Submitted in part fulfilment for the degree of BSc

3D digital watermarking through application of spectral decomposition

David Kelly

2021-22

Supervisor: Adrian Bors

1 Executive Summary

1.1 Statement of ethics

The following report and project did not include the use of any human participants. No confidential data is at risk of being compromised or need for informed consent. No ethical consequences could arise. All software used is appropriately licensed.

1.2 Project Goals

- A review of literature regarding 3D watermarking
- Implementation of 3D watermarking algorithm
- Testing of algorithm and comparison of some currently proposed solutions.

1.3 Project Lifecycle.

The lifecycle which was used for this piece of work was fairly straightforward. The projects requirements and chosen algorithm were chosen through research using relevant literature. This chosen algorithm was then implemented according to the before mentioned requirements. Then a cyclical process of testing followed by modification of the program followed to increase its effectiveness and hit more requirements. Finally, the results of the testing were assessed, and conclusions were made on the effectiveness of the algorithm.

1.4 Document Overview

3D digital watermarking is the act of embedding data into 3D models while minimizing visible distortion to the object. This data must be able to be extracted out the object successfully and be robust to attacks.

With the ever increase of 3D digital media digital watermarking will become a much bigger field once people look for a solution to the problems it solves.

This project looks at creating an 3D digital watermarking algorithm which will meet the criteria above. The different ways this could be achieved was gained through extensive research and analysis. This resulted in the choice of an algorithm that uses spectral decomposition to achieve these aims.

The algorithm was tested against said criteria and underwent attacks to challenge its robustness. For the most part the algorithm meets the stated criteria and is able to embed information into 3D objects without much visible distortion.

2 Introduction

2.1 Watermarking introduction

Watermarking is the act of imbedding information within content such that the content's integrity is maintained. The origins of this technique date back to 13th century Italy where Italian paper manufacturers would use watermarking to identify paper that they had produced. The method of watermarking they used was one of adding patterns to their paper moulds through use of a thin wire. Paper created using their moulds would be thinner in the areas where the wire was and thus the patterns could be seen imprinted onto the paper. By the 18th century watermarking was a commonplace occurrence being employed in numerous forms to verify authenticity of goods and stop counterfeiting.

Digital watermarking is a digital evolution of this medieval concept. Just as the Italian paper manufacturers of old needed to prove the authenticity of their physical paper we now need to verify the authenticity of our digital goods moving through the information age. The research into digital watermarking started more than forty years ago mainly focusing on how to embed information into images and audio files. In recent years however research has mainly focussed on embedding information into three-dimensional media. This is due to the prevalence of 3D objects on the web thanks to widely available software which anyone can learn and use to create these objects. With this rise in 3D creation digital watermarking is needed by artists to protect their work.

3D watermarking is a relatively new area of research when compared to 2D watermarking. The techniques used in 2D watermarking[1] cannot be easily adapted to be used with 3D objects as 3D objects are more complex and can be represented numerous ways. However, the overall concept and aim is the same. The way watermarking is achieved is through a process of embedding and extracting. Firstly, the watermark and embedding medium are fed into an embedding algorithm which embeds the watermark into the medium by editing properties of the medium. In the case of an image this could be the colour of pixels and in 3D models it could be the movement of vertices. Extraction is essentially the reverse of this process. The watermarked medium is fed into an extraction algorithm potentially alongside the original unwatermarked medium and the watermark is extracted from the medium.

2.2 3D digital watermarking applications

In the modern age 3D media is extremely prevalent and thus 3D digital watermarking has many applications.

For example, video games have exploded in popularity in the past decade. Most modern videogames make heavy use of 3D meshes to create their virtual environments. Proof of ownership is extremely important to the companies which create these meshes as they require copyright protection for the things they have put so much time into creating. On top of this many individual creators sell their models online for a living so people can use them in their games. Protecting their creations from piracy is important as creating 3D models is their livelihood.

Another application of 3D digital watermarking is in 3D printing[2]. In the past decade 3D printing has become far more prevalent as 3D printers become cheaper and more available to the general public. 3D printing involves taking digital 3D models and printing them into physical creations. Many people are creating physical models and selling their creations through this. However, if someone has access to the object file they can easily print and sell these models themselves. Therefore, 3d digital watermarking can help these creators prove ownership of their 3d printed goods.

Tracking distribution channels is another application 3D watermarking would be useful for. Here watermarking would be used to identify the chain of intermediaries the 3D digital goods have passed through. At each stage a new unique digital watermark could be inserted into the mesh to identify the intermediary it has passed through. This would be useful to track the sale and reselling of 3D digital goods.

2.3 3D digital watermarking Terminology

2.3.1 Blind and Non-blind

A Non-blind watermarking algorithm is one which requires the original unwatermarked mesh in order to extract the watermark from the mesh. This original mesh is usually used to align the meshes and for comparison so that the data can be extracted correctly.

A blind algorithm is one which does not require the original mesh for extraction. Blind algorithms are desirable as they do not need the usage of things such as databases to get the unwatermarked meshes.

2.3.2 Robust and fragile

Robust algorithms are designed to embed watermarks with the intent of thwarting attacks made against the mesh. The aim is to have watermarks that cannot be broken or removed. These kinds of algorithms are used in things like copyright protection where you do not want the copyright label removed from the product

Fragile algorithms are designed to break very easily. The watermark is not to be used as a proof of ownership but more like an indicator to see whether the product has been tampered with or not.

2.4 Conclusion

In conclusion 3D digital watermarking is a fascinating upcoming new technology which will prove vital moving into the modern age filled with create digital works. There are a lot of different ways you can perform 3D digital watermarking and just as many applications.

3 Literature review

3.1 Methods of directly modifying mesh geometry

R. Ohbuchi, H. Masuda, and M. Aono : Watermarking 3-Dimensional Polygonal Models (1997)

The first attempts at 3D watermarking were laid out by Ohbuchi in 1997. The method he describes is one of embedding data into 3D models through modification of either their vertex coordinates, their vertex topology, or both. The paper describes five different blind embedding algorithms to watermark 3D objects.

The first method “Triangle Similar Quadruple Embedding” involves finding four similar adjacent triangles in the mesh and using them to embed two bytes of the watermarks code. It achieves this through modification of the triangles edge relationships. The second method “Tetrahedral Volume Ratio Embedding” involves modifying ratios of pairs of volumes in the model in order to embed the data. The third method “Triangle Strip Peeling Symbol-Sequence-Embedding” involves peeling off triangle strips off the model in order to insert the data. The fourth method “Polygon Stencil Pattern Embedding” involves cutting out human visible patterns into the mesh. The final method “Mesh Density Pattern Embedding” involves taking a curved surface model and converting it into a triangular mesh through increasing the number of triangles in certain parts of the model and inserting the data in these parts.

All of these methods represent valid ways to insert watermarks into 3D meshes. However, as the author admits none of these methods are that resistant under attacks, especially attacks that modify the topology of the mesh such as mesh simplification.

Nonetheless this paper servers as a great introduction to 3D watermarking and how it is done. It introduces the concept of robustness and stresses its importance.

Benedens, O. : Geometry-based watermarking of 3-D polygonal models. (1999)

In this Paper Bendens addresses the fundamentals of geometry-based watermarking and then presents his own algorithm which modifies normal distribution to embed data into a 3D mesh.

He talks about the difference between private and public algorithms and their use-cases. He also introduces the idea of embedding multiple watermarks into the same mesh using different keys and how this could be useful for things such as identifying resellers.

The main aim of his proposed algorithm is to improve robustness over previously suggested methods with emphasis on more protection against mesh simplification. This method involves separating the normal vectors to the objects surface into bins based on how similar they are to each other. The normals in each bin are used to encode the data through different means. One way suggested is to encode data based on the mean angular deviation of the bin contents to the bin centre normal. The other way suggested is to encode based on the percentage of the normal in a bin that fall within a certain angle to the bin centre.

This solution to 3D watermarking has increased success in opposing mesh simplification but its not perfect. However, Bendens does suggest something very interesting to improve on the algorithm. He suggests embedding the watermark multiple times inside the mesh and then using majority voting on the extracted data to reduce errors. The more the watermark is embedded the more likely extraction will succeed.

Harte, T., Bors, A.: Watermarking 3d models. (2002)

In this paper an algorithm is proposed in which a watermark is embedded through alteration of the location of select vertices. This process is split into two stages. The first stage is involved in finding suitable locations in the mesh where the watermark can be embedded. The second involves the alteration of the vertex's location in the suitable locations. In order to find suitable vertices to change first the neighbourhoods of each vertex are found where a neighbourhood represents the vertices that the selected vertex is connected to. Then suitable vertices are chosen based on how close they are to the vertices in their neighbourhood. If their distance is below a threshold they are suitable. This selection process is done for the purpose of only changing areas where there is determined to be low human perceptibility to changes. Embedding is either done through calculating the normals of the selected neighbourhood and moving the vertices according to this or using elliptical bounding volumes to determine which way to move them. An interesting idea that is presented in the embedding stage is that of embedding multiple of the same watermark in the object to protect the object from cropping attacks.

3.2 Methods based on multi-resolution mesh analysis

. Praun, E., Hoppe, H., Finkelstein, A.: Robust mesh watermarking. (1999)

This paper describes a method of watermarking which makes use of multiresolution analysis. Their approach details first converting the original mesh into a multiresolution format which consists of a coarse base mesh and a sequence of refinement operations. These refinement operations cause the greatest geometric change to the mesh. The aim is to embed the data into basis functions over the mesh. In order for the method to be robust it is important for the basis functions to correspond to large perceptually significant features of the mesh. Therefore, the refinement operations are used to define a scalar basis function over each operations corresponding neighbourhood. In order to embed the data these basis functions are multiplied by a coefficient and added to the 3D coordinates of the mesh vertices. In order to extract the watermarks the original mesh is needed. The watermarked mesh is brought into the same coordinate frame as the original and resampled to create a mesh with the same connectivity. Then all is needed is to take the differences between the vertex coordinates of each to create a vector of 3D residuals which are used to extract the watermark from.

