The University of Sydney

SCHOOL OF ELECTRICAL AND INFORMATION ENGINEERING

PROJECT CLEARANCE FORM

Unit of Study Code and Name

Thesis A ELEC 4712

This is to certify that my student

Student Name Louis Angelo Policarpio

SID 470358803

Has:

- **Returned all books and reference material;**
- **Returned all equipment and keys; and**
- **Tidied their work place.**

SUEIE Academic supervisor:

Signature:  **Date:** 26 May 2022

Name: Dr. Vera Miloslavskaya

External supervisor (if applicable):

Signature:  **Date:** 28/May/2022

Name: Prof. Masahiro Takatsuka

The University of Sydney

Faculty of Engineering

School of Electrical and Information Engineering

Year 2021

A thesis submitted in partial fulfilment of a Bachelor of Engineering Honours (Software Engineering) Degree

Student Name	**Louis Angelo Policarpio**
SID	**470358803**
Unit of Study Code and Name	**Thesis A** **ELEC 4712**
Supervisor	**EIE: Dr. Vera Miloslav kaya, Prof. Branka Vucetic** **External: Prof. Masahiro Takatsuka**
Title	**Interactive visualisation of Self-organising map**

Statement of Achievements

The work presented in this thesis is the result of the original research carried out by myself, in collaboration with my supervisors, while enrolled in the School of Electrical and Information Engineering at the University of Sydney as a for the Bachelor of Software Engineering. The studies were conducted under the supervision of Dr. Vera Miloslavskaya , Prof. Branka Vucetic and Prof. Masahiro Takatsuka . No part of this work has been submitted for any other degree or award in any other university or educational institution.

The following aspects of the thesis are believed to be important achievements:

- Comprehensive literature review Providing an overview of self-organising maps
- Proposed solution of 2D Projection of 3D self-organising maps. Utilising Spherical SOMs, map projections and interaction, instead of the conventional toros with a square or rectangular projection
- Comprehensive literature review outlining and comparing 2D Projection Methods Suitable for spherical self-organising maps
- Performance and complexity analysis of proposed solution, outlining the systems limitations regarding geodesic dome frequency
- Proposed Improvements and extension of work

Contents

1. Introduction

A self-organizing map (SOM) is an artificial neural network used to simplify high-dimensional data into low-dimensional data whilst maintaining the data's topological structure, developed by Professor Teuvo Kohonen in the 1980s [1]. The purpose of dimensionality reduction is to aid in "data analysis, clustering problems, and visualization of high dimensional datasets" [2]. There are two main procedures when implementing a SOM, they are training and mapping/visualisation. Training is the process of high-dimensional data into low-dimensional data, and visualisation is the process mapping the low-dimension data to map that is interpretable.

When building a SOM, the following factors need to be considered, lattice structure, lattice size, and weight vector initialization method. Lattice structure refers to the shape of each grid. The lattice size refers to the number of nodes within the lattice. The weight vector initialisation method is the method in which the weights are assigned to each vector in the lattice. Changes in the structure and the size can impact the visualisation of the SOM, whereas the initialisation method impacts the efficiency of the training process [2].

The structure of the lattice can vary in dimension, for example, a flat 2D plane grid or a 3D spherical grid can be used. The different dimensions have their advantages and disadvantages, a 2D SOM allows for greater readability however it reduces accuracy of the visualisation due to the problem known as the border effect. The border effect occurs due to the nodes on the edge/ border of a 2D SOM lattice having fewer neighbouring nodes. Fewer neighbouring nodes means that the nodes on the edge will have a lower chance of being updated. There have been some mathematical solutions to solve the border effect such as "the heuristic weighting rule method by Korhonen (2001) and local-linear smoothing by Wand and Jones (1995)" [3], however, the simplest method is to use a 3D lattice [3].

The most straightforward way to create a 3D lattice is to connect boundaries (left connected to right or top connected to bottom), forming a torus. The torus lattice removes the border effect; however, it reduces the readability and area of each grid varies greatly. It is important for a SOM to have "equal geometrical treatment" as it can impact the accuracy off the

visualisation. The preferred method for a 3D lattice is a spherical lattice, which has greater readability than a torus and more equal sized grids, whilst combatting the border effect [3].

However, a spherical visualisation is still harder to read and interpret than a 2D lattice. The curved surface of a sphere distorts distance, and as a sphere is a 3D shape all data cannot be viewed at once. This work aims to propose a solution to this issue by presenting methods to project the sphere into 2D and providing interactive interface.

2. Literature Review

2.1. SOM

2.1.1. SOM Training

The SOM training involves placing a lattice on the multidimensional dataset. A value in the dataset is then selected, and the closest node in the lattice is selected and moved towards it. The process is repeated and as it progresses the lattice nodes move into the centres of clusters in the data set. The moving process is referred to as training and function that determines how the nodes move is known as the neighbourhood function. [2] The neighbourhood function selects the winning element in the data set by calculating the Euclidian distance for the current lattice node, from that the winning node and its neighbouring nodes weight vectors are adjusted. The adjustment will continue with the neighbours of the neighbouring nodes until the update radius is reached (the distance from the winner node) [4]. After training has been completed the lattice with its updated weight vectors is the result.

2.1.2. SOM Visualisation

There are numerous visualisation methods used to represent a SOM. Common consist of class representation maps, U-matrix, and component planes. Class representation maps are a 2D representation of the SOM, which observes the frequency of dataset classes in each node of the lattice. There are many methods of implementing class representation maps. In Ponmalai and Kamath's paper, the class representation is focused on the frequency in which a class appears in each node, assigning a unique colour to each class. There are 2 methods outlined in the paper. The first method simply identifies the class of highest frequency in the node and displays its colour. The second gets the average colour using the frequency of each colour. The first method results in clearer visualisation as the classes and the borders between them are easier to understand. The second method alone is not preferred as the average colour is

hard to interpret [2]. As the number of classes increases the harder it is to interpret the data. However, Brereton's paper utilises the best matching unit (BMU) to colour the node. In this implementation, each node in the lattice represents a node. The BMU is derived by calculating the Euclidean distance between each dataset element and each node. The class with the closest BMU will colour that node. If there are nodes of equal distance an average of the colours will be used. [5]. Using either method of class representation would be suitable for the interactive 2D projection of the spherical SOM. This is as they provide a clear visualisation of the SOM.

The U-matrix was developed by Ultsch and Siemon [5]. Ponmalai & Kamath's paper states that it illustrates the spacing of the nodes in the lattice [2]. Whilst Brereton's paper states that it illustrates, "the similarity of a unit to its neighbours." Both outlined methods aim to display the shape of the data, however, Brenton's method gives a clearer representation of the groupings of the data. Moreover, both methods do not give information on individual classes, but the overall grouping. Either method of implementing U-matrices would be suitable, as both methods display a single visualisation of the data that can be applied to a 2D lattice.

Component planes create separate visualisations for each class, depending on the method of the visualisation of the classes may vary. For example, in Brenton's paper, the BMU of each node is used [5], whilst in Ponmalai & Kamath's weight vectors are used [2]. The component planes allow for comparison of classes, and a greater understanding of each class, however, it does not achieve one of the main goals of the SOM reducing the dimensions of the data. This method can be implemented however would require the creation on several visualisations so more care would be need so that the system runs efficiently and without error.

From the mentioned visualisation methods, the class representation and a U-matrix are best suited for implementation in the interactive spherical SOM and its 2D projection. However, implementation of the visualisation feature is outside the scope of the thesis, as the focus of this thesis is the 2D projection and interactive visualisation of a spherical SOM. Once that is complete either of the visualisation methods can be utilised on the 2D projection and interactive visualisation of a spherical SOM.

2.2. Projection of sphere to 2D space

There are various methods to project a sphere into 2D space, and for years cartographers have been developing projection methods. When projecting a sphere into 2D space it is inevitable for there to be distortion, therefore properties that need to be retained (not distorted) need to be selected and prioritised. There are 4 main categories of projections based on what geometric properties they aim to preserve.

The categories of projections are equal-area, conformal, equidistant and compromise. Equal area projections aim to maintain the same area of the grids on the projection; however, this does not mean the shape of each grid is the same. Conformal projections aim to preserve the local angles and shapes on the projection; and are only essential for surveying and navigating with a protractor. Equidistant projections preserve the distances between points along a direction and are used to determine distances between different points. Compromise projections make sacrifices in area, angle, and distances to "balance the distortion" [6]. In a SOM, it is most important to retain an equal area, therefore, equal-area projections are the preferred projection method.

2.2.1. Equal-area projections

Equal-area projections aim to maintain the area of each grid in the lattice, not angles or shapes. This can be visualised by placing ellipses on the sphere and comparing the transformation of the ellipses after the projection. For equal-area projection, the area of the ellipses should be similar for all ellipses on the projection whilst the shape and angles of the ellipses can vary, which can be seen in figure 1 [7]. For equal-area projections either lines or points are used to represent the poles of the sphere. If points are used there will be substantial amounts of distortion on the poles. When lines are used there will still be distortion but to a lesser degree, however, the grids will be compressed vertically [3]. As distortion is more uniform when poles are lines, it is the more suited option for equal-area projections in SOMs.

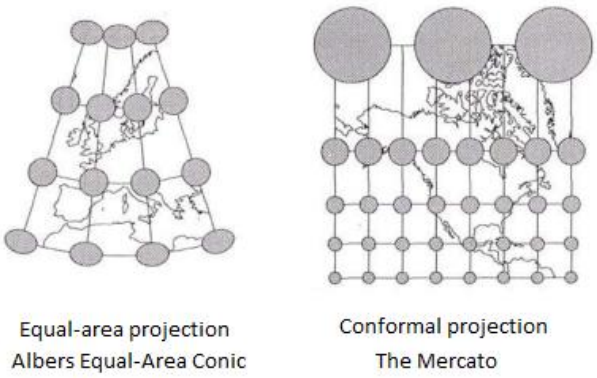


Figure 1- comparing the area of maps using circles

2.2.2. Adaptive composite map

As the created projection is going to be interactive, we can look at how interaction is handled on Web maps [8]. Web map creators have faced issues with online maps and making them interactive. The issue of interactive maps is that when users zoom and interact with the map areas of the map become more distorted. Cartographers have produced two solutions to this issue, first, apply the same protection for different scales. Secondly, cartographers can change the map projection based on the scale and location that the user is viewing. The second method is known as the adaptive composite map, whilst in the first method "excessive distortions" at certain scales are inevitable [8]. The map transitions and combines different projections providing an equal area for all scales and views of the map.

