Resistance of Infinite Networks

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Sharma Maharaj

Annandale-on-Hudson, New York
May, 2015

Abstract

We study the resistance of infinite electrical networks that contain a single source and a sink at infinity. We will be exploring the total resistance on the infinite binary tree, line, grid, and hyperbolic grid using two different methods. The two methods are discrete forms of Laplace's equation and the heat equation. These two methods are used to find the potentials of nodes in the networks. By modeling the resistance of large subnetworks in Sage we are able to estimate the resistance of infinite networks. Using Laplace's equation we were able to determine the resistance on the binary tree and line. Using the heat equation we were able to obtain a resistance for all four specified networks. These two methods may prove useful for more complicated infinite networks.

Contents

List of Figures

Dedication

To my brothers Vinod, Narcisso, Amin, my sister Sarita, and my beloved Arobi, may your potentials never converge.

Acknowledgments

It was Sabrina Bryan who once said, “You can do anything as long as you have the passion, the drive, the focus, and the support.” Without the support of my family, friends, advisors, and mentors the completion of this project would not be possible. First and foremost I would like to thank my advisor James Belk for your constant guidance and mentorship, this project would truly not be possible without you. To my mother, thank you for the wisdom you passed on to me. I would not have come nearly as far without it. To my father, thank you for the strength you passed on to me, without it I would not have the drive to never give up even through failure. To my mentor Domenec, thank you for giving me my love for math.

Matthew Deady, thank you for showing me that I can accomplish things I thought impossible through hard work and patience. John Cullinan, thank you for showing me that grades are not in any way a measure of intelligence and that exams are just that, exams. To Jane Duffstein, the weekly meetings to discuss everything that happened through the week as well as the progress I made definitely helped during senior year.

Lastly I would like to thank my friends Borahnie Garcia, and Casey Tong, those nights you guys would stay up with me playing video games, listening to the ideas and concepts, and asking me questions on my project really helped me to get a grip on exactly what I was doing. To Christeina Wade I owe you more than you know for all you have done for me, too much to actually list here, without you I don’t think I would be as happy with myself as I am now. Lastly I would like to thank my PS4, Evolve and Crota for reminding me that its ok to take a break every now and then.

Introduction

An infinite network is an infinite collection of sources, sinks, and resistors that are arranged in a particular manner. We will be looking at infinite networks that have only one source, and a "sink" at infinity, where every resistor in the network has a resistance of 1Ω. It would be intuitive to think that an infinite network would have infinite resistance. However, as we shall see, certain infinite networks such as the binary tree do have a finite resistance.

Another infinite network known to have a finite resistance is the **hyperbolic grid**. The hyperbolic grid is a certain graph drawn in the hyperbolic plane, shown in Figure 1. Every rectangle in the hyperbolic grid has one rectangle to each side, two rectangles below, and half of a rectangle above it. The hyperbolic grid is analogous to the normal grid drawn in the Euclidian plane in that all rectangles of the hyperbolic grid are congruent, meaning they all are the same size.

There exists a close relation between electrical networks and random walks on graphs. Given a graph G and a starting vertex v, an associated **random walk** is a path that starts at v and travels to some random vertex adjacent to v, and continues by traveling to a new random adjacent vertex at each step. We say that a random walk is **transient** if the

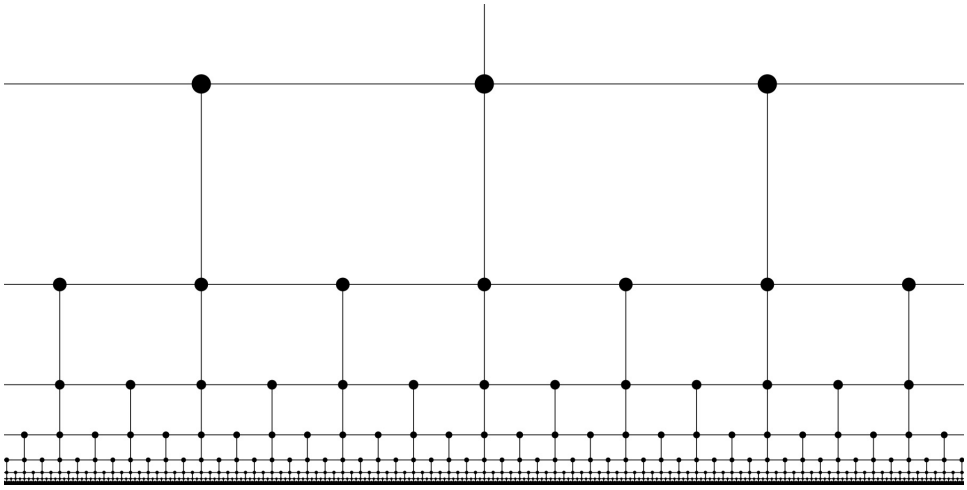

Figure 1

probability of returning to the starting vertex is less than one. It is known that a random walk on an infinite network is transient if and only if the network has finite resistance (see [1]). It is also known that the infinite hyperbolic grid is in fact transient (due to the fact that it is non-amenable, see [3]), so the resistance must be finite. However, it is not known what the resistance of the hyperbolic grid actually is.

Electric networks are also related to harmonic functions on graphs. A real valued function on the vertices of some graph G is a **harmonic function** if the value of each vertex is the average of the values of all adjacent vertices. This is a discrete analogy of continuous harmonic functions. For the networks we consider, the potential function on the vertices of a network is harmonic except at the source and sink. The potential function satisfies two boundary conditions for our infinite networks. The first being, the potential of the source is always a constant of one. The second condition tells us, if we take any sequence of distinct vertices on an infinite network the potentials should tend towards zero.

While we know the potential function on an infinite network is harmonic it is hard to figure out what those potentials actually are. To solve this we will take a large subnetwork of our infinite network and solve the potentials and resistance of the subnetwork. Then

we will take a larger subnetwork and again solve for the potentials and resistance. We will continue this trend of taking larger networks and solving for potentials and resistance until the potentials of the vertices start to converge, along with the resistance. The goal is to develop numerical methods in Sage that can create these large subnetworks and find the potentials and resistances of them in order to find the potentials and resistance on the infinite network.

We will be using harmonic functions along with two different methods in order to explore the feasibility of using numerical approximation in Sage to estimate the resistances of infinite networks. The two methods we can use to estimate the resistance of the infinite networks involve using Laplace's equation and the heat equation for networks. The first method we will attempt to use is a discrete form of Laplace's equation. This method starts by taking a finite subnetwork of an infinite network, and creates an equation for every vertex in the subnetwork. The equations represent the fact that the potential of the vertex is equal to the average potential of all adjacent vertices. After all vertices are assigned an equation we will create a matrix using all equations, and the last step is to use Gaussian elimination to determine the potentials of each vertex. From this point, we calculate the potentials of increasingly larger subnetworks and determine if the potentials of the vertices are converging. If the potentials converge to a value less than one then we know that the infinite network is transient and has a finite resistance, however if all potentials converge to one as we take larger and larger subnetworks then we say the network has an infinite resistance.

The second method is the heat equation. The heat equation works much better for larger subnetworks than Laplace's equation because it eliminates the need to create matrices. The heat equation starts by taking some finite subnetwork of an infinite network and assigns every vertex that is not the source a potential of zero, and assigns the source a potential of one. Then the heat equation will enter an iterative process, that will happen

some k times, where at each iteration the values of the potentials of every vertex, except for the source and sink, will become the average of the potential of all adjacent vertices. After many iterations we expect the potentials to converge to the harmonic function of the subnetwork. We will then take a larger subnetwork and repeat the process. We will continue this process of taking larger subnetworks until the potentials of the subnetworks converge to the harmonic function on the infinite network.

We test both methods on different infinite networks including the infinite binary tree, the infinite line, and the infinite grid. The finite resistance of the infinite binary tree is known to be 1Ω, while the infinite line is known to have infinite resistance. The infinite grid is also well understood but more complicated than both the infinite line and binary tree. We will use the grid to test the limits of both of our methods. Lastly it is known that the infinite hyperbolic grid does in fact have finite resistance, although it is not known exactly what this resistance is. We will use both methods to estimate what the finite resistance on the hyperbolic grid is.

We use code written in Sage in order to create these subnetworks and find the resistance along them. Starting with the `Digraph` data structure, we begin by creating a subnetwork and then calculate the potentials either with Laplace's equation or the heat equation. Since the potential functions of the binary tree and line are well understood we will use both methods on those two networks to verify that our code is working properly. Next we will attempt to solve for the resistance of the infinite grid, and hyperbolic grid.

Both methods provided very clear results on all networks. Using the Laplace equation we were able to determine the finite resistance on the infinite binary tree to be 1Ω which is exactly what we expected it to be. We were also able to confirm that the infinite line has infinite resistance. When attempting to find the resistance of the infinite grid we came across a problem: Laplace's equation works well for small subnetworks with fewer than 10,000 or so vertices but failed as the subnetworks became larger than this. This is due

to the fact that as we take larger subnetworks Laplace's equation would have to create larger matrices to account for the new vertices. Eventually the function would take more than a week to give any results for our subnetwork, and in later cases the computer ran out of memory and could not produce any result. Using this method we were able to find the potentials and resistance of the first 12 subnetworks of the grid, after this the function would break down. We attempted to use Laplace's equation on the hyperbolic grid but were only able to produce the potentials and resistances of the first five subnetworks. It became apparent that if Laplace's equation did not work for the grid it definitely would not work for the hyperbolic grid as the hyperbolic grid adds an exponential number of vertices for each new subnetwork.

We were able to solve the problem given by Laplace's equation using the heat equation, which allowed us to get rid of the matrix and rely strictly on the subnetwork. To verify this method does in fact work we use the heat equation on the infinite binary tree and line and were able to produce the exact same results as Laplace's equation. Using our results from the heat equation we were able to find a best fit function for the resistance of fine grids that was within 1×10^{-3} accuracy of the actual resistance. Surprisingly the resistance of the finite subnetworks of the infinite grid appears to grow logarithmically with grid size.

In order to find the total resistance on the hyperbolic grid we used Laplace's equation on small subnetworks of the hyperbolic grid. Then we used the heat equation to ensure both methods produced the same result. After confirming that both methods gave the same result we used the heat equation to find the potentials and resistances of two more subnetworks of the hyperbolic grid. In the next step we used the heat equation to determine potentials of larger subnetworks and estimate the total resistance of the hyperbolic grid, which turned out to be around 0.4847Ω.

1
Basic Networks

This chapter we will focus on building certain notions for networks. In Section 1.1 we will define what an electrical network is and what the two laws are for electrical networks. Then we will show exactly how we may determine the total resistance on one of these networks, and demonstrate exactly how we can find the total resistance starting with finding the potentials of all vertices in the network. Then in Section 1.2 we will define what a harmonic function on a network is, and how harmonic functions apply to networks. Lastly in Section 1.3 we will show how the rules for resistors in series and parallel can be derived

1.1 Electrical Networks

Before we define what a network is we have to first understand what a graph is. A graph is a collection of vertices and edges that are arranged in a certain way. There are two types of graphs, connected and disconnected. We can think of a disconnected graph as being "split," meaning that if you were to start at some arbitrary vertex and followed the edges to try and get to every other vertex you would not be able to. We can think of a connected graph as the opposite, where if you started at some arbitrary vertex and followed along

the edges you could reach any other vertex in the graph. In this project we will only be dealing with connected graphs. An electrical network can be thought of as a graph on which electricity flows. Each edge on the graph is assigned two values, a **current** I and a **resistance** R. The current can be either positive or negative. We say that a current is positive between two vertices A and B if the current moves from A to B and we say the current is negative if the current moves from B to A. Each vertex on the graph is also assigned a **potential** V.

Networks operate under two laws. The first law is known as Ohm's Law.

Ohm's Law. The product of the current and resistance on some edge is equal to the potential difference between the two vertices connected to that edge, (See [4] for a full treatment).

$$\Delta V = IR$$

Here ΔV is the potential difference between the two vertices connected to the edge, I is the current, and R is the resistance of the edge. When we take the difference of the two vertices we always use the potential of the vertex the current is traveling to minus the potential of the vertex the current is coming from.

The second law is referred to as Kirchhoff's current law.

Kirchhoff's Current Law. This law states that the current going into a vertex must equal the current out of it. We can also think of this as the total amount of current coming into as well as leaving a vertex must sum to zero, where the current out of a vertex is negative. (See [4] for a full treatment).

Figure 1.1.1 illustrates the idea of Kirchhoff's current law for a vertex of degree four. For the networks we are considering there are two vertices that can ignore Kirchhoff's current law. The first vertex is called the **source** which is the vertex in the network that

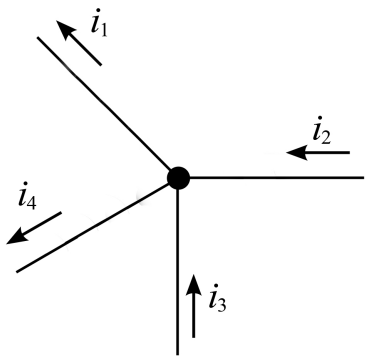

Figure 1.1.1: $i_2 + i_3 = i_1 + i_4$

always has a constant potential of one and a positive net current flow. The other is the sink that always has a constant a potential of zero and can have a net negative flow of current.

1.1.1 Units

In physics when we talk about resistances and currents we usually use units to describe them. For example if we have some network with a source, we say the source has a potential of 1a (amp). Another example is if we have an edge with a resistance 1, we say that the edge has resistance 1Ω(Ohm). These units are important as they help to distinguish exact amounts current or resistance. In this paper we will be mostly ignoring units. We will assume that any currents I are measured in amperes (a), resistances R are measured in ohms (Ω), and potentials are measured in volts (V). We can also assume from this point on that anytime we say network we are referring to an electrical network.

1.1.2 Total Resistance

In any given network every edge is given some resistance R. Using Ohm's law we can write the resistance on any edge, is given by the quotient of the difference in potential of the two vertices connected to the edge and the current along that edge.

$$R = \frac{\Delta V}{I}$$

We can also talk about the total resistance on a network.

Definition 1.1.1. The **total resistance** on any network is given by,

$$R_{total} = \frac{\Delta V_{total}}{I_{total}}$$

Where the total current is the total current out of the source. For finite networks the total current can also be given by the total current into the sinks. We know that, due to Kirchhoff's current law, that no vertex in our network can produce any current, except for the source, therefore the total current in the network must be coming from the source. In order to find the total potential change we need to take the difference of the point where current starts to follow in the network and where it ends. We know that in all networks the current starts at the source and ends at a sink. So the total potential difference will always be known as the source and sink are fixed constants. Determining the total resistance becomes a matter of figuring out how much current leaves the source. If we can determine the total current then we can find the total resistance in the same way that we found the resistance on an edge. This equation treats the entire network as if it was one resistor with resistance R_{total}. In this paper we will assume that in any given network the resistance of every edge in the network is 1Ω.

We will now show how one can use Kirchhoff's and Ohm's Laws to find the total resistance of closed networks.

Example 1.1.2. We will use a relatively simple network with one source and one sink. Larger networks can have more than one source and sink. The network we will be using will have three resistors. As seen in Figure 1.1.2. All of these resistors will have a resistance of 1Ω. It is important to note that there is an edge between the sink and the resistors, however this edge has no resistance and in any network an edge with no resistance does not effect the network in any way.

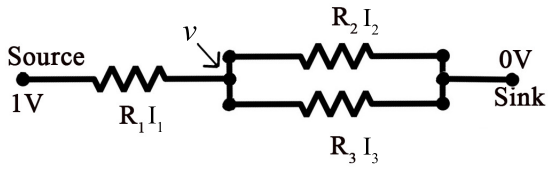

Figure 1.1.2: Simple network

This network has a source with a potential of $1V$ and a sink with a potential of $0V$. We also have a vertex v in the network with an unknown potential. In order to find the total resistance we need to find the potential of all vertices in the network. In order to find the unknown potential we will start by using Kirchhoff's current law to determine all of the currents involved in the network. Due to Kirchhoff's law the current flowing into any vertex is equal to the current flowing out of that vertex. For this network finding the current will be simple as there is only one vertex in the middle so all of the current in the network must be passing through this point. The current starts at the source and travels along the first resistor to our vertex v, we will call this current I_1. At this point the current splits into the remaining two resistors R_2 and R_3, we will call the current along these edges I_2 and I_3 respectively. The current then comes together again to go to the sink. This leads us to the first equation of the network.

$$I_1 = I_2 + I_3 \tag{1.1.1}$$

We can now use Ohm's Law to find the rest of the equations for the network. Ohm's law tells us that the potential difference between two vertices is equal to the product of the current between them and the resistance. This will be relatively simple as the resistance of all the resistors are 1Ω. This means for this network the current along any edge is equal

to the change in potential of the vertices. For our first edge we have

$$\frac{1-v}{1} = I_1 \tag{1.1.2}$$

Similarly for the edges R_2 and R_3 we have

$$\frac{v-0}{1} = I_2 \tag{1.1.3}$$

$$\frac{v-0}{1} = I_3 \tag{1.1.4}$$

This gives us a system of equations that we can rewrite and solve for.

$$I_1 - I_2 - I_3 = 0$$

$$v + I_1 = 1$$

$$I_2 - v = 0$$

$$I_3 - v = 0$$

We have four equations with four unknowns solving for the potential of the unknown vertex in the network we obtain a value of $v = \frac{1}{3}$. We can now use Ohm's law to find the total resistance in the network. We know that in order to find the total resistance We need to determine the total potential change in the network which is one. Now we need to find the total current. We know that all of the current that could ever be in the network comes from the source, because all other vertices in the network must obey Kirchhoff's current law. For this particular network we know that all of the current must be traveling along the first edge, as it is the only edge connected to the source. So our total current is equal to I_1, which we solved for. This leaves us with the equation.

$$1 - 0 = \frac{2}{3}R \tag{1.1.5}$$

solving for R we obtain

$$R = \frac{3}{2} \tag{1.1.6}$$

This tells us that we have a total resistance of $\frac{3}{2}\Omega$.

