) all possible shortest paths
% from one vertex to the in node should pass through the same kinds of
% edges as all possible shortest paths from another vertex to the in node.
%
% Inputs:   coords      = n-by-3 matrix, rows give coordinates of vertices
%           in          = ending node (optional)
%
% Output:   layers      = 1-by-n matrix, entries give layers of vertices

%% Initialization
if nargin == 1; in = 1; end
n   = size(coords, 1);                      % Number of vertices
adj = adjacency(coords);                    % Adjacency matrix

%% Create nodedepth
% nodedepth gives the minimum number of steps from each vertex to in node.
nodedepth = -1 * ones(1,n);                 % Initialize
nodedepth(in) = 0;                          % The depth of in node is 0
q = in;                                     % Start search for all vertices
while ~isempty(q)                           % While not all vertices have
                                                % been found
    currentnode = q(1);                     % Take the first vertex in list
    q(1) = [];                              % Remove it from list
    for i = 1:n                             % Loop through all vertices
        if adj(i,currentnode) && nodedepth(i) == -1
                                            % If vertex is adjacent to
                                                % first vertex in list
                                                % and vertex hasn't been
                                                % found yet...
            nodedepth(i) = nodedepth(currentnode) + 1;
                                                % record its depth
            q = [q i]; %#ok<*AGROW>         % Add vertex to end of list
        end
    end
end
```

```
%% Find E, F, Fs, and faces
E   = sum(sum(adj))/2;          % Number of edges
F   = E-n+2;                    % Number of faces
if (2*E/F) ~= round(2*E/F)      % The average face size is NOT an integer
    Fs  = input('Type face sizes as a row vector, ex: [3 4]\n');
else
    Fs = 2*E/F;
end
[faces, cycles] = facefinder(coords, nodedepth, Fs);

%% Create catalog of edges by face sizes
% Each row has the format [minvertex maxvertex minfacesize maxfacesize].
% The vertices are the endpoints of the edge.
% The face sizes are of the faces that the edge separates.
% Only the first two columns are populated right now.
edges   = zeros(E, 4);          % Initialize
ind     = 0;
for i = 1:n                     % Loop through combinations
    for j = i+1:n                   % of non-repeating vertices
        if adj(i,j)             % If vertices are neighbors,
            ind = ind + 1;          % add the edge to catalog
            edges(ind, 1) = i;
            edges(ind, 2) = j;
        end
    end
end
if any(edges(:,1:2) == 0)       % Not all the edges were found
    error('Not all edges found')
end
if ind ~= E                     % Too many edges were found
    error('Too many edges')
end

%% Create face2edges
% Each row gives the indices of the edges that make up each face's sides.
% The number of finite entries in each row equals the size of the face.
face2edges = Inf*ones(size(faces));         % Initialize
for i = 1:F                                 % Loop through faces
    ind = 0;
    for j = 1:sum(isfinite(faces(i,:)))-1   % Loop through edges in face
        edge = sort(cycles(i,j:j+1));
        for k = 1:size(edges,1)             % Find edge index in catalog
            if edges(k,1:2) == edge
                ind = ind + 1;
                face2edges(i,ind) = k;      % Add edge index to face2edges
```

```
            end
        end
    end
    edge = sort([cycles(i,1) cycles(i,j+1)]);
                                            % Add final edge in face
    for k = 1:size(edges,1)
        if edges(k,1:2) == edge
            ind = ind + 1;
            face2edges(i,ind) = k;
        end
    end
    face2edges(i,:) = sort(face2edges(i,:));
                                            % Sort edges by index
end

%% Create face2faces
% Each row gives the indices of the faces that border each face.
% The number of finite entries in each row equals the size of the face.
face2faces = Inf*ones(size(faces));         % Ininialize
for i = 1:F                                 % Loop through faces
    for j = 1:sum(isfinite(faces(i,:)))     % Loop through edges in face
        edgeind = face2edges(i,j);          % Find edge index
        [faceinds, ~] = find(face2edges == edgeind, 2);
                                            % Find both face indices that
                                                % contain edge
        face2faces(i,j) = faceinds(faceinds ~= i);
                                            % Record other face index
    end
end

