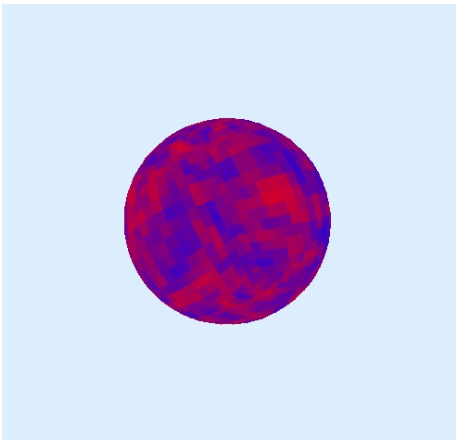


Modeling Heat Conduction on a Spherical Surface With Iterative Methods

Dylan Harrison
&
Evan Hauer

5/18/16

Contents

1 Abstract **4**

2 Introduction **4**
- 2.1 Newton's Law of Heating and Cooling . . . 4
- 2.2 A Small Example System of ODEs . . . 5
- 2.3 Partitioning The Surface of A Sphere . . . 5
- 2.4 A Discussion of the Resulting System . . . 6

3 Methods **7**
- 3.1 Eulers Method . . . 7
- 3.2 Modified Eulers Method . . . 8
- 3.3 Midpoint Method . . . 8
- 3.4 Runge-Kutta . . . 9
- 3.5 Finite-Difference . . . 10

4 Implementation **12**
- 4.1 On Choosing Java for Implementation . . . 12
- 4.2 A Superficial Description of Some Objects Defined . . . 12
  - 4.2.1 Vertex . . . 12
  - 4.2.2 Edge . . . 12
  - 4.2.3 Polygon . . . 12
  - 4.2.4 Model . . . 13
- 4.3 Extraction Of Adjacency Matrix A . . . 13
- 4.4 $T_{n+1} = hAT_n + T_n$: An Iterative Approach . . . 13
- 4.5 Using The Program . . . 14
  - 4.5.1 Installation . . . 14
  - 4.5.2 Usage . . . 14

5 Results **14**
- 5.1 n = 1, 6-Celled Tessellation . . . 15
  - 5.1.1 Visualization . . . 15
  - 5.1.2 Adjacency Matrix . . . 15
  - 5.1.3 Solution . . . 16
- 5.2 N = 2, 24-Celled Tessellation . . . 16
  - 5.2.1 Visualization . . . 16
  - 5.2.2 Adjacency Matrix . . . 17
  - 5.2.3 Solution . . . 17
- 5.3 N = 3, 54-Celled Tessellation . . . 18
  - 5.3.1 Visualization . . . 18
  - 5.3.2 Adjacency Matrix . . . 18

5.3.3 Solution 18
5.4 N = 4, 96-Celled Tessellation 19
5.4.1 Visualization 19
5.4.2 Adjacency Matrix 19
5.4.3 Solution 19
5.5 Conclusion 19

6 Bibliography **20**

1 Abstract

The authors of this paper, Dylan Harrison and Evan Hauer, sought to model and visualize the diffusion of heat over a spherical surface. A Java application was developed to generate and display polygonal spheres, for which each polygon of the sphere represents a cell with a temperature variable. The evolution of the temperature values of each cell, over time, is dependent upon that of its neighboring cells. From the polygonal model of the sphere, a matrix representing the adjacency of each cell with one another may be derived, with which a mathematical model has been developed using Newton's Law of Heating and Cooling. This yields very large systems of ODEs. Due to the size of these systems, an iterative solution is implemented in the Java application the Authors developed in order to approximate its direct solution. The following is a discussion of the development of the aforementioned mathematical model, the rudiments of the Java application that was developed, the various iterative methods available for implementation, and finally, examines some data that has been procured from the application.

2 Introduction

2.1 Newton's Law of Heating and Cooling

$$\frac{dT}{dt} = -k(T - T_{env})$$

where

$$\begin{cases} \frac{dT}{dt} & \equiv \text{the rate at which temperature, T changes with respect to time, t} \\ k & \equiv \text{coefficient of heat transmission} \\ T & \equiv \text{temperature function} \\ T_{env} & \equiv \text{temperature of environment} \end{cases}$$

Newton's Law of heating and cooling describes the way in which the rate at which the temperature of some object changes with respect to the temperature of it's surrounding environment. It indicates that when the difference between the temperature of an object and its environment is large, its change in temperature will be large, while for smaller differences, the temperature changes by a smaller degree.

The solution of this equation will tend to be of the form of a monotone increasing or decreasing exponential function, for which T(t) increases or decreases toward T_{env} as time, t, increases toward infinity.

2.2 A Small Example System of ODEs

For the sake of clarification, a small system of two ODEs can demonstrate some aspects of the topic of this paper, from which the over all nature of the problem may be better understood.