1.2 Harmonic Functions on a Graph

Using Ohm's and Kirchhoff's laws to determine the potentials is just one method, there is another way to determine the potentials of vertices in networks. It is possible to have a some function f on a graph G such that $f : V(G) \to \mathbb{R}$ where $V(G)$ is the set of vertices of G. There is a special class of functions that can be applied to networks, these functions are called harmonic and can help determine the potentials in a network. A harmonic function on a network will take the vertex set of the network and map it to the real numbers, these numbers would be the potentials of the vertices in the network.

Definition 1.2.1. A **harmonic** function on a graph G is a function h where $h\colon V(G) \to \mathbb{R}$ where $V(G)$ is the vertex set of G, and for all $v \in G$ we have

$$h(v) = \frac{f(v_1) + f(v_2) + \ldots + f(v_k)}{k}$$

where $v_1, v_2, \ldots, v_n$ are the vertices adjacent to v

The definition states that h is a harmonic function on a graph G if h takes the vertex set of G and maps its to the real numbers, moreover if we took any vertex $v \in V(G)$ the value of the function at v is the average of value of the function of all vertices that are adjacent to v.

Proposition 1.2.2. Let N be a network and suppose that every edge in N has resistance 1Ω. Then the electric potential on $N$is harmonic except at the source and sink.

Proof. Let N be some network and let v be some vertex in N. Also let $v_1, v_2, \ldots, v_n$ be all vertices that are adjacent to v and $I_1, I_2, \ldots, I_n$ be the current on the edges between v and $v_1, v_2, \ldots, v_n$. Then for every vertex v_k that is adjacent to v we can use Ohms law to

describe the relationship between v and all adjacent v_k

$$h(v) - h(v_1) = I_1 R$$

$$h(v) - h(v_2) = I_2 R$$

$$\vdots$$

$$h(v) - (v_n) = I_n R$$

Where h is the potential function and $I_1, I_2, \ldots, I_n$ are the currents from v to all adjacent vertices v_k.We can solve all of these equations for v to obtain

$$h(v) = I_1 R + h(v_1)$$

$$h(v) = I_2 R + h(v_2)$$

$$\vdots$$

$$h(v) = I_n R + h(v_n)$$

Adding all of the equations together gives us

$$nh(v) = I_1 R + v_1 + I_2 R + v_2 + \ldots + I_n R + v_n.$$

Kirchhoff's law tells us that the sum of the currents along each of the edges connected to v must add to zero, thus we are left with the equation

$$nh(v) = h(v_1) + h(v_2) + \ldots + h(v_n)$$

We then solve for v to obtain

$$h(v) = \frac{h(v_1) + h(v_2) + \ldots + h(v_n)}{n}$$

□

This tells us that the potential of every vertex v in a given network is the average of all of the vertices that are adjacent to v. This is analogous to the harmonic function on

a plane where our vertices are given by some (x, y) for $x, y \in \mathbb{R}$, and our function h is defined by $h : \mathbb{R}^2 \to \mathbb{R}$

From now on we will let $h(v)$ denote the potential of vertex v in the network for $v \in N$. There are two boundary conditions for h. The first is a restriction on the source being $h(v_{\text{source}}) = 1$. The second is a condition on the sink $h(v_{\text{sink}}) = 0$. These two constraints must always be true no matter the size of the network.

Example 1.2.3. We can now use the harmonic method to solve for the total resistance of the network in figure 1.2.1. Using the average potential method we obtain an equation for every vertex in the network.

$$\begin{aligned} h(v_{\text{source}}) &= 1 \\ h(v_2) &= \frac{h(v_{\text{source}}) + h(v_3) + h(v_{\text{sink}})}{3} \\ h(v_3) &= \frac{h(v_{\text{source}}) + h(v_2) + h(v_{\text{sink}})}{3} \\ h(v_{\text{sink}}) &= 0V \end{aligned}$$

This gives us four equations and two unknowns, which means we can turn these equations into a matrix. However it would be easier if we avoided using fractions in our matrix so we are going to rewrite the equations.

$$\begin{aligned} Source &= 1 \\ 3v_2 - h(v_{\text{source}}) - h(v_3) - h(v_{\text{sink}}) &= 0 \\ 3v_3 - h(v_{\text{source}}) - h(v_2) - h(v_{\text{sink}}) &= 0 \\ Sink &= 0 \end{aligned}$$

We can turn this set of equations into a matrix where the rows are the equations and the columns correspond to the vertices in the network. The last column in the matrix will correspond to the potential values of the vertices.

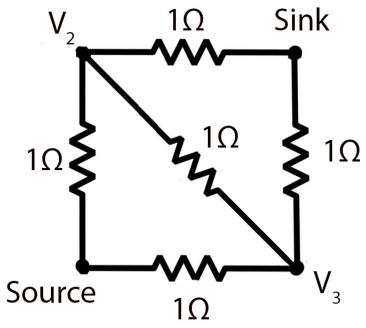

Figure 1.2.1: Network K_4 with an edge missing

$$M = \begin{bmatrix} \text{Source} & v_2 & v_3 & \text{Sink} & \text{Potential} \\ 1 & 0 & 0 & 0 & 1 \\ -1 & 3 & -1 & -1 & 0 \\ -1 & -1 & 3 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

We then row reduce this matrix to its row echelon form.

$$M = \begin{bmatrix} \text{Source} & v_2 & v_3 & \text{Sink} & \text{Potential} \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & .5 \\ 0 & 0 & 1 & 0 & .5 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

We see that by reducing the matrix to row echelon form we are able to obtain $h(v_2)$ and $h(v_3)$, the boundary conditions are also fulfilled which we expected. Now that we know the potential values of out vertices we can start to figure out exactly what the total resistance in the network is going to be. Just as before we use Ohm's law to help us. The first thing we will ned to figure out is the total current out of the source. We already know the potentials of the two vertices connected to the source so we can solve for the current along those edges using Ohm's law. Since both potentials are the same we know that the two edges will produce the same amount of current. Substituting into Ohm's law we are left

with $1 - 0.5 = IR$. We know that every resistor has a value of 1Ω. This gives us that the current out of one edge is $I = 0.5$. This means that the total current in the network is 1. Now that we know the total current we can solve for total resistance.

$$\Delta V_{\text{total}} = I_{\text{total}} R_{\text{total}} \tag{1.2.1}$$

By filling in our variables we are left with $1 - 0 = 1R$. It is not hard to see at this point that the total resistance in this network is $R = 1$.

1.3 Some Other Rules for Networks

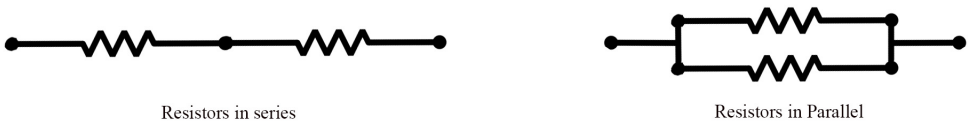

Figure 1.3.1: Resistors in series and parallel

There are two rules that can be derived from Kirchhoff's and Ohm's laws. The two rules are for resistors in **series**, and those in **parallel**, and any combination of the two. For any network we can think of a set of resistors in series as two or more resistors that are in a sequence with each other. This means that if current were flowing through the series the current would only have one possible route to take. Similarly resistors in parallel are two or more resistors that are parallel to each other. This means that if current were flowing

through the resistors there would be more than one possible route the current could take. We can see an example of resistors in series and parallel in figure 1.3.1.

Resistors in series and parallel follow certain rules. These rules can be directly derived from Kirchhoff's and Ohm's law.

Proposition 1.3.1. Given any set of resistors n in series, the total resistance R_{total} of the series is given by $\sum_{i=0}^{n} R_n$, where R_i is the resistance of resistor i in the series.

Proof. We will start by first determining the total current in the network. We know that due to Kirchhoff's current law that the only node that can produce any current is the source, and since all resistors are in sequence the current along each edge must be the same from the source to the sink. Therefore we know:

$$I_{total} = I_1 = I_2 = \ldots = I_n$$

Now since we know that the current and resistance is the same along each edge the potential difference must be the same along each edge, so the total potential drop of the series must be given by the sum of all of the potential drops between adjacent vertices.

$$\Delta V_{\text{total}} = \Delta V_1 + \Delta V_2 + \Delta V_3 + \ldots + \Delta V_n$$

Now that we know the total potential and total current we can use Ohm's law to determine the total resistance.

$$R_{total} = \frac{V_{\text{total}}}{I_{\text{total}}}$$

Substituting into Ohm's equations gives us:

$$\begin{aligned} &= \frac{V_1 + V_2 + V_3 + \ldots + V_n}{I_1} \\ &= \frac{V_1}{I_1} + \frac{V_2}{I_2} + \frac{V_3}{I_3} + \ldots \frac{V_n}{I_n} \\ &= R_1 + R_2 + R_3 + \ldots + R_n \end{aligned}$$

□

For a set of resistors in series we sum the resistance of every resistor, this gives us the total resistance of the series. This rule is very useful for some set of resistors in series as it will treat the series as one big resistor with one resistance.

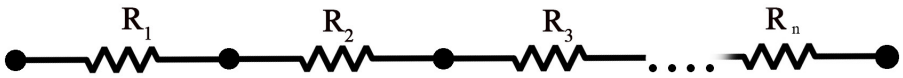

Figure 1.3.2: Set of n resistors in series

This idea of summing the resistors is clearly shown in image 1.3.2. for this n set of resistors we find the total resistance with $\sum_{k=1}^{n} R_k$. A similar rule can be derived for resistors in parallel.

Proposition 1.3.2. Given any set of resistors n in parallel, the total resistance R of the parallel resistors is given by

$$\frac{1}{R_{total}} = \frac{1}{R_1} + \frac{1}{R_2} + \ldots + \frac{1}{R_n}.$$

Proof. To prove this we take any set of resistors n that are in parallel with each other, connected to two vertices A and B. We notice first that each resistor shares the same two vertices. This implies that the potential difference across each resistor is the same such that:

$$\Delta V_{\text{total}} = \Delta V_1 = \Delta V_2 \ldots \Delta V_n$$

Next we need to find the total current in he network. We know that the total current is given by the amount of current out of vertex A and into vertex B. We also know that each resistor has an individual current. This means that the total current is a sum of all of the individual currents along each edge.

$$I_{\text{total}} = I_1 + I_2 + \ldots + I_n \tag{1.3.1}$$

It follows from Ohm's law that the total resistance is given by

$$R_{total} = \frac{\Delta V}{I_{total}} = \frac{\Delta V}{I_1 + I_2 + \ldots + I_n} \implies \frac{1}{R_{total}} = \frac{I_1 + I_2 + \ldots + I_n}{\Delta V} \tag{1.3.2}$$

$$= \frac{I_1}{\Delta V} + \frac{I_2}{\Delta V} + \ldots + \frac{I_n}{\Delta V} \tag{1.3.3}$$

$$= \frac{1}{R_1} + \frac{1}{R_2} + \ldots + \frac{1}{R_n} \tag{1.3.4}$$

□

For a set of resistors in parallel we take the reciprocal of every resistor and sum them together to get the reciprocal of the total resistance, to find the total resonate we simply take the reciprocal one more time.

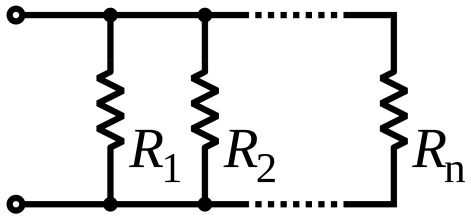

Figure 1.3.3: Resistors in series

Figure 1.3.3 shows an n set of resistors in parallel with each other. In order to find R_{total} we first take $\frac{1}{R_{total}} = \sum_{k=1}^{n} \frac{1}{R_k}$, The last step is to take the reciprocal and we have the total resistance. While the rules for resistors in series and parallel provide a very efficient way to find total resistance of certain networks it does not work for all networks. The graph K_4

is one of these networks. Such networks are called forbidden minors. This simply means if we have some network N such that K_4 is a subnetwork we can not use the rules for resistors in series and parallel to find the total resistance.

We will show how to find the total resistance of a network using the rules for series and parallel.

Example 1.3.3. We will use the network given in Figure 1.1.2. Where every resistor has a resistance one 1Ω. It is clear from the figure that we have a pair of resistors in parallel, these parallel resistors are in series with a third resistor. We can solve for the total resistance of the two resistors in parallel.

$$\frac{1}{R_{parallel}} = \frac{1}{1} + \frac{1}{1}.$$

This shows that the two resistors in parallel have a resistance of $\frac{1}{2}$. Adding this with the last resistor in series gives a total resistance of $\frac{3}{2}$, which is the result we expected.

While the rules for resistors in series offer an easy way to determine the total resistance of networks it is not a solution to all networks. We can take the network given in Figure 1.2.1. We can see there is no clear way to tell if the edges on the network are in series or parallel with each other. This results from the network $K_{2,2}$. The resistance of any network with $K_{2,2}$ as a subnetwork can not be found only using the rules for resistors in series and parallel.

2
Infinite Networks

In this chapter we will be discussing the various types of infinite networks and what it means for an infinite network to have finite resistance. We will start in Section 2.1 by first looking at the infinite binary tree and line. In Section 2.2 we will show how one may find the total resistance on one of the infinite networks which can be finite. We give the example of the binary tree and show that the resistance is in fact finite. In Section 2.3 we will go on to show exactly how one may use subnetworks of infinite networks, as well as the methods presented in Chapter 1, which include methods such as the average potential rule to determine the potentials on infinite networks. In the last Section 2.4 we will present the idea of the infinite grid and how the grid evolves as we take certain subnetworks. Another of these infinite networks is the hyperbolic grid which we will introduce in Section 2.5, to begin with we will define this grid and then show what the shapes of the subnetworks are.

2.1 Rules for Infinite Networks

An infinite electrical network is an electrical network that has a source as some point and an infinite set of edges and vertices, as well as sinks at infinity. We can think of the

network on a coordinate plane with the source being the point (0,0) and has sinks that are any point that is infinitely away from the source. There are an infinite amount of infinite networks, a couple of examples include shapes such as the infinite binary tree as seen in figure 2.1.1, and the infinite line figure 2.1.2.

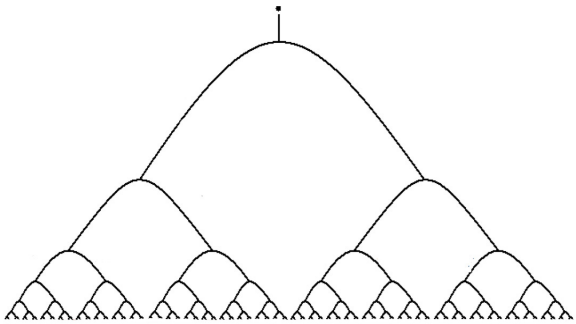

Figure 2.1.1: Infinite Binary Tree

Figure 2.1.2: Infinite Line Network

On these infinite networks it is possible to have a potential at each vertex even though the sinks are at infinity by setting a boundary condition for the networks. Because of the unique way that infinite networks are drawn we need to set a boundary condition. This

will ensure that we can account for the "sinks" at infinity. If we did not account for the sinks at infinity then every vertex in the network would end up with a potential of one.

Boundary Condition for Infinite Networks. Let N be an infinite network, and let S be any set of distinct vertices in N. Then we say

$$\lim_{n \longrightarrow \infty} h(v_n) = 0.$$

Where h is the potential function and v_n is some vertex in S.