%% Create facetypes
% Each entry takes the form "a.bcd...", where a is the number of sides of
% the given face, and bcd... give the number of sides of each bordering
% face in ascending order of size.
facetypes = strings(F, 1);                  % Initialize
for i = 1:F                                 % Loop through faces
    facesize = sum(isfinite(face2faces(i,:)));
                                            % Find size of face
    adjfaces = zeros(1, facesize);          % Initialize array of
                                                % bordering face sizes
    for j = 1:facesize                      % Loop through bordering faces
        adjfaces(j) = sum(isfinite(face2faces(face2faces(i,j),:)));
                                            % Find size of bordering face
                                                % and record it
    end
    adjfaces = sort(adjfaces);              % Sort sizes in ascending order
```

```
    facetypes(i) = string(facesize)+"."+strjoin(string(adjfaces),'');
                                            % Record entry
end

%% Complete catalog of edges by face sizes
% Each row has the format [minvertex maxvertex minfacesize maxfacesize].
% The vertices are the endpoints of the edge.
% The face sizes are of the faces that the edge separates.
% The final two columns are populated right now.
for ind = 1:E                               % Loop through edges
    for k = 1:size(faces,1)                 % Loop through faces
        if ismember(edges(ind,1), faces(k,:)) && ...
                ismember(edges(ind,2), faces(k,:))
                                            % If the edge is in the face...
            if ~edges(ind,3)                    % and it's the first found,
                edges(ind,3) = double(facetypes(k));
                                                % add it to catalog
            else                                % and it's second found,
                edges(ind,4) = double(facetypes(k));
                                                % add it to catalog
                break
            end
        end
    end
    edges(ind,3:4) = sort(edges(ind,3:4));
end
if any(edges == 0)                          % Some edge wasn't added
    error('Catalog of edges is incomplete')
end

%% Create edge dictionary
% Each key has the form num2str([minvertex maxvertex])
% which looks like 'minvertex  maxvertex' (two spaces).
% Each value has the form [minfacesize maxfacesize].
% Essentially, convert catalog of edges into a dictionary
edgedict = containers.Map('keyType','char','ValueType','any');
                                            % Initialize
for i = 1:E                                 % Loop through edges
    edgedict(num2str(edges(i,1:2))) = edges(i,3:4);
                                            % Add edge to edgedict
end

%% Create list of dists between vertices
% The dot products between the in node and all other vertices are
% calculated, and then sorted in descending order.
% The dot product is a stand-in for distance, since the vertices are
```

```
% equispaced about the origin.
dists = [];                                 % Initialize
for j = 1:n                                 % Loop through vertices
    SUM = dot(coords(in,:), coords(j,:));   % Compute dot product
    if not(ismembertol(SUM, dists, 10^-10)) % Add dist to array
        dists = [dists SUM];
    end
end
dists = sort(dists,'descend');              % Sort distances

%% Create preliminary layers, based on distance
% The first go at layers is just based on the distance of each vertex from
% the in node.
distlayers = zeros(1,n);                    % Initialize
for i = 1:length(dists)                     % Loop through dists
    for j = 1:n                             % Loop through vertices
        if abs(dot(coords(in,:), coords(j,:)) - dists(i)) < 10^-10
                                            % If dist between vertex and
                                                % in node matches dist...
            distlayers(j) = i;                  % add to distlayers
        end
    end
end

%% Create vertpath dictionary
% Each vertex index is a key
% Each value is an a-by-b matrix, where a is the number of unique (in
% vertices) shortest-length paths from the vertex to the in node, and b is
% (nodedepth+1). Each row gives a shortest-length path from the vertex to
% the in node, including both ends.
vertpathdict = containers.Map('keyType','double','ValueType','any');
                                            % Initialize
for depth = 0:max(nodedepth)                % Loop through depths
    verts = find(nodedepth == depth);       % Find vertices with same depth
    for vert = verts                        % Loop through vertices
        vertpath = vertpathfinder(adj, nodedepth, vertpathdict, in, vert);
                                            % Find all shortest-length
                                                % paths from vertex to
                                                % in node
        vertpathdict(vert) = vertpath;      % Record in dictionary
    end
end

%% Createedgepath dictionary
% Each vertex index is a key
% Each value is a 1-by-2*c*e matrix, where c is the number of unique (in
```