2.2.3. Aitoff's equal area using Lambert azimuthal projection

David Aitoff proposed a transformation method in 1889. The method utilises the following equal-area projections methods: Hammer, the Eckert–Greifendorff, quartic authalic projections and the Lambert azimuthal projection. To perform the transformation with the Lambert azimuthal projection, we multiply the abscissa (the distance from the vertical axis) by a chosen factor, then divide its longitude by that given factor. The process of the transformation can be seen in figure 2 and the bellow equation where B is the selected factor [8]. This method allows for adaptive equal-area projections. However, this method utilises

points for the projection poles, as previously stated making the points of the projection points lead to more distortion at the poles.

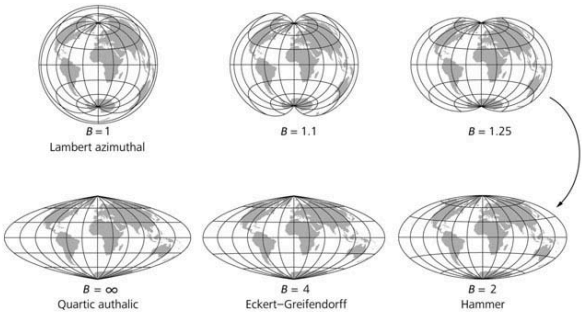


Figure 2- Aitoff's equal-area using Lambert azimuthal projection

The formula for the Lambert azimuthal projection:

Where x and y are the coordinates, ϕ and $\boldsymbol{\lambda}$ respectively are the latitude and longitude transformation factor and B is the factor that straitens the abscissa.

2.2.4. Adaptive Equal-area Projection using Wagner's method

The optimal solution of an Adaptive Equal-area projection would be to have a projection that transforms from the Lambert azimuthal projection, has lines as poles and retains an equal area whilst it transforms. However, a method that meets these requirements has not been found. Therefore, the Wagner's transformation method may be used instead.

Wanger's method was proposed by Karl Heinrich Wagner in 1932 [9]. There are three versions of this method, however, only one maintains the area. In this work when referring to Wagner's transformation method, the method prescribed in Canters (2002) [10] as Wagner's second transformation method is outlined in figure 3. The transformation process begins with creating a segment on the sphere, bounded by the bounding parallel (ϕ_B) and bounding meridian ($\boldsymbol{\lambda}_B$), which are mirrored by the equator and meridian, respectively (step 1 figure 3). After that, the sphere is projected onto the newly created segment (step 2 figure 3). Next, the segment is scaled using the scale factor calculated using the below formula (step 3 figure 3)

$$scale\ factor = \frac{1}{\sqrt{m \cdot n}}$$

$$where, \qquad m = sin\phi_B\,, \qquad n = \frac{\lambda_B}{\pi}$$

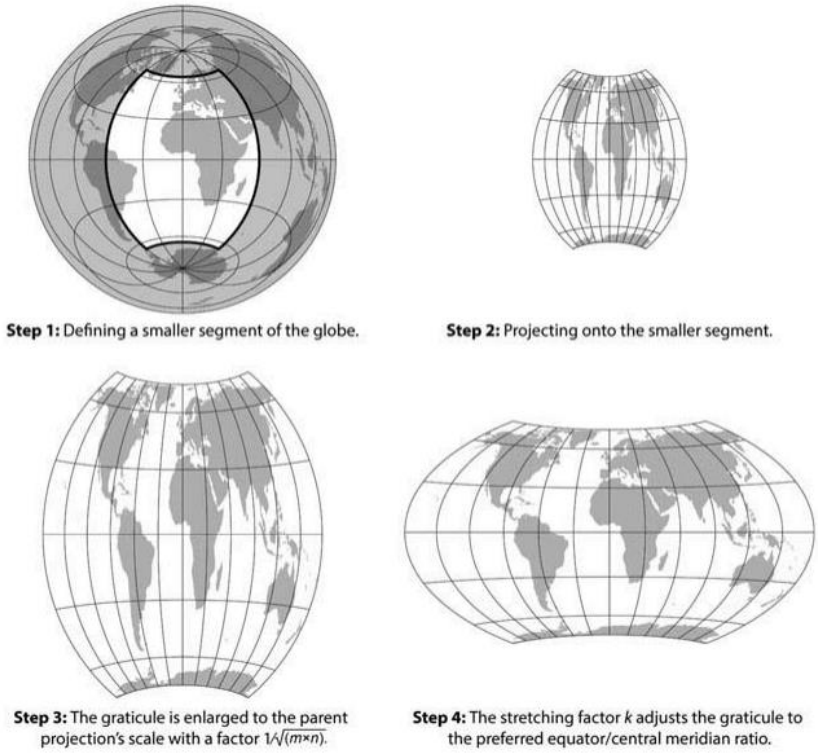


Figure 3 - Equal-area Projection using Wagner's method

Finally, the segment is then stretched using stretching factor k. The value of k varies depending on the desired equator, meridian ratio (step 4 figure 3). After stretching the segment is now the final Wagner projection. The general formula for Wanger's method is:

$$x = \frac{k}{\sqrt{m \cdot n}} f_x(\theta, n\lambda)$$

$$y = \frac{1}{k\sqrt{m \cdot n}} f_y(\theta, n\lambda)$$

Where f_x and f_y are the coordinates on the original projections, $\sin\theta = m \cdot \sin\phi_B$, $m = \sin\phi_B$, $n = \boldsymbol{\lambda}_B/\boldsymbol{\pi}$ [8].

2.2.5. Wanger's transformation method combined with Lambert azimuthal projection

To allow for an adaptive equal-area projection with lines as poles Lambert azimuthal with Wagner's method can be used. The following formula can be used to generate the Lambert azimuthal projection with Wagner's transformation method:

where ϕ and λ are the latitude and longitude, sinθ = m·sinϕ, m, n and k are from variables used in Wagner's transformation, in which, m = $\sin\phi_B$, n = $\boldsymbol{\lambda}_B/\boldsymbol{\pi}$, k = $\sqrt{p \cdot \sin\frac{\phi_B}{2} \Big/ \sin\frac{\lambda_B}{2}}$,

such that ϕ_B and $\boldsymbol{\lambda}_{B\ are}$ the bounding parallel and the bounding meridian [8]. The resulting

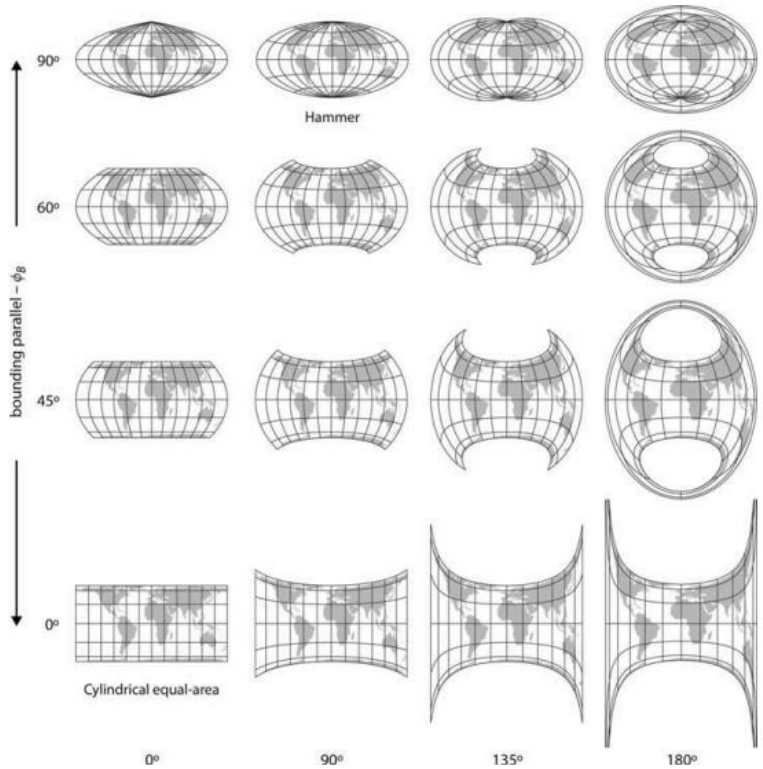


Figure 4 - Wanger's transformation method combined with Lambert azimuthal projection

projections can be seen in figure 4 where p =2.

As the degree of the bounding parallel increases the lines of latitude are more bent, the same occurs for the bounding meridian and the lines of longitude. Moreover, the bounding meridian increases the length of the poles. Several key projections can be generated using this method. First a cylindrical projection is formed when both the bounding parallels and meridians are 0°. When the parallel is 90° Aitoff's transformation method with the Lambert

azimuthal projection can be seen with Aitoff's transformation being formed with a pounding parallel and meridian of 90° and 180° [8].

Šavrič and Jenny assessed the projections generated from the Lambert azimuthal projection with Wagner's method via two factors distortion analysis and aesthetic appearance. Distortion analysis was done through computing indices for scale and angular distortion. Aesthetics were assessed through expert opinion, in which map projection experts created the projections that they believed were graphically the best to them [8].

The results for the scale distortion analysis were as follows. Projections with [8]:

- lowest angular distortion:
    - Bounding parallel < 50°
    - 2 > ratio p >2.7
- Lowest Scale distortion:
    - 35° > Bounding parallel > 75°
    - 1.7 > ratio p > 2.4

As previously stated, angular distortion is not important for SOMs whilst scale distortion may aid in improving the readability of the SOM. Thus, when generated projections should aim to fall in the above criteria for lowest scale distortion.

The results from the aesthetic analyse found that experts aimed to balance the stretching in the equator and the shape distortion on the sides ("Australia, South America, and East or South-East Asia" on a map of earth). All expects chose projections with lines as poles and curved borders on the meridians. Half of the experts set the ratio p to 2 as it reflects the real ratio, and the length of the poles were adjusted to balance the distortion that the p ratio caused. Moreover, most projections created were in the criteria for the lowest scale distortion [8].

2.2.6. Šavrič and Jenny's final selection of projections

Through the aforementioned analysis of distortion and aesthetic appearance, Šavrič and Jenny selected four candidates to include into the adaptive projections. The candidates include the projection with:

1. the best scale distortion index
2. the best angular distortion index

3. the mean values of the expert's suggestions
4. equator and meridian ratio of 2

Table 1 shows these four candidates (in bold) as well as other common projections that can be transformed using Wagner's method on the Lambert azimuthal projection and ranks them by mean error in scale distortion. All methods except the best angular distortion have the same amount of error in scale distortion and performed better than most of the other common projection methods (aside from Wagner VII). The best angular distortion does not

Table 1. Weighted mean error in the overall scale distortion.