The limit tells us that if we take any sequence of v_n distinct vertices in N then $h(v_n)$ should tend toward zero. This boundary condition ensures that we are accounting for the sink at infinity. We will use the binary tree to illustrate an infinite network under this condition.

Definition 2.1.1. The **infinite binary tree** is a network where each vertex in the tree is assigned an integer value n. The values are arranged in such a way that vertex n is adjacent to the three vertices $2n$, $2n+1$, and $\lfloor \frac{n}{2} \rfloor$. Where the source is not connected to $\lfloor \frac{n}{2} \rfloor$, and the sinks are not connected to $2n$ and $2n+1$.

Example 2.1.2. We can take the infinite binary tree and an example, the potentials of the binary tree are known.

As we can see in figure 2.1.3, even though the binary tree is in fact infinite and does not contain any real sinks it is possible using two boundary conditions that we obtain potentials for each vertex. What is even more interesting is that due to the symmetry of the binary tree every vertex in a given depth of the tree will have the same potential. These potentials provide us with a harmonic function for the binary tree, where the potential of any vertex is given by the average of all adjacent vertex potentials. When we say depth of the tree we can imagine the source at depth zero, the two nodes connected to the source

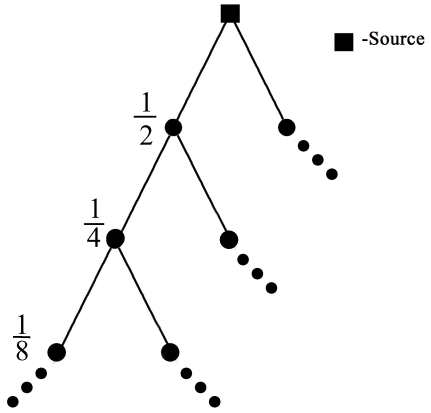

Figure 2.1.3: Potentials of the Infinite Binary Tree

would have depth one, the 4 nodes connected to those nodes would have depth two, and so on.

Proposition 2.1.3. The potential of a certain depth n of the infinite binary tree is given by the function $h = \frac{1}{2^n}$

Proof. In order to show that the function $\frac{1}{2^n}$ is the function that does in fact determine the way the potentials along the infinite binary tree change we need to show that the function is harmonic and converges to zero. We know that in order for a function on a network to be considered harmonic the value of the function at any vertex must by the average of all values of the adjacent vertices. This means that if we take some depth n we get the value of any vertex v_n is given by.

$$v_n = \frac{\frac{1}{2^{n-1}} + \frac{1}{2^{n+1}} + \frac{1}{2^{n+1}}}{3}$$

We can rewrite the equation for simplicities sake as.

$$3v_n = \frac{1}{2^{n-1}} + \frac{1}{2^{n+1}} + \frac{1}{2^{n+1}}$$

Simplifying the right hand side we are able to obtain.

$$\begin{aligned} 3v_n &= \frac{1}{2^{n-1}} + \frac{1}{2^{n+1}} + \frac{1}{2^{n+1}} \\ &= \frac{4}{2^{n+1}} + \frac{1}{2^{n+1}} + \frac{1}{2^{n+1}} \\ 3v_n &= \frac{6}{2^{n+1}} \\ &= \frac{2}{2^{n+1}} \\ &= \frac{1}{2^n} \end{aligned}$$

This shows that the function $\frac{1}{2^n}$ is in fact harmonic and can be used to find the potential of some node of depth n on the binary tree. We also know that $\frac{1}{2^n}$ converges to zero as n tends towards infinity. This result directly complements the potential that we expected the tree to have and the boundary condition we have set for our infinite networks. □

2.2 Finding the Total Resistance

Ohm's law gives us the equation for finding the total resistance of a network. However there is not a way to determine if a network has an infinite resistance using this law as infinity is not a number.

Definition 2.2.1. Given some infinite network N we say the resistance of N is infinite if there exists no harmonic function that satisfies the boundary condition for infinite networks.

2.2.1 Resistance on the Infinite Binary Tree

Based on the values of the potentials of the binary tree we can find the total resistance along the infinite tree. We know that, by rearranging Ohm's law, that the total resistance of any network is given by

$$R_{total} = \frac{\Delta V_{total}}{I_{total}} \tag{2.2.1}$$

That is the total resistance equals the quotient of the total change in potential and total current. We know that the total potential change in the network is one, as the network starts at the source and ends at any of the sinks, who's potential are always known. The last thing to do would be to figure out the total current in the network. This can be done by calculating the total current out of the source, this is because we know that the only vertex in the network that can produce a current is the source all other vertices must follow Kirchhoff's current law. In order to find the total current out of the source we would need to use Ohm's law on the two vertices directly connected to the source which we will call V_1 and V_2. We know on the infinite tree the potential of V_1 and V_2 are 0.5 as seen in figure 2.1.3. This leaves us with two equations.

$$V_1 \implies I = \frac{1 - 0.5}{1} \tag{2.2.2}$$

$$V_2 \implies I = \frac{1 - 0.5}{1} \tag{2.2.3}$$

We can see that the total current of each branch of the tree is $I = \frac{1}{2}\Omega$. To find the total resistance along B we add the current of both V_1 and V_2 to get a value of $I_{\text{total}} = 1$. We can now use Ohm's law to find the total resistance of the network.

$$R_{total} = \frac{1}{1} \tag{2.2.4}$$

Solving for R_{total} we obtain a value of 1Ω. This method relied very heavily on the fact that we already knew the potentials of the vertices on the infinite binary tree. However, there is another method that we can use to determine the total resistance of infinite networks.

2.3 Subnetworks

The boundary condition we have set for our infinite networks along with the use of harmonic functions are required for any infinite network to have a finite resistance. However,

we have used the example of the binary tree where the potentials are known. If we were to use such a method for an infinite network such as the hyperbolic grid whose potentials are unknown we come to a problem. There is a method we can use to avoid such a situation. If we have some infinite network N, we can take subnetworks of N in such a way that the first subnetwork is a subnetwork of the second, and so on.

$$N_0 \subseteq N_1 \subseteq N_2 \subseteq \ldots \subseteq N$$

The sinks of the subnetwork are chosen where if there exists a $v \in N$ whose edges are not in the subnetwork then we say that v is a sink. For each subnetwork created we use the average potential law to solve for the potentials of the vertices and resistance of the subnetwork then move to the next larger subnetwork. As we take larger and larger subnetworks we should notice the potential of the vertices start to converge to a certain potential.

Potential Convergence Principle. If an infinite network has finite resistance then the potentials of the subnetworks should converge to the harmonic function h on N. Also if an infinite network N has infinite resistance then the potentials of the subnetworks should converge to one.

This principle leaves us with a constraint for our subnetworks.

$$\lim_{k \longrightarrow \infty} h_k(v_n) = h(v_n)$$

In order to illustrate how the convergence principle will work we will first demonstrate, using the binary tree, how the sub networks are chosen. Then we will show how the boundary condition along with the convergence principal can be used to determine the resistance of the infinite binary tree. We will consider the case of the infinite binary tree, which we will denote B. For this network we will ignore the trivial subnetwork B_0 where the

source is attached to nothing but sinks. The first subnetwork, B_1, contains two unknown vertices directly attached to the source. B_1 will have one vertex between the source and sinks.

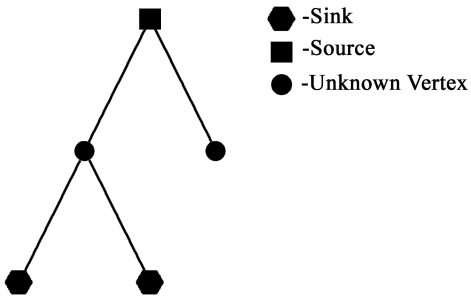

Figure 2.3.1: B_1

B_1 is solved fairly easily as the only vertex where there is an unknown $h(v)$ is the vertex between the source and two sinks. Using the average potential rule we obtain a potential of $h(v) = .33$. Symmetry tells us that the other unknown vertex must also have the same potential. In this case we do not need to set up a matrix as there would only be one unknown variable that we could easily solve for. The next subnetwork of the binary tree becomes a little more complicated. B_2 will have three unknown vertices as we can see in Figure 2.3.2.

B_2 is going to require a matrix solution to help find the potentials of the unknown vertices. B_2 can easily be solved by hand to find the potentials, however this will become increasingly difficult as we take larger and larger B_k. We will show in the next chapter a solution to help find all $h(v_n)$ for all $v_n \in B$ and solve for the total resistance of B which is known to be finite.

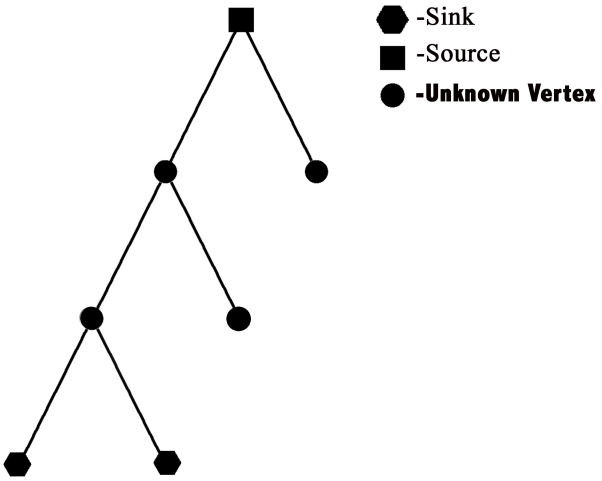

Figure 2.3.2: B_2

Resistance Convergence Principle. If an infinite network has finite resistance then the resistances of the subnetworks should converge to the total resistance of the infinite network.

2.3.1 Total Resistance on the Binary Tree subnetworks

We know that the infinite binary tree has a finite resistance then the resistance every subnetwork of the infinite binary tree must converge to the total resistance of the infinite binary tree.

Proposition 2.3.1. The total resistance of some B_k subtree is given by $R = \frac{2^{n+1}-1}{2^{n+1}}$

Proof. We will prove this using induction. Base Case $n = 0$: We know that by definition B_0 is the subnetwork simply given by the source connected to two sinks. Using Ohm's Law we can find the total current in the network. We know that the current in both resistors

in B_0 are given by $1-0=IR$, where $R=1$ so we know the current out of either resistor is 1 amp. So the total current is given by the source is 2 amps. Now that we have the total current and the total potential drop we can use Ohm's Law to find the total resistance. $1-0=2R$ solving for R we obtain a value of $R_{total}=\frac{1}{2}$. Which is equivalent to $\frac{2^1-1}{2^1}=\frac{1}{2}$

Induction Step: Assume that the total resistance of any B_n is given by $R=\frac{2^{n+1}-1}{2^{n+1}}$. Now assume we have some B_{n+1}. We will notice first that the binary tree B_{n+1} is really the source connected to two resistors that are both connected to a binary tree of length n. Since we know that the total resistance of some a binary tree of length n is given by $R=\frac{2^{n+1}-1}{2^{n+1}}$. We can change the binary tree of length $n+1$ into a binary tree of depth two were the source is connected to two resistors, R_1, and R_2 of resistance one and R_1, and R_2 are each connected to one resistor of resistance $R=\frac{2^{n+1}-1}{2^{n+1}}$. As seen in figure 2.5.1.

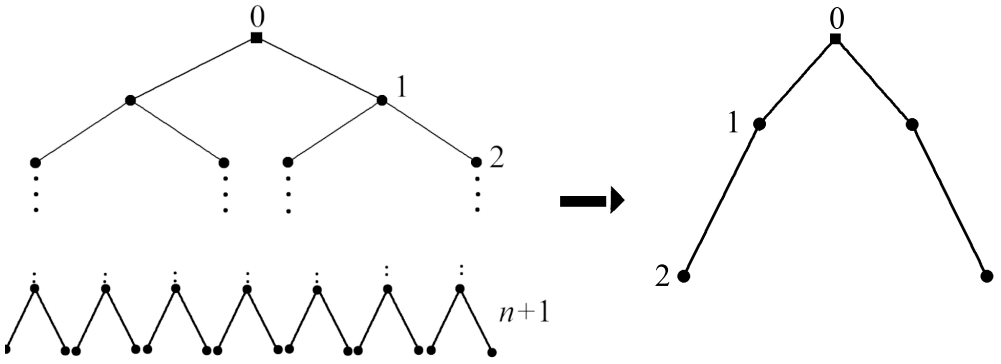

Figure 2.3.3: Binary Tree transformation

We now see that we have a simple case of resistors in series and parallel. First note that each branch in the tree is now a series of two resistors, we know that using the rule for resistors in parallel the total resistance of each branch is given by $R=\frac{2^{n+1}-1}{2^{n+1}}+1$. We also notice that the two branches are in parallel with each other, this means that we can

use our law for resistors in parallel to obtain the equation.

$$\begin{aligned}\frac{1}{R_{\text{total}}} &= \frac{1}{\frac{2^{n+1}-1}{2^{n+1}}+1} + \frac{1}{\frac{2^{n+1}-1}{2^{n+1}}+1} \\ &= \frac{1}{\frac{2^{n+2}-1}{2^{n+1}}} + \frac{1}{\frac{2^{n+2}-1}{2^{n+1}}} \\ &= \frac{2^{n+1}}{2^{n+2}-1} + \frac{2^{n+1}}{2^{n+2}-1} \\ &= \frac{2^{n+2}}{2^{n+2}-1}\end{aligned}$$

To find the total resistance on B_{n+1} we simply take the reciprocal of our equation and obtain $R_{\text{total}} = \frac{2^{n+2}-1}{2^{n+2}}$. Which give us the desired result. □

This result for the binary tree produces a sequence of functions that will converge to one as we take large subnetworks, which is the total resistance of the infinite binary tree.

2.4 The Infinite Grid

The infinite grid that we will denote G, is a network that resembles a network embedded in the Euclidian plane.

Definition 2.4.1. The infinite grid is described in the Euclidian plane as all vertices having the form (m, n) where m and n are integers. Where each vertex (m, n) is adjacent to the vertices $(m+1, n), (m-1, n), (m, n-1)$, and $(m, n-1)$.

Due to the way the resistors are connected there is no way to tell if the network has any parallel or series sets of resistors as seen in figure 2.3.1, We can imagine figure 2.3.1 goes on indefinitely do infinity.

The infinite grid behaves just as the other infinite networks. We can find the $h(v)$ for $v \in G_k$ of the network using the matrix method and average potential rule. This will help conclude if the potentials of our network in fact fit the criteria for infinite networks to have a finite resistance. This in turn will help us to solve for the total resistance. With

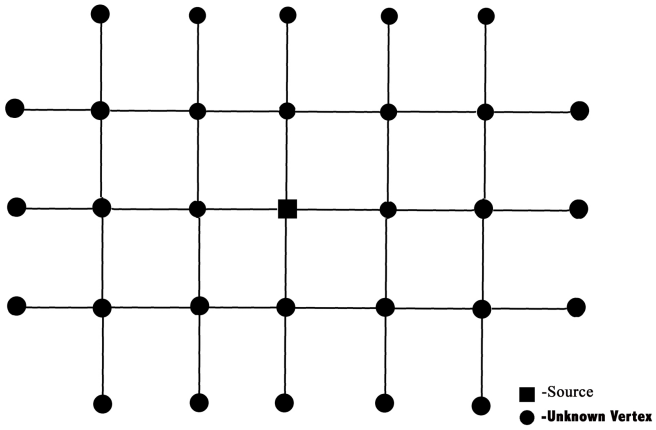

Figure 2.4.1: The infinite grid

the infinite grid we will show the trivial case where the source is connected to nothing but sinks to illustrate how exactly the network will grow as we take larger and larger G_k. The trivial subnetwork, G_0 is the source connected to four sinks.

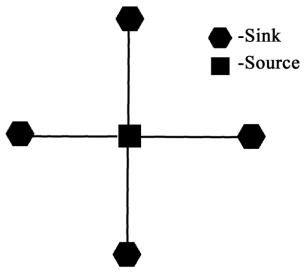

Figure 2.4.2: Trivial Grid

We can see that with this first network there are no unknown vertices so there is nothing to solve. Now we will take G_1.