This method shows good robustness against a wide variety of attacks such as vertex reordering, addition of noise similarity transforms cropping smoothing simplification and insertion of a second watermark.

Qiu, J.J., Ya, D.M., Jun, B.H., Sheng, P.Q.: Watermarking on 3D mesh based on spherical wavelet transform. (2004)

In this paper an algorithm is based on the idea of spherical wavelet transform. The method decomposes the original mesh into a series of details at different scales through use of this transform and then the watermark is embedded into the different level of details. This process begins by first performing global spherical parameterization to the model. This is the process of mapping the mesh into a sphere so that the 3D model can be defined as spherical signals. Then uniform spherical sampling must be performed as aspherical wavelet transform needs geometrical signals to be uniformly sampled over the sphere. Then sphere wavelet forward transform is applied. Through this, areas where local bumpiness can be found which if changed wouldn’t be that discernible to a human eye. The watermark is then embedded into these areas. Then in order to get the mesh back into an acceptable form the inverse of the spherical wavelet transformation is performed, and the watermarked mesh is resampled to recover the topology connectivity of the original model.

Cotting, D., Weyrich, T., Pauly, M., Gross, M.: Robust watermarking of point-sampled geometry. (2004)

In this paper an digital watermarking scheme is explained which is designed for polygonal data to unstructured point clouds. The method it uses goes as follow. The mesh is split into patches and then each patch is mapped into the space of eigenfunctions of an Laplacian operation to obtain a decomposition of the singular patch. The watermark is then embedded into the patches low frequency components to minimize low frequency components.

F. CayreP. Rondao-AlfaceF. Schmitt, Benoıt Macqb, H. Maıtre, Application of spectral decomposition to compression and watermarking of 3D triangle mesh geometry

In this paper a method of watermarking is presented that also works through spectral analysis and spectral decomposition. It splits up meshes into submeshes and then computes the eigenvectors of each of these meshes. From here it embeds the data through editing the meshes spectra so that minimal distortion is caused. It does this through symmetrical movements of these spectra.

3.3 Conclusion

Through my research I have found two camps of watermarking. One being direct action onto mesh properties such as vertices and one focusing on the connectivity of the mesh and changing it in the spectral domain.

Again through research I have detirmend to focus on the latter camp as they show more resistance to more kinds of attacks and overall less visible distortion when embedding.

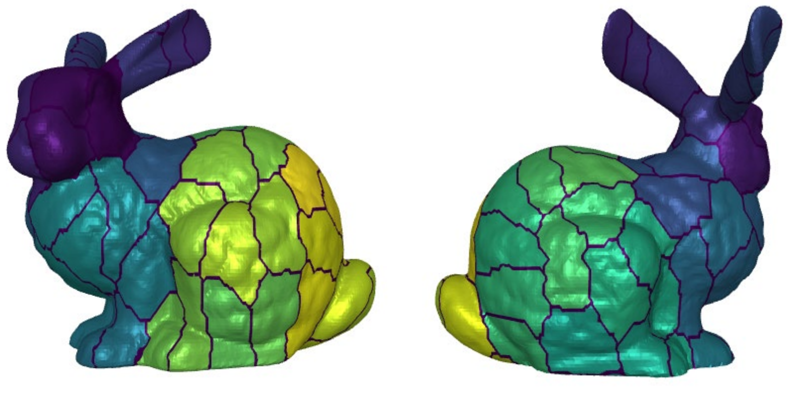
For this reason I have chosen to primarily base my program on the work in [11] as it seems to be a fairly simple way of changing spectral information of an object and seems to be resistant against most attacks.

4 Methodology

4.1 Embedding

4.1.1 Mesh partitioning

The method which I have chosen involves computing eigenvalues and eigenvectors of a Laplacian matrix. Computing these eigenvalues and eigenvectors is very time consuming with it being O(N^3). For larger meshes with thousands of vertices and faces this method becomes unviable unless the mesh is split into smaller sub-meshes. In order to partition the mesh MeTiS software[8] was employed. MeTiS splits meshes based on connectivity and not geometrical position which is important for splitting the mesh even when it has been spatially altered. In the images below you can see an example of the bunny mesh split using this method. If you look closely you can see that each sub-mesh is surrounded by triangles which are coloured purple. These represent the edges of each sub-mesh and are not used to embed data in. I implemented this as there was overlap between the sub-meshes in terms of faces. On the edges of sub-meshes you get vertices which get sorted into different sub-meshes but together they create a face. This resulted in overlap in which groups certain faces were in. The effect of this was that the partitioning was inconsistent between embedding and extracting resulting in the extraction process failing as it did not correctly find the start of the watermark and extracted the watermark incorrectly. Removing these edge faces from the equation means there is no such inconsistency, and the data can be extracted successfully. A mapping is created in this process so that the sub-meshes can be put back together again.



Bunny model partitioned into 40 sub-meshes: front facing view (a), back facing view (b).

4.1.2 Data encryption

The data which is to be embedded is encrypted in order to be more secure in case an attacker is able to extract it. The data is represented as a binary string. This binary string is shuffled randomly based on a secret key. This secret key is needed to decrypt the watermark in the extraction process. It was important to implement this to have added security. If an adversary was able to reengineer the extraction processes they would have access to the watermark so the watermark itself must be encrypted.

4.1.3 3D model representation

In the field of computer graphics 3D models are represented as meshes made up triangles or polygons. Triangle meshes are the norm as any other shape can be made from multiple triangles. A triangle mesh M is represented as a tuple where M = (V,T) where V = {v0,…,vn} is the set of vertices and T = {t0,…,tn} is the set of triangles. The degree di of a vertex refers to how many vertices the vertex is connected to. The neighborhood of a vertex vi is the set vi* = {vj ∈ V : vi ~ vj}. Each vertex is specified in cartesian coordinates as V = (X,Y,Z) $\subset \mathbb{R}^3$.

4.1.4 Embedding data

For each sub-mesh which is created from partitioning the mesh we follow the same process to embed data into it. For every sub-mesh we follow a process of spectral decomposition to embed the data which is done as follows:

Firstly, we must compute the Laplacian matrix of the sub-mesh. Which is defined as followed:

$$L_{ij} = \begin{cases} 1 & If\ i\ == \ j \\ -d_i^{-1} & If\ i\ is\ a\ neighbour\ of\ j \\ 0 & otherwise \end{cases}$$

The eigenvectors of L form an orthogonal basis $\mathbb{R}^N$ therefore eigeinvalues can be considered as pseudo-frequencies of the geometry defined over the mesh graph. These egien values are bounded by 0 and 2. The transformed vectors of the submesh are obtainedby projecting X Y and Z of each vector over the basis functions. The eigenvalues are ordered by decreasing value giving us:

$$B^{-1}LB$$

The columns of B are the eigenvectors of L and the columns of B^{-1} are the dual basis functions. This transformation gives us three vecots of dimension N each representing the X Y and Z of the vectors. These are called spectra and defined as followed:

$$\mathrm{S} = \mathrm{BX}$$

$$\mathrm{T} = \mathrm{BY}$$

$$\mathrm{U} = \mathrm{BZ}$$

Using these spectra is how we will be embedding the actual watermark into each sub-mesh. The aim is to embed 64 bits of information into the mesh as many times as possible. The amount of times the data is inserted depends on the chosen strength.

First we calculate the sum of the powers of each spectra which defines the power spectrum of the Laplacian of V. Then the strength is used to determine a threshold where only the vectors with power sums under this threshold will be considered for insertion.

The actual insertion takes place as follows. S T and U are sorted by size and set as a minimum, intermediate and maximum. From this we get interval between the minimum and maximum and divide it into two equal intervals as defined as followed:

$$W_0 = [minimum; maximum + \frac{(maximum - minimum)}{2}]$$

$$W_1 = [minimum + \frac{(maximum - minimum)}{2}; maximum]$$

The intermediate value will have to lie inside one of these intervals. Now if the bit to be embedded is a zero and the intermediate lies in W_0 then the bit will not be inserted. However if the intermediate lies in W_1 then the intermediate gets flipped with respect to the centre of [min,max]. This is a perfectly symmetrical procedure so can be reversed but also is a good way of spreading the bits pseudo-randomly through meshes as it relies on multiple factors for a bit to be embedded. An adversary would have to know how many sub-meshes the model was broken into and the strength used to embed the watermark to reverse the process.

Now that we have the data embedded in the spectras of the sub-mesh we need to get it back into its X Y Z form. This is a simple process of just multiplying B^{-1} by each spectra as follows:

$$X = B^{-1}S$$

$$Y = B^{-1}T$$

$$Z = B^{-1}U$$

Finally using the mapping that was created when we partitioned the mesh we can reconstruct the mesh into its full form.

4.5 Extracting

4.5.1 Mesh Registration

Before the extraction process can begin we need to estimate the optimal rotation, scaling and translation to get the watermarked model back to its initial scale and location in the case where the object has been changed. This process is extremely important in order to get the watermark back successfully. To estimate this the iterative closest point method was used.

4.5.2 extraction

The extraction process begins the same as the embedding process. First the watermarked mesh is partitioned into sub-meshes and then spectra are calculated from these sub-meshes. However instead of embedding bits into these spectra we are retrieving them. This is done similar the embedding process where you get the intervals. However, all that is needed in the case of extraction is to read whether the intermediate lies in W_0 or W_1.