Wagner VII	0.37
Best scale ($\phi_B = 57.5°, p = 2.02$)	**0.38**
Experts' mean ($\phi_B = 61.9°, p = 2.03$)	**0.38**
Ratios equal to 2 ($\phi_B = 65.1°, p = 2$)	**0.38**
Best angular ($\phi_B = 28°, p = 2.43$)	**0.43**
Hammer	0.43
Eckert–Greifendorff	0.45
Quartic authalic	0.47

perform as well due to its meridian begins curved. Curved meridians cause compression on the polar areas and stretching along the sides of the projection, thus would not be the optimal projection to use. The remaining three have the same mean error in scale distortion and visually are similar, the difference being the lengths of the central meridian and pole lines. The experts' mean has better mean angular deformation and thus is recommended to be included in the transformations [8].

2.2.7. Final set of projections for adaptive composite map projection

The set of maps Šavrič and Jenny selected for adaptive composite map projection are:

1. The new equal area pseudo-cylindrical projection created from the average of expert results ($\phi_B = 61.9$, $p = 2.03$). Which is suggested to be used as the original map.
2. The Lambert azimuthal projection (with $\lambda_B = 180$, $\phi_B = 90$, and $p = \sqrt{2}$)
3. Cylindrical equal-area projection, for large scale mapping ($\lambda_B = 0$, $\phi_B = 0$, and $p = 2$).
4. Wagner VII (with $\lambda B = 60$, $\phi B = 65$, and $p = 2$). [8]

Utilising these maps allows for equal area projections with lines as poles suited to the SOM. The transformation also allows for interaction through zooming in on the projection which would be a useful feature for the interactive SOM.

2.4. Geodesic dome

A lattice is needed in the creation of SOM and maintaining uniformity in the grids of the lattice is essential for producing accurate SOM visualisations. Therefore, for spherical SOMs a lattice structure should aim to maintain the number of direct neighbours and the distances between those neighbours. There are only 5 methods in which uniformity can be achieved: through the platonic polyhedra which consist of the tetrahedron, cube, octahedron, icosahedron, and dodecahedron. A Geodesic dome is created through tessellation on the platonic polyhedra. The method of tessellation was made by Fuller in 1975 [12], in which tessellation occurs through dividing the faces of the polyhedron into triangles. The subdivision is done so that the new edges are parallel to the edges of the polyhedron face. Out of the 5 platonic polyhedra, the closest to a sphere is the icosahedron. The icosahedron geodesic dome has the smallest variance in edge length, uniform neighbours (all nodes having 6 neighbours except the 12 from the original icosahedron). Therefore, as the icosahedron geodesic dome is the closest to a sphere, has uniform distances and a uniform number of neighbours it is the most suitable lattice shape for a spherical SOM [3].

2.4.1 Data structure of the geodesic dome

Data structures for SOMs need to have "fast vertex indexing," due to many of the neighbourhood searches required in the training process. Wu and Takatsuka's article [9] states 3 methods for creating a data structure for geodesic domes utilised for SOMs. The two commonly implemented methods, the first being to create the adjacency matrix for all the vertices. The second is to use an array containing all vertices and have each element of that array contain pointers to its neighbours. The first method has drawbacks in space efficiency taking $O(n^2)$, where n is the number of grids, due to its use of adjacency matrices. The second method uses pointers instead thus does not have the same issue; however, greater care is needed due to many pointers. Moreover, both methods have issues with further tessellation.

The third method is Wu and Takatsukas solution to the problems of the other two methods, reducing the size complexity to O(n), not relying on many pointers, and allowing for further tessellation on the sphere [12]. For the third method, the dome is unwrapped into a 2D lattice with the axes U and V as shown in figure 5. The lattice is then skewed such that U and V are orthogonal. Thus, square grids are created by joining the two triangular grids and removing the centre diagonal line as seen in figure 6 and the full transformed matrix in figure 7. As the

lattice is formed from unwrapping a sphere all vertices around the perimeter of the matrix will be duplicated (e.g., A, A'', …, A'''' from figure 5 are the same point) these points are stored in the data structure to ensure that they are maintained. There are only a small number of duplicates thus the size complexity is not affected and remains at O(n) [11].

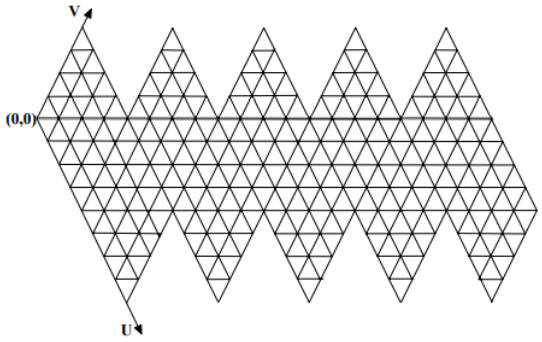


Figure 5

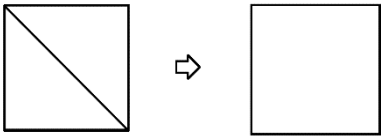


Figure 6

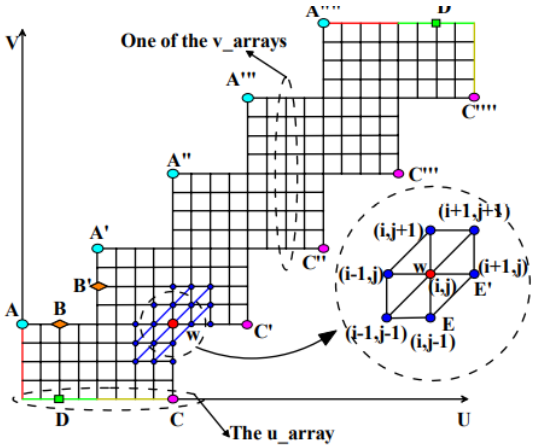


Figure 7

For this thesis, a simplified data structure will be utilised as performing the SOM training process and thus neighbourhood searches are not within the scope. However, it is recommended when implementing a 2D projection and interactive visualisation of a spherical SOM to use the third method proposed by Wu and Takatsuka as it solves errors from the other two implementations, reducing the size complexity to O(n), not relying on many pointers, and allowing for further tessellation of the sphere.

3. Methodology

3.1. Program Architecture

The programming language that was chosen for implementation was Python. This language was selected due to its Geodesic Dome libraries and its numerous visualisation methods. A web framework is utilised for storage of dome coordinates and visualisation. Python has two main web frameworks Django and Flask. The difference between the benefits and drawbacks of these frameworks is negligible. Django was chosen due to familiarity and experience in the framework. The visualisation is done through HTML and JavaScript, with the aid of D3.js. For the database SQLite is used as it is light weight and barebones system, which is all that is required for this project. A data base is used to store the starting vertices of both the Geodesic dome and its projection.

3.2. Geodesic Dome

As stated in Section 2.4, the visualisation of the 3D SOM can be achieved through a Geodesic dome. This application will thus generate a geodesic dome which will act as a representation of the 3D SOM. The Application will Display all visible edges and hide edges that are covered by the dome faces in front of them.

3.2.1. Geodesic Dome Structure

The co-ordinates (x, y, z) of all vertices on the Geodesic dome must be stored so that the dome can be drawn. Furthermore, the triangle vertices of dome must be stored (the vertices that are connected forming triangles), which is done thorough storing the indexes of all the co-ordinates of the vertices of the triangle in an array. These features have all been implemented by the python library geodome 1.0.0.

3.2.2. Geodesic Dome Implementation

The geodesic dome, its coordinates and triangles will be stored in the database. There are three models in the database: Geodome, Triangle and Coordinate. The Triangle and the coordinate models both contain the foreign keys of the Geodome (the id(Primary key) of the Geodesic dome), and have a one-to-many relationships, in which there is one Geodome to many triangles or coordinates. The structure can be seen below:

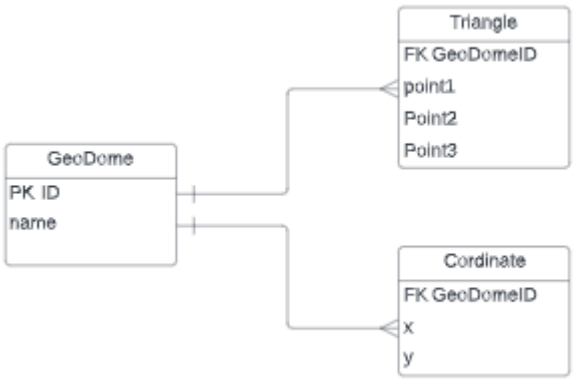


Figure 8

To obtain the Coordinates for the Geodesic dome the python library geodome 1.0.0. is utilised. To create the dome, function GeodesicDome(freq) is called, and its output is stored in a variable named dome, where freq is the frequency defined in Section 2.1.2. To get the coordinates of that dome, dome.get() is called this gets an array of coordinates [x, y, z] for

each vertex that in the geodesic dome. To get the triangles, dome.triangle() is called which returns an array of the indices for each of the three points of the triangles. These indices correspond to an index of a vertex in the geodesic dome array. The arrays of the coordinates and triangles are then looped through and stored in the database.

```
# Get the frequency of the dome (the amount of tessellation)
1.  freq = int(post.get("freq"))

# Create the dome and get an array of coordinates and the triangles of the dome
2.  dome = GeodesicDome(freq)
3.  domeOrds = dome.get_vertices()
4.  domeTri = dome.get_triangles()

# Get a name to give the dome
5.  newName = post.get("name")

# Create the SQL dome object
6.  dome = GeoDome(name= newName)
7.  dome.save()

8. ordArr = []
9. triArr = []


#Loop through array of cords create SQL objects and post all of them to the DB
10.  for i in range(len(domeOrds)):
11. newCoOrd = CoOrdDome( geoDome = dome, x = domeOrds[i][0], y
             = domeOrds[i][1], z = domeOrds[i][2])
12. CoOrdDome.objects.bulk_create(ordArr)

13. for i in range(len(domeTri)):
14.    newTri  =  Triangle(geoDome  =  dome,  point1  =  domeTri[i][0],  point2
   = domeTri[i][1], point3 = domeTri[i][2])
15. Triangle.objects.bulk_create(triArr)
```

3.2.3. Triangle Implementation

The Dome is a 3D object thus when being displayed edges and faces that are behind other edges and faces need to be hidden. This can be seen in the results figure 13 which compares dome with edges hidden and one where all are shown. This process is outlined in section 3.4. hidden surface removal

3.3. 2D projection

3.3.1. Vertices

The vertices of the dome need to be converted to the 2D for the projection. The Literature review (section 2) addresses several methods for the conversion of the 3D vertex to 2D. Two algorithms are addressed in section 2: Lambert Azimuthal Projection and Wagner projection. The program utilises Wanger's Transform algorithm as both algorithms have similar scale distortion however, Wagner's Transformation allows for the creation of a pseudo-cylindrical projection created from the average of experts' suggestions. This allows for a projection that is both appealing graphically and maintains scale distortion. The general formula for Wanger's method is:

$$x = \frac{k}{\sqrt{m \cdot n}} f_x(\theta, n\lambda)$$

$$y = \frac{1}{k\sqrt{m \cdot n}} f_y(\theta, n\lambda)$$

Where, ϕ and λ are the latitude and longitude, ϕB and $\boldsymbol{\lambda}$B are the bounding parallel and the bounding meridian, sinθ = m sin ϕ, m = sinϕB, n = $\boldsymbol{\lambda}$B/$\boldsymbol{\pi}$. Moreover, to get the pseudo-cylindrical projection created from the average of experts (ϕB = 61.9, p = 2.03).