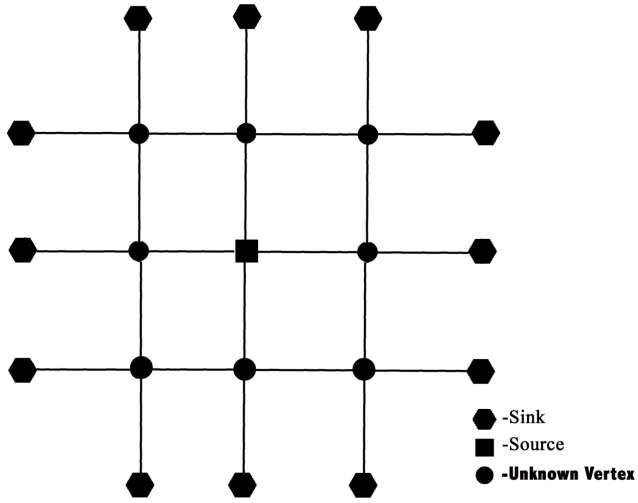

Figure 2.4.3: subnetwork 1

We see that by adding one layer to our network we add eight new vertices. The sinks of the previous layer are turned into unknown vertices and then a few new vertices are added in and connected in a specific way. The last step would be to add new sinks into the network such that there is a sink connected to every vertex at the boundary and two sinks connected to the vertex in each corner. This process adds 12 new sinks and a total of 16 new vertices to G_1. This will cause the resulting matrices to add 192 new entries just from moving from G_0 to G_1. Following this process for each G_k we take we will add an exponential amount of vertices and an even greater amount of entries to the resulting matrix. The sub grids are given in such a way that for any sub grid G_k all vertices (m, n) exist where $|m| \leq k + 1$ and $|n| \leq k + 1$.

2.5 The Hyperbolic Grid

The infinite hyperbolic grid H, as seen in figure 2.4.1 is another infinite network whose total resistance cannot be found by only using the rules for resistors in parallel and series. While H looks very similar to the infinite grid G there are subtle differences.

Definition 2.5.1. The hyperbolic grid is described in the hyperbolic plane as all vertices having the form $(m2^n, 2^n)$ where m and n are integers. Where each vertex $(m2^n, 2^n)$ is adjacent to either three or four vertices depending on if m is even or odd. If m is even then the vertex is adjacent to the vertices $(m2^n, \frac{2^n}{2}), ((m+1)2^n, 2^n)$ and $((m-1)2^n, 2^n)$. If m is even then the vertex is adjacent to $(m2^n, \frac{2^n}{2}), ((m+1)2^n, 2^n), ((m-1)2^n, 2^n)$, and $(m2^n, 2(2^n))$.

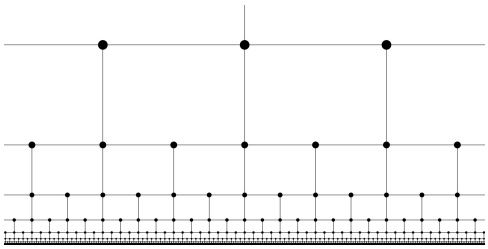

Figure 2.5.1: Hyperbolic Grid

We will assume that our infinite network H will work in the same way all infinite networks have done so far, by using the matrix method coupled with the average potential law. However this network has a very different set up in the way that the H_k will change. In oder to show this we will start with showing the trivial H_0 in order to demonstrate exactly how the different H_k will change.

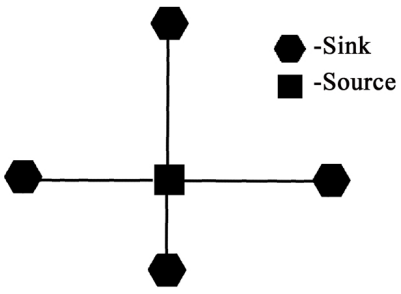

Figure 2.5.2: H_0

The trivial H_0 provides a good starting place for us as it only deals with the four vertices attached to the source. It will also show us exactly how H_0 will change as we move along H_k from $k = 0$ to $k = 1$.

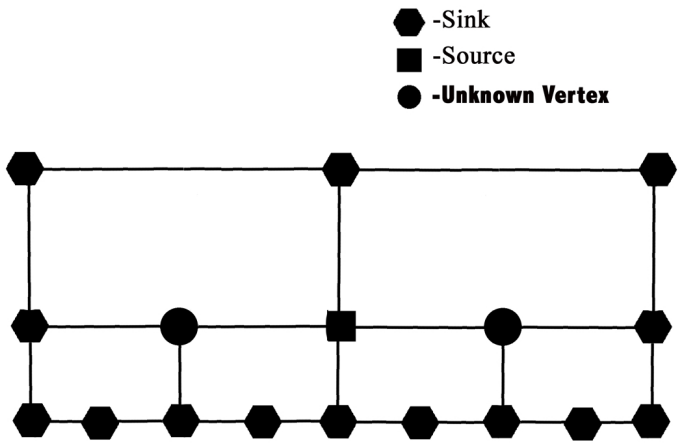

Figure 2.5.3: H_1

We can see that just as with the infinite grid a noticeable amount of vertices were added, specifically 16 new ones, this in tun will cause the resulting matrix to increase dramatically in size. As seen in figure 2.4.3

3
The Discrete Laplace Equation

In this chapter we will show a Sage program that computes the potentials on large subnetworks using Laplace's equation. First the program uses the `Digraph` data structure to create the subnetwork. Then using the `Matrix` data structure the program will create an equation for each vertex in the subnetwork that follows the average potential rule. After this the program will turn these equation into a matrix and row reduce the matrix. In the last step the program will return a specified vertex along with the vertex potential.

In each case we will start by describing the Sage function used to construct the subnetwork. The code to construct the subnetworks is different for the different networks. However, all of these functions will start by first having a blank directional graph, then the function will insert the source of the network into the graph and construct the entire graph from the source usually through some iterative process. After the network is drawn there is another function that we will use that will help to solve for the potentials of the vertices in the network. The code to determine the potentials is different for the various networks we will be solving, however there are a few fundamental steps that are the same for each potential function. Each of the potential functions creates a matrix, this matrix

will take the vertices in the network and create a column that correspond to a vertex. The rows of the matrix will correspond to the equations used to find the potential of each vertex, as shown in example 1.2.3. The code will then add an extra column at the end of the matrix that will account for the potentials of each vertex, as seen in figure 3.0.1. When we row reduce the matrix the last column will have the potential for each vertex in the subnetwork. If we take the number of vertices n, the size of the matrix is given by $n^2 + n$

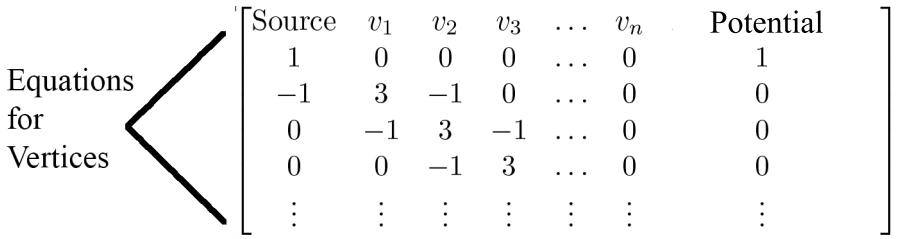

Figure 3.0.1: The Laplace Matrix for a Subnetwork

After using this method we learn that it is reasonable effective on small subnetworks with fewer than 10,000 vertices. After this point when we run the code the matrix that is produced becomes too large for the computer to process and the computer runs out of random access memory. This presents a problem as we can only gather a finite amount of data. With this limitation on the code we were able to show that the potentials of the infinite binary tree seem to be approaching $\frac{1}{2^n}$ which is what we expected. We were also able to show that the potentials of the infinite line were in fact all approaching one, meaning the total resistance was infinite.

Lastly we were able to use this method to compute the resistance of finite subnetworks of the grid up to size 12. Then surprisingly, based on the data we obtained, we were able

to determine that the resistances of the finite grid seem to be growing logarithmically with size, as we see in section 3.4.

Given the size restriction of Laplace's equation we were not able to conclude if the potentials of the hyperbolic grid are converging to some number less than one, as we see in section 3.5.

3.1 Calculating the Potentials

In order to find the total resistance on a network N, we need to show the potentials on large subnetworks converge. Finding $h(v)$ for $v \in N_k$ for small values of k along the line and binary tree by hand is possible but become increasingly difficult as $k \to \infty$ due to the fact that the matrices become larger and larger. In order to deal with the larger matrices we will be using a code that works by first drawing some N_k and then produces a matrix to row reduce and give us the potentials at some specified vertex.

3.1.1 Binary Tree Code

We start with the pseudo code that draws the subnetwork of the binary tree of a certain depth.

1: **function** DRAW BINARY TREE($Size$)
2: $SubTree = [v_1]$
3: $v_1 = Source$
4: **if** $Size \leq 0$ **then return** v_1 attached to sinks v_2,v_3
5: **else**
6: $n = 0$
7: **while** $n <$ $Size$ **do**
8: for each sink v_n attach v_n to v_{2n}, v_{2n+1}
9: $n = n + 1$ **return** $SubTree$

10: **end while**

11: **end if**

12: **end function**

This function has only one argument of $Size$ that helps it to determine the depth of the sub-tree. Our functions follows a specific number of steps in order to first create the subnetwork of the binary tree that we will be using.

1. Lines 2-3: Create a subnetwork with only one vertex, this vertex is the source.

2. Lines 3-4: If the required size of the subnetwork is less than or equal to zero then return the subnetwork of a source connected to two sinks.

3. Lines 5-9: If the size required is greater than zero enter a while loop that iterates $Size$ times. At each iteration attach every current sink v_n in the subnetwork to the sinks v_{2n} and v_{2n+1}.

4. Line 10: Return a copy of the subnetwork.

After the subnetwork has been created another function will take the subnetwork of the Draw Binary Tree function and start solving for the potentials of every vertex in the subnetwork.

1: **function** Find Potential of Binary Tree(subnetwork, Vertex)

2: $V =$ [list of vertices in subnetwork]

3: $v_1 = Source$

4: $M = len(V)$ by $len(V) + 1$ Matrix of all zeroes

5: **for** each sink $v_i \in$ v **do**

6: $MatrixM_{ii} = 1$

7: **end for**

8: **for** v_n not a sink **do**

9: v_i, v_j, v_k = vertices connected to v_x

10: $M_{nn} = 3$

11: $M_{ni} = -1$

12: $M_{nj} = -1$

13: $M_{nk} = -1$

14: **end for**

15: $M_{11} = 1$

16: $M_{1n} = 1$

17: A = Reduced Row Echelon form(M)

18: **for** $v_i \in A$ **do**

19: **if** v_i = Vertex **then**: **return** (Vertex, Potential)

20: **end if**

21: **end for**

22: **end function**

Just as the DrawBinaryTree function, this new potential function follows a specific number of steps in order to determine the potential values of each vertex in our subnetwork. The function takes as the argument a subnetwork, and a specified vertex.

1. Lines 2-3: Take all vertices in the subnetwork and puts them into a list called "V", then assigns the vertex v_1 in this list as the source.

2. Line 4: Take the length of V and create a matrix that is this length by length plus one with all entires being zero.

3. Lines 5-6: Take V and iterate through it to determine each vertex v_i that is a sink and change M_{ii} to a one.

4. Lines 8-13: Iterate through V to find all vertices v_m that are not sinks and change M_{mm} to a three. After this it will take the three vertices v_i, v_j, and v_k that are attached to the vertex v_m and change M_{ii},M_{jj} and M_{kk} to a negative one.

5. Lines 15-16: Account for the source in the matrix by changing the first entry in the first row to a one and the last entry in the first row to a one.

6. Lines 17-20: Row reduce the matrix so each column, except for the last column, has only one entry and the last column contains the potentials of all of the vertices. Then go through the matrix and return the specified vertex in the argument and the potential of that vertex.

3.1.2 Binary Tree Results

We will start by finding the harmonic function h on the binary tree B_k and trying to determine if the potentials are converging to the points as seen in figure 2.1.3. We know that due to the symmetry of the binary tree the vertices on each depth will have the same $h(v)$. We took the potential values of B_k from $0 \leq k \leq 6$.

Depth of Tree	B_1	B_2	B_3	B_4	B_5	B_6
$h(0)$	1	1	1	1	1	1
$h(1)$	.430	.460	.480	.492	.496	.498
$h(2)$	0	.140	.200	.225	.238	.244
$h(3)$	—	0	.096	.111	.118	.121
$h(4)$	—	—	0	.047	.055	.058
$h(5)$	—	—	—	0	.023	.027
$h(6)$	—	—	—	—	0	.013

The values in this table represent the first six subnetworks of the infinite binary tree. These values give us the potential of each depth of B. The source is depth 0 with $h(0) = 1$, all of the vertices that are directly connected to the source are depth 1 with $h(1) = 0.5$. The vertices directly connected to those are depth 2 and so on an so forth. We know that we

want these values to be a good estimate of the potentials on B. In order to tell if the h values we have obtained from B_6 are a good estimates we will take $6 \leq k \leq 10$

Depth of Tree	B_6	B_7	B_8	B_9	B_{10}
0	1	1	1	1	1
1	.498	.498	.499	.499	.499
2	.244	.245	.246	.248	.249
3	.121	.122	.122	.123	.124
4	.058	.060	.061	.061	.062
5	.027	.029	.030	.030	.031
6	.013	.011	.012	.014	.014
7	0	.005	.005	.006	.007
8	—	0	.002	.002	.003
9	—	—	0	.001	.001
10	—	—	—	0	.001

As we expected, from figure 2.1.3, $h(1) = 0.5$. This table also fits both of our limit constraints perfectly, we see that the values of the network do in fact go to zero as we travel from the source to the sink. We also notice that as we take B_k for large k our $h(V(B_k))$ are approaching the values we expected.

B_6 has 127 vertices, and creates a matrix with 16,129 entries, while this is not close to infinity it is a very large portion, and one that seems to give us the limit of at least the first few layers of the tree. We can see that even adding four layers adds an exponential amount of vertices and an even greater amount of entries to our matrix. This data seems to also solidify the values for $h(1)$ and $h(2)$, such that $h(1) = 0.5$ and $h(2) = 0.25$ which is exactly what we expected.

3.2 Potentials Along the Infinite Line

It is not the case however that every infinite network will have finite resistance. For example we will take the infinite line L. Proceeding as we have done with the infinite binary tree. The code for the infinite line works in almost the exact same way the infinite tree worked.

3.2.1 Infinite Line Code

The following code will outline the steps to drawing the infinite line and then finding the potentials along those lines.

1: **function** DRAW LINE($Size$)
2: $SubNetwork = []$
3: $n = 0$
4: **while** $n <Size$ **do**
5: for each sink v_n attach v_n to v_{n+1} and v_{n-1}
6: $n = n + 1$
7: **end while**
8: **return** $SubNetwork$
9: **end function**

Just as the last function to draw the binary tree, this new function only needs the $Size$ of the line before it can create the subnetwork. However unlike the binary tree this code only has one step. The function will iterate $Size$ times, and during each iteration it will connect the two sinks in the subnetwork to one sink each. Just as before after the line is created we will use a different code that will determine the potential values of the nodes.

1: **function** FIND POTENTIAL LINE(subnetwork, Vertex):
2: $V =$ List of vertices in subnetwork
3: $v_1 = Source$
4: $M = len(V)$ by $len(V) + 1$ Matrix of all zero's
5: **for** each sink v_i **do**:
6: $M_{ii} = 1$
7: **end for**
8: **for** Vertices v_x that are not a sink **do**

9: v_i, v_j = vertices connected to v_x

10: $M_{xx} = 2$

11: $M_{xi} = -1$

12: $M_{xj} = -1$

13: **end for**

14: $M_{11} = 1$

15: $M_{1n} = 1$

16: A = Reduced Row Ecehlon form(M)

17: **for** $v_i \in$ A **do**

18: **if** v_i = Vertex **then**: **return** (Vertex, Potential)

19: **end if**

20: **end for**

21: **end function**

Just as with the binary tree potential function this function takes the subnetwork and a specified vertex as the argument and follows almost the same exact steps as the binary tree potential function.

1. Lines 2-4: Create a list, v, of all vertices in the subnetwork. Then denote the source as v_1. Then creates a matrix that is the length of v by the length of $v+1$ of all zeroes

2. Lines 5-7: Iterate through all of there vertices and for each sink v_i, will change M_{ii} to one.

3. Lines 8-13: Iterate through all vertices and for each vertex v_x that is not a sink, change M_{xx} to a two, and for both adjacent vertices v_i and v_j change M_{ii} and M_{jj} to a negative one.

4. Lines 14-19: The last steps are that same as before the code will take the first row of the matrix and change the first and last entries to a one. Lastly we row-reduce

the matrix and return the potential of the specified vertex given in the argument of the function.

3.2.2 Potentials on the Line

For the infinite line we look at how potentials along a certain subnetwork change starting with L_{10}. We are able to take a larger sample of L_k due to the fact that we only add two new vertices to every L_k unlike the binary tree which added 2^k vertices at every layer. In this table we will have zero be the source, one will be the two vertices connected directly to source and their potentials, two be the vertices connected to one with their potentials, and so on.