```
% edge types) shortest-length paths from the vertex to the in node, and e
% is the number of edges in each shortest-length path (= nodedepth).
% Each row contains the edge types of all shortest-length paths to in node,
% one after the other in a single row. Duplicate paths using the same
% number of the same edge type are removed. The number of each edge type
% within a path does not affect the final result, but all edges are
% included to make the matrix sizes match.
% For example: if the edge types of the four shortest-length paths are as
% follows:
%   {3,4},{5,6},{5,6}
%   {3,4},{4,5},{4,5}
%   {3,4},{4,5},{4,5}
%   {3,4},{3,4},{5,6}
% The a-by-2*e matrix will then be
% [3 5 5 4 6 6
%  3 4 4 4 5 5
%  3 4 4 4 5 5
%  3 3 5 4 4 6]
% After eliminating the extraneous third row and reshaping to 1-by-2*c*e
% matrix, the matrix that is saved into edgepathdict would be
% [(3 5 5 4 6 6) (3 4 4 4 5 5) (3 3 5 4 4 6)]
% where the parentheses indicate the three unique paths.
edgepathdict = containers.Map('keyType','double','ValueType','any');
                                          % Initialize
for vert = 1:n                            % Loop through vertices
    matrix = zeros(size(vertpathdict(vert)));
                                          % Initialize matrix
    matrix(:,1) = [];                     % Trim matrix to correct size,
                                              % a-by-e, where a is the
                                              % number of unique (in
                                              % vertices) shortest-length
                                              % paths.
    vpd = vertpathdict(vert);             % Copy vertpathdict for vertex
    for i = 1:size(matrix,1)              % Loop through rows
        for e = 1:size(matrix,2)          % Loop through edges
            edge = sort(vpd(i,e:e+1));    % Identify vertices of edge
            matrix(i,e,1:2) = edgedict(num2str(edge));
                                          % Find edge type and record
                                              % into matrix, which is now
                                              % an a-by-e-by-2 array
        end
    end
    if vert == in                         % If already at in node...
        edgepathdict(vert) = matrix;          % record in dictionary
        continue
    else                                  % Otherwise, clean matrix
```

```
        for i = 1:size(matrix, 1)           % Loop through rows
            temp = reshape(matrix(i,:,:), size(matrix, 2), 2);
                                            % Reshape 1-by-e-by-2 array
                                                % into e-by-2 matrix
            temp = sortrows(temp);          % Sort edges in row by smaller
                                                % face then by larger face
            temp = reshape(temp, 1, size(matrix, 2), 2);
                                            % Reshape e-by-2 matrix back
                                                % into 1-by-e-by-2 array
            matrix(i,:,:) = temp;           % Record sorted row into matrix
        end
        matrix = [matrix(:,:,1) matrix(:,:,2)];
                                            % Reshape a-by-e-by-2 array
                                                % into a-by-2*e matrix.
                                                % Each row represents an
                                                % edge path to the in node
        matrix = unique(matrix, 'rows');    % Eliminate repeated edge paths
        matrix = reshape(matrix', 1, numel(matrix));
                                            % Reshape into 1-by-2*c*e
        edgepathdict(vert) = matrix;   % Record matrix into dictionary
    end
end

%% Create vertlayers
% Vertlayers is described in the help for this function.
vertlayers      = zeros(1,n);               % Initialize
vertlayers(in)  = 1;                        % in node is always layer 1
offset          = 0;                        % Offset is used for splitting
                                                % layers
for i = 2:max(distlayers)                   % Loop through distlayers
    verts   = find(distlayers == i);        % Find vertices with same dist
                                                % from in node
    epdmaxlen = 0;                          % Initialize longest shortest-
                                                % length path for vertices
    for j = 1:length(verts)                 % Loop through vertices
                                                % to find longest shortest-
                                                % length path
        if length(edgepathdict(verts(j))) > epdmaxlen
                                            % If length is longer than
                                                % current max...
            epdmaxlen = length(edgepathdict(verts(j)));
                                                % replace max with length
        end
    end
    epds = zeros(length(verts), epdmaxlen); % Initialize matrix of edge
                                                % path rows
```