3.3.2. 2D projection Implementation

To Convert the 3D coordinates of the geodesic dome to their 2D projection we must be obtaining the longitude and latitude of the 3D coordinates. This can be done by obtaining the spherical coordinate which allows us to obtain their longitude, latitude, and radius of a coordinate. The function used to convert the Cartesian co-ordinate to Spherical co-ordinates is the following:

```
1. def sphericalCordConvert(x,y,z):
2.     spherCodord = []
```

```
        radius =  sqrt((x*x) + (y*y) + (z*z))
        longitude = math.atan2(y,x)
        latitude = math.atan2(z,sqrt(x*x+y*y).real)


        spherCodord.append(radius)
        spherCodord.append(lng)
        spherCodord.append(lat)
    return spherCodord
```

Wagner's transformation is also converted into a function in the code which takes in an array of the new coordinates and outputs the new 2D coordinates for the projection. The following code is the transformation functions is the following:

```
def wagnerTransform(boundParrallel,p,long, lat):
    k = sqrt(2*p*math.sin(boundParrallel/2)/math.pi).real
    m = math.sin(boundParrallel)
    theta = math.asin(m * math.sin(lat) ).real


    x = ((k/sqrt(m)) * (    (long   * math.cos(theta))/(math.cos(theta/2))   )
          ).real
    y = (2/(k * sqrt(m))) * math.sin(theta/2)
    coOrd = [round(x.real,5),round(y.real,5)]
    return coOrd
```

The conversion will be occurring on the server side in the views.py file, as the initial coordinates of the 2D projection will be stored in the database.

3.3.3. Edges Implementation

After obtaining all the vertices we can now display the edges. This is done thought looping through the array of triangles which gives us the indices of the vertices, which we can then draw as lines connecting them and forming triangles. However, as the triangle array is for a dome there will be triangles in the array that should not be shown. These are the triangles that connect the vertices on the edges of the projection which can be seen in the results figure 14 .This process is outlined in section 3.4. hidden surface removal

3.4. hidden surface removal

hidden surface removal is a process in computer graphics in which surfaces that are not meant to be seen are identified to be hidden. For this program hidden surface removal needs to be applied on both the dome and its projection. For the dome faces that are behind others

should be hidden and for the projections lines that are on the perimeter that connect to other lines on the perimeter that are not adjacent to them should be hidden.

To do this we identify the direction that the triangle is facing. As the vertices in the triangle are stored in a clockwise direction, we can use the normal of the vectors to determine what direction the triangle is facing as seen in figure 9. We do this by getting cross product of the vectors if the normal is positive that means that the triangle if facing towards us and is in the front thus should be shown(depicted by the red in figure 9). If the normal is negative the triangle is behind us and should not be shown (depicted by the blue in figure 9). This process is Also the same for the projection.

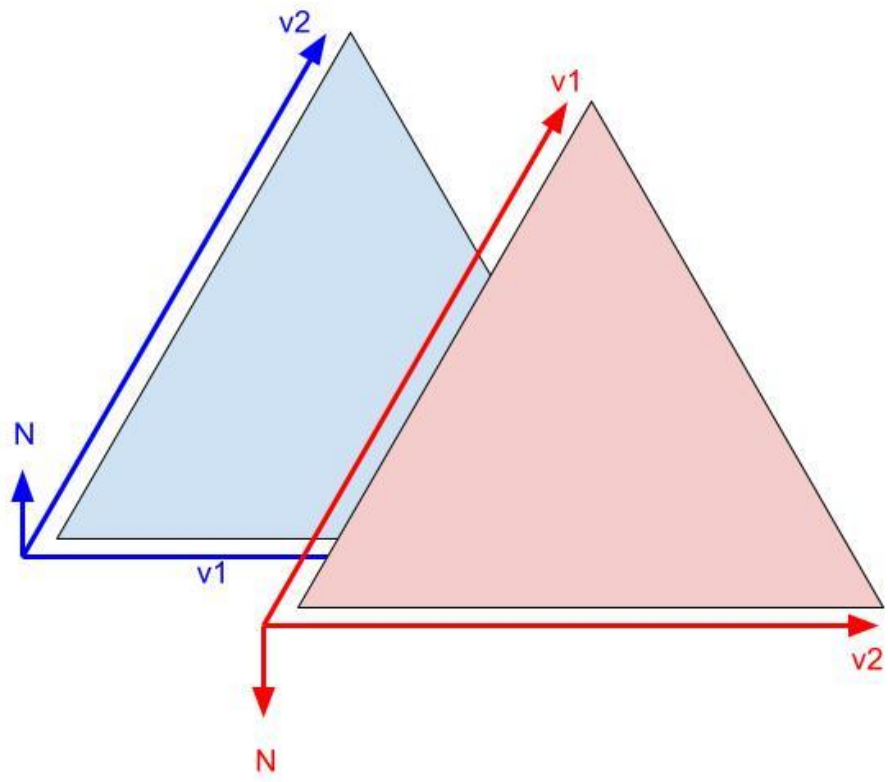


Figure 9

The process can be seen in the code extract:

```
1.    var vector1 = math.subtract(math.matrix(ords[p1]) , math.matrix(ords[p2]));
2.    var vector2 = math.subtract(math.matrix(ords[p1]) , math.matrix(ords[p3]));
3.    var normal = math.det([vector1, vector2]);
```

```
    normal = math.dot([normal, [0,0,1]]);
    if( normal > 0){
         projLines(ords[p1][0], ords[p1][1], ords[p2][0], ords[p2][1], "visible", id + "_0", svg);
         projLines(ords[p2][0], ords[p2][1], ords[p3][0], ords[p3][1], "visible", id + "_1", svg);
         projLines(ords[p3][0], ords[p3][1], ords[p1][0], ords[p1][1], "visible", id + "_2", svg);
    }else{
         projLines(ords[p1][0], ords[p1][1], ords[p2][0], ords[p2][1], "hidden", id + "_0", svg);
         projLines(ords[p2][0], ords[p2][1], ords[p3][0], ords[p3][1], "hidden", id + "_1", svg);
         projLines(ords[p3][0], ords[p3][1], ords[p1][0], ords[p1][1], "hidden", id + "_2", svg);
         }
     }
```

3.5. Rotation

As the Geodesic dome is a 3D object all faces of the dome will not be visible at the same time. To aid in the readability we provide a 2D projection. However, the issue of faces not being not visible remains, as the relationship between the hidden faces of the geodesic dome and the 2d projection cannot be drawn. To solve the rotation of the sphere can be implemented. Rotation of the 2D projection is implemented as it allows the user to interact with the 2d projection if the projection is zoomed in and not all the projection is visible.

3.5.1. Mouse movement implementation

For rotation, we first need to get the mouse input. To do this we get the x and y coordinates of the mouse when it is clicked and set it to null when it Is released. Then we get the mouse movement when the x and y are not null (mouse is clicked) and update the x and y to follow the movement of the mouse. The following code outlines this process.

```
function rotation(dome_svg, proj_svg,  triangles){
    var startX = null;
    var startY = null;
    document.getElementById("dome_display").onmousedown = function (e) {
        startX = e.clientX;
        startY = e.clientY;
    };
    document.getElementById("dome_display").onmouseup = function (e) {
        startX = null;
        startY = null;
    };
    document.getElementById("dome_display").onmousemove = function (e) {
```

```
13.         if (startX == null) {
14.             return;
15.         }
16.         if (startY == null) {
17.             return;
18.         }
```

In the `onmousemove` function we also need to update the coordinates with the size of the drag. This is done through creating a distance vector from the origin of the click to new position of the mouse. The length of this vector is the distance that the mouse has moved. The angle of the rotation is also calculated this done though getting the cross product of the distance vector and the axis which for our case is [0,0, -1]. Theta is also calculated, theta is the amount that the sphere will rotate and is calculated by dividing the distance by some scalar such that it is between 0 and 2 PI, this process can be seen in the following code:

```
 //distance vector
19.         var dx = e.clientX - startX;
20.         var dy = e.clientY - startY;
21.         var vector = math.matrix([dx, -dy, 0]);
22.         var dist = parseFloat(Math.sqrt(Math.pow(dy, 2) + Math.pow(dx, 2)));
//angle
23.         var axis = math.matrix([0, 0, -1]);
24.         var rotationV= (math.cross(vector, axis));
//between 0 and 2pi
25.         Var scalar = 200;
26.         var theta = Math.min(Math.max(parseFloat(dist / scalar), 0), 2 * Math.PI);
```

3.5.2. Geodesic dome rotation implementation

The dome rotation is done through updating all the coordinates of the dome by multiplying the coordinates by the rotation matrix. The rotation matrix:

$$R = \begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}.$$

Where, u is the coordinates of the rotation vector and θ is theta the distance of rotation, both calculated in the previous section (3.5.1.1 Mouse movement implementation). After those values are substituted into the rotation matrix the matrix is then multiplied by the original coordinates of the geodesic dome. The result of that multiplication will be the updated coordinates of the dome. This process can be seen in the following function:

```
1.  function rotationMatrix(theta, coord, rotationV) {
```

```
    var x = math.subset(pVect, math.index(0));
    var y = math.subset(pVect, math.index(1));
    var z = math.subset(pVect, math.index(2));

    var sum = Math.sqrt(x ** 2 + y ** 2 + z ** 2);
    if(sum === 0){
        x = 0
        y = 0
        z = 0
    }else{
        x = math.subset(rotationV, math.index(0)) / sum;
        y = math.subset(rotationV, math.index(1)) / sum;
        z = math.subset(rotationV, math.index(2)) / sum;
    }

    var r = math.matrix([
       [
            Math.cos(theta) + Math.pow(x, 2) * (1 - Math.cos(theta)),
            x * y * (1 - Math.cos(theta)) - z * Math.sin(theta),
            x * z * (1 - Math.cos(theta)) + y * Math.sin(theta)
        ], [
            y * x * (1 - Math.cos(theta)) + z * Math.sin(theta),
            Math.cos(theta) + Math.pow(y, 2) * (1 - Math.cos(theta))
            y * z * (1 - Math.cos(theta)) - x * Math.sin(theta)
        ], [
            z * x * (1 - Math.cos(theta)) - y * Math.sin(theta)
            z * y * (1 - Math.cos(theta)) + x * Math.sin(theta)
            Math.cos(theta) + Math.pow(z, 2) * (1 - Math.cos(theta))
        ]
    ]);
    var res = math.multiply(r, coord);
    return res;
}
```

All Coordinates of the geodesic dome go through this process and the values of dome on the front end are updated and the dome rotates on screen.