Suppose that two objects are placed in contact with one another, both with the same propensity to transfer heat, k, equal to one, and with initial temperatures $T_1(0) = t_1$, and $T_2(0) = t_2$. Then the solution to the following system of ODEs describes the evolution of the temperature of the two objects as a function of time.

$$\begin{aligned} \frac{dT_1}{dt} &= -(T_1 - T_2) \\ \frac{dT_2}{dt} &= -(T_2 - T_1) \end{aligned}$$

This equation may be written in vector form as:

$$\begin{bmatrix} \frac{dT_1}{dt} \\ \frac{dT_2}{dt} \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \end{bmatrix}$$

This system has eigenvalues:

$$\begin{aligned} \lambda_1 &= -2 \\ \lambda_2 &= 0 \end{aligned}$$

and eigenvectors:

$$\begin{aligned} v_1 &= (-1, 1)^T \\ v_2 &= (1, 1)^T \end{aligned}$$

and so, the general solution is:

$$\begin{bmatrix} T_1 \\ T_2 \end{bmatrix} = C_1 \begin{bmatrix} -1 \\ 1 \end{bmatrix} e^{-2t} + C_2 \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

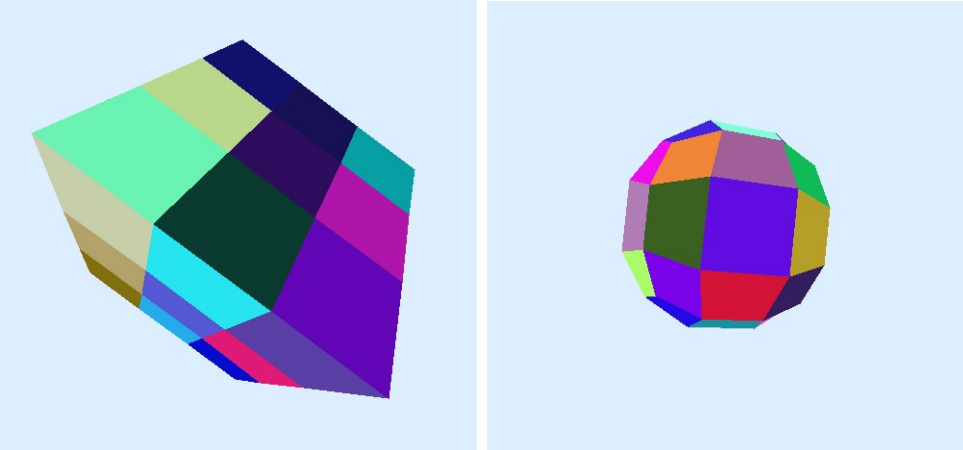
The coefficients, C_1 and C_2, can be solved for, given the initial conditions described above. As expected, the solution shows that the temperature of both objects approach the same temperature as time, t, approaches infinity.

2.3 Partitioning The Surface of A Sphere

The objective of this project is to model heat diffusion over the surface of a sphere. Rather than searching for a continuous solution, we seek to discretize the surface into cells of approximately equal surface area, each of whose temperature as a function of time is proportionate to those of its neighboring cells.

While each of the five Platonic Solids tessellate the sphere into equal, regular polygonal cells, such a solid can have at most twenty faces (the icosahedron). This project seeks to approximate the surface of the sphere to a resolution which may increase without theoretical bounds. There is no way to equally partition the surface of the sphere beyond the Platonic Solids, so this property must be abandoned for higher resolutions.

We have chosen to begin with the cube as a base tessellation of the unit sphere, and from there, partition each face into nXn equal squares. Once this partitioning is complete, each vertex of the model may be normalized, placing them on the surface of the unit sphere. This gives a fair approximation of the surface of the unit sphere. Below is a depiction of an $n = 3$ tessellation before and after normalization.



2.4 A Discussion of the Resulting System

As a cube has six faces, tessellations of those faces into nXn cells gives the approximate sphere with $6n^2$ cells for $n \in \{1, 2, 3, ...\}$. To each cell there is a corresponding differential equation, for which the rate of change of its temperature is proportionate to the temperature of its adjacent cells and itself. Cells are defined to be adjacent to one another if and only if they share an edge. In this way, for each cell on the tessellated sphere, that cell has a Von Neumann Neighborhood, whereby each cell is adjacent to four other cells. This property led to the selection for this tessellation, as it was anticipated that a general method to derive the adjacency matrix of this configuration would be forthcoming. That did not prove to be so.
We have the following vector equation of ODEs:

$$\vec{T'} = A\vec{T}$$

where
$\begin{cases} \vec{T'} \equiv \text{a column vector of } T'_m, \text{ for each cell, m} \\ A \equiv \text{an mXm adjacency matrix} \\ \vec{T} \equiv \text{a column vector of temperatures } T_m \text{ for each cell, m} \end{cases}$

This system is of order $O(n^2)$, and thus, increases in computational complexity rapidly as n increases. It is not practical to find the eigenvalues and eigenvectors of this system for larger systems, and thus, not practical to solve directly. We use Euler's iterative method for this system in order to determine the evolution of each cell's temperature over time.