4.5.3 Error correction

The extraction process is not always perfect and thus some sort of error detection is needed. I have opted for majority voting which takes in all the extracted bite strings and compares them against each other. For each bite in the string whichever one has come up the most will be chosen. The higher the strength of the algorithm the higher the amount of bits inserted and the more likely correct strings will be the majority.

4.6 Conclusion

Above I have described the algorithm I have chosen to implement. It focuses on spectral decomposition in order to embed data in a non-visible way.

5 Testing

5.1 Parameters

Number of Sub-meshes: N

The number of sub-meshes is controlled through a parameter N. It controls how many sub-meshes the mesh is split into before it has the data embedded into it. Through experimentation I determined a good value for N was formulated as: (Number of vertices / 500). Splitting meshes into too many sub-meshes can take a while but splitting them into too few made the embedding algorithm take too long. I found this simple formula provided an excellent balance.

Strength: α

Strength determines how “strong” the embedding process is. It is a value between 0 and 1 and determines the threshold which is used to decide whether to embed in a frequency or not. The list of frequencies is taken in and then strength is used to find the α percentile of the frequencies. Any frequencies which are higher than this percentile are modified. This means that the higher the strength is the higher the number of frequencies which will be modified. The higher the strength the more obvious the visual changes are but more data is embedded and thus the algorithm is more robust.

Amplitude: A

Amplitude determines how much noise is added to the mesh in the random noise attack. A determines how much a single vertex will move when under the attack.

Smoothness: S

Smoothness determines how much the mesh is smoothed when under the smoothness attack.

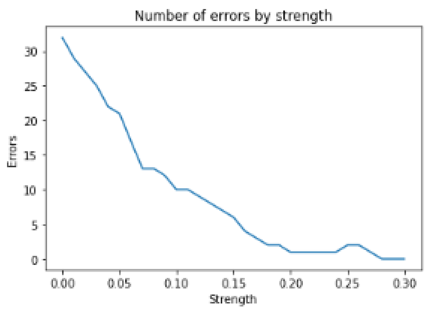
SimpleStrength: β

SimpleStength determines how “strong” the mesh simplification attack is. It is a value between 0 and 1 and determines how much the mesh is simplified by. It is multiplied by the number of vertices of a mesh to figure out how many vertices the mesh will be reduced to.

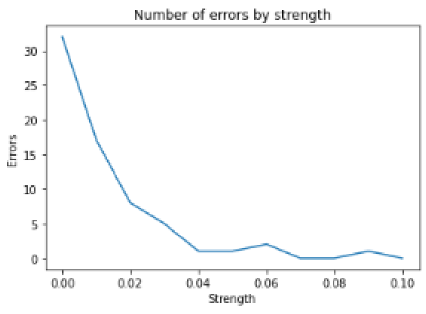
5.2 Errors by strength

When extracting the watermark there is not a certainty that the watermark will be correctly extracted from each sub-mesh. However, by increasing the strength of the embedding process more bits are inserted into the mesh and thus it is more likely that the watermark will be extracted successfully. Below are graphs which plot strength against the number of errors which occurred when extracting. The number of errors is determined by comparing the original watermark with the extracted one. The graphs start at a strength of 0 and finish when the number of errors reaches zero. After this point any increase in strength would not change the number of errors further.

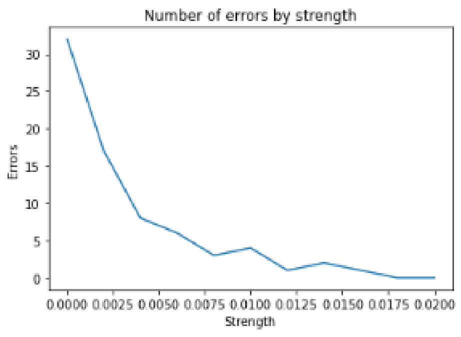
Elephant mesh:



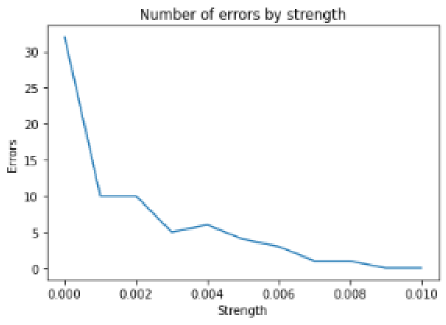
Athena mesh:



Venus mesh:



Bunny mesh:



The graphs above clearly show that the number of errors when extracting decreases when strength increases, and it shows that it takes different amount of strength of each mesh to achieve zero errors. They also show that reaching zero errors isn't a case of errors always reducing with every strength increase as shown in some instances of errors increasing with strength.

Model	**No. of vertices**	**No. of faces**	**No. of sub-meshes**	**Strength to get 0 errors**	**Capacity (bits)**
Elephant	**623**	**1,148**	**1**	**0.28**	**175**
Athena	**4,721**	**9,395**	**9**	**0.1**	**368**
Venus	**19,847**	**43,357**	**40**	**0.02**	**375**
Bunny	**35,947**	**69,451**	**72**	**0.01**	**420**

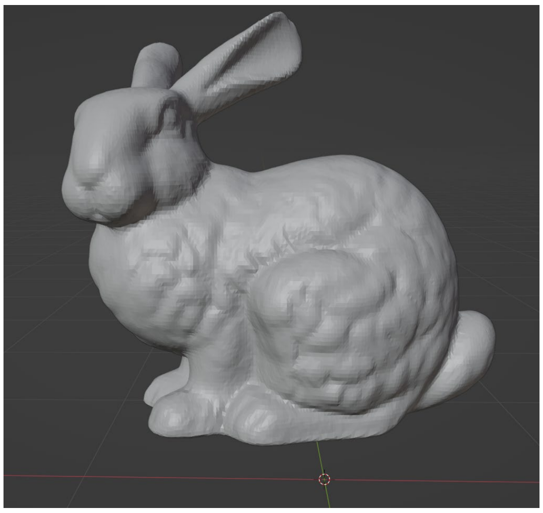
In the table above it can be clearly seen that the higher the number vertices in the mesh the lower the strength needed to reach zero errors. This makes sense as more vertices means that the percentage of vertices that pass the strength threshold will be more and thus more vertices will be used to embed the watermark in for lower strengths. This can also be shown in the table as the capacity of the higher vertex meshes are still higher with a lower strength.

5.3 Visibility

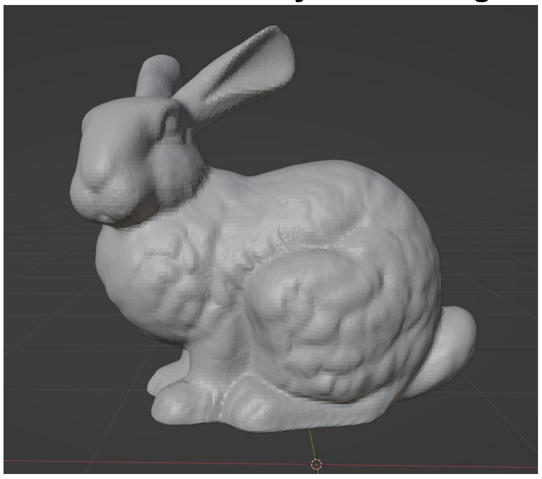
This section will focus on the visual change of the meshes when the watermark has been embedded into it at different strength levels.

Below I have used the bunny model as an example of the effect increasing strength levels have on the distortion of the mesh's visibility. The levels of strength go from 0.01 to 0.9. I have chosen to start at a strength of 0.01 as this is the strength level where the extraction begins to return 0 errors.

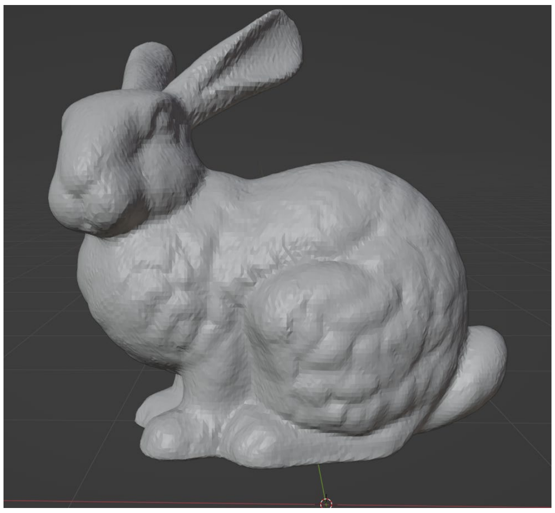
Unwatermarked mesh



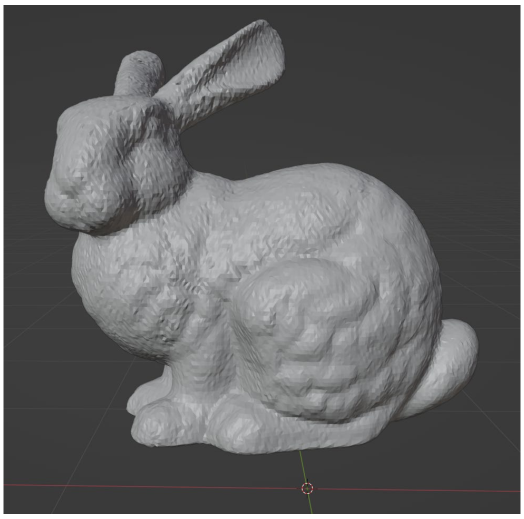
Watermarked bunny with strength α = 0.01