3.5.3. Projection rotation implementation

For the rotation of the 2d projection the updated coordinates are converted to spherical coordinates so they can be read into Wagner's transformation algorithm. After going through Wagner's transformation result is the 2D coordinates for the projection. The vertices of the projection are then updated. After this the lines that are needed to be drawn are calculated and the lines connecting the points are updated.

3.6. Selecting vertices

To aid in the user's interoperation of the data clicking on a vertex on the dome will increase the size of that vertex as well as the corresponding vertex in the other dome. This process can be seen in figure 10. This is simply done by adding an onclick to the attribute of the vertex of the dome, which then updates its size. Then when the projection is next updated, it will copy the vertex size of the dome

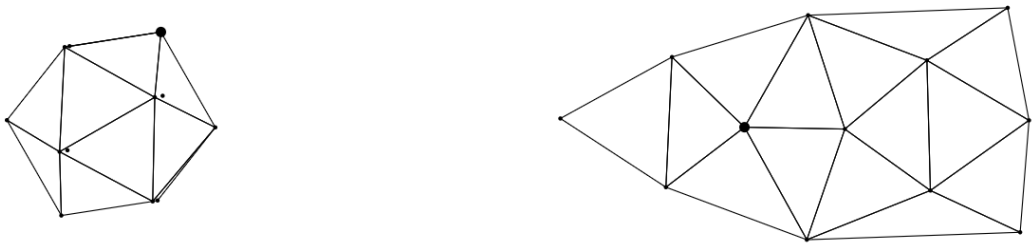


Figure 10

4. Simulation Results

4.1. Display

The points of the dome and projection are as expected as seen in figure 11. From this the lines are drawn.

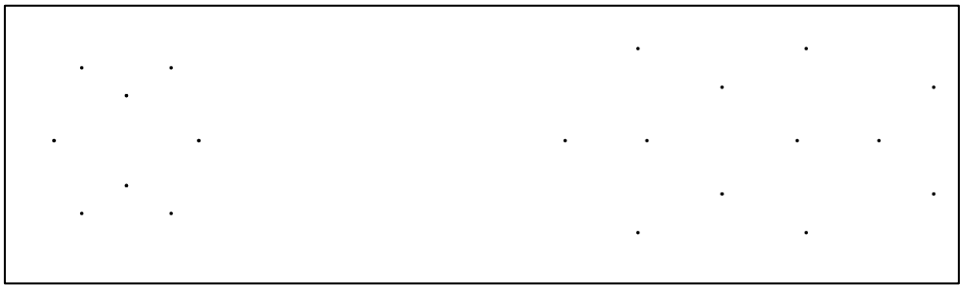


Figure 11

Figure 12 shows the displaying of geodesic domes at different frequencies. Starting with 0 at the top and ending with 3 at the bottom. All projections and domes appear as expected and as such there is no error in the display method for either the dome or projection.

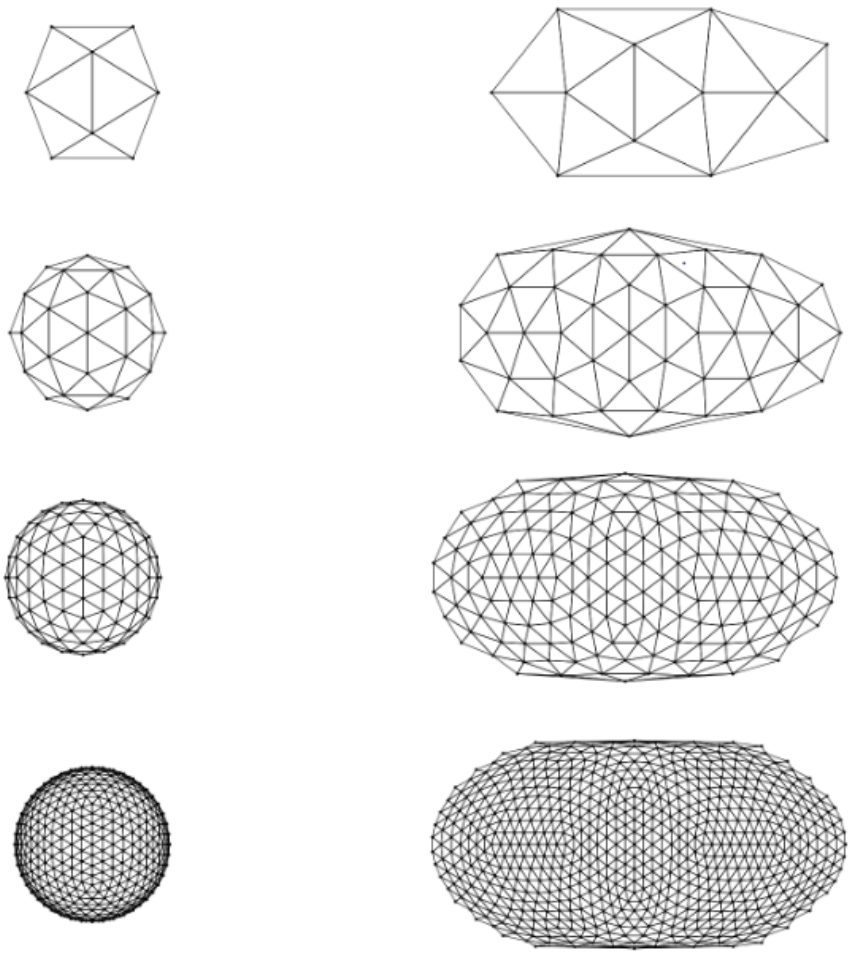


Figure 12

Figure 13 shows lines of the geodesic dome, the image on the right shows how the dome would look like if the edges were not hidden. The image on the right of figure 13 shows the current output of the application. As shown in the figure the dome on the right hides all the lines that would be behind the other faces on the dome and thus is working as expected.

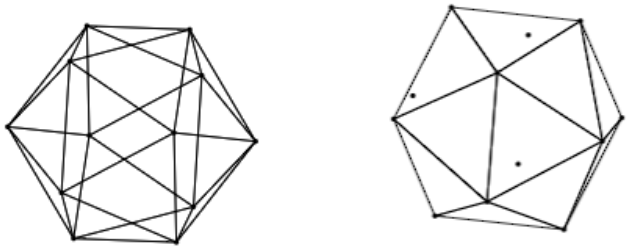


Figure 13

Figure 14 shows the lines dawn for the projection. The image on the right shows all triangles present in the Triangle array. The projection on the right of figure 11 shows the current output of the projection. As shown in the figure the projection on the right hides all the lines that would be crossing the projection and thus is working as expected.

Figure 14

4.2. Rotation

For rotation the intended functionality is that the user will. Clicking, holding, then moving the mouse within the div (content division) highlighted in figure 15, will cause the dome to rotate, and the 2d projection to shift. For the Dome mouse drags to the left will rotate the dome to the left and the same for the other corresponding directions. The Purpose of this rotation is firstly to allow the user to view all the data that would be displayed on the dome. Secondly, to allow the user to allow the user to understand the connection between the Dome and projection.

Rotation using the mouse is working as expected. As the div contains both the dome and the map dragging the mouse will on top of the projection will also rotate the dome and map. However, there is an existing bug, where if the user clicks within the div and release outside the div, the application will not be notified of the release. This results in the dome rotating with the mouse whilst the left click is not being held. However, Once the user clicks back onto the div the Application will begin to perform normally.

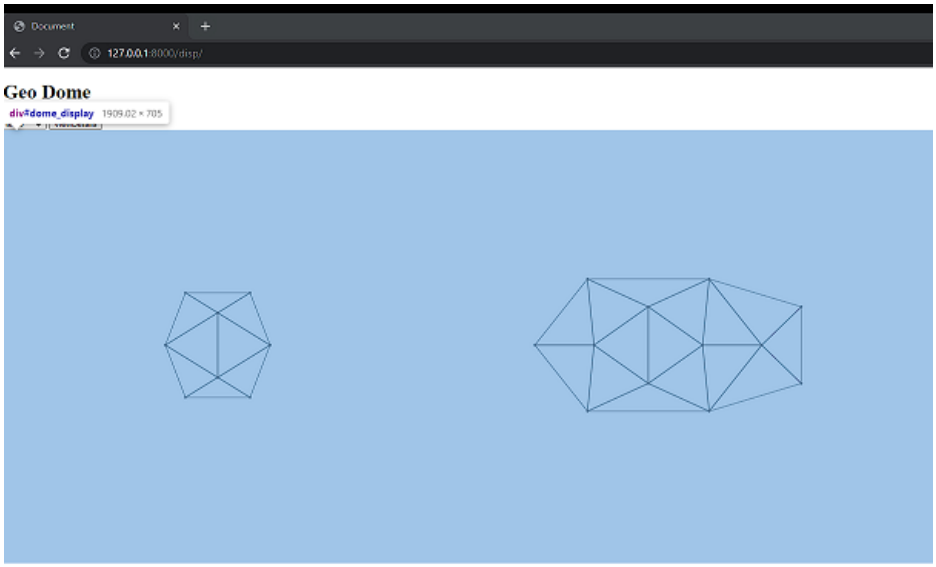


Figure 15

The rotation works Smoothly without any noticeable lag or delay for domes of with inputted frequencies of 0 and 1. With domes of tessellation 2 rotation is working however there is stuttering and lagging during the rotation. Domes of tessellation 3 also still rotate, however there is significant lagging and shuttering to the point in which the application would not be good to use. At Tessellation 4 the Application is so slow to the point in which users cannot tell whether the dome follows the drag, and the application should not be used.