Vertex in L_{10}	0	1	2	3	4	5	6	7	8	9	10
Potential	1	.9	.8	.7	.6	.5	.4	.3	.2	.1	0

We know that there cannot exist a vertex in our network with a potential that is greater than the source because of the way the average potential rule works, as well as the boundary conditions we have in place for the network. Meaning that layer one is most likely approaching some limit. In order to determine if layer one is in fact approaching a limit we can take larger L_k. This time we will take $k = 13$.

Vertex in L_{13}	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Potential	1	.923	.846	.769	.692	.615	.538	.461	.384	.307	.230	.135	.076	0

It seems apparent that $h(L(v))$ for all $v \in L$ seem to be approaching the value one. This data only represents the first 13 vertices that go out in either direction of the source. However this only gives us a matrix of $2n$ entries, where n is the number of vertices. We have already shown that the program can produce a matrix of 2,047 vertices. So we will take L_{1023}. I will only list the first few values of the matrix produced as listing 1023 values would be excessive.

Vertex on L_{1023}	0	1	2	3	4	5	6	7	8	9	10	11	...	1023
Potential	1	.999	.999	.999	.999	.999	.999	.999	.999	.999	.999	.999	...	0

It is very obvious now that all of the vertices seem to be converging to the value of one in the infinite line L. Due to this fact the convergence principal tells us that the total resistance on the line is infinite.

3.3 Resistance of the Infinite Grid

We have seen how infinite networks can be both finite and infinite in total resistance. However it was the case that both the infinite binary tree and line could be described using the rules for parallel and series to help find the total resistance. There are networks where this is not the case, we will take both the infinite hyperbolic and regular grids. Which we will denote H and G respectively. Let us first consider the case of the infinite grid G as the symmetry will make determining the total current and resistance easier.

We will proceed the way we have done with the infinite line and tree. First we start with a program that will draw the grid and then calculate the potential values of each vertex in the grid.

3.3.1 Grid Code

The following will describe the code method for drawing the grid and finding the potentials of each of its vertices.

```
1: function DrawGrid(Size)
2:     if Size = 0 then
3:         returnSource connected to four sinks
4:     elseSize != 0
5:         Grid = []
6:         XValue = Size
```

```
7:        YValue = Size
8:        while YValue != −YValue - 1 do
9:            for  x in range(−XValue,XValue) do
10:               insert in to Grid (XValue,YValue)
11:               YValue = YValue − 1
12:           end for
13:       end while
14:   end if
15: return Grid
16: end function
```

The function for creating a grid is more difficult than the binary tree and line.

1. Lines 2-3: If the *Size* that is required for the grid is less than or equal to zero then the code will return the trivial grid of the source connected to four sinks.

2. Lines 3-7: If the *Size* is appropriate then we start by creating a blank grid with no vertices, and take *Size* and set it equal to our starting x-value and starting y-value.

3. Lines 8-11: Enter a while loop that will loop over from the negative y-value to the positive y-value in steps of one, and at each iteration of the loop the code will take every x from the negative x-value to the positive x-value range and insert the point corresponding to the current x-value and y-value.

4. Line 15: Returns a copy of the current subnetwork.

The code for the grid is more complicated due to the fact that there is a nested range loop inside of a while loop. The nested loop and while loop are responsible for the creation of all vertices and edges. The code to determine the potentials follows a very similar route to the line and binary tree.

1: **function** FINDPOTENTIALGRID(Sub-Grid, Vertex)
2: $V =$ List of vertices in subnetwork
3: $v_1 = Source$
4: $M = len(V)$ by $len(V) + 1$ Matrix of all zero's
5: **for** each sink v_i **do**:
6: $M_{ii} = 1$
7: **end for**
8: **for** Vertices v_k that are not a sink **do**
9: $v_i, v_j, v_m, v_n =$ vertices connected to v_x
10: $M_{kk} = 4$
11: $M_{xi} = -1$
12: $M_{xj} = -1$
13: $M_{xm} = -1$
14: $M_{xn} = -1$
15: **end for**
16: $M_{11} = 1$
17: $M_{1n} = 1$
18: $A =$ Reduced Row Escehlon form(M)
19: **for** $v_i \in$ A **do**
20: **if** $v_i =$ Vertex **then**: **return** (Vertex, Potential)
21: **end if**
22: **end for**
23: **end function**

This potential function follows the exact same formula as the potential functions for the line and binary tree.

1. Lines 2-6: Create a list, V, of all the vertices and defines the source as v_1. Then the code creates a matrix that is the length of V by the length of $v + 1$ of all zeroes. Then Iterate though all vertices v_i and for every vertex that is a sink will change M_{ii} to a one.

2. Lines 7-14: Iterate through all vertices and for every vertex v_k that is not a sink it will change M_{kk} to a four, the code will then take the four adjacent vertices and change the four corresponding vertices in M_k to a negative one.

3. Lines 15-21: Change the first and last entry in the first row of M to a one. Then row-reduce the matrix and return the specified vertex with its potential.

3.3.2 Potentials on the Grid

We will start with G_1 as seen in figure 2.3.3. We will show the potential values of G_3 in order to demonstrate that G does in fact follow the boundary conditions we have set. After this we will only show how the potential of the vertex $(1, 0)$ changes as we take larger subnetworks, due to the fact that for each subnetwork we take an exponential number of vertices are added, this would create tables of tens of thousands of entries.

Vertices	Potentials
(0,0)	1
{(1,0), (-1,0), (0,1), (0,-1)}	.3333
{(1,1), (-1,-1), (-1,1), (1,-1)}	.2826
{(2,0), (-2,0), (0,2), (0,-2)}	.1739
{(2,1), (-2,1), (-1,2), (1,-2), (2,-1), (-2,-1), (1,2), (-1,-2)}	.1304
{(2,-2), (-2,2), (2,2), (-2,-2)}	.0652

G_3 has 45 vertices and creates a matrix with 2025 entries. This goes to show how even taking small values of G_k produces very large networks with even larger matrices. In order to find the total resistance of this network we will take the next few G_k and only list $h((1, 0))$.

G_k	2	3	4	5	6	7	8	9	10	11	12
$h((1,0))$	.3333	.4347	.4887	.5236	.5488	.5680	.5833	.5960	.6067	.6159	.6239

It is hard to tell from this data if there is a certain value that the vertex (1,0) is approaching or not. However from the data that we have gathered it seems completely possible that the infinite grid might also have an infinite resistance just as the infinite line where the potential of every vertex will eventually be one as $k \to \infty$.

3.4 Approximate Formula for Grid Resistance

Since a random walk on the grid is not transient[3] the infinite grid is known to have an infinite resistance. So we expect the resistances of the grid subnetworks to diverge to infinity. In this section we find an approximate formula for our grid subnetworks that fits our data surprisingly well. In order to do this we will be using the least squares method. This method works by first taking each k value in G_k and creating some $f(k)$, where $f(k)$ should be the value of the resistance on G_k, along with the actual resistance, R_k. Then it will measure the error between our $f(k)$ and the actual resistances. The smaller we can make this error the better idea we will have on how exactly the resistances are changing as we increase our value of k. We would ideally want this error to be zero.

$$\text{Least Squares Error} = \sum_{i=1}^{n} (R_i - f(k_i))^2 \tag{3.4.1}$$

That is, the error of the best fit function is given by the sum of the difference of the total resistance of G_k and the best fit function squared. We will start by determining the resistance for G_k for $0 \leq k \leq 5$. We already have the potentials of the point $(1, 0)$. Using this we can determine the total current in the network, first by solving for the total current along the edge from the source to the point $(1, 0)$ and multiply this by 4. Then we can use Ohm's Law to solve for the total resistance of the current subnetwork.

G_k	Potential at (1,0)	Total resistance
2	.3333	0.374999981
3	.4347	0.442307686
4	.4887	0.488970584
5	.5236	0.524875531
6	.5488	0.554101716
7	.5680	0.578760374
8	5833	0.600093278
9	5960	0.618893992
10	.6067	0.635701882
11	.6159	0.65090013
12	.6239	0.664770564

Now we will plot the total resistance as a function of G_k.

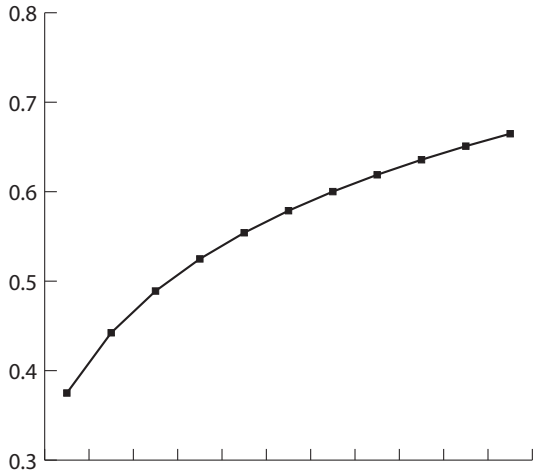

Figure 3.4.1: Resistances of Sub-Grids

We can see from the plot that the best fit function must be something along the lines of either a logarithmic function or a square root function. We attempted to use a square root formula of the form $A + B\sqrt{x + C}$ and a logarithmic function of the form $A + B\log(x + C)$, since the log function provided a smaller error we will only show the results of the logarithmic function. The set up for finding the best fit line will be a linear program where

we will be minimizing the total error. After running our program through Microsoft Excel we are left with the table.

H_k	Total Resistance	$A+B\log(x+C)$	(Total Resistance - $(A+B\log(x+C))^2$)
2	0.374999981	0.37642678	2.03575×10^{-6}
3	0.442307686	0.441815001	2.42738×10^{-7}
4	0.488970584	0.488208681	5.80496×10^{-7}
5	0.524875531	0.524194416	4.63918×10^{-7}
6	0.554101716	0.553596902	2.54837×10^{-7}
7	0.578760374	0.57845635	9.24308×10^{-8}
8	0.600093278	0.599990583	1.05464×10^{-8}
9	0.618893992	0.618985122	8.30478×10^{-9}
10	0.635701882	0.635976317	7.53146×10^{-8}
11	0.65090013	0.651346723	1.99445×10^{-7}
12	0.664770564	0.665378803	3.69955×10^{-7}
			Sum of error = 4.33374×10^{-6}

Where $A = 0.264644879$, $B = 0.371331439$, and $C = 0$. We let Excel change the values of A, B and C and Excel claimed that the best value for C was zero. As we can see the least squares error of our best fit line and the actual resistance is something along the magnitude of 10^{-3} which is a relatively small error. Even when we compare the actual resistances to the ones produced by the best fit function we see that the best fit function is correct to the third decimal place or better. In order to see this relationship more clearly we can graph the resistances and the best fit function.

We can clearly see from the graph that our best fit function seems to show exactly how the resistances of the different G_k will change over time. What is even more interesting is that there is no C value for our best fit function as seen in Figure 3.4.2, this seems to suggest that we have found almost exactly how the resistances will change as we take larger and larger subnetworks. If this function does in fact dictate how the total resistance will change then it is easy to see that the resistance on G is infinite, as we expected because we know that $\log(x)$ functions do not converge as $x \to \infty$.

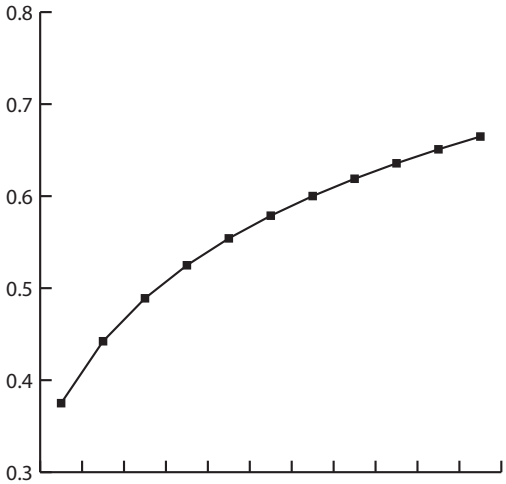

Figure 3.4.2: Resistance given by best fit

3.5 Resistance of the Infinite Hyperbolic Grid

The hyperbolic grid is another example of a grid whose total resistance can not be found by simply using the rules of series and parallel. We will consider the infinite hyperbolic grid H. The program for this code works much different from any of the programs we have seen before, due to the symmetry of the grid.

3.5.1 Hyperbolic Grid Code

The following code demonstrates how we can use Sage to draw some subnetwork of the hyperbolic grid of a certain size.

1: **function** DrawHyperbolicGrid($Size$)
2: **if** $Size \leq 0$ **then**
3: **return**$(0, 1)$ connected to $(1, 1)(-1, 1)(0, 2)(0, 1/2)$
4: **else**
5: $SubNetwork$ = [Blank Digraph]

6: $XValue = 2^{Size}$

7: $YValue = 2^{Size}$

8: $N = -Size$

9: **while** $N < Size$ **do**

10: **for** vertices in range($-XValue$, $XValue$) **do**

11: insert into SubNetwork($XValue, YValue$) and connect to appropriate vertices

12: **end for**

13: $YValue = YValue/2$

14: $N = N + 1$

15: **end while**

16: **end if**

17: **return** SubNetwork

18: **end function**

The code for drawing the hyperbolic grid follows the same structure as the grid as one is a modified version of the other.

1. Lines 1-3: Return the trivial hyperbolic grid of the source connected to four sinks if $Size$ is less than or equal to zero.

2. Lines 4-8: If the size is appropriate, define the variables starting x and y value, and the number of layers N.

3. Lines 9-12: Enter a range loop from the negative x value to the positive x value in steps of one and insert the vertex $(x, YValue)$ into the subnetwork and connect this vertex to its appropriate vertices.

4. Lines 13-14: After one iteration of the loop from the negative x value to the positive x value increase the value of N by one and divide the y value by two.

5. Line 17: Return a copy of the subnetwork.

The code works the same as all the other "draw" functions that we have seen so far. Taking as an argument the size of the subnetwork. The potential function for the hyperbolic grid is slightly different than all the potential function we have seen so far due to the fact that not every vertex has the same symmetry. Some vertices have three adjacent vertices and some have four.

1: **function** FIND POTENTIAL HYPER(subnetwork, Vertex)
2: $V =$ List of vertices in subnetwork
3: $v_1 = Source$
4: $M = len(V)$ by $len(V) + 1$ Matrix of all zero's
5: **for** each sink v_i **do**:
6: $M_{ii} = 1$
7: **end for**
8: **for** vertices v_k that are not a sink **do**
9: **if** the number of adjacent vertices is three **then**
10: $v_i, v_j, v_p =$ vertices connected to v_k
11: $M_{kk} = 3$
12: $M_{ki} = -1$
13: $M_{kj} = -1$
14: $M_{kp} = -1$
15: **else if** the number of adjacent vertices is four **then**
16: $v_i, v_j, v_m, v_n =$ vertices connected to v_k
17: $M_{kk} = 4$
18: $M_{ki} = -1$
19: $M_{kj} = -1$

20: $M_{km} = -1$

21: $M_{kn} = -1$

22: **end if**

23: **end for**

24: $M_{11} = 1$

25: $M_{1n} = 1$

26: $A =$ Reduced Row Ecehlon form(M)

27: **for** $v_i \in A$ **do**

28: **if** $v_i =$ Vertex **then**: **return** (Vertex, Potential)

29: **end if**

30: **end for**

31: **end function**

The potential function adds another step, in that it has to first differentiate between vertices that have either three or four adjacent vertices.

1. Lines 2-6: Create a list, V, of all the vertices and defines the source as v_1. Then create a matrix that is the length of V by the length of $v+1$ of all zeroes. After this the code will iterate though all vertices v_i and for every vertex that is a sink will change M_{ii} to a one.

2. Lines 7-14: Determine if the vertex has three adjacent vertices and change the corresponding matrix values accordingly

3. Lines 15-23: If the vertex we are iterating through has four adjacent vertices instead then alter the matrix accordingly.

4. Lines 24-31: Terminate by first setting the first row of the matrix to be the source and then row-reducing and returns the specified vertex in the argument and its potential

3.5.2 Hyperbolic Grid Results

We will again start with H_1 and keep track of the four vertices that are directly connected to the source. Just as with the grid we will not list the potentials of each vertex as there are too many to keep track of even in H_2, just one step up from the trivial case.

H_k	(0,2)	(0,.5)	(1,1)	(1,-1)
2	.3295	0	.3825	.3825
3	.4151	.3370	.4738	.4738
4	.4464	.4133	5080	.5080
5	.4579	.4416	.5224	.5224

It is hard to tell from the table if the potentials are approaching any limits. Based on the data in the table if we were to collect a few more data points perhaps we would be able to determine such a limit. We run into the same problem of the program running out of memory due to the size of the matrices that are created. We stop at H_6 which contains 6191 vertices and produces a matrix of 38,328,481 entires. Any matrix greater than this will truly be impossible to row reduce. This data shows that we need to create a new program in order to help find the potentials and then total resistance of the hyperbolic grid.