$$\vec{T'} \approx \frac{\vec{T_{n+1}} - \vec{T_n}}{h}$$
$$\vec{T_{n+1}} = hA\vec{T_n} + \vec{T_n}$$

---

3 Methods

3.1 Eulers Method

Eulers method was invented in the late 1700s by Leonhard Euler. This is a simple numerical method for approximating the solution to differential equations. To use this equation, we can be given the differential equation and initial conditions. The initial slope and is used to predict the future points. The equation given for this method is:

$$W_{i+1} = w_i + h * f(t_i, w_i) - Equation 1.0$$

Here the wi+1 is the prediction of the next step. The initial conditions are the represented by the wi, where the I is the counter for this equation. This method requires a step size, which is the change in time between each prediction of the next point of the differential equation. The smaller the step size the more accurate the predictions will be.
For example, if we are given the differential equation:

Y = y + 4*t -2 from 0 to 2 where y(0) = 0.5; h= 0.5

Following the equation 1.0

$i = 0 - - - - - - - - - > w0 = y(0) = 0.5$
$i = 1 - - - - - - - - - > w1 = w0 + 0.5(w0 + 4 * 0.0 - 2) = 0.5 + 0.5(-1.5) = -0.25$
$i = 2 - - - - - - - - - > w2 = w1 + 0.5(w1 + 4 * 0.5 - 2) = -0.25 + 0.5(-0.25) = -0.375$
$i = 3 - - - - - - - - - > w3 = w2 + 0.5(w2 + 4 * 1.0 - 2) = -0.375 + 0.5(1.625) = 0.4375$

The above gives the approximations for the differential equation. The above is a relatively large step size or change in time so we can expect the error to be large compared to the actual solution.

3.2 Modified Eulers Method

The Modified Eulers method is an adaptation of Eulers Method (Previously Shown). This method provides a more accurate approximation than Eulers method. This solves a system of differential equations. The equation for the modified Eulers method is as follows:

$$W_{i+1} = w_i + h/2[f(t_i, w_i) + f(t_{i+1}, w_i + h * f(t_i, w_i))]$$
$$\text{Where } i \in \{0, 1, 2, 3, ...n-1\}$$

N is the max iterations that the method will undergo. As you can see above there is an extra parameter provides an infinite number of second-order runge- kutta formulas. Thus increasing the accuracy of the approximations. As before wi+1 is the approximation, wi is the initial conditions, f(ti,wi) is the differential equation with the initial conditions, and h is the step size.
For example:
Using the modified eulers method on the following differential equation:

$$Y = y - t^2 + 1$$
$$\text{from 0 to 2}$$
$$y(0) = 0.5$$
$$W_{i+1} = 1.22w_i - 0.0088i^2 - 0.008i + 0.216$$

$i = 0 - - - - - - - - - - - - - - - - - > w_0 = y(0) = 0.5$
$i = 1 - - - - - - - - - - - - - - - - - > w_1 = 1.22(0.5)0.0088(0)^2 + 0.008(0) + 0.216 = 0.826$
$i = 2 - - - - - - - - - - - - - - - - - > w_2 = 1.22(0.826)0.0088(0.2)^2 0.008(0.2) + 0.216 = 1.20692$

The answers to the above equations are the approximations that are being made for the differential equation. The initial conditions for the differential equation are the y(0) = 0.5. This allows the method to take place. Like the Eulers method the modified it is required that there are initial conditions.

3.3 Midpoint Method

This method is another adaptation of the original Eulers method. It provides a better approximation of the average slope for the interval. The method uses the Euler's method to shoot to an approximated midpoint. The error introduced by replacing the term in the Taylor method with its approximation has the same order as the error for the method, so the Runge-Kutta method is produced in this way. This is a second order method, therefore the error is proportional to h^3. H being the step size. The formula for the midpoint method is as follows:

$$W_{i+1} = w_i + h[f(t_i + h/2, w_i + h/2 * f(t_i, w_i))]$$
$$\text{Where } i = 0, 1, 2, 3, ...N - 1$$

The above uses the same notation as before where I is the counter, ti and wi are the initial conditions, and f is the differential equation. This method provides greater accuracy than the previous due to the fact of the midpoint approximation. It in a way average the two approximations together. Similarly to the previous methods this is required to have initial conditions and the differential equations.
An example of this method is below:

$$Y = yt^2 + 1$$
$$\text{from 0 to 2}$$
$$\text{y(0)} = 0.5$$
$$Wi + 1 = 1.22w_i - 0.0088i^2 - 0.008i + 0.218$$

$$i = 0 - - - - - - - - - - - - - - - - > w_0 = y(0) = 0.5$$
$$i = 1 - - - - - - - - - - - - - - - - > w_1 = 1.22(0.5) - 0.0088(0)^2 - 0.008(0) + 0.218 = 0.828$$
$$i = 2 - - - - - - - - - - - - - - - - > w_2 = 1.22(0.826) - 0.0088(0.2)^2 - 0.008(0.2) + 0.218 = 1.21136$$

These iterations continue on until the counter reaches 2. Below is the comparison between the second order methods of the midpoint and the modified Eulers method.