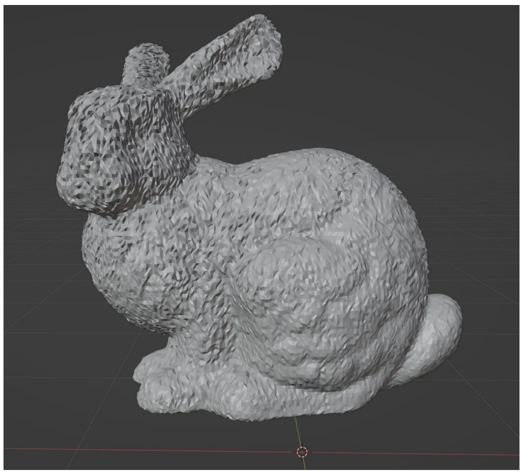
Watermarked bunny with strength α = 0.05



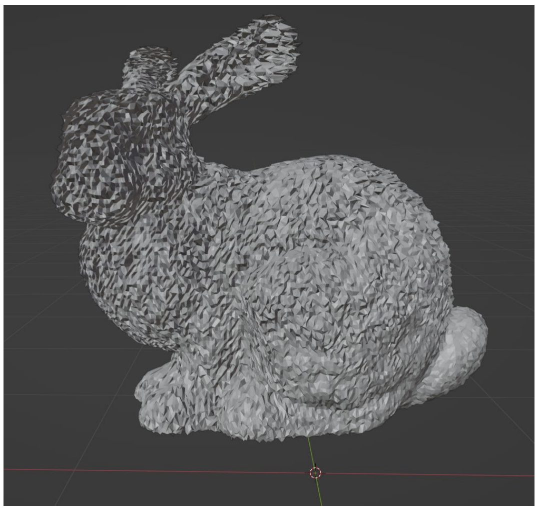
Watermarked bunny with strength α = 0.1



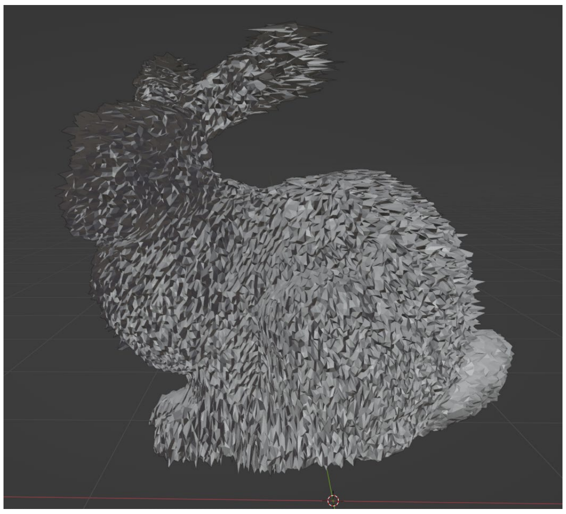
Watermarked mesh with strength α = 0.25



Watermarked mesh with strength α = 0.5



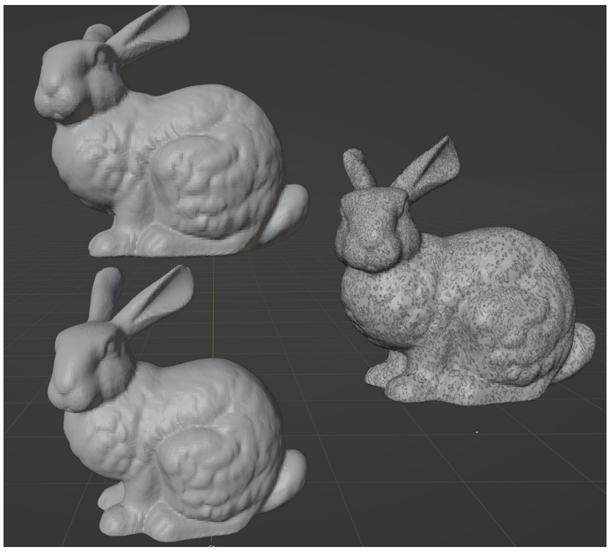
Watermarked mesh with strength α = 0.9



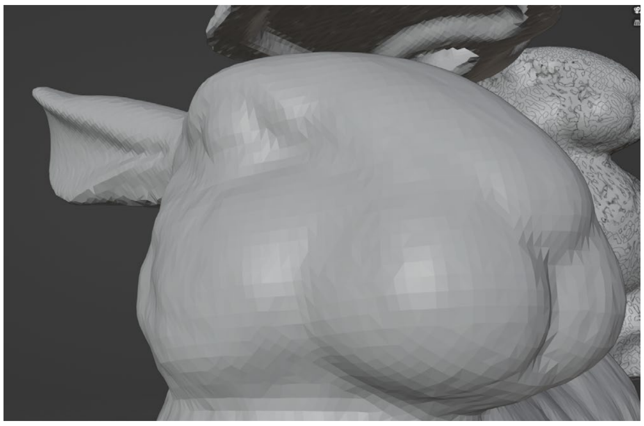
Through the examples above it is clear to see that the bunny becomes increasingly visibly distorted the larger the strength is. While the examples with higher strengths are more robust to attacks all of the examples above can have the watermark extracted out with zero errors.

Below I show the bunny mesh compared to their watermarked mesh which achieves zero errors for the first time.

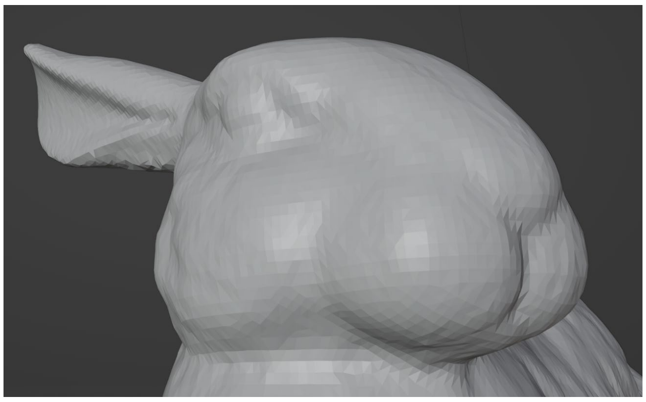
Original mesh (bottom left), Watermarked mesh (top left), original and watermarked mesh superimposed to see differences (right).



Close up on original bunny.



Close up on watermarked bunny.



Through the above images we can see how indistinguishable the changes are to the human eye. The only way we can see the differences is to superimpose both of them and show where they exactly differ.

5.4 Distortion

While simply looking at how the mesh has changed after embedding the watermark is a useful way to assess distortion it is more helpful if we have a way to mathematically measure it. I have achieved this through three different methods.

Root mean square error

Root mean square error is used to calculate the average distance of a vertex in the watermarked mesh from its corresponding vertex in the original mesh. It is defined as follows:

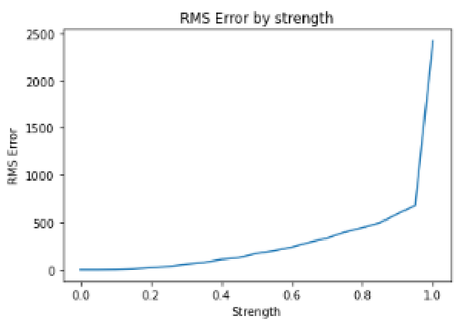
$$RMS = \sqrt{\frac{(V_O - V_W)_0^2 + (V_O - V_W)_1^2 + (V_O - V_W)_2^2 + \ldots + (V_O - V_W)_n^2}{N}}$$

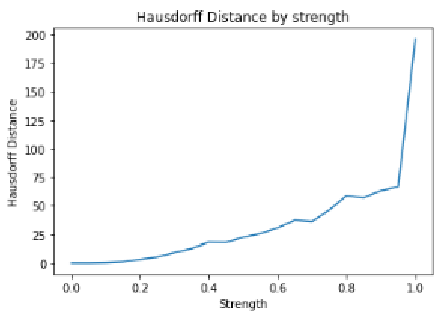
Where V_O represents the vertex of the original mesh, V_W represents the vertex of the watermarked mesh and N represents the number of vertices.

Hausdorff distance

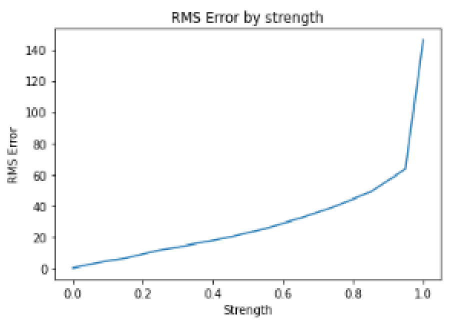
Hausdorff distance measures how far two subsets of a metric space are from each other. The Hausdorff distance is the longest distance between from a vertex in one mesh to the closest vertex in the other mesh. This is achieved in practise by looping through the set of vertices in one mesh and finding the distance from each to its closest in the other mesh. The largest of these distances in the hausdorff distance.