4.2.1. Rotation movement analysis

To test the expected rotation of the dome a single triangle in the dome and the projection was highlighted and the movement was tracked. The horizontal and vertical movement was tested. To test the horizontal movement a drag to the right was observed. The drag continued one full 360-degree rotation occurred.

The result of this tracking can be seen in figure 16. The movement of the triangle on the dome is intuitive(a drag up rotates upward), whilst the movement of the triangle on the projection was not. On the projection when dragging up the triangle moves in a circle, in a clockwise direction (anti clockwise when down).

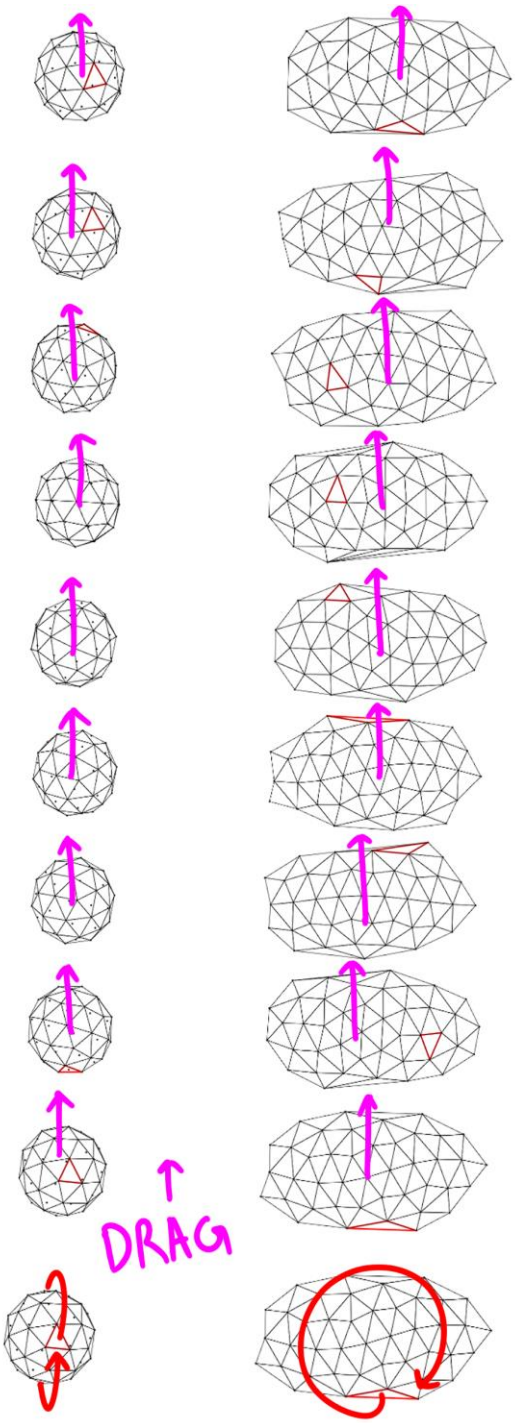


Figure 16

Horizontal movement was also recorded, the result in figure 17. Again, the dome movement is as expected the triangle rotates with intuitive movement (a drag to the left rotates the dome to the left). However, For the projection, there are two points of rotation on the left and right sides of the projection. When dragging to the right there the left points rotate in an

anti-clockwise direction and the right in a clockwise direction. (Clockwise and anti-clockwise respectively for drags to the left).

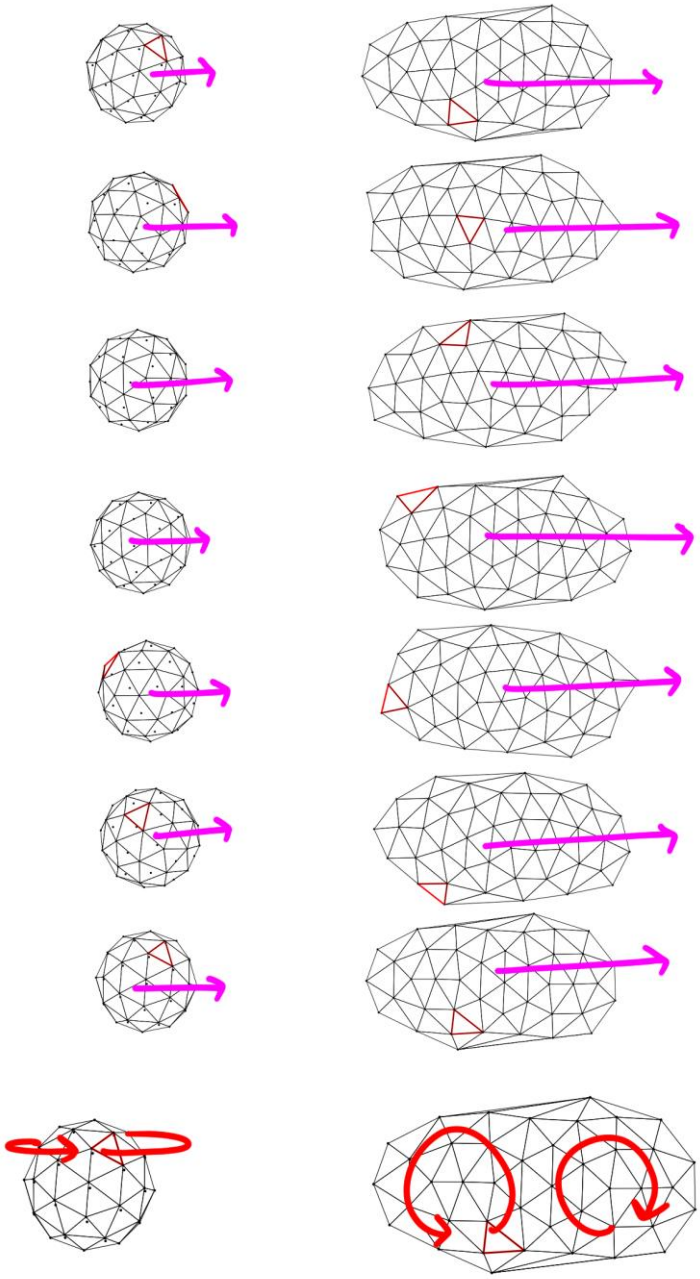


Figure 17

There are no technical problems with the dome and projection rotation although a better rotation method could be utilised. Currently the rotation method is based on the rotation of

the dome, The drag updates the co-ordinates of the domes co-ordinates, and those points are reprojected updating the 2D projection. This causes the projection rotations to be counter intuitive as a drag to the left does not move the triangle to the left, right does not move the dome to the right. Instead, the rotation follows the domes rotation, so the triangles move in circular motions.

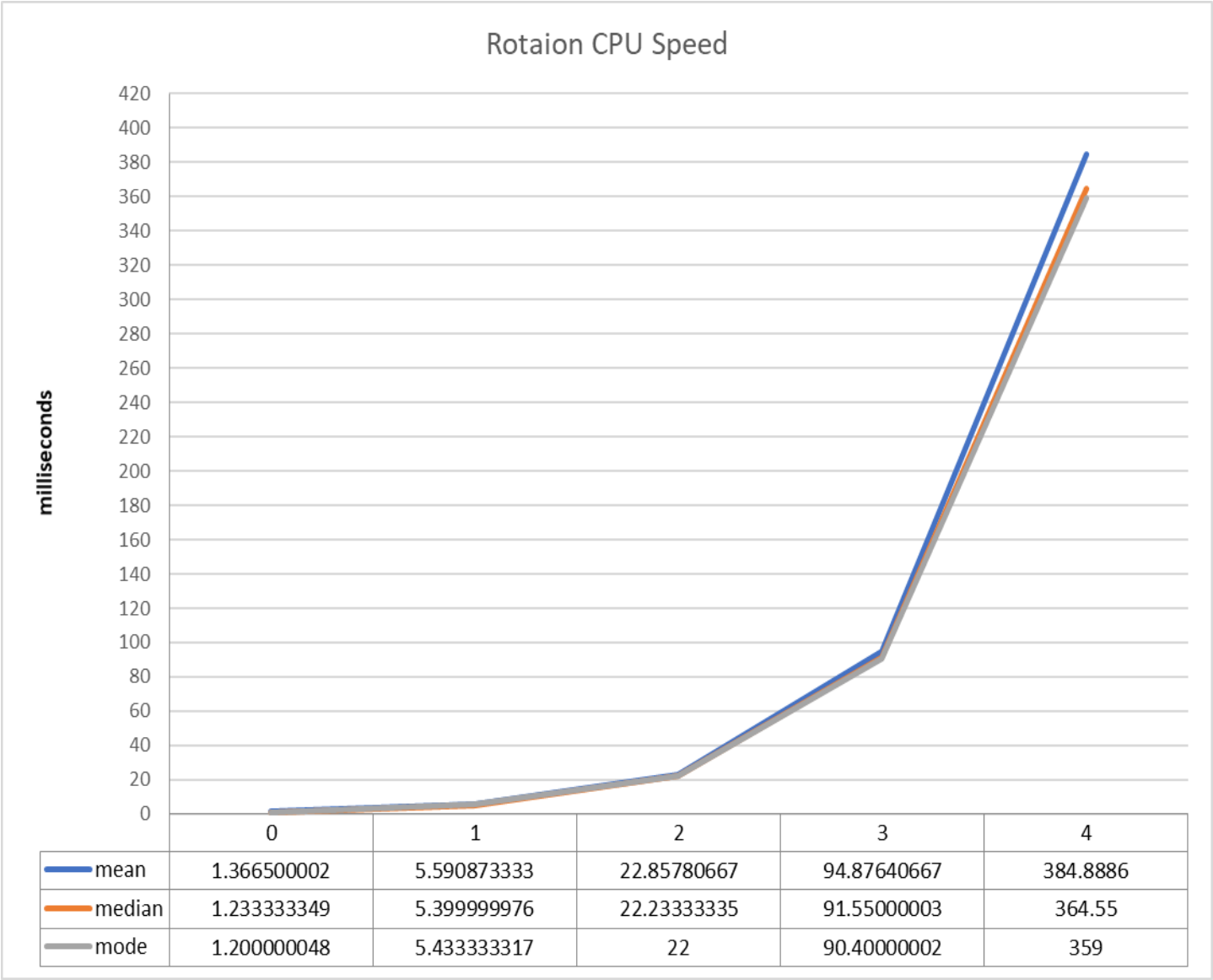
As previously stated, the purpose of the rotation is to allow all data to be visible on the dome, which the dome rotation succeeds in achieving. However, the second goal to show the correlation between the dome and the 2d projection was not full satisfied. When looking at the sphere we can understand the projections rotation. However, looking at the projection the rotation of the sphere is not clear. An additional method needs to be implemented to have the rotation on the map rotate the sphere, such that when the user drags to the right the triangle also moves to the right and the sphere would then follow.