Table 5.6

		Midpoint		Modified Euler	
t_i	$y(t_i)$	Method	Error	Method	Error
0.0	0.5000000	0.5000000	0	0.5000000	0
0.2	0.8292986	0.8280000	0.0012986	0.8260000	0.0032986
0.4	1.2140877	1.2113600	0.0027277	1.2069200	0.0071677
0.6	1.6489406	1.6446592	0.0042814	1.6372424	0.0116982
0.8	2.1272295	2.1212842	0.0059453	2.1102357	0.0169938
1.0	2.6408591	2.6331668	0.0076923	2.6176876	0.0231715
1.2	3.1799415	3.1704634	0.0094781	3.1495789	0.0303627
1.4	3.7324000	3.7211654	0.0112346	3.6936862	0.0387138
1.6	4.2834838	4.2706218	0.0128620	4.2350972	0.0483866
1.8	4.8151763	4.8009586	0.0142177	4.7556185	0.0595577
2.0	5.3054720	5.2903695	0.0151025	5.2330546	0.0724173

3.4 Runge-Kutta

The runge -kutta method is a higher order method. The most common use of this method is the order of 4. It takes four steps to get across an interval, at each step it estimates a quarter of the interval. This manner allows for a more accurate approximation of the next point. This is a more advanced and complex way to solve for the differential equation, although this is the most accurate of the way of the following methods since the more steps taken will allow for greater accuracy. The method is

represented as follows:

$$
\begin{aligned}
K1 &= h * f(t_i, w_i) \\
K2 &= h * f(t_i + h/2, w_i + 1/2 * K1) \\
K3 &= h * f(t_i + h/2, w_i + \ * K2) \\
K4 &= h * f(t_{i+1}, w_i + K3) \\
W_{i+1} &= w_i + 1/6(K1 + 2K2 + 2K3 + K4) \\
i &= 0, 1, 2, 3, ...n - 1
\end{aligned}
$$

The Ks here are the partial steps in between the step size of the overall equation. This is the partial step size referred to above. The w_i and t_i are the initial conditions like before, h is the step size, and f is the differential equation.
An Example is as follows:

$$
\begin{aligned}
Y &= y - t^2 + 1 \\
&\text{from 0 to 2} \\
&\text{y(0)} = 0.5 \\
W_0 &= 0.5 \\
K1 &= 0.2 * f(0, 0.5) = 0.2(1.5) = 0.3 \\
K2 &= 0.2 * f(0.1, 0.65) = 0.328 \\
K3 &= 0.2 * f(0.1, 0.664) = 0.3308 \\
K4 &= 0.2 * f(0.2, 0.8308) = 0.35816 \\
W_1 &= 0.5 + 1/6(0.3 + 2(0.328) + 2(0.3308) + 0.35816) = 0.8292933
\end{aligned}
$$

The Runge-Kutta method requires order of 4 requires four evaluations per step. Thus it gives a more accurate answers than the Euler method. Similarly the second order methods of the Midpoint method and modified Eulers method are less accurate due to more steps of the order of 4.

3.5 Finite-Difference

This is a method for again approximating the values for differential equations. This creates a grid in which to approximate the values within the grid size. In general, the method replaces the derivatives by finite differences. These are primarily used for larger sample sizes vs the previous methods that can be done for all sizes. This uses boundary conditions that help to keep the function within the grid size. This method is modeled by:

$$2[(h/k)^2 + 1]w_{i,j} - (w_{i+1,j} + w_{i-1,j}) - (h/k)^2(w_{i,j+1} + w_{i,j-1}) = -h^2 * f(x_i, y_j)$$

Where

$$i = 1, 2, ...N - 1 \text{ and } j = 1, 2, ...m - 1$$

$$W_{0j} = g(x_0, y_j)$$

$$w_{nj} = g(x_n, y_j)$$

$$w_{i0} = g(x_i, y_0)$$

and

$$w_{im} = g(x_i, y_m)$$

For each i = 1,2, ... N-1

$$j = 0, 1, ...m$$

where

w_{ij} approximates $u(x_i, y_j)$

The i and j in this case are counters for the two different variables. The x_i and y_j are show the placement in the grid. This method gives us a system of equations that can then be solved by using iterative methods such as the SOR, Jacobi, or newtons method.