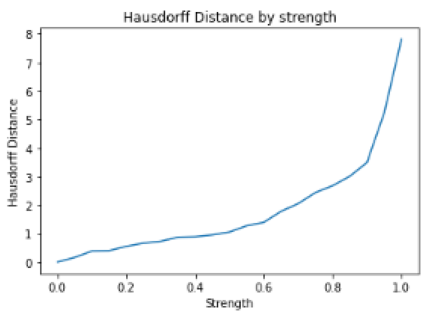
Elephant:



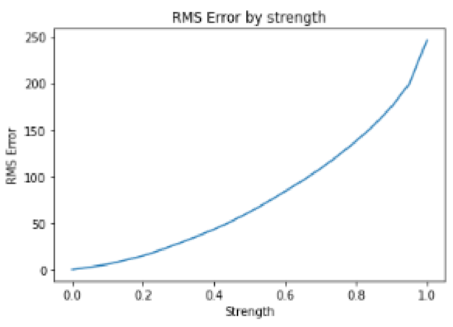


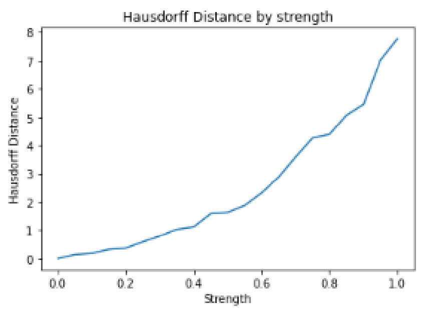
Athena:



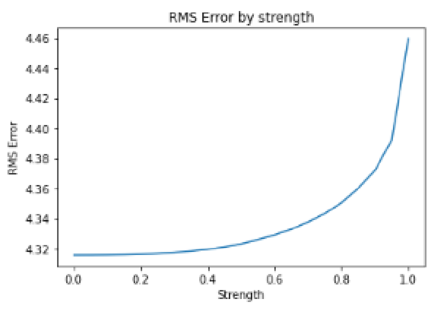


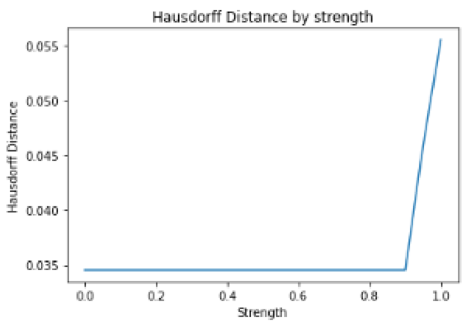
Venus:





Bunny:





5.5 Capacity and Time

Model	No. of vertices	No. of faces	No. of sub-meshes	Strength to get 0 errors	Capacity (bits)
Elephant	**623**	**1,148**	**1**	**0.28**	**175**
Athena	**4,721**	**9,395**	**9**	**0.1**	**368**
Venus	**19,847**	**43,357**	**40**	**0.02**	**375**
Bunny	**35,947**	**69,451**	**72**	**0.01**	**420**

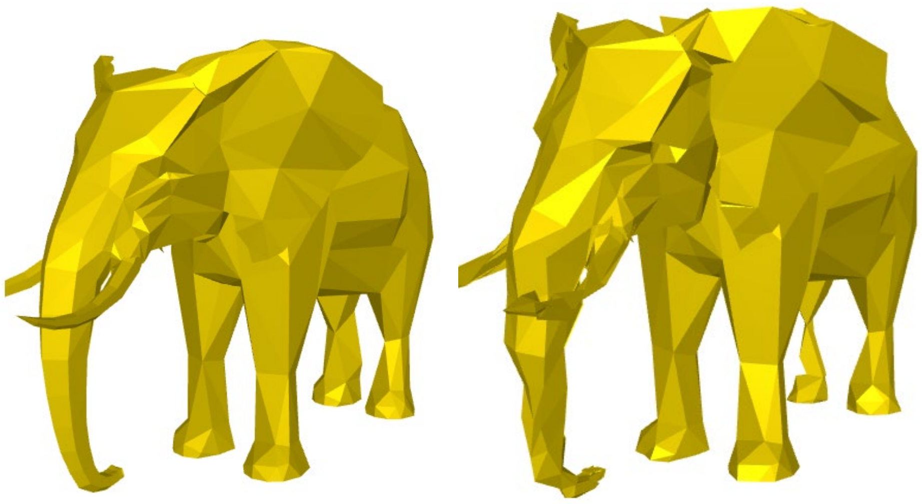
Model	No. of vertices	No. of faces	No. of sub-meshes	Strength to get 0 errors	Time (Seconds)
Elephant	**623**	**1,148**	**1**	**0.28**	**0.35**
Athena	**4,721**	**9,395**	**9**	**0.1**	**4.17**
Venus	**19,847**	**43,357**	**40**	**0.02**	**9.78**
Bunny	**35,947**	**69,451**	**72**	**0.01**	**14.72**

5.6 Attacks

5.6.1 Robustness against translation, rotation, and scaling

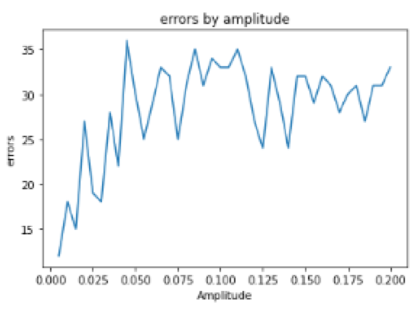
The algorithm is robust against translation rotation and scaling. This is thanks to the alignment that is done before extracting the watermark from the mesh. Iterative closest point (ICP) is used to align meshes and make sure that they are the same size rotation and placement before attempting to extract the watermark.

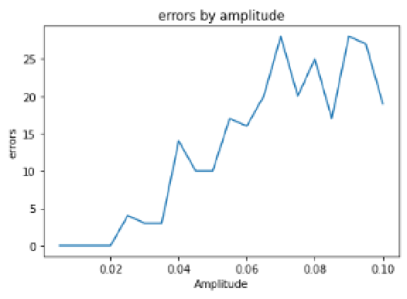
5.6.2 Robustness against noise



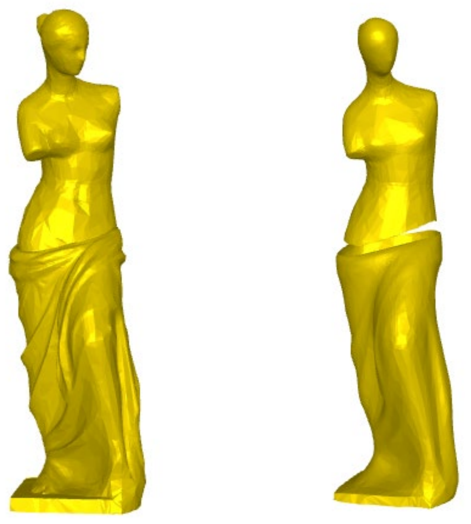
Noise attacks are attacks which randomly apply noise to each vertex in the mesh. This is done by multiplying a vertex's position by a random number and an amplitude value. The algorithm is fairly resistant to these attacks but as can be seen below if you turn up amilitude to a large amount the algorithms stops being effective.

Elephant: **Athena:**



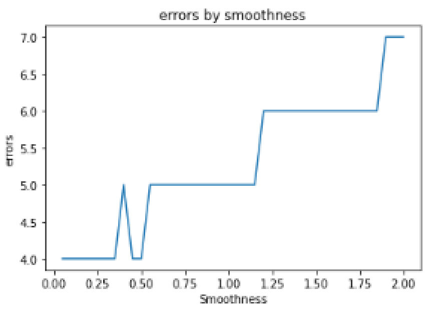


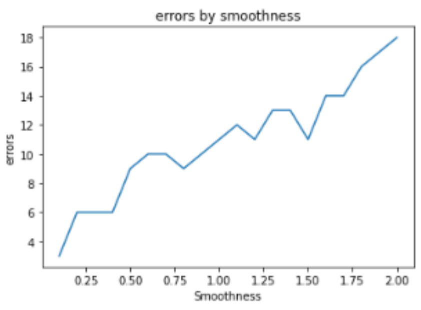
5.6.3 Robustness against smoothing



Smoothing attacks like noise attacks also change the vertices positions but in a deliberate way so to smooth out the mesh and retain some semblance of itself. Like noise attacks the algorithm does fairly well at lower smoothness values but stuggles the more the smoothness rises.

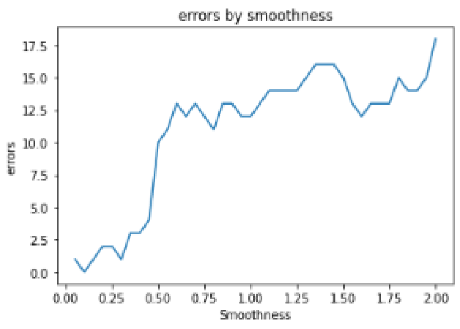
Elephant: **Athena:**

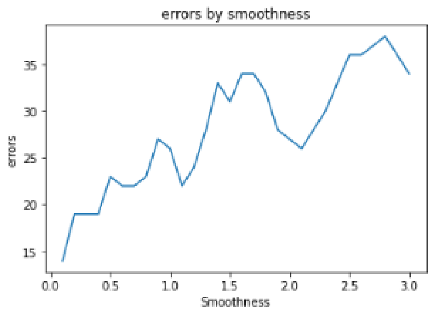




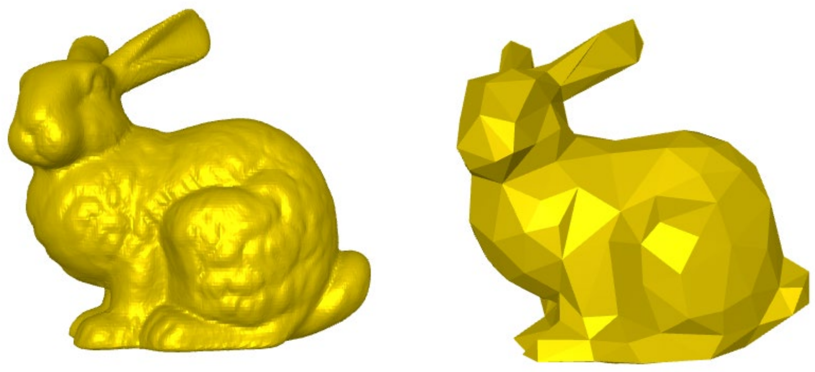
Venus:

Bunny:



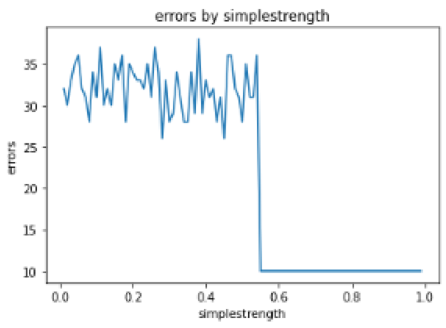


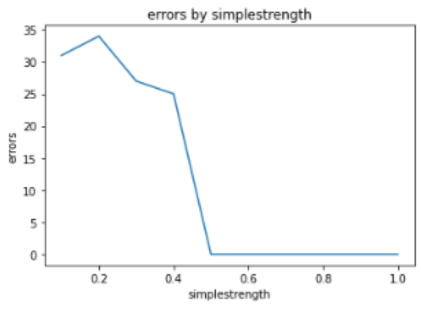
5.6.4 Robustness against simplification



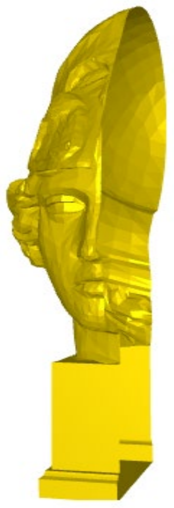
Mesh simplification is the act of reducing a meshes number of vertices and faces in order to simplify a mesh. This could be used by an attacker to destroy the watermark embedded in the mesh as it reduces the amount of vertices in the mesh and therefore gets rid of vertices which carry information. Fortunately, due to mesh alignment the chosen method is fairly resistant to mesh simplification. As shown in the graph below the algorithm is useless past a certain level of simplification but It can still extract correctly even with over 40 percent of the vertices gone.

Elephant: **Venus:**





5.6.5 Robustness against cropping



Cropping is an attack where the mesh is split into different parts. This disrupts how the mesh is partitioned and removes a lot of the embedded data. Unfortunatly because of how my partitioning system works I was unable to make the algorithm robust against mesh cropping. For whatever reason the mesh gets partitioned differently when cut and is unable to extract the data successfully out. If I had more time I would look to implement a better allignment algorithm that takes into account where the mesh has been cut and partition the mesh based on this.

5.7 Conclusion

The algorithm has been assessed in terms of the amount of errors occurred when extracting, capacity, time, distortion and robustness against attacks. The algorithm performed excellent when it came to visible distortion in the objects but not as well when it came to robustness under attacks.

6 Conclusion

6.1 Evaluation of performance

I set out to create a 3d digital watermarking algorithm that can embed data into 3d objects without visible distortion. In this aspect the algorithm is a success. I also had the goal of creating a robust and secure algorithm. While I believe the algorithm is secure due to the nature of spectral decomposition and use of encryption I do think the algorithm could be more robust. It does fine against some amount of smoothing and noise but eventually fails. It can resist half of mesh simplification but cannot deal with mesh cropping.

6.2 Applications

This algorithm could be used in a myriad of situations. My ideal use for this algorithm would be to protect users models from piracy by embedding watermarks into them. I think this algorithm would be a good fit as it does not ruin models looks when embedding the watermark.

6.3 Improvements.

The main area for concern is the algorithms lack of robustness against attacks. This is mainly caused by the implementation of the partitioning algorithm. If I were to make any improvements it would definitely be to how this algorithm works and also to how meshes are aligned before undergoing extraction.

7 Bibliography

[1] https://www.sciencedirect.com/topics/engineering/image-watermarking

[2] https://www.tctmagazine.com/additive-manufacturing-3d-printing-news/rights-reserved-new-digital-watermarking-3d-printing/

[3]

R. Ohbuchi, H. Masuda, and M. Aono : Watermarking 3-Dimensional Polygonal Models (1997)

[4] Benedens, O. : Geometry-based watermarking of 3-D polygonal models. (1999)

[5] Harte, T., Bors, A.: Watermarking 3d models. (2002)

[6] Praun, E., Hoppe, H., Finkelstein, A.: Robust mesh watermarking. (1999)

[7] Qiu, J.J., Ya, D.M., Jun, B.H., Sheng, P.Q.: Watermarking on 3D mesh based on spherical wavelet transform. (2004)

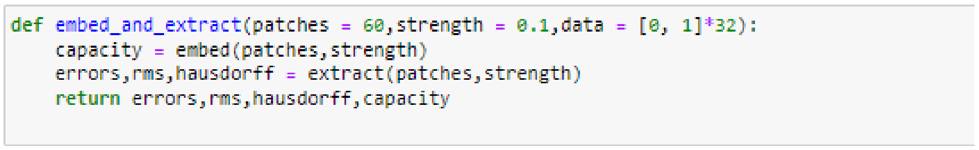
[8] Cotting, D., Weyrich, T., Pauly, M., Gross, M.: Robust watermarking of point-sampled geometry. (2004)

[9] F. CayreP. Rondao-AlfaceF. Schmitt, Benoıt Macqb, H. Maıtre, Application of spectral decomposition to compression and watermarking of 3D triangle mesh geometry

[10] Karypis, G., Kumar, V.: MeTiS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fillreducing orderings of sparse matrices, Version 4.0, University of Minnesota, Department of Computer Science (1998)

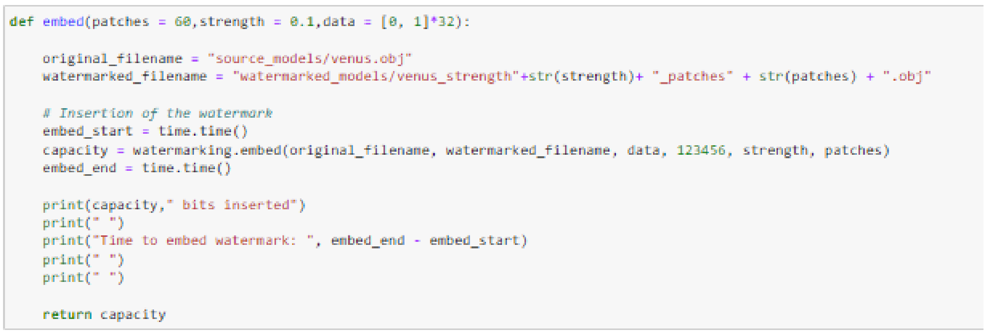
8 Appendix

8.1 Embed and extract



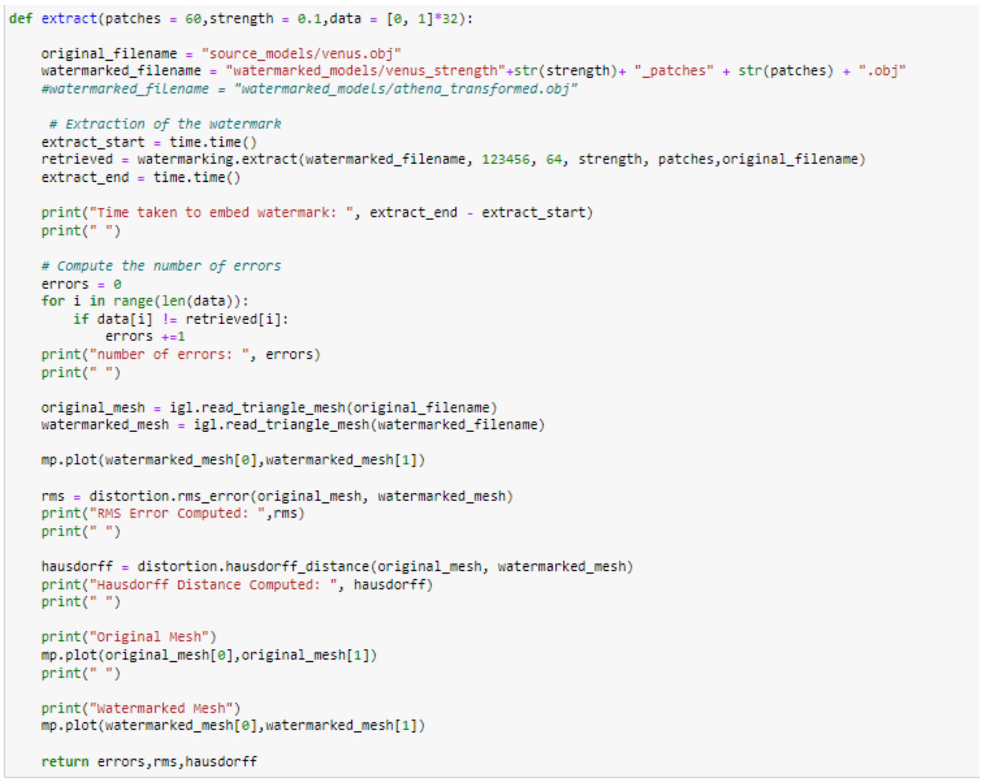
```python
def embed_and_extract(patches = 60,strength = 0.1,data = [0, 1]*32):
    capacity = embed(patches,strength)
    errors,rms,hausdorff = extract(patches,strength)
    return errors,rms,hausdorff,capacity
```