4.2.2. Rotation Speed testing

Dome frequency	length	mean	median	mode
1	500	1.443000002	1.299999952	1.200000048
1	1000	1.363800004	1.200000048	1.200000048
1	5000	1.2927	1.200000048	1.200000048
1	**avg**	**1.366500002**	**1.233333349**	**1.200000048**
2	500	5.649599998	5.399999976	5.399999976
2	1000	5.5932	5.399999976	5.399999976
2	5000	5.52982	5.399999976	5.5
2	**avg**	**5.590873333**	**5.399999976**	**5.433333317**
3	500	22.7444	22	22
3	1000	22.8464	22.20000005	22
3	5000	22.98262	22.5	22
3	**avg**	**22.85780667**	**22.23333335**	**22**
4	500	94.2188	91.30000007	90.60000002
4	1000	94.4977	91.45000005	90.60000002
4	5000	95.91272	91.89999998	90
4	**avg**	**94.87640667**	**91.55000003**	**90.40000002**
5	**500**	**384.8886**	**364.55**	**359**

Table 1

As the speed of rotation was identified as an issue of the application, testing was done to understand and solve the problem. The times were taken when the function began and ended, from those times the runtime of the function was calculated. From that the 3 averages calculated seen in table 1 and figure 18. As expected, there is exponential increase in the time it takes to run the application, this reflects the sudden drops in performance in the application when we increase the frequency of the dome.



	0	1	2	3	4
mean	1.366500002	5.590873333	22.85780667	94.87640667	384.8886
median	1.233333349	5.399999976	22.23333335	91.55000003	364.55
mode	1.200000048	5.433333317	22	90.40000002	359

Figure 18

A hypothesis for the slowing of the rotation function is that the number of elements drawn is too much for the program to render. To support this, we can look at the number of objects

drawn for each dome of varying frequency. Table 2 shows the number of objects needed to draw a dome.

Dome frequency	# of Triangles	# of vertecies	# of edges
0	20	12	60
1	80	42	240
2	320	162	960
3	1280	642	3840
4	5120	2562	15360
5	10242	10242	61440
6	81920	40962	245760

Table 2

When a triangle is drawn and updated all vertices and edges are updated, even the ones that are not visible. From this we can see the expansional growth in the number of objects for each dome. The Program starts struggling when the dome frequency is 2. At a frequency of three there are 1122(162+ 960 = 1122) objects drawn. Compared to frequency 1 are only 482 (42 + 240 = 482) objects drawn. Thus, methods need to be utilised to reduce the amount that the application is rendering.

Firstly, we could remove the drawing of vertices, drawing of vertices is not needed when all lines are rendered correctly. The points in which the lines meet is good enough at showing the vertices. However, there will still be calculations for each vertex as they are used to find the beginning and end of each line. Moreover, as seen in the above table vertices make up the smallest number of objects in the geodesic dome. Therefore, this change would not be suitable.

Moreover, another possibility would be to store triangles instead of lines. If Triangles draw and stored instead of lines this could improve the speed. Looking at the above table the number of edges is by far the largest number of objects drawn. If we were to use triangles instead of lines, we would have 3 times less the number of objects (not including vertices). For example, for frequency 2 there would be 482 (320 + 162) instead of 1122 (960 +162) objects this is a difference of 640 objects more than 2 times less than the original. As the

number of objects increase exponentially this will also reduce the number of objects exponentially. However, this is not possible with the current architecture of the application as D3js only draws regular triangles. New libraries will need to implement.

Another hypothesis is that the logic used for rotation is inefficient. To test this the Time complexity of the function needs to be calculated though Big O notation. Figure 19 shows the Time complexity of the function in big O notation. The function as a complexity of O(n).

```
1. Function rotation (){ O(n) + O(n) + O(n) + O(1) + O(1) = O(n)
    1. document.getElementById("dome_display").onmousedown = function () O(1)
    2. document.getElementById("dome_display").onmouseup = function ()O(1)
    3. document.getElementById("dome_display").onmousemove = function () { O(n)
        1. dome_svg.selectAll("circle").datum(function () { O(n) * (O(1) + O(1)) = O(n)
            1. var spherCodord = sphericalCordConvert(x, y, z); O(1)
            2. var newProjCoOrd = wagnerTransform(); O(1)
    4. proj_svg.selectAll("circle").datum(function () O(n)
    5. for(let i = 0; i < triangles.length ; i ++) O(n) * (O(1) + O(1)) = O(n)
        1. visibleTriangles() ; O(1)
        2. visibleTriangles() ; O(1)
```

Figure 19

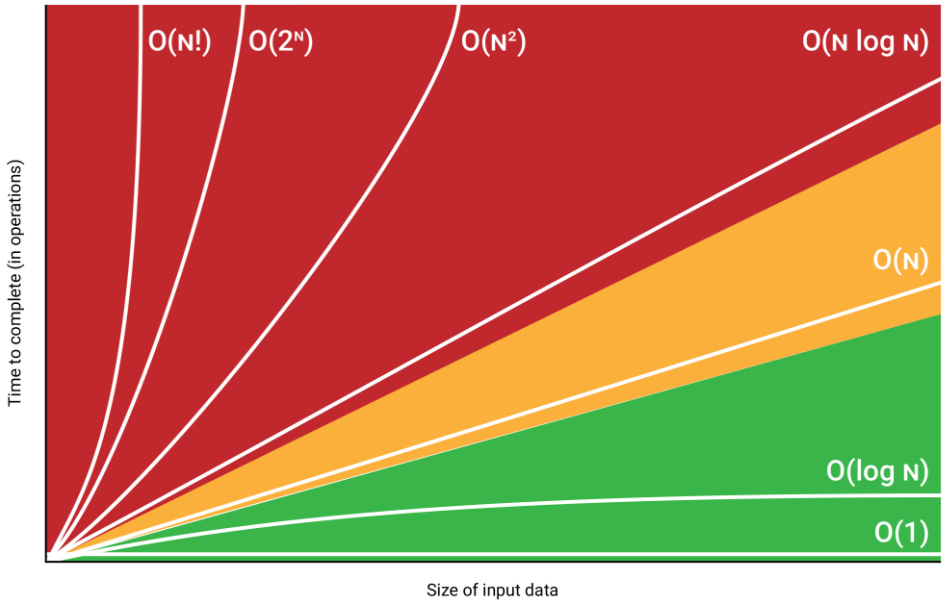


Figure 20

Figure 20 above shows time to complete a big O task with increasing number of inputs (n). O(n) is linear with linearly increasing n, therefore the time to complete an operation O (n) will follow the number of inputs directly. The largest set of inputs used within the rotate function is the number of Objects in the Geodesic dome. Using table 2 we can graph the number of

objects in the dome, as shown seen in figure 21. The figure shows an exponential growth of objects as we increase the frequency of the dome. Therefore, the time it takes to complete the rotation function will also increase exponentially with the frequency of the dome. The rotation function must update all objects in the dome this mean all n vertices must accessed. Thus, the time complexity is caped at n.

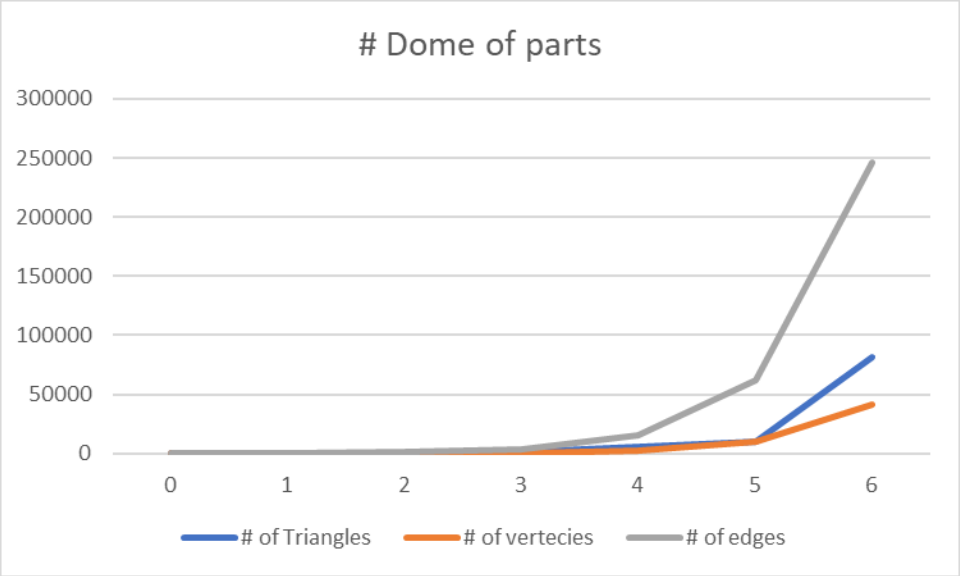


Figure 21

A solution to this would be multi-threading. Multi-threading allows programs to run in parallel meaning multiple processes can be performed at the same time thus improving the performance and speed of program. Threading is meant to be implanted in algorithms that utilise independent entities[14]. This is suitable for the function as there are sections of code that are choke points. In which all objects of the dome must be looped through. Such as the selecting and updating all circles and the drawing triangles loop. However, JavaScript is not made for multi-threading, having little threading capability. Thus, threading would not be suited for a JavaScript Implementation If this program were to be repeated a framework that supports threading such as java would be better suited.