In Math 415 we were introduced to the Finite-Difference Methods. This method utilizes making a grid with boundary conditions. This is the part that we found to be helpful in creating an adjacency matrix that describes the face of the cube. The user inputs a number, n, this number is then used as the number of rows and columns. Choosing the step size, h and k, n another selection part that makes the matrixes into equal parts. H and K are created by the following:

$$H = (b - a)/n$$
$$K = (d - c)/n$$

Here the interval [a,b] is the starting and ending points along the x-axis, the interval [c,d] is the y-axis. The grid is then provided a rectangle R by drawing vertical and horizontal lines with the point coordinates (x_i, y_j)

$$x_i = a + i * h$$
$$y_j = c + j * k$$

Here i and j are used as interval of time. For example, I = 0,1, 2, ... n and the same for j. This method was utilized since it creates a spare matrix. In this example the face of the cube that is broken up into a grid that models the face of the cube. This method can be used to model each face individually. What is meant by modeling each face is that it was successful in creating the adjacency matrix for one face. Unfortunately, this is only a piece of the puzzle in our case.

This method broke down when it came to modeling the entire cube together. The

Finite method was unreliable when coming to creating the adjacency matrix for the entire cube together. That is with all six faces. I believe that the reason for this is because the method of numbering each face and then using the numbered faces to relate each cube to another proved difficult to compute.

The program that was used to find the relationship between the faces for each cube are located in the appendix. The program would put a 1 in the spots where the two numbered faces would be touching. Otherwise the spots were left blank. In the space that was the numbered face itself there would also be a 1. This was to make sure the continued cooling path that will be talked about later on.

4 Implementation

4.1 On Choosing Java for Implementation

Most of the work involved in this project has been in developing the Java program which models, visualizes and carries out the aforementioned calculations. Java was chosen for its object oriented-ness. This property lends to developing larger applications while keeping its complexity manageable. The desired visualizations were likely not possible in matlab.

4.2 A Superficial Description of Some Objects Defined

4.2.1 Vertex

A vertex is defined as a 3-tuple corresponding to its x, y, and z coordinates in three dimensional space. Some operations defined for vertices are translations, rotations, scaling, normalizing, etc... Most importantly, the operation, equals, is defined to compare two vertices and return a true or false, boolean value depending on whether their distance is to within a provided tolerance.

4.2.2 Edge

An Edge very simply consist of an origin vertex, and an endpoint vertex. The only operation defined on Edges that is of importance within the context of this paper is the undirectedEquals method. This method returns a true/false boolean value when comparing two edges. If two edges have the same origin and endpoint, or the endpoint of one Edge is the origin of the other, and its origin is the others' endpoint (i.e. they differ only by direction), These are decidedly equal.

4.2.3 Polygon

A Polygon is a list of (hopefully co-planar) edges for which the endpoint vertex of each edge in the list is the origin vertex of the edge that follows it in the list, and for

which the origin vertex of the first edge in the list is the endpoint vertex of the final edge in the list. Some further restrictions are also necessary, such that edges in the list can not cross one another, etc.. In this project, the polygons also double as cells. To each polygon, there is a temperature value, and a history value. the history value records samples of its temperature as it changes over time. Building from the comparison methods described for the Edge and Vertex objects, the Polygon object defines a method isAdjacent. This method compares two polygons and returns true if the two polygons share an edge.

4.2.4 Model

A Model is a list of polygons and their corresponding adjacency matrix. This object defines the method, generateNNCellCube. This method constructs the vertices, edges, and polygons of the sphere in question. In brief, the method works by defining the faces of the cube and their nXn tessellation in separate parts, and then stitches them together. Beginning with one sheet, the next is collated by introducing polygons, one by one. If the introduced polygon contains a vertex that is equal (to within tolerance) to a vertex already introduced, that polygon's vertex is discarded and replaced by the one already introduced. In this way there are no redundant vertices in the model, and the Edge method, equals, works as intended. Once the sheets are stitched together, we iterate through each vertex in the model and normalize them, one by one. This places each vertex on the surface of the unit sphere.

4.3 Extraction Of Adjacency Matrix A

Once the Model is completed, It is straight forward to extract the model's adjacency matrix. In a nested for loop structure, we iterate through each polygon/cell, say cell u in the model, and compare it to each other cell, say cell v in the model with the Polygon objects isAdjacent method. If cell u isAdjacent to cell v, then the adjacency matrix element A_{uv} is set to one. If cell u is cell v, then the adjacency matrix element A_{uv} is set to negative four (the elements along the diagonal of A are all -4). Adjacency matrix elements A_{ij} are all zero elsewhere. Reflecting on the matrix developed for the small system of two ODEs appearing earlier in the paper, we see that these are the correct values for the non-zero entries of A, as each cell in this model is adjacent to four other cells, whereas in the small example, each cell was only adjacent to one other cell. In a manner similar to the small example, we have chosen to restrict the scope of the project, such that the coefficient of heat permeability of each cell is equal to one. This whole process is also carried out within the generateNNCube method defined in the Model class.

4.4 $T_{n+1} = hAT_n + T_n$: An Iterative Approach

Still within the Model object, is defined the method computeAx. This method carries out Euler's method for iterative solutions to ODEs on the whole vector equation.