8.2 Embed start



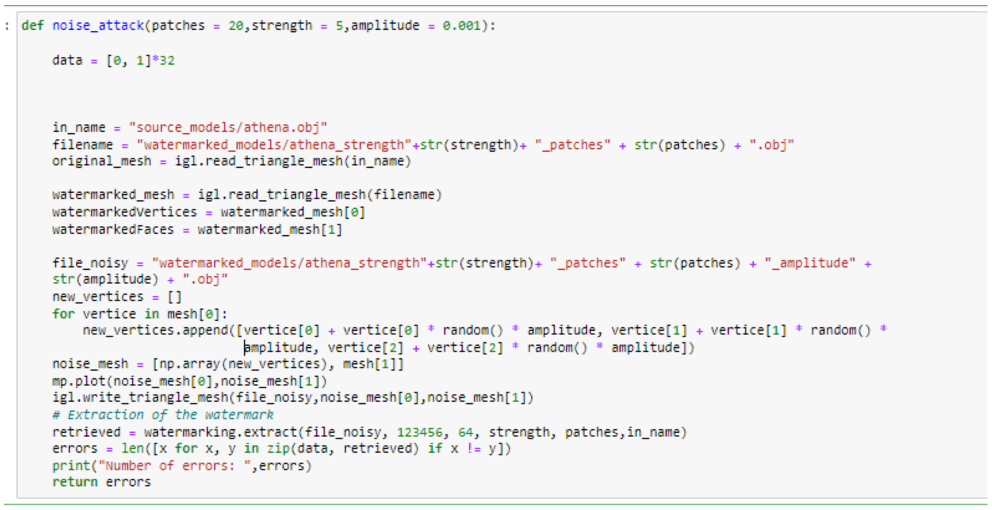
```python
def embed(patches = 60,strength = 0.1,data = [0, 1]*32):

    original_filename = "source_models/venus.obj"
    watermarked_filename = "watermarked_models/venus_strength"+str(strength)+ "_patches" + str(patches) + ".obj"

    # Insertion of the watermark
    embed_start = time.time()
    capacity = watermarking.embed(original_filename, watermarked_filename, data, 123456, strength, patches)
    embed_end = time.time()

    print(capacity," bits inserted")
    print(" ")
    print("Time to embed watermark: ", embed_end - embed_start)
    print(" ")
    print(" ")

    return capacity
```

8.3 Extract start



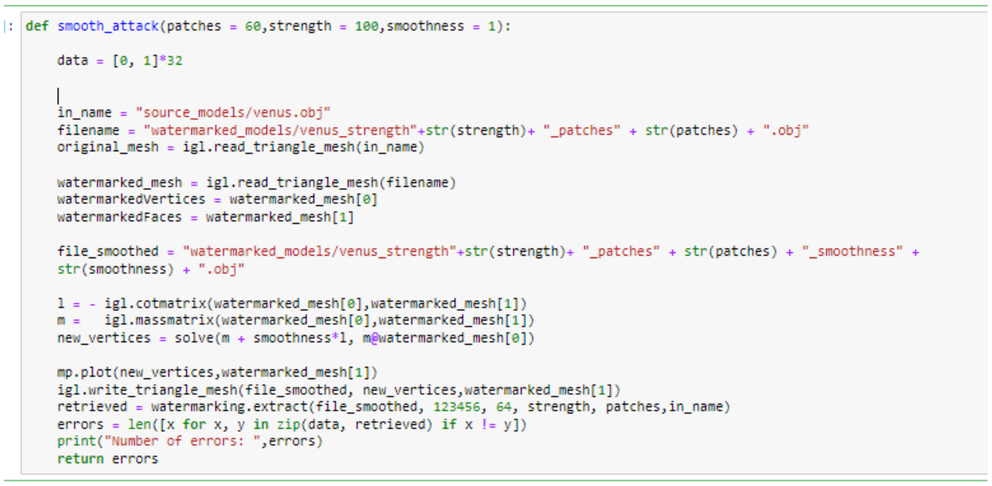
```python
def extract(patches = 60,strength = 0.1,data = [0, 1]*32):

    original_filename = "source_models/venus.obj"
    watermarked_filename = "watermarked_models/venus_strength"+str(strength)+ "_patches" + str(patches) + ".obj"
    #watermarked_filename = "watermarked_models/athena_transformed.obj"

     # Extraction of the watermark
    extract_start = time.time()
    retrieved = watermarking.extract(watermarked_filename, 123456, 64, strength, patches,original_filename)
    extract_end = time.time()

    print("Time taken to embed watermark: ", extract_end - extract_start)
    print(" ")

    # Compute the number of errors
    errors = 0
    for i in range(len(data)):
        if data[i] != retrieved[i]:
            errors +=1
    print("number of errors: ", errors)
    print(" ")

    original_mesh = igl.read_triangle_mesh(original_filename)
    watermarked_mesh = igl.read_triangle_mesh(watermarked_filename)

    mp.plot(watermarked_mesh[0],watermarked_mesh[1])

    rms = distortion.rms_error(original_mesh, watermarked_mesh)
    print("RMS Error Computed: ",rms)
    print(" ")

    hausdorff = distortion.hausdorff_distance(original_mesh, watermarked_mesh)
    print("Hausdorff Distance Computed: ", hausdorff)
    print(" ")

    print("Original Mesh")
    mp.plot(original_mesh[0],original_mesh[1])
    print(" ")

    print("Watermarked Mesh")
    mp.plot(watermarked_mesh[0],watermarked_mesh[1])

    return errors,rms,hausdorff
```

8.4 Noise attack



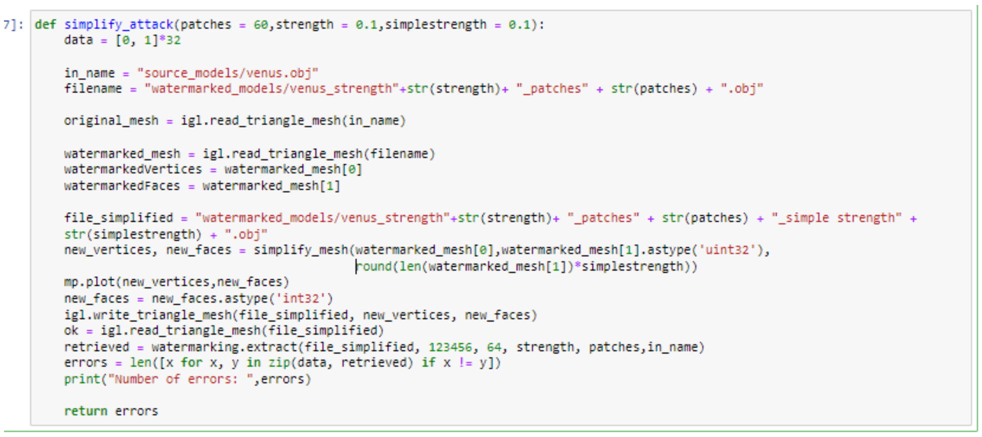
```
: def noise_attack(patches = 20,strength = 5,amplitude = 0.001):

      data = [0, 1]*32


      in_name = "source_models/athena.obj"
      filename = "watermarked_models/athena_strength"+str(strength)+ "_patches" + str(patches) + ".obj"
      original_mesh = igl.read_triangle_mesh(in_name)

      watermarked_mesh = igl.read_triangle_mesh(filename)
      watermarkedVertices = watermarked_mesh[0]
      watermarkedFaces = watermarked_mesh[1]

      file_noisy = "watermarked_models/athena_strength"+str(strength)+ "_patches" + str(patches) + "_amplitude" +
      str(amplitude) + ".obj"
      new_vertices = []
      for vertice in mesh[0]:
          new_vertices.append([vertice[0] + vertice[0] * random() * amplitude, vertice[1] + vertice[1] * random() *
                              amplitude, vertice[2] + vertice[2] * random() * amplitude])
      noise_mesh = [np.array(new_vertices), mesh[1]]
      mp.plot(noise_mesh[0],noise_mesh[1])
      igl.write_triangle_mesh(file_noisy,noise_mesh[0],noise_mesh[1])
      # Extraction of the watermark
      retrieved = watermarking.extract(file_noisy, 123456, 64, strength, patches,in_name)
      errors = len([x for x, y in zip(data, retrieved) if x != y])
      print("Number of errors: ",errors)
      return errors
```

8.5 Smooth attack



```
|: def smooth_attack(patches = 60,strength = 100,smoothness = 1):

       data = [0, 1]*32


       in_name = "source_models/venus.obj"
       filename = "watermarked_models/venus_strength"+str(strength)+ "_patches" + str(patches) + ".obj"
       original_mesh = igl.read_triangle_mesh(in_name)

       watermarked_mesh = igl.read_triangle_mesh(filename)
       watermarkedVertices = watermarked_mesh[0]
       watermarkedFaces = watermarked_mesh[1]

       file_smoothed = "watermarked_models/venus_strength"+str(strength)+ "_patches" + str(patches) + "_smoothness" +
       str(smoothness) + ".obj"

       l = - igl.cotmatrix(watermarked_mesh[0],watermarked_mesh[1])
       m =   igl.massmatrix(watermarked_mesh[0],watermarked_mesh[1])
       new_vertices = solve(m + smoothness*l, m@watermarked_mesh[0])

       mp.plot(new_vertices,watermarked_mesh[1])
       igl.write_triangle_mesh(file_smoothed, new_vertices,watermarked_mesh[1])
       retrieved = watermarking.extract(file_smoothed, 123456, 64, strength, patches,in_name)
       errors = len([x for x, y in zip(data, retrieved) if x != y])
       print("Number of errors: ",errors)
       return errors
```