4
The Heat Equation

The heat equation offers a different approach than Laplace's equation for calculating the potentials of large finite networks. The problem with Laplace's equation is the fact that as we take larger and larger subnetworks the matrices we produce also get larger, and as we have seen the function for the equation fails for subnetworks of a certain size, due to the fact that a computer could not calculate the row reduced form of the matrix. The heat equation serves to fix this by completely removing the need for a matrix and using only the vertices. We describe the theory behind this in Section 4.1 and the code in Section 4.2. In particular the heat equation works for the hyperbolic grid H_k for $5 \leq k \leq 7$ as we will see in section 4.3, which is where the function for Laplace's equation broke down. We can think of the heat equation as the way heat would flow in a network that was perfectly insulated and had some initial temperature distribution on it.

4.1 Heat Flow in a Network

Definition 4.1.1. Given some finite network N with a source and sinks the **temperature functions** $t_0, t_1 \dots t_n$ are defined recursively as

$$t_0(v) = \begin{cases} 1 & \text{if } v = \text{Source} \\ 0 & \text{otherwise} \end{cases}$$

$$t_j(v) = \begin{cases} 1 & \text{if } v = \text{Source} \\ 0 & \text{if} v = \text{Sink} \\ \frac{t_{j-1}(v_1)+t_{j-1}(v_2)+\dots+t_{j-1}(v_n)}{n} & \text{otherwise} \end{cases}$$

Where $\{v_1, v_2, \dots, v_n\}$ are the vertices that are adjacent to v.

We can think of the heat equation as a sequence of functions that start with an initial temperature "at time zero," and then as we let time increase the temperature of the system will slowly come to converge to some point. For any given network we say that t_0 is the initial temperature such that:

$$t_0(v) = \begin{cases} 1 & \text{if } v = \text{Source} \\ 0 & \text{otherwise} \end{cases}$$

Now as we increase the value of j we will notice that the values will begin to change according to the previous j giving us a recursive definition for t.

$$t_k(v) = \begin{cases} 1 & \text{if } v = \text{Source} \\ 0 & \text{if} v = \text{Sink} \\ \frac{t_{j-1}(v_1)+t_{j-1}(v_2)+\dots+t_{j-1}(v_n)}{n} & \text{otherwise} \end{cases}$$

Where $\{v_1, v_2, \dots, v_n\}$ are the vertices that are adjacent to v. This recursive function give us a sequence of functions. This sequence of functions should also converge to the harmonic function on the infinite network. It turns out that as the value of j increases the value of the vertices in a network under the heat equation will start to converge to the potential of the vertex if it were under the Laplace equation.

Temperature Convergence Principle. Let $h_k(v)$ be the potential function of some network N_k then the temperature governed by the functions $t_1, t_2, \ldots, t_j$ on N_k converge to h_k.

$$\lim_{j \longrightarrow \infty} t_j(v) = h_k(v)$$

Where $t_j(v)$ are the sequence of functions that produce a value for our vertex v by the heat equation, and $h(v)$ is the potential of the vertex v produced by using Laplace's equation. It is important to note that the sequence of functions t_j depends on the size of the subnetwork N_k and h_k.

Example 4.1.2. Let us take the subnetwork of the line L_2 as seen in figure 4.0.1. There are two unknown vertices in this subnetwork. We will also call our heat function t_j to be the temperature distribution of the network after j hours, and also define t_0 to be the initial temperature distribution on the network.

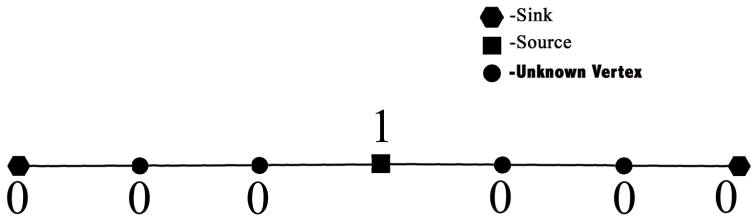

Figure 4.1.1: t_0 on L_2

We will start in the same way that we did using Laplace's equation and say that the source always has a temperature of one and the sinks always have a temperature of zero. We will also define the initial temperature of any vertex that is not the source to have an initial temperature of zero. So our t_0 is defined to be piecewise.

$$t_0(v) = \begin{cases} 1 & \text{if } v = Source \\ 0 & otherwise \end{cases}$$

Where the source is assigned a value of one and every other vertex is assigned a value of zero. Now the heat equation says that every time we increase the value of j by one we have to also take each vertex in the network and change the value to be the average of the values of all adjacent vertices, we can show this change by using figure 4.0.1, as we take j from zero to one.

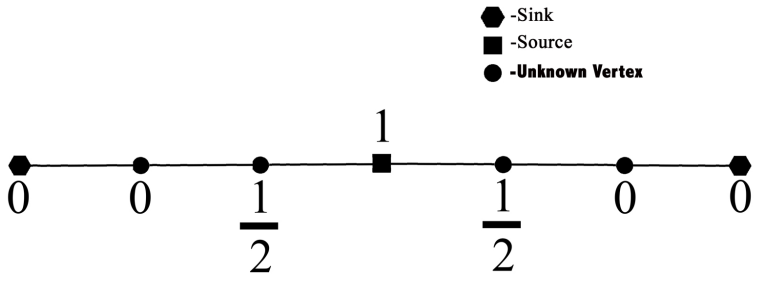

Figure 4.1.2: t_1 on L_2

We can see from figure 4.0.2 that as we moved from $j = 0$ to $j = 1$ the value of the vertices that are adjacent to the source changed from zero to one half. This is exactly the average of zero and one. So our function for t_1 becomes.

$$t_1(v) = \begin{cases} 1 & \text{if } v = Source \\ 0 & \text{if } v = Sink \\ \frac{t_0(v-1)+t_0(v+1)}{2} & otherwise \end{cases}$$

Where v+1, and v-1 represent the vertices that are adjacent to the vertex in question. We also notice that the in the function to find t_1 we have to use t_0 meaning that in general we are going to have a recursive definition. For any line network the general definition of t_j is.

$$t_j(v) = \begin{cases} 1 & \text{if } v = Source \\ 0 & \text{if } v = Sink \\ \frac{t_{j-1}(v-1)+t_{j-1}(v+1)}{2} & otherwise \end{cases}$$

We can use this equation to calculate the first 10 iterations of the line given in figure 4.0.2. In the following table we will define layer 1 to be the two vertices connected to the source, layer 2 will be the two vertices connected to layer 1, and layer 3 will be the 2 sinks at the end of the network.

Layer	1	2	3
Iteration Number			
1	.500	0	0
2	.500	.250	0
3	.625	.250	0
4	.625	.312	0
5	.656	.312	0
6	.656	.328	0
7	.664	.328	0
8	.664	.332	0
9	.666	.332	0
10	.666	.333	0

As we take j to be larger we will notice the value of the vertices start to converge to some point, after we see the values start to converge we take the next larger subnetwork, in this case it will be the subnetwork L_3 which simply adds two more vertices to the line so that we have a total of three unknown vertices, and repeat the process we preformed on L_2. We continue this process until we get a large enough subnetwork and take large k such that as we take larger subnetworks the value of the vertices do not change very drastically. We can think of this value as the steady state of the network as time goes on forever.

4.2 The Heat Function

Calculating the way that heat might move along by hand for some N_k would be relatively simple for small k but again would become increasingly difficult for larger k as the number

of vertices will grow. Not to mention you would have to repeat the process a number of times before you can get an accurate representation of what the potential at a certain vertex might be. In order to more efficiently use this method we will use a code that will preform this process for any N_k.

4.2.1 Heat Function Code

The following code will describe exactly how the heat equation will work for any network. We will use the draw functions that were described in chapter three.

```
1: function HeatFuntion(IterationNumber, SubNetwork, Vertex)
2:     for Vertices in SubNetwork do
3:         b = [V = 0 if not source, V = 1 if source]
4:     end for
5:     N = 0
6:     while N < IterationNumber do
7:         for every vertex V that is not a source do
8:             Take all adjacent vertices
9:             A = Average of adjacent vertices
10:            Set V:0 in b to V:A
11:            N = N + 1
12:        end for
13:    end while
14:    for all vertices V in b do
15:        if V = Vertex then
16: return that vertex and its value
17:        end if
18:    end for
```

19: **end function**

This new function is very different from the previous as it will work for any subnetwork that we use, where as before we would have to modify our algorithm slightly in order to account for the specific entries in the matrix that needed to change.

1. Lines 1-4: The code will take every vertex in the given subnetwork and create a new dictionary where if the vertex is not the source it will define as zero, and if the vertex is the source it will define it as one.

2. Lines 5-13: We will then iterate through all vertices and for each vertex v that is not a source or sink the code will take the assigned values of all adjacent vertices and change the value of the vertex v to the average of the values of the adjacent vertices. This process will happen the same number of times as the argument *IterationNumber*.

3. Lines 14-19: After the iterative process of changing the values of all vertices the code will go through the dictionary b and return the vertex specified in the argument of the function along with its value.

4.3 Heat Equation on the Hyperbolic Grid

The concept of the heat function will certainly reduce the run-time of the functions and can get results at a much faster pace than the potential function, but how exactly do we know that it will work? This is simple we will take the potential values and compare them to the heat function output for different iterations on the hyperbolic grid. We have the potentials for H_k for $0 \leq k \leq 5$.

G_k	(0,2)	(0,.5)	(1,1)	(1,-1)
2	.3295	0	.3825	.3825
3	.4151	.3370	.4738	.4738
4	.4464	.4133	5080	.5080
5	.4579	.4416	.5224	.5224

We will now use the HeatFunction to determine what the number of iterations are that we should perform in order to get within four decimal places of precision of the potential values we obtained using Laplace's method. For our iteration numbers we will start at 10 and take steps of 10 until we get the known value. We will also only be looking at the point (0,2) for each H_k

H_k	10	20	30	40	50	60	70	Potential
2	.3295							.3295
3	.4111	.4150	.4151					.4151
4	.4276	.4436	.4459	.4463	.4464			.4464
5	.4291	.4511	.4561	.4574	.4577	.4578	.4579	.4579

There seems to be a very clear pattern between the size of the subnetwork and the number of iterations that need to be performed before we are able to get within 4 decimal places of the potential that we previously calculated. This progression also seems to be linear as the number of iterations seem to go up by 20 when k is increased by one. There must be an equation to solve for the number for iterations needed on any H_k. Using simple algebra we arrive at a linear equation.

$$y = 20x - 30 \tag{4.3.1}$$

If this equation does in fact give us the total number of iterations that need to be performed by the heatFunction in order to get within four decimal places of the potential then we know that for $k = 6$ we will have to perform 90 iterations, and for $k = 7$ we will need 110 iterations. Expanding the table of potentials:

H_k	(0,2)	(0,.5)	(1,1)	(1,-1)
2	.3295	0	.3825	.3825
3	.4151	.3370	.4738	.4738
4	.4464	.4133	5080	.5080
5	.4579	.4416	.5224	.5224
6	.4622	.4541	.5289	.5289
7	.4641	.4599	.5319	.5319

Based on the new data it seems as if the four points connected to the source do in fact converge to some point.

4.3.1 Geometric Convergence

Now that we have compiled reasonably sound data we can estimate what the potentials of the vertices are converging to. We know that when we have some sequence of numbers that converges we can say that $\{x_n\}_{n=1}^{\infty}$ converges if $\lim_{n\longrightarrow\infty} x_n = L$, we will try to use geometric convergence to write $x_n \approx L - Ca^n$, for some constant C and a. In order to estimate the limits we have to find C, and a. Finding these two constants are just a matter of simple algebra. To solve for a let us take x_{n+2}, x_{n+1}, and x_n. Then we can write.

$$\begin{aligned}
\frac{x_{n+2} - x_{n+1}}{x_{n+1} - x_n} &= \frac{L - Ca^{n+2} - L + Ca^{n+1}}{L - Ca^{n+1} - L + Ca^n} \\
&= \frac{-a^{n+2} + a^{n+1}}{-a^{n+1} + a^n} \\
&= \frac{-a^{n+1}(a+1)}{-a^n(a+1)} \\
&= \frac{-a^{n+1}}{-a^n} \\
&= a
\end{aligned} \tag{4.3.2}$$

We can use this method to determine if there is in fact an a value that can be used in $x_n + Ca^n \approx L$ that will work for all x_n. This method should work between any set of x_n, x_{n+1}, and x_{n+2}. In order to determine if there is an a value that works for any sequence of three points we will preform this analysis on the four vertices we have been looking at so far.

Vertex	x_n	x_{n+1}	x_{n+2}	$\frac{x_{n+2}-x_{n+1}}{x_{x+1}-x_n}$
(0,2)	.3295	.4151	.4464	0.3654
	.4151	.4464	.4579	0.3674
	.4464	.4579	.4622	0.3785
	.4579	.4622	.4640	0.4081
(1,1)(-1,1)	.3825	.4738	.5080	0.3749
	.4738	.5080	.5224	0.4194
	.5080	.5224	.5289	0.4522
	.5224	.5289	.5319	0.4740
(0, .5)	0	.3370	.4133	0.2264
	.3370	.4133	.4416	0.3707
	.4133	.4416	.4540	0.4385
	.4416	.4540	.4599	0.4700

It is interesting that the a values we have obtained from our data vary, this could be due to some error, as the variations in a seem to be very small. This could also mean that a is in fact converging to some point as we take more data points, this must be the case if the potentials are converging to some point. If this is the case we can get a sense of the limits that our vertices are converging to by taking the last a value in the table and using it as the a value for all x_n. Now that we have an a value that we can use for each vertex we can find the C value in $x_n + Ca^n \approx L$. Which we can write as $x_n \approx L - Ca^n$. Now if we take the difference between any three points x_n and x_m we are left with.

$$\begin{aligned} x_n - x_{n+1} &= L - Ca^n - L + Ca^{n+1} \\ &= -Ca^n + Ca^{n+1} \\ &= C(-a^n + a^{n+1}) \\ \frac{x_n - x_{n+1}}{-a^n + a^{n+1}} &= C \end{aligned} \tag{4.3.3}$$

Again we should be able to use any two potentials from the same vertex and determine what the C value is that we will need to use in order to get any idea if these points are in fact converging to some limit. We will preform this analysis on the potentials we have managed to obtain so far.

Vertex	a	n	$n+1$	x_n	x_{n+1}	$\frac{x_n - x_{n+1}}{-a^n + a^{n+1}}$
$(0,2)$	.4081	2	3	.3295	.4151	.8682
		3	4	.4151	.4464	.7774
		4	5	.4464	.4579	.6998
		5	6	.4579	.4622	.6490
		6	7	.4622	.4640	.6490
$(0,\frac{1}{2})$	.4700	2	3	0	.3370	2.8787
		3	4	.3370	4133	1.3870
		4	5	.4133	.4416	1.0939
		5	6	.4416	.4540	1.0208
		6	7	.4540	4599	1.0208
$(1,1)$,$(-1,1)$	.4740	2	3	.3825	.4738	.7721
		3	4	.4738	5080	.6108
		4	5	.5080	.5224	.5405
		5	6	.5224	.5289	.5156
		6	7	.5289	.5319	.5156

Based on the table we can see that there must be some C value that each vertex is approaching. If this is the case using the last C value obtained for each vertex should suffice to find the approximate limit. This could prove that each vertex is in fact approaching some limit L as we increase the size of H_n. Now that we have a value for a and C we can use the equation $x_n + Ca^n \approx L$ to determine the approximate limit the hyperbolic grid approaches.