I would have liked to develop this method further and implement more sophisticated iterative methods, such as the Runga-Kutta method, and more so, to have developed a means to define a vector forcing function to accompany it. Unfortunately, time was not permitting. The method, computeAx, takes one parameter which operates as the h value in the equation. This number is defined elsewhere in the program.

4.5 Using The Program

4.5.1 Installation

The most straightforward way to run the project is to take the following steps: (1) Run Eclipse; (2) File->New->Java Project; (3) Type whatever file name you like in the text field (4) Click Finish (near the bottom of the prompt) a new folder should appear in the package explorer window with your file name; (5) Copy all 9 .java source files provided; (6) Right-Click on the new folder that appeared in the package explorer window and select paste. There is a more elegant method to "import" projects into eclipse, however that does not seem to work reliably, and is not very straight forward.

4.5.2 Usage

From the Eclipse toolbar, near the top of the window, there is a green arrow icon. Clicking that runs the program. From the package explorer window, double click the folder with the name you have chosen. Navigate to Src->(Default Package). The nine provided source files should appear there. Double clicking on the file name test.java opens it for editing in the Eclipse editor window. Near line 35 are a few parameters that can be altered: tf, the amount of time to iterate over, n, the number of iterations to partition the time interval into, and different positive integers may be passed to the generateNNCube method to generate more finely tessellated spheres. I have not experimented with values much larger than 20, as this is enough to run slowly on my computer. The Program will generate output until the number of iterations is reached, and then carry on indefinitely without doing much. The final block of output is a list of vectors corresponding to the temperature values of each cell sampled throughout the computation. This output May be copy-pasted into matlab.

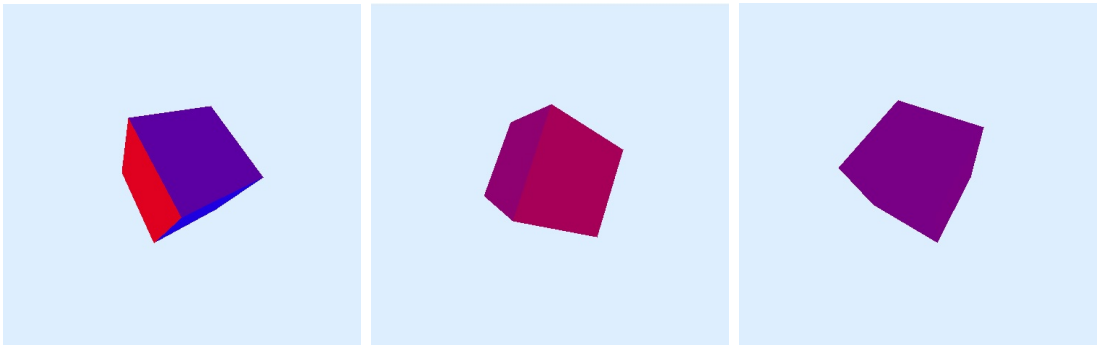
5 Results

For experimentation, we examine the results from tessellations of the unit sphere by increasing degree. To each cell in the tesselation we give an initial random value between $\pm$ 500.0. In order to indicate temperature visually, we color cells with the highest temperature bright red, cells with the lowest temperature are colored bright blue, and those temperatures are in between are colored an interpolated color between bright red and bright blue to a degree that is proportionate to it's relative temperature. We use Euler's iterative method on each system, whereby the initial time, t_0 is 0, and

the final time, t_f is one unit of time. This interval is divided into 1000 iterative steps. Throughout the computation, the temperature of each cell is logged a total of 20 times at even intervals. (Please note that I was not able to devise a method to capture each of the three visualization snapshots all from the same simulation, they are in fact, snapshots of separate simulations captured at various phases.)