```
    for j = 1:length(verts)                 % Loop through vertices
        if length(edgepathdict(verts(j))) < epdmaxlen
                                            % If length is less than
                                                % max length...
            edgepathdict(verts(j)) = padarray(edgepathdict(verts(j)), ...
                [0 epdmaxlen-length(edgepathdict(verts(j)))], 0, 'post');
                                            % pad row with zeros at end
        end
        epds(j,:) = edgepathdict(verts(j)); % Record row into matrix
    end
    epds = unique(epds, 'rows', 'stable');  % Find unique rows
    for vert = verts                        % Loop through vertices
        [~, epdrow] = ismember(edgepathdict(vert), epds, 'rows');
                                            % Find vertex's row in matrix
        vertlayers(vert) = i + offset + epdrow - 1;
                                            % Record layer of vertex
    end
    offset = offset + size(epds, 1) - 1;    % Adjust offset, which is a
                                                % measure of how much layer
                                                % splitting is occurring
end
end
```

C.5 Findhits

```
function [symlayerhits] = findhits(coords, in)
% findhits.m
% [symlayerhits] = findhits(coords, in)
% Author: Kevin Stern
%
% Takes a n-by-m coordinate matrix, where n is the number of vertices
% and m is the dimension of the space in which the coordinates live,
% and computes the equivalent resistances from the given node to all
% other nodes. If no node is given, the first node is taken.
% Lovasz's hanging method is used in this function, simplifying by layers.
% The hitting time H(a,b) is the average number of random walk steps it
% takes to walk from node a to node b. Lovasz gives the formula
% H(a,b) = 1 + (1/deg(a))*sum_(v in N(a))H(v,b),
% where deg(a) is the number of neighbors of node a, and N(a) denotes the
% neighbors of node a.
% After collecting the nodes into symmetric layers, we can rewrite as
% H({a_n},b) = 1 + (1/sum_(a_i in a_n)deg(a_i))*
%                        (sum_(a_i in a_n)sum_(v_i in N(a_i))H(v_i,b)),
% where {a_n} denotes the list of all nodes in a layer, and
% H({a_n},b) denotes the (identical) hitting time of each node in a layer.
% The hitting time can be converted to resistance by dividing by the total
% number of edges in the network.
%
% Inputs:   coords      = n-by-3 matrix, rows give coordinates of vertices
%           in          = ending node (optional)
%
% Output:   symlayerhits    = 1-by-l matrix, entries give hitting times of
%                                layers in exact fractional form

%% Initialization
if nargin == 1; in = 1; end
adj = adjacency(coords);                    % Adjacency matrix
layers = findlayers(coords, in);            % Find layers
A = eye(max(layers)-1);                     % Decimal coefficient matrix
symA = sym('A',size(A));                    % Symbolic coefficient matrix
b = ones(max(layers)-1,1);                  % Decimal RHS vector
symb = sym(b);                              % Symbolic RHS vector

%% Construct matrix equation
for inlayer = 2:max(layers)                 % Loop through layers
    inverts = find(layers==inlayer);        % Find vertices in layer
    denom = 0;                              % Initialize denominator
    for invert = inverts                    % Loop through vertices
        denom = denom + sum(adj(invert,:)); % Denominator is equal to sum
```

```
                                                % of degrees of vertices
                                                % in layer
    end
    for outlayer = 2:max(layers)            % Loop through layers
        numer = 0;                          % Initialize numerator
        outverts = find(layers==outlayer);  % Find vertices in layer
        for invert = inverts                % Loop through vertices in
                                                % first loop layer
            for outvert = outverts          % Loop through vertices in
                                                % second loop layer
                if adj(invert,outvert)      % If vertices are adjacent...
                    numer = numer - 1;          % decrement numerator
                end
            end
        end
        if inlayer == outlayer              % If layers are the same...
            symA(inlayer-1,outlayer-1) = 1 + sym(numer)/sym(denom);
                                                % find fraction, add one,
                                                % and record in matrix
        else                                % If layers are not the same...
            symA(inlayer-1,outlayer-1) = sym(numer)/sym(denom);
                                                % find fraction and record
                                                % in matrix
        end
    end
end

%% Solve matrix and compute/display results
symlayerhits = [0;symA\symb];               % Find symbolic inverse
[symlayerhits,I] = sort(symlayerhits);      % Sort hitting times
newlayers = zeros(size(layers));            % Initialize reordered layers
for layer = 1:max(layers)                   % Loop through layers
    newlayers(layers==I(layer)) = layer;    % Record reordered layer
end
disp('newlayers')                           % Display reordered layers
for layer = 1:max(newlayers)                % Loop through reordered layers
    disp([layer find(newlayers==layer)])    % Display layer number and
                                                % vertices in layer
end
end
```