Vertex	a	C	n	x_n	$x_n - Ca^n$
(0,2)	.4081	.6490	2	.3295	.2214
			3	.4151	.3710
			4	.4464	.4284
			5	.4579	.4505
			6	.4622	.4592
			7	.4640	.4628
$(0,\frac{1}{2})$	.4700	1.0280	2	0	-.2255
			3	.3370	.2310
			4	.4133	.3635
			5	.4416	.4182
			6	.4540	.4430
			7	.4599	.4547
(1,1),(-1,1)	.4740	.5156	2	.3825	.2667
			3	.4738	.4189
			4	.5080	.4820
			5	.5224	.5100
			6	.5289	5230
			7	.5319	.5292

It seems at first glance that the none of the vertices are approaching any kind of limit. However we notice that the change between each of the individual $x_n - Ca^n$ are getting smaller as we take larger values of n. This means that the change of the change is in fact getting smaller and implies that the values of $x_n - Ca^n$ are in fact converging to some point. Even more interesting is that the change between the first two entries in the last column for each vertex is on the order of 1×10^{-1} while the change in the last two entries of the last column for each vertex are on the order of $1 \times 10^{-}3$. This could imply that the first two decimal places are in fact a good approximation for what the limit of each of the vertices are approaching. With this we have a approximate limit for each of our vertices where:

$$\lim_{x \longrightarrow \infty} h(H_x(0,2)) \approx .4628$$

$$\lim_{x \longrightarrow \infty} h(H_x(0,\frac{1}{2})) \approx .4599$$

$$\lim_{x \longrightarrow \infty} h(H_x(1,1),(-1,1)) \approx .5292$$

4.3.2 Resistance on the Hyperbolic Grid

Now that we have some approximate potential for the vertices directly connected to the source on H we can solve for the total resistance using Ohm's law. First recall that Ohm's law tells us that the total resistance of a network is equal to the total potential change divided by the total current in the network. We know the total potential change in the network is one as the source starts at one and the sinks are all zero. Finding the total current in the network is simply a matter of finding the total current out of the source. This will be relatively simple as we have already determined the potential values of all vertices connected to the source. We will start with the point $(0, 2)$, which we determined to have a potential of about .46. Using Ohm's Law we can write, $1 - .46 = I_{(0,2)}$. This means that $I_{(0,2)} = .54$. Similarly we can find the currents along the other three edges as $I_{(0,\frac{1}{2})} = .55$, $I_{(1,1),(-1,1)} = .48$. To find the total current out of the source we simply add up the current along all of these edges, we will add the .48 twice as it counts for two separate edges. The total current is given by, $I_{\text{total}} = .54 + .55 + 2(.48) = 2.05$. Now that we have the total current we can solve for the total resistance on H using Ohm's law.

$$1 - 0 = 2.05R$$

$$1 = 2.05R$$

$$\frac{1}{2.05} = R$$

This implies that H has an approximate total resistance of about .4847.

$$\begin{aligned} h(v_{\text{source}}) &= 1 \\ h(v_2) &= \frac{h(v_{\text{source}}) + h(v_3)}{2} \\ h(v_3) &= \frac{h(v_2) + h(v_{\text{sink}})}{2} \\ h(v_{\text{sink}}) &= 0V \end{aligned}$$

$$M = \begin{bmatrix} \text{Source} & v_2 & v_3 & \text{Sink} & \text{Potential} \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & .66 \\ 0 & 0 & 1 & 0 & .33 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_k	2	3	4	5	6	7	8	9	10	11	12
$h((1,0))$	12	32	60	96	140	192	252	320	396	480	572

Appendices

Appendix A
Binary Tree Code

```
def DrawBinaryTree(Size,Vertex):
    if Size == 0:
        print "why you do dis"
        return
    elif Size == 1:
        d = {1:[2,3]}
        return d
    else:
        d = ({1:[2,3]});
        g = d.copy();
        N = 0;
        while N < Size:
            for i, k in g.iteritems():
                for l in k:
                    x = 2*l
                    y = 2*l+1
                    insertInToLib(l,x,d)
                    insertInToLib(l,y,d)
            g = d.copy()
            N = N+1
        for i, k in d.items():
            for l in k:
                while k.count(l) > 1:
                    k.remove(l)
    if Size > 6:
        FindPotentialTree(d,Vertex)
        DrawBinaryTree(Size-1,Vertex)
```

```
    else:
        return

def FindPotentialTree(Library,Vertex):
    x = 0
    lst = []
    for i,k in Library.items():
        for l in k:
            lst.append(l)
    for i in lst:
        if lst.count(1)>1:
            lst.pop(i)
    for i in lst:
        if i>x:
            x = i
    MaxValue = x


    g = DiGraph(Library)
    temp = len(g.get_vertices())+1
    MatrixSize = []
    temp2 = 0
    while temp2<temp:
        MatrixSize.append(0)
        temp2 = temp2+1


    lst = []
    for x in MatrixSize:
        for y in MatrixSize:
            lst.append(0)
    BlankMatrix = np.array(lst).reshape(len(MatrixSize), len(MatrixSize))


    ComparingVector = [1]
    for i, k in Library.items():
        for l in k:
            ComparingVector.append(l)
    ComparingVectortmp = [1]
    for i in ComparingVector:
        if i not in ComparingVectortmp:
            ComparingVectortmp.append(i)
    ComparingVector = ComparingVectortmp
```

```
Ilist = []
Llist = []
for i in Library.items():
    Ilist.append(i[0])
for i, k in Library.items():
    for l in k:
        Llist.append(l)
StartingPoint = -1
for i, k in Library.items():
    for l in k:
        if l in Llist and l not in Ilist:
            x = ComparingVector.index(l)
            BlankMatrix[StartingPoint][x] = 1
            StartingPoint = StartingPoint-1



StartingPoint3 = 1
for i, k in Library.items():
    for l in k:
        if l in Llist and l in Ilist:
            x = ComparingVector.index(l)
            BlankMatrix[StartingPoint3][x] = 3
            a = 2*l
            b = 2*l+1
            c = floor(l/2)
            a1 = ComparingVector.index(a)
            BlankMatrix[StartingPoint3][a1] = -1
            b1 = ComparingVector.index(b)
            BlankMatrix[StartingPoint3][b1] = -1
            c1 = ComparingVector.index(c)
            BlankMatrix[StartingPoint3][c1] = -1
            StartingPoint3 = StartingPoint3+1
FinalRow = []
for i in BlankMatrix[0]:
    FinalRow.append(0)
FinalRow[len(BlankMatrix[0])-1] = 1
if 1 in ComparingVector:
    x = ComparingVector.index(1)
    FinalRow[x] = 1
BlankMatrix[0] = FinalRow
A = matrix(RR, BlankMatrix)
A = A.delete_rows([StartingPoint3])
A = A.rref();
```

```
#print A
HAHA = []
for i in A:
    for l in i:
        if l!=0 and l!=1:
            HAHA.append(l)
LOL = []
for i in HAHA:
    x=i/1.0
    LOL.append(x)
#print LOL
FinalLib = {}
Finaltemp = []
for i in A:
    x = []
    for l in i:
        x.append(l)
    Finaltemp.append(x)
for i in Finaltemp:
    for l in i:
        if l == 1:
            a = i.index(l)
            b = ComparingVector[a]
            insertInToLib(b,i[len(i)-1],FinalLib)
for i, k in FinalLib.items():
    if i == Vertex:
        print (i,k[0]/1.0)
return
```

Appendix B
Line Code

```
def DrawLine(LineSize,Vertex):
    if LineSize == 0:
        print "why you do dis?"
    elif LineSize == 1:
        g = DiGraph({(0,0):[(1,0),(-1,0)]});
        return g.plot()
    else:
        d = ({(0,0):[(1,0),(-1,0),(0,1)]});
        g = d.copy();
        N = 0;
        while N < LineSize:
            for i, k in g.iteritems():
                for l in k:
                    if l[0]>0:
                        x = (l[0]+1,l[1])
                        insertInToLib(l,x,d)
                    elif l[0]<0:
                        x = (l[0]-1,l[1])
                        insertInToLib(l,x,d)
            g = d.copy()
            N = N+1
        for i, k in d.items():
            for l in k:
                while k.count(l) > 1:
                    k.remove(l)
        for i, k in d.items():
            for l in k:
```

```
                if l[1]>1:
                    k.remove(l)
                if l[0].abs()>LineSize:
                    k.remove(l)
        g = d.copy()
        for i in g:
            if i[1]>=1:
                d.pop(i)
        FindPotentialLine(d,Vertex)
        #DrawLine(LineSize-1,Vertex)
def FindPotentialLine(Library,Vertex):
    x = 0
    lst = []
    for i,k in Library.items():
        for l in k:
            lst.append(l)
    for i in lst:
        if lst.count(1)>1:
            lst.pop(i)
    for i in lst:
        if i[0]>x:
            x = i[0]
    MaxValue = x



    g = DiGraph(Library)
    temp = len(g.get_vertices())+1
    MatrixSize = []
    temp2 = 0
    while temp2<temp:
        MatrixSize.append(0)
        temp2 = temp2+1



    lst = []
    for x in MatrixSize:
        for y in MatrixSize:
            lst.append(0)
    BlankMatrix = np.array(lst).reshape(len(MatrixSize), len(MatrixSize))


    ComparingVector = [(0,0)]
    for i, k in Library.items():
```

```
        for l in k:
            ComparingVector.append(l)
    ComparingVectortmp = [(0,0)]
    for i in ComparingVector:
        if i not in ComparingVectortmp:
            ComparingVectortmp.append(i)
    ComparingVector = ComparingVectortmp
    print ComparingVector


    StartingPoint = -1
    for i, k in Library.items():
        for l in k:
            if (l[0].abs() == MaxValue or l[1] == 1):
                x = ComparingVector.index(l)
                BlankMatrix[StartingPoint][x] = 1
                StartingPoint = StartingPoint-1


    StartingPoint3 = 1
    for i, k in Library.items():
        if i[0].abs() < MaxValue and i != (0,0):
            x = ComparingVector.index(i)
            BlankMatrix[StartingPoint3][x] = 2
            StartingPoint3 = StartingPoint3+1


    StartingPoint4 = 1
    for i,k in Library.items():
        if i[0].abs() < MaxValue and i != (0,0):
            x = (i[0]+1,i[1])
            y = (i[0]-1,i[1])
            a = ComparingVector.index(x)
            b = ComparingVector.index(y)
            BlankMatrix[StartingPoint4][a] = -1
            BlankMatrix[StartingPoint4][b] = -1
            StartingPoint4 = StartingPoint4 +1


    FinalRow = []
    for i in BlankMatrix[0]:
        FinalRow.append(0)
    FinalRow[len(BlankMatrix[0])-1] = 1
    if (0,0) in ComparingVector:
        x = ComparingVector.index((0,0))
```

```
        FinalRow[x] = 1
    BlankMatrix[0] = FinalRow
    A = Matrix(RR, BlankMatrix)
    A = A.delete_rows([StartingPoint4])
    A = A.rref()
    HAHA = []
    for i in A:
        for l in i:
            if l!=0:
                HAHA.append(l)
    LOL = []
    for i in HAHA:
        x=i/1.0
        LOL.append(x)
    print LOL
    FinalLib = {}
    Finaltemp = []
    for i in A:
        x = []
        for l in i:
            x.append(l)
        Finaltemp.append(x)
    for i in Finaltemp:
        for l in i:
            if l == 1:
                a = i.index(l)
                b = ComparingVector[a]
                insertInToLib(b,i[len(i)-1],FinalLib)
    for i, k in FinalLib.items():
        if i == Vertex:
            pass
            #print (i,k)
    return
```