5.1 n = 1, 6-Celled Tessellation

5.1.1 Visualization

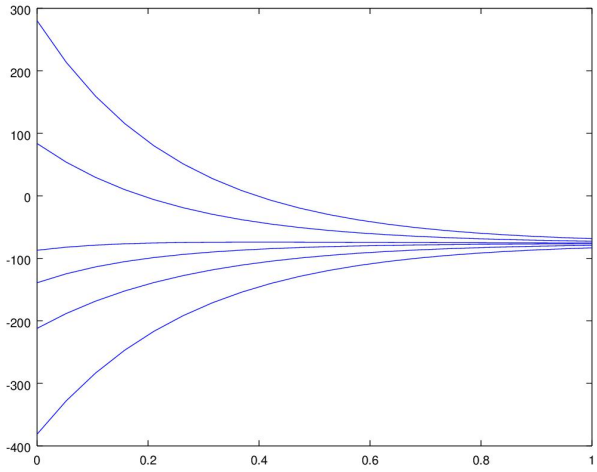


snapshots of the n = 1 (cube) tessellation at t_0, $t_{0.5}$, t_1

5.1.2 Adjacency Matrix

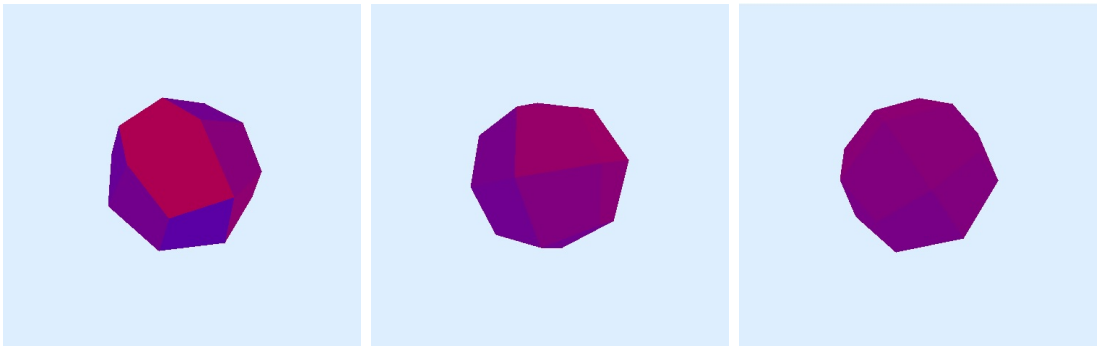
$$\mathrm{A} = \begin{bmatrix} -4.0 & 1.0 & 0.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & -4.0 & 1.0 & 0.0 & 1.0 & 1.0, \\ 0.0 & 1.0 & -4.0 & 1.0 & 1.0 & 1.0, \\ 1.0 & 0.0 & 1.0 & -4.0 & 1.0 & 1.0, \\ 1.0 & 1.0 & 1.0 & 1.0 & -4.0 & 0.0, \\ 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & -4.0, \end{bmatrix}$$

5.1.3 Solution



5.2 N = 2, 24-Celled Tessellation

5.2.1 Visualization



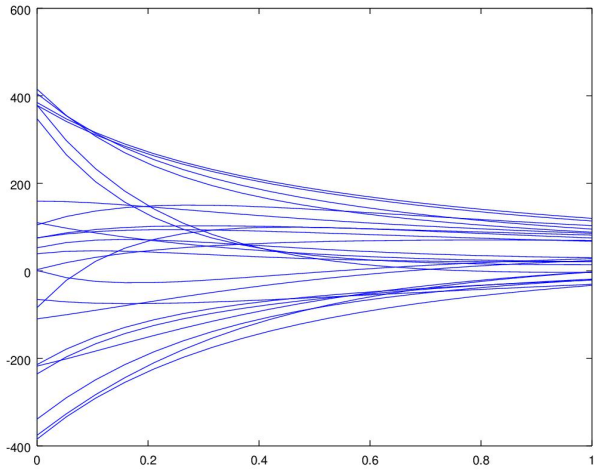
snapshots of the n = 2 tessellation at t_0, $t_{0.5}$, t_1

5.2.2 Adjacency Matrix

$$A =$$

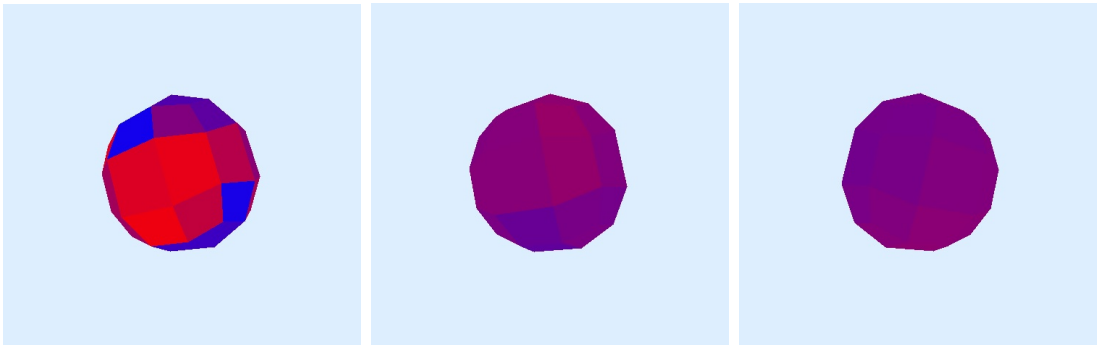
$$\left[\begin{array}{cccccccccccccccccccccccc}
-4.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\
1.0 & -4.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
1.0 & 0.0 & -4.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 1.0 & -4.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 0.0 & 0.0 & -4.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & -4.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 & -4.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & -4.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & -4.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & -4.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 & -4.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & -4.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & -4.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & -4.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 & -4.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & -4.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & -4.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & -4.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 1.0 & -4.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & -4.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -4.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & -4.0 & 1.0 \\
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & -4.0 \\
0.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0
\end{array}\right]$$