4.2.3. Projection Speed testing

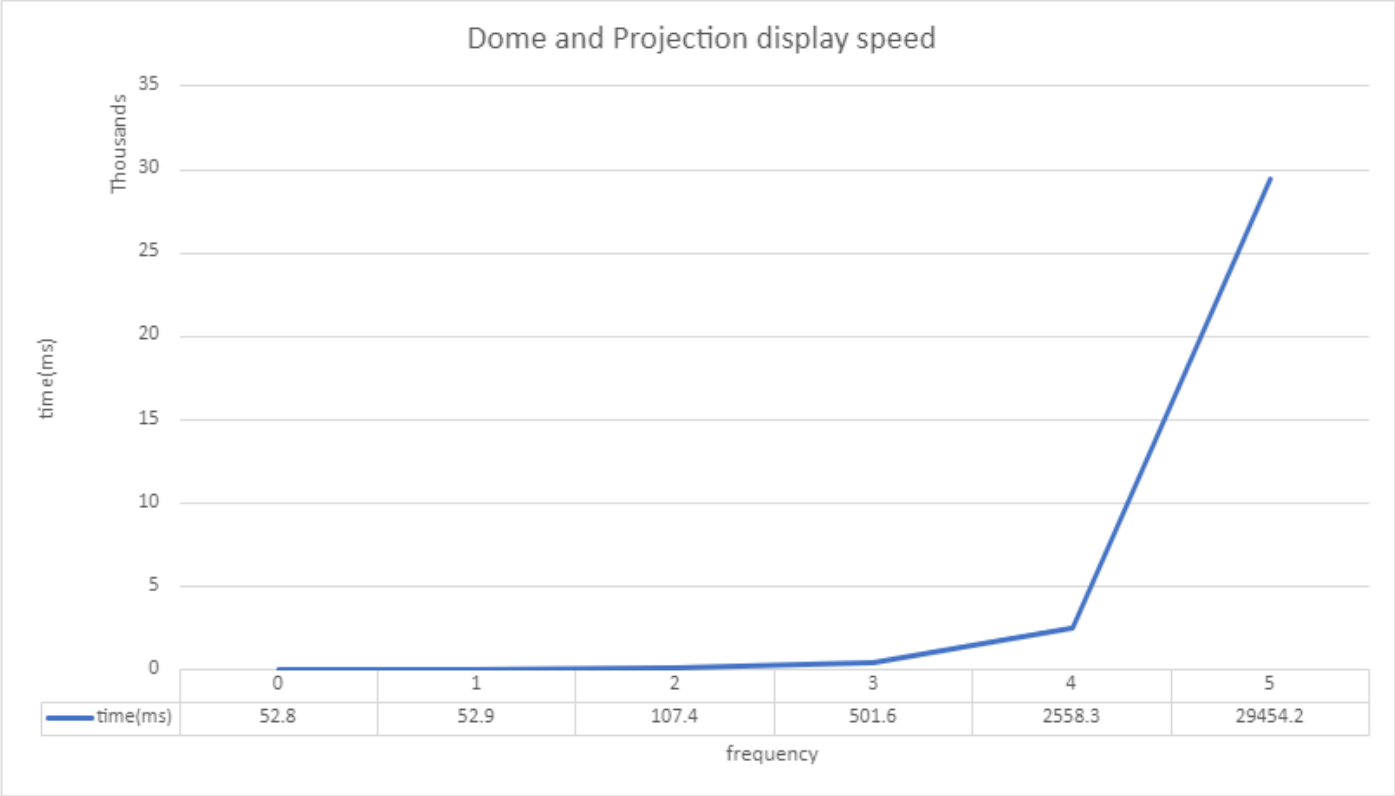


Figure 22

The above figure displays the speed it takes to fully display a dome and projection. The speed it takes to load the dome is less of an issue than rotation. This is as rotation requires frequent updates per mouse move whilst display is only called once. Although, the time it takes to display is not a large issue we can still observer it. Comparing figure 22 and figure 18 the graphs follow the same pattern. As the number of vertices increase time it takes to display.

4.2.3. Optimal Dome Frequency

From the previous data we can determine the viable dome frequencies for the application. To do so we must consider the framerate. Most applications will aim for 60FPS which would be 16.67ms per frame. However, the accepted floor for an applications FPS is around 30FPS, 33.33ms per frame[13]. From Figure 18 we can see that dome of frequency 0 - 2 are within that range (2 having an average of 22ms). Thus, the optimal acceptable frequencies of the domes for the application are 0 - 2 inclusive. We can also see that the load time for a frequency 2 dome would be acceptable with an average speed of 107.4ms

5. Conclusion

A self-organizing map (SOM) is an artificial neural network used to simplify high-dimensional data into low-dimensional data whilst maintaining the data's topological structure, developed by Professor Teuvo Kohonen in the 1980s [1]. The purpose of dimensionality reduction is to aid in "data analysis, clustering problems, and visualization of high dimensional datasets" [2].

The structure of as SOM can vary in dimension, for example, a flat 2D plane grid or a 3D spherical grid can be used. The different dimensions have their advantages and disadvantages, a 2D SOM allows for greater readability however it reduces accuracy of the visualisation due to the problem known as the border effect, the simplest method to prevent the border effect is to use a 3D lattice [3].

The most straightforward way to create a 3D lattice is to connect boundaries (left connected to right or top connected to bottom), forming a torus. The torus lattice removes the border effect; however, it reduces the readability and area of each grid varies greatly. It is important for a SOM to have "equal geometrical treatment" as it can impact the accuracy off the visualisation. The preferred method for a 3D lattice is a spherical lattice, which has greater readability than a torus and more equal sized grids, whilst combatting the border effect [3].

However, a spherical visualisation is still harder to read and interpret than a 2D lattice. The curved surface of a sphere distorts distance, and as a sphere is a 3D shape all data cannot be viewed at once. This work proposed a solution to this issue by presenting a method to project the sphere into 2D and providing interactive interface.

The chosen method to project the sphere to 2D was the Wanger's Transform algorithm as the projection has low scale, angle, and area distortion. This means that the shapes and area of the grids will remain relatively consistent, which is essential for SOMs. Wagner's Transformation also allows for the creation of a pseudo-cylindrical projection created from the average of experts' suggestions outlined in section 2: Lambert Azimuthal Projection and Wagner projection. This allows for a projection that is both appealing graphically and maintains scale distortion.

Rotation was added as an interactive method , The Purpose of this rotation is firstly to allow the user to view all the data that would be displayed on the dome. Secondly, to allow the user to allow the user to understand the connection between the Dome and projection. The first goal was done successfully however, rotation was not intuitive for the projection. The projection rotations are counter intuitive as a drag to the left does not move the triangle to the left, right does not move the dome to the right. Instead, the rotation follows the domes rotation, so the triangles move in circular motions. The option needs to be added for the rotation to occur about the projection, such that drags left move the triangles left and the same for the other directions. The other interactive method was click on vertices, clicking on vertices increased their size booth on the projection and dome, this allows for the easier tracking of data and connection of the dome and sphere to be done.

Through the performance analysis in Section 4, The limitation of the application was determined regarding geodesic dome frequency. It was Found that dome with a frequency 0-2 inclusive were suitable as their framerate were in comfortable levels for the human eye. Also recombination to use an approach that could allow for threading for future versions.

Possible extensions to the application would be the aforementioned additional rotation method that focuses on the projection. Another would be the adaptive zooming feature addressed in the literature review(section 2). This feature would allow the method of projection to change based on the size of the 2D projection. This would allow for better area preservation as different projections have better area distortion at different sizes. This Could be done by changing the variables of the Wagner Transformation based on the zoom level as well as adding other projection algorithm such as Aitoff's transformation.

6. References

[1] M. Dubravko, "Brief Review of Self-Organizing Maps," Zagreb, 2017.

[2] R. Ponmalai and C. Kamath, "Self-Organizing Maps and Their Applications to Data Analysis," Lawerence Livermore National Laboratory, 2019.

[3] Y. Wu and M. Takatsuka, "Spherical Self-Organizing Map Using Efficient Indexed Geodesic Data Structure.," *Neural networks 19.6,* p. 900–910, 2006.

[4] K. Teuvo, "NeuralNetworks," *Essentials of the self-organizing map,* vol. 37, pp. 52-65, 2013.

[5] R. G. Brereton, "Self organising maps for visualising and modelling," *Chemistry Central journal,* vol. 6, no. 2, pp. 4-15, 2012.

[6] Jenny, B. Šavrič, Bojan, Arnold, Nicholas, Marston, Brooke, Preppernau and Charles, "A Guide to Selecting Map Projections for World and Hemisphere Maps," in *A Guide to Selecting Map Projections for World and Hemisphere Maps*, 2017, pp. 213-228. ("Choosing a Map Projection | SpringerLink") ("Choosing a Map Projection | SpringerLink")

[7] D. R. Steinwand, J. A. Hutchinson, and J. P. Snyder, "Map Projections for Global and Gontinental Data," *Photogrammetric Engineering & Remote Sensing,,* vol. 61, no. 12, pp. 1487-1.49, 1995.

[8] B. Šavrič and B. Jenny, "A new pseudocylindrical equal-area projection for," *International Journal of Geographical Information,* vol. 28, no. 12, pp. 2373-2389, 2014.

[9] D. Aitoff, "Projections des cartes géographiques," *Atlas de géographie moderne,* 1889.

[10] K. Wagner, "Die unechten Zylinderprojektionen: Ihre Anwendung und ihre Bedeutung für," *Mathematisch-Naturwissenschaftliche Fakultät,* 1931.

[11] F. Canters, Small-scale map projection design, London: New York: Taylor & Francis, 2002.

[12] Y. Wu and M. Takatsuka, "the Geodesic Self-Organizing Map and Its Error Analysis.," in *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38*, vol. 5, Australian Computer Society, Inc., 2005, pp. 343-351.

[13] Apple Inc, "Frame Rate (iOS and tvOS)," Apple Inc, 2018 . [Online]. Available: https://developer.apple.com/library/archive/documentation/3DDrawing/Conceptual/MTLBestPracticesGuide/FrameRate.html#:~:text=Most%20apps%20target%20a%20frame,frame%20rate%20to%20avoid%20jitter.&text=The%20minimum%20acceptable%20frame%20rate,time%20gaming%2. [Accessed 22 05 2022].

[14] IBM, "Benefits of threads," IBM, 20 01 2022. [Online]. Available: https://www.ibm.com/docs/en/aix/7.1?topic=programming-benefits-threads. [Accessed 4 05 2022].

Code: https://github.com/LouisPolicarpio/Interactive-visualisation-of-spherical-SOM