Appendix C
Grid Code

```
def MakeGrid(Gridsize, Vertex):
    if Gridsize == 0:
        return "Need a size bigger than 0"
    elif Gridsize == 1:
        g = {(0,0):[(0,1),(0,-1),(1,0),(-1,0)]};
        return g
    else:
        d = ({(0,0):[(0,1),(1,0),(-1,0),(0,-1)]});
        g = d.copy();
        N = 0;
        while N < Gridsize:
            for i, k in g.iteritems():
                for l in k:
                    if (l[0] == 0 and l[1]>0):
                        x = (l[0],l[1]+1)
                        insertInToLib(l,x,d)
                    if (l[0] == 0 and l[1]<0):
                        x = (l[0],l[1]-1)
                        insertInToLib(l,x,d)
                    if (l[0]>0 and l[1]==0):
                        x = (l[0]+1,l[1])
                        insertInToLib(l,x,d)
                    if (l[0] < 0 and l[1]==0):
                        x = (l[0]-1,l[1])
                        insertInToLib(l,x,d)
            g = d.copy()
            N=N+1
```

```
        g = d.copy()
        M = 1
        while M < Gridsize:
            for i,k in g.iteritems():
                for l in k:
                    if (l[0] == (M-1) and l[1]>0 and l[1] < Gridsize):
                        x = (l[0]+1,l[1])
                        insertInToLib(l,x,d)
                    if (l[0] == -(M-1) and l[1]<0 and l[1].abs() < Gridsize):
                        x = (l[0]-1,l[1])
                        insertInToLib(l,x,d)
                    if (l[0]>0 and l[1]== (M-1) and l[0] < Gridsize):
                        x = (l[0],l[1]+1)
                        insertInToLib(l,x,d)
                    if (l[0]<0 and l[1]== -(M-1) and l[0].abs() < Gridsize):
                        x = (l[0],l[1]-1)
                        insertInToLib(l,x,d)
            g = d.copy()
            M=M+1
        g = d.copy()
        B = 1
        while B < Gridsize:
            for i,k in g.iteritems():
                for l in k:
                    if(l[0]== -(B-1) and l[1]>0 and l[1] < Gridsize):
                        x = (l[0]-1,l[1])
                        insertInToLib(l,x,d)
                    if(l[0]== (B-1) and l[1]<0 and l[1].abs() < Gridsize):
                        x = (l[0]+1,l[1])
                        insertInToLib(l,x,d)
                    if(l[1]== -(B-1) and l[0]>0 and l[0] < Gridsize):
                        x = (l[0],l[1]-1)
                        insertInToLib(l,x,d)
                    if(l[1]== (B-1) and l[0]<0 and l[0].abs() < Gridsize):
                        x = (l[0],l[1]+1)
                        insertInToLib(l,x,d)
            g = d.copy()
            B=B+1
        g = d.copy()
        for i, k in g.items():
            for l in k:
                if (l[0].abs()==l[1].abs() and l[0] > 0 and l[1]>0):
                    x = (l[0]+1,l[1])
                    y = (l[0],l[1]+1)
                    insertInToLib(l,x,d)
```

```
                    insertInToLib(l,y,d)
                if (l[0].abs()==l[1].abs() and l[0]<0 and l[1]<0):
                    x = (l[0]-1,l[1])
                    y = (l[0],l[1]-1)
                    insertInToLib(l,x,d)
                    insertInToLib(l,y,d)
                if (l[0].abs()==l[1].abs() and l[0]<0 and l[1]<0):
                    x = (l[0]-1,l[1])
                    y = (l[0],l[1]-1)
                    insertInToLib(l,x,d)
                    insertInToLib(l,y,d)
                if (l[0].abs()==l[1].abs() and l[0]>l[1] and l[0]>0 and l[1]<
                    x = (l[0]+1,l[1])
                    y = (l[0],l[1]-1)
                    insertInToLib(l,x,d)
                    insertInToLib(l,y,d)
                if (l[0].abs()==l[1].abs() and l[0]<l[1] and l[0]<0 and l[1]>
                    x = (l[0]-1,l[1])
                    y = (l[0],l[1]+1)
                    insertInToLib(l,x,d)
                    insertInToLib(l,y,d)
        g = d.copy()
        for i, k in d.items():
            for l in k:
                if(l[0]!=l[1] and l[0]<Gridsize and l[1]<Gridsize and l[0] >
                    x = (l[0]+1,l[1])
                    insertInToLib(l,x,d)
                if(l[0]!=l[1] and l[0]<Gridsize and l[1]<Gridsize and l[0] >
                    x = (l[0],l[1]+1)
                    insertInToLib(l,x,d)
                if(l[0]!=l[1] and l[0].abs()<Gridsize and l[1].abs()<Gridsize
                    x = (l[0]-1,l[1])
                    insertInToLib(l,x,d)
                if(l[0]!=l[1] and l[0].abs()<Gridsize and l[1].abs()<Gridsize
                    x = (l[0],l[1]-1)
                    insertInToLib(l,x,d)
                if(l[0]!=l[1] and l[0].abs()<Gridsize and l[1].abs()<Gridsize
                    x = (l[0],l[1]-1)
                    insertInToLib(l,x,d)
                if(l[0]!=l[1] and l[0].abs()<Gridsize and l[1].abs()<Gridsize
                    x = (l[0]+1,l[1])
                    insertInToLib(l,x,d)
                if(l[0]!=l[1] and l[0].abs()<Gridsize and l[1].abs()<Gridsize
                    x = (l[0],l[1]+1)
                    insertInToLib(l,x,d)
```

```
                if(l[0]!=l[1] and l[0].abs()<Gridsize and l[1].abs()<Gridsize
                    x = (l[0]-1,l[1])
                    insertInToLib(l,x,d)
        for i, k in d.items():
            for l in k:
                while k.count(l) > 1:
                    k.remove(l)
        for i, k in d.items():
            for l in k:
                if(l[0].abs() > Gridsize or l[1].abs() > Gridsize):
                    k.remove(l)
        for i,k in d.items():
            if(i[0].abs()>Gridsize or i[1].abs()>Gridsize):
                del d[i]

        FindPotential(d,Vertex)
        #d = DiGraph(d)
        #d.plot().show()
        #G = d.kirchhoff_matrix();
        #print G
        #for i in G.rows():
         #   print i
        #G = d.plot()
        #return G.show()

def FindPotential(Library,Vertex):
    x = 0
    lst = []
    for i,k in Library.items():
        for l in k:
            lst.append(l)
    for i in lst:
        if lst.count(1)>1:
            lst.pop(i)
    for i in lst:
        if i[1]>x:
            x = i[1]
    MaxValue = x
    g = DiGraph(Library)
    temp = len(g.get_vertices())+1
    MatrixSize = []
    temp2 = 0
    while temp2<temp:
        MatrixSize.append(0)
        temp2 = temp2+1
```

```
lst = []
for x in MatrixSize:
    for y in MatrixSize:
        lst.append(0)
BlankMatrix = np.array(lst).reshape(len(MatrixSize), len(MatrixSize))
ComparingVector = [(0,0)]
for i, k in Library.items():
    for l in k:
        ComparingVector.append(l)
ComparingVectortmp = [(0,0)]
for i in ComparingVector:
    if i not in ComparingVectortmp:
        ComparingVectortmp.append(i)
ComparingVector = ComparingVectortmp
StartingPoint = -1
for i, k in Library.items():
    for l in k:
        if (l[0].abs() == MaxValue or l[1].abs() == MaxValue):
            x = ComparingVector.index(l)
            BlankMatrix[StartingPoint][x] = 1
            StartingPoint = StartingPoint-1
StartingPoint3 = 1
for i, k in Library.items():
    if len(k)< 2:
        pass
    elif (len(k) >= 2 and i!= (0,0)):
        if len(MatrixSize) - StartingPoint3 >= StartingPoint.abs():
            x = ComparingVector.index(i)
            BlankMatrix[StartingPoint3][x] = 4
            StartingPoint3 = StartingPoint3+1
ThisOne = StartingPoint3
for i, k in Library.items():
    if(len(k) == 2):
        if len(MatrixSize) - ThisOne >= StartingPoint.abs():
            x = ComparingVector.index(i)
            BlankMatrix[ThisOne][x] = 4
            ThisOne = ThisOne+1
StartingPoint4 = 1
for i, k in Library.items():
    if(len(k) >= 2 and i!=(0,0)):
        if i[0]==0 and i[1]<0 :
            a = (i[0]+1,i[1])
            x = ComparingVector.index(a)
            BlankMatrix[StartingPoint4][x] = -1
            b = (i[0]-1, i[1])
```

```
        y = ComparingVector.index(b)
        BlankMatrix[StartingPoint4][y] = -1
        c = (i[0],i[1]-1)
        z = ComparingVector.index(c)
        BlankMatrix[StartingPoint4][z] = -1
        d = (i[0],i[1]+1)
        w = ComparingVector.index(d)
        BlankMatrix[StartingPoint4][w] = -1
        StartingPoint4 = StartingPoint4 +1
    elif i[0]<0 and i[1]==0 :
        a = (i[0]+1,i[1])
        x = ComparingVector.index(a)
        BlankMatrix[StartingPoint4][x] = -1
        b = (i[0]-1, i[1])
        y = ComparingVector.index(b)
        BlankMatrix[StartingPoint4][y] = -1
        c = (i[0],i[1]-1)
        z = ComparingVector.index(c)
        BlankMatrix[StartingPoint4][z] = -1
        d = (i[0],i[1]+1)
        w = ComparingVector.index(d)
        BlankMatrix[StartingPoint4][w] = -1
        StartingPoint4 = StartingPoint4 +1
    elif i[0]==0 and i[1]>0 :
        a = (i[0]+1,i[1])
        x = ComparingVector.index(a)
        BlankMatrix[StartingPoint4][x] = -1
        b = (i[0]-1, i[1])
        y = ComparingVector.index(b)
        BlankMatrix[StartingPoint4][y] = -1
        c = (i[0],i[1]-1)
        z = ComparingVector.index(c)
        BlankMatrix[StartingPoint4][z] = -1
        d = (i[0],i[1]+1)
        w = ComparingVector.index(d)
        BlankMatrix[StartingPoint4][w] = -1
        StartingPoint4 = StartingPoint4 +1
    elif i[0]>0 and i[1]==0 :
        a = (i[0]+1,i[1])
        x = ComparingVector.index(a)
        BlankMatrix[StartingPoint4][x] = -1
        b = (i[0]-1, i[1])
        y = ComparingVector.index(b)
        BlankMatrix[StartingPoint4][y] = -1
        c = (i[0],i[1]-1)
```

```
            z = ComparingVector.index(c)
            BlankMatrix[StartingPoint4][z] = -1
            d = (i[0],i[1]+1)
            w = ComparingVector.index(d)
            BlankMatrix[StartingPoint4][w] = -1
            StartingPoint4 = StartingPoint4 +1
        elif i[0]>0 and i[1]<0:
            a = (i[0]+1,i[1])
            x = ComparingVector.index(a)
            BlankMatrix[StartingPoint4][x] = -1
            b = (i[0],i[1]-1)
            y = ComparingVector.index(b)
            BlankMatrix[StartingPoint4][y] = -1
            c = (i[0]-1,i[1])
            z = ComparingVector.index(c)
            BlankMatrix[StartingPoint4][z] = -1
            d = (i[0],i[1]+1)
            w = ComparingVector.index(d)
            BlankMatrix[StartingPoint4][w] = -1
            StartingPoint4 = StartingPoint4 +1
        elif i[0]>0 and i[1]>0:
            a = (i[0]+1,i[1])
            x = ComparingVector.index(a)
            BlankMatrix[StartingPoint4][x] = -1
            b = (i[0],i[1]-1)
            y = ComparingVector.index(b)
            BlankMatrix[StartingPoint4][y] = -1
            c = (i[0]-1,i[1])
            z = ComparingVector.index(c)
            BlankMatrix[StartingPoint4][z] = -1
            d = (i[0],i[1]+1)
            w = ComparingVector.index(d)
            BlankMatrix[StartingPoint4][w] = -1
            StartingPoint4 = StartingPoint4 +1
        elif i[0]<0 and i[1]>0:
            a = (i[0]+1,i[1])
            x = ComparingVector.index(a)
            BlankMatrix[StartingPoint4][x] = -1
            b = (i[0],i[1]-1)
            y = ComparingVector.index(b)
            BlankMatrix[StartingPoint4][y] = -1
            c = (i[0]-1,i[1])
            z = ComparingVector.index(c)
            BlankMatrix[StartingPoint4][z] = -1
            d = (i[0],i[1]+1)
```

```
                w = ComparingVector.index(d)
                BlankMatrix[StartingPoint4][w] = -1
                StartingPoint4 = StartingPoint4 +1
            elif i[0]<0 and i[1]<0:
                a = (i[0]+1,i[1])
                x = ComparingVector.index(a)
                BlankMatrix[StartingPoint4][x] = -1
                b = (i[0],i[1]-1)
                y = ComparingVector.index(b)
                BlankMatrix[StartingPoint4][y] = -1
                c = (i[0]-1,i[1])
                z = ComparingVector.index(c)
                BlankMatrix[StartingPoint4][z] = -1
                d = (i[0],i[1]+1)
                w = ComparingVector.index(d)
                BlankMatrix[StartingPoint4][w] = -1
                StartingPoint4 = StartingPoint4 +1
FinalRow = []
for i in BlankMatrix[0]:
    FinalRow.append(0)
FinalRow[len(BlankMatrix[0])-1] = 1
if (0,0) in ComparingVector:
    x = ComparingVector.index((0,0))
    FinalRow[x] = 1
BlankMatrix[0] = FinalRow
A = Matrix(RR, BlankMatrix)
A = A.delete_rows([StartingPoint4])
A = A.rref()
for i in A:
    pass
    #print i
HAHA = []
for i in A:
    for l in i:
        if l!=0:
            HAHA.append(l)
LOL = []
for i in HAHA:
    x=i/1.0
    LOL.append(x)
print LOL
FinalLib = {}
Finaltemp = []
for i in A:
    x = []
```

```
        for l in i:
            x.append(l)
        Finaltemp.append(x)
    for i in Finaltemp:
        for l in i:
            if l == 1:
                a = i.index(l)
                b = ComparingVector[a]
                insertInToLib(b,i[len(i)-1],FinalLib)
    for i, k in FinalLib.items():
        if i == Vertex:
            print (i,k)
    return
```

Appendix D
Hyperbolic Grid Code

```
def drange(start, stop, step):
    r = start
    while r < stop:
        yield r
        r += step


def DrawHyperbolicGrid(Size,Vertex):
    if Size == 0:
        print "why you do dis?"
        d = {(0,0):[]}
        return d
    elif Size == 1:
        d = {(0,0):[(0,1),(0,-1),(1,0),(-1,0)]}
        return d
    else:
        d = {}
        N = -Size
        M = 2^Size          # horizontal step size
        Y = 2^Size          # current y-value
        Max_X = 2^Size
        Min_Y = 2^(-Size+1)
        while N < Size:
            g = d.copy()
            for x in drange(-Max_X, Max_X, M):
                C = (x,Y)
                a = x+M
```

```
                b = Y/2
                A = (a,Y)
                B = (x,b)
                if A[0] <= Max_X:
                    insertInToLib(C,A,d)
                if B[1] >= Min_Y:
                    insertInToLib(C,B,d)
            Y = Y/2
            M = M/2
            N = N+1
        for i, k in d.items():
            for l in k:
                while k.count(l) > 1:
                    k.remove(l)
                if i == l:
                    k.remove(l)
                if l == ():
                    k.remove(l)
        for i,k in d.items():
            for l in k:
                if l[0] > Max_X:
                    k.remove(l)
                if l[1] < Min_Y:
                    k.remove(l)
        if Size > 6:
            FindPotentialHyper(d,Vertex)
            DrawHyperbolicGrid(Size-1,Vertex)
        else:
            return
def FindPotentialHyper(Library,Vertex):
    x = 0
    lst = []
    for i,k in Library.items():
        for l in k:
            lst.append(l)
    for i in lst:
        if lst.count(1)>1:
            lst.pop(i)
    for i in lst:
        if i[1]>x:
            x = i[1]
    MaxValue = x
    #print MaxValue
```

```
g = DiGraph(Library)
temp = len(g.get_vertices())+1
MatrixSize = []
temp2 = 0
while temp2<temp:
    MatrixSize.append(0)
    temp2 = temp2+1
#print MatrixSize



lst = []
for x in MatrixSize:
    for y in MatrixSize:
        lst.append(0)
BlankMatrix = np.array(lst).reshape(len(MatrixSize), len(MatrixSize))



ComparingVector = [(0,1)]
for i, k in Library.items():
    for l in k:
        ComparingVector.append(l)
    ComparingVector.append(i)
ComparingVectortmp = [(0,1)]
for i in ComparingVector:
    if i not in ComparingVectortmp:
        ComparingVectortmp.append(i)
ComparingVector = ComparingVectortmp
#print ComparingVector



Max_X = 0
for i in ComparingVector:
    if i[0] > Max_X:
        Max_X = i[0]
Min_Y = Max_X
for i in ComparingVector:
    if i[1] < Min_Y:
        Min_Y = i[1]
Max_Y = 0
for i in ComparingVector:
    if i[1] > Max_Y:
```

```
            Max_Y = i[1]



    StartingPoint1 = 1
    for i in ComparingVector:
        if i[0].abs() == Max_X or i[1] == Min_Y or i[1] == Max_Y:
            I = ComparingVector.index(i)
            BlankMatrix[StartingPoint1][I] = 1
            StartingPoint1 = StartingPoint1 + 1
        elif i != (0,1):
            I = ComparingVector.index(i)
            a = (i[0]+i[1],i[1])
            b = (i[0]-i[1],i[1])
            c = (i[0],i[1]*2)
            d = (i[0],i[1]/2)
            if a in ComparingVector:
                A = ComparingVector.index(a)
            if b in ComparingVector:
                B = ComparingVector.index(b)
            if c in ComparingVector:
                C = ComparingVector.index(c)
            if d in ComparingVector:
                D = ComparingVector.index(d)
            if c in ComparingVector and d in ComparingVector and a in Compari
                BlankMatrix[StartingPoint1][I] = 4
                BlankMatrix[StartingPoint1][A] = -1
                BlankMatrix[StartingPoint1][B] = -1
                BlankMatrix[StartingPoint1][C] = -1
                BlankMatrix[StartingPoint1][D] = -1
                StartingPoint1 = StartingPoint1 + 1
            elif a in ComparingVector and b in ComparingVector and d in Comp
                BlankMatrix[StartingPoint1][I] = 3
                BlankMatrix[StartingPoint1][A] = -1
                BlankMatrix[StartingPoint1][B] = -1
                BlankMatrix[StartingPoint1][D] = -1
                StartingPoint1 = StartingPoint1 + 1
    FinalRow = []
    for i in BlankMatrix[0]:
        FinalRow.append(0)
    FinalRow[len(BlankMatrix[0])-1] = 1
    if (0,1) in ComparingVector:
        x = ComparingVector.index((0,1))
```

```
        FinalRow[x] = 1
    BlankMatrix[0] = FinalRow
    BlankMatrix[0][0] = 1
    A = Matrix(RR, BlankMatrix)
    A = A.delete_rows([StartingPoint1])
    A = A.rref()
    FinalLib = {}
    Finaltemp = []
    for i in A:
        x = []
        for l in i:
            x.append(l)
        Finaltemp.append(x)
    for i in Finaltemp:
        for l in i:
            if l == 1:
                a = i.index(l)
                b = ComparingVector[a]
                insertInToLib(b,i[len(i)-1],FinalLib)
    for i, k in FinalLib.items():
        if i == Vertex:
            for j in k:
                j = float(j)
                print (i,j)

    return
```

Appendix E

Hyperbolic Heat Function

```
def DrawHyperbolicGrid(Size,Vertex):
    if Size == 0:
        print "why you do dis?"
        d = {(0,1):[]}
        return d
    elif Size == 1:
        d = {(0,1):[(0,1/2),(0,2),(1,0),(-1,0)]}
        return d
    else:
        d = []
        Xvalue = 2^Size
        Yvalue = 2^Size
        Step = 2^Size
        N = -Size
        while N < Size:
            for x in drange(-Xvalue,Xvalue,Step):
                a = (x,Yvalue)
                d.append(a)
            Yvalue = Yvalue/2
            Step = Yvalue
            N = N+1
        return d

def HeatHyper(IterationNumber, Library, Vertex,Vertex1,Vertex2,Vertex3):
    ComparingVector = []
    for i in Library:
        ComparingVector.append(i)
```

```
MaxValue = 0
for i in ComparingVector:
    for l in i:
        if l>MaxValue:
            MaxValue = l
MinValue = 1
for i in ComparingVector:
    for l in i:
        if l<MinValue and l>0:
            MinValue = l
#print MinValue
#print MaxValue
#print ComparingVector
a = {}
b = {}
c = {}
for i in ComparingVector:
    if i!=(0,1):
        insertInToLib(i,0,a)
    elif i == (0,1):
        insertInToLib(i,1,a)
for i in ComparingVector:
    x= (i[0]+i[1],i[1])
    y= (i[0]-i[1],i[1])
    z= (i[0],i[1]*2)
    w= (i[0],i[1]/2)
    if x in ComparingVector:
        insertInToLib(i,x,b)
    if y in ComparingVector:
        insertInToLib(i,y,b)
    if z in ComparingVector:
        insertInToLib(i,z,b)
    if w in ComparingVector:
        insertInToLib(i,w,b)
for i,k in b.items():
    for l in k:
        if k.count(l) > 1:
            k.remove(l)
for i,k in b.items():
    if len(k) > 4:
        for l in k:
            if k.count(l) > 1:
                k.remove(l)
for i,k in a.items():
    for l in k:
```

```
            while k.count(l) > 1:
                k.remove(l)
    N = 0
    while N < IterationNumber:
        c = a.copy()
        for i, k in a.items():
            if i != (0,1) and i[0].abs()!= MaxValue and i[1].abs() != MaxValu
                for items, keys in b.items():
                    if i == items:
                        HAHA = []
                        for key in keys:
                            for it, ks in a.items():
                                if key == it:
                                    HAHA.append(ks[0])
                        l = sum(HAHA)/float(len(HAHA))
                        k[0] = l
        N = N+1
    for i,k in a.items():
        if i == Vertex:
            print (i,'----',k)
    for i,k in a.items():
        if i == Vertex1:
            print (i,'----',k)
    for i,k in a.items():
        if i == Vertex2:
            print (i,'----',k)
    for i,k in a.items():
        if i == Vertex3:
            print (i,'----',k)
```

Bibliography

[1] Peter G. and Snell Doyle J Laurie, *Random walks and electric networks*, AMC **10** (1984), 12.

[2] James W and Floyd Cannon William J and Kenyon, *Hyperbolic Geometry*, Flavors of Geometry **31** (1997), 59–115.

[3] Wolfgang Woess, *Random walks on infinite graphs and groups*, Cambridge university press, England, 2000.

[4] David and Resnick Halliday Robert and Walker, *Fundamentals of physics extended*, John Wiley & Sons, United States, 2010.