5.2.3 Solution



5.3 N = 3, 54-Celled Tessellation

5.3.1 Visualization

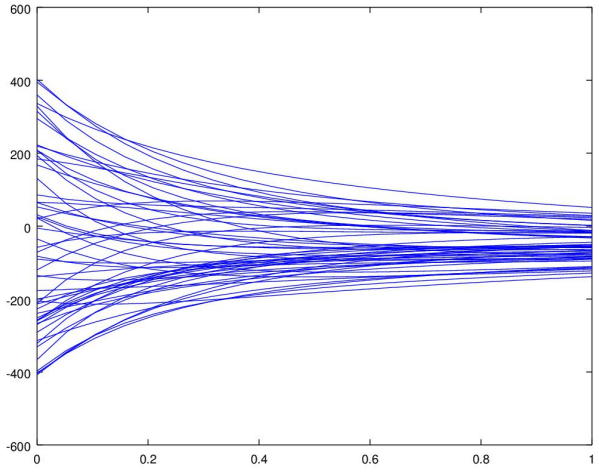


snapshots of the n = 3 tessellation at t_0, $t_{0.5}$, t_1

5.3.2 Adjacency Matrix

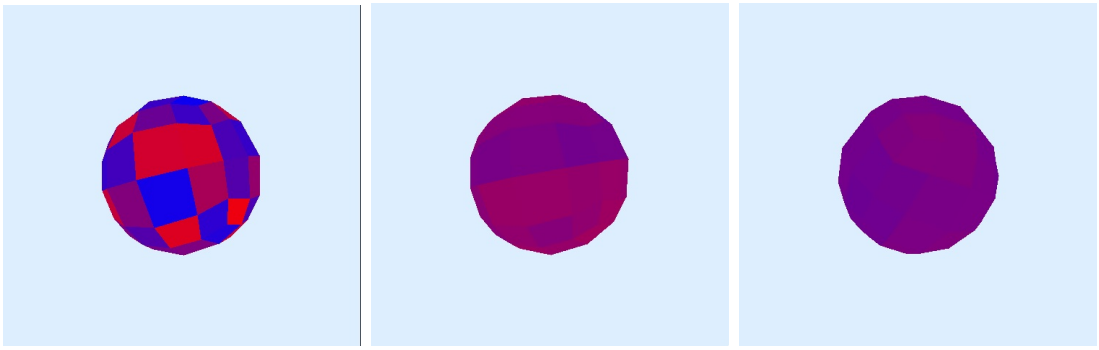
The 54-Celled Tessellation adjacency matrix is far too large to fit on this page.

5.3.3 Solution



5.4 N = 4, 96-Celled Tessellation

5.4.1 Visualization

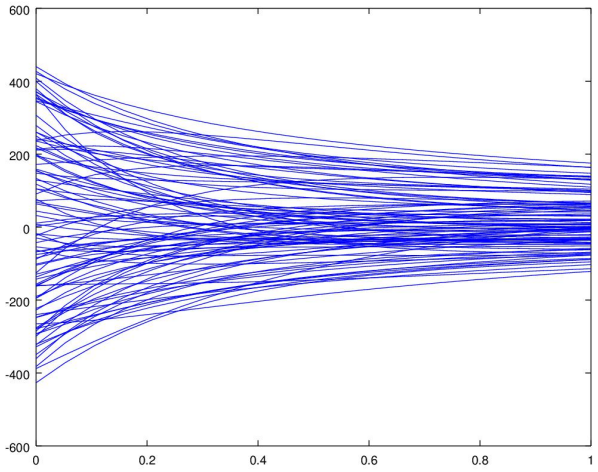


snapshots of the n = 4 tessellation at t_0, $t_{0.5}$, t_1

5.4.2 Adjacency Matrix

The 96-Celled Tessellation adjacency matrix is far too large to fit on this page.

5.4.3 Solution



5.5 Conclusion

While The application developed for this project has no theoretical limit on how large the degree of tessellation may grow, it is very difficult to express the data procured from larger simulations as the size of the data quickly grows too large. The title page depicts a much larger simulation of about n = 20. At larger degrees of tessellation than that, computation becomes nearly too intensive for the author's computer. The solution plots depict the temperature values that each cell took on throughout the simulation.

This was accomplished by having each cell record its own temperature at regular intervals. Once the simulation is complete, each cell prints it's history of temperatures in a matlab style vector assignment statement. These output strings are then copied from the Eclipse output console and pasted into a very simple Matlab script which plots them all at once. Each Solution plot has an apparent trend whereby the temperature of each cell approaches a universal average temperature. It is interesting to observe the plots near to the y-axis. I suspect that lines with greater rates of change depict cells that end up in neighborhoods with vastly opposite temperatures during the initial randomization.

---

6 Bibliography

References

[1] Faires, J. Douglas., and Richard L. Burden, Numerical Methods, Pacific Grove, CA, Thomson/Brooks/Cole, 2003. Print.

[2] Other Differential Equations, Newton's Law of Cooling, N.p., n.d. Web, 14 May 2016