8.6 Simplify attack

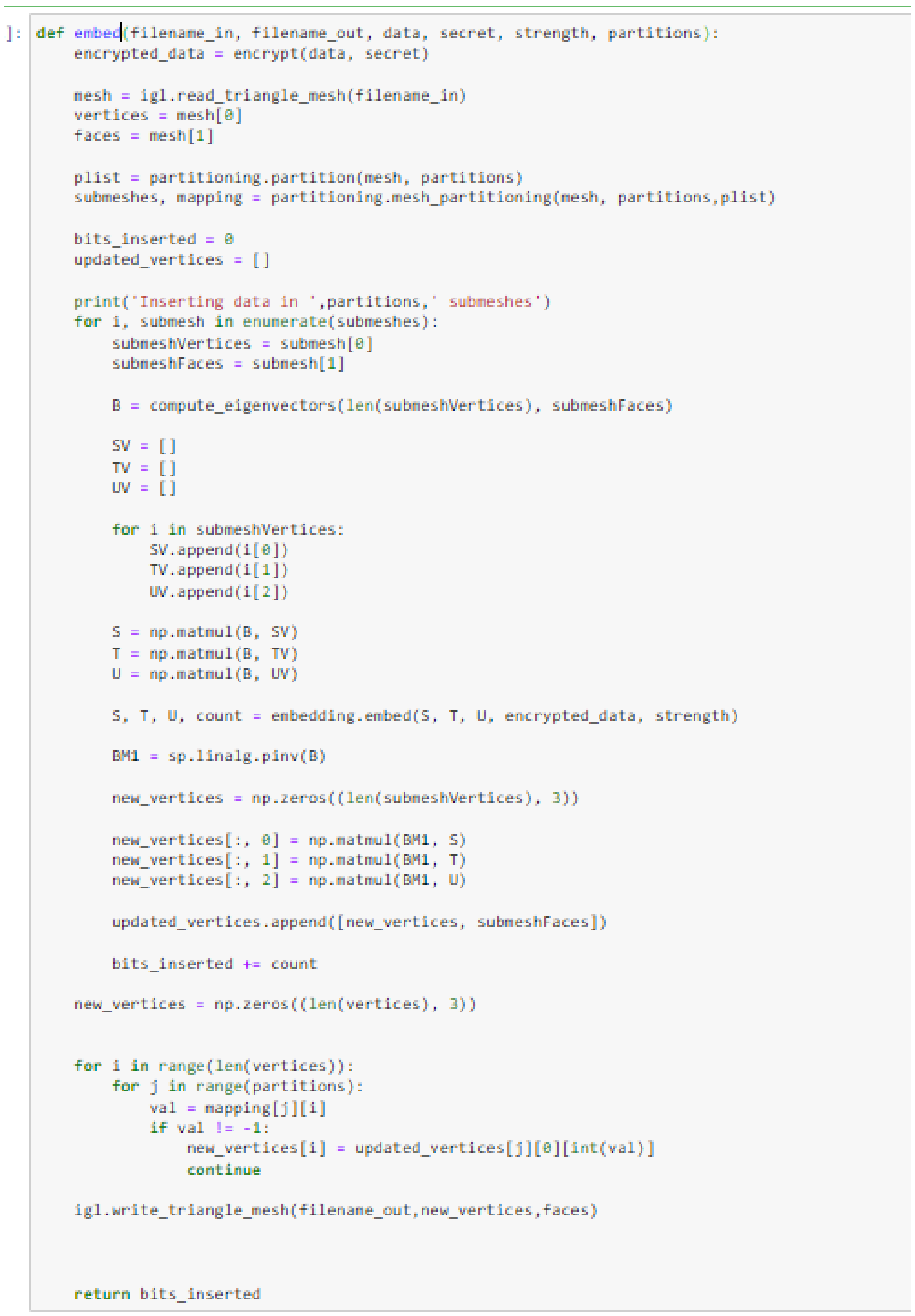


```
7]: def simplify_attack(patches = 60,strength = 0.1,simplestrength = 0.1):
        data = [0, 1]*32

        in_name = "source_models/venus.obj"
        filename = "watermarked_models/venus_strength"+str(strength)+ "_patches" + str(patches) + ".obj"

        original_mesh = igl.read_triangle_mesh(in_name)

        watermarked_mesh = igl.read_triangle_mesh(filename)
        watermarkedVertices = watermarked_mesh[0]
        watermarkedFaces = watermarked_mesh[1]

        file_simplified = "watermarked_models/venus_strength"+str(strength)+ "_patches" + str(patches) + "_simple strength" +
        str(simplestrength) + ".obj"
        new_vertices, new_faces = simplify_mesh(watermarked_mesh[0],watermarked_mesh[1].astype('uint32'),
                                       round(len(watermarked_mesh[1])*simplestrength))
        mp.plot(new_vertices,new_faces)
        new_faces = new_faces.astype('int32')
        igl.write_triangle_mesh(file_simplified, new_vertices, new_faces)
        ok = igl.read_triangle_mesh(file_simplified)
        retrieved = watermarking.extract(file_simplified, 123456, 64, strength, patches,in_name)
        errors = len([x for x, y in zip(data, retrieved) if x != y])
        print("Number of errors: ",errors)

        return errors
```

8.7 Main

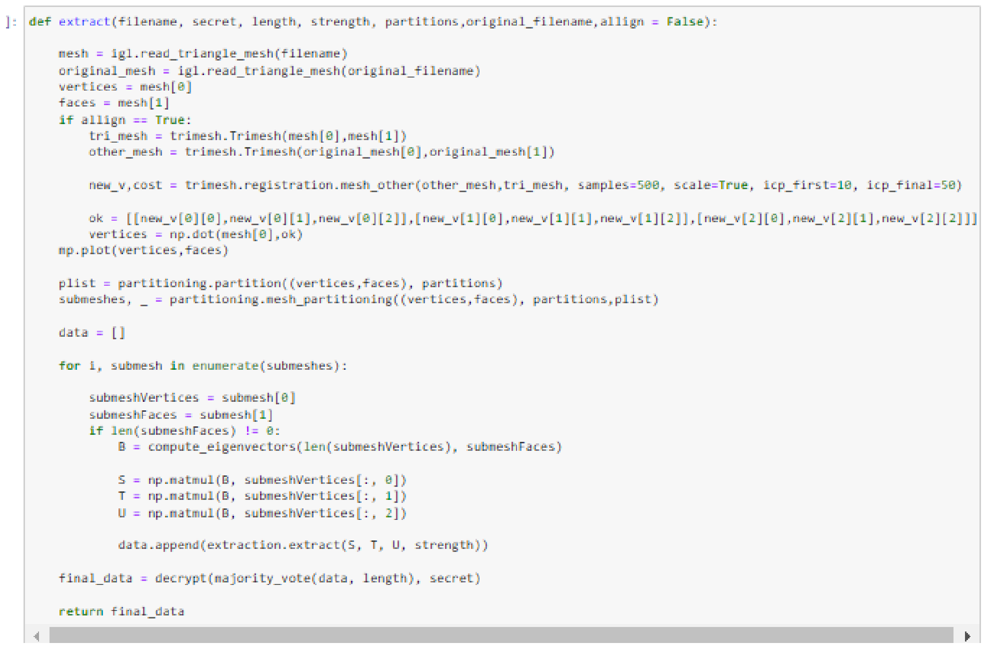
```python
if __name__ == '__main__':

    errorlist = []
    rmslist = []
    hauslist = []
    capacitylist = []
    modellist = ["elephant","athena","venus","bunny","hand"]


    mesh = igl.read_triangle_mesh("source_models/venus.obj")
    num_submeshes = round(len(mesh[0])/500)
    strengths = [x * 0.01 for x in range(29,29)]

    for i in strengths:
        errors,rms,hausdorff, capacity= embed_and_extract(num_submeshes,i)
        errorlist.append(errors)
        rmslist.append(rms)
        hauslist.append(hausdorff)
        capacitylist.append(capacity)

    x = strengths
    y = errorlist
    plt.title("Number of errors by strength")
    plt.xlabel("Strength")
    plt.ylabel("Errors")
    plt.plot(x,y)
    plt.show()

    x = strengths
    y = rmslist
    plt.title("RMS Error by strength")
    plt.xlabel("Strength")
    plt.ylabel("RMS Error")
    plt.plot(x,y)
    plt.show()

    x = strengths
    y = hauslist
    plt.title("Hausdorff Distance by strength")
    plt.xlabel("Strength")
    plt.ylabel("Hausdorff Distance")
    plt.plot(x,y)
    plt.show()

    x = strengths
    y = capacitylist
    plt.title("Capacity by strength")
    plt.xlabel("Strength")
    plt.ylabel("Capacity (bits)")
    plt.plot(x,y)
    plt.show()

    smoothlist = [x * 0.1 for x in range(1, 31)]
    smooth_errors = []
    for i in smoothlist:
        smooth_errors.append(smooth_attack(num_submeshes,0.02,i))

    x = smoothlist
    y = smooth_errors
    plt.title("errors by smoothness")
    plt.xlabel("Smoothness")
    plt.ylabel("errors")
    plt.plot(x,y)
    plt.show()


    simplestrengths = [x *  0.1 for x in range(1, 11)]
    simple_errors = []
    for i in simplestrengths:
        simple_errors.append(simplify_attack(num_submeshes,0.09,i))

    x = simplestrengths
    y = simple_errors
    plt.title("errors by simplestrength")
    plt.xlabel("simplestrength")
    plt.ylabel("errors")
    plt.plot(x,y)
    plt.show()

    amplitudelist = [x * 0.004 for x in range(1, 21)]
    noise_errors = []
    for i in amplitudelist:
        noise_errors.append(noise_attack(num_submeshes,0.09,i))

    x = amplitudelist
    y = noise_errors
    plt.title("errors by amplitude")
    plt.xlabel("Amplitude")
    plt.ylabel("errors")
    plt.plot(x,y)
    plt.show()
```

8.8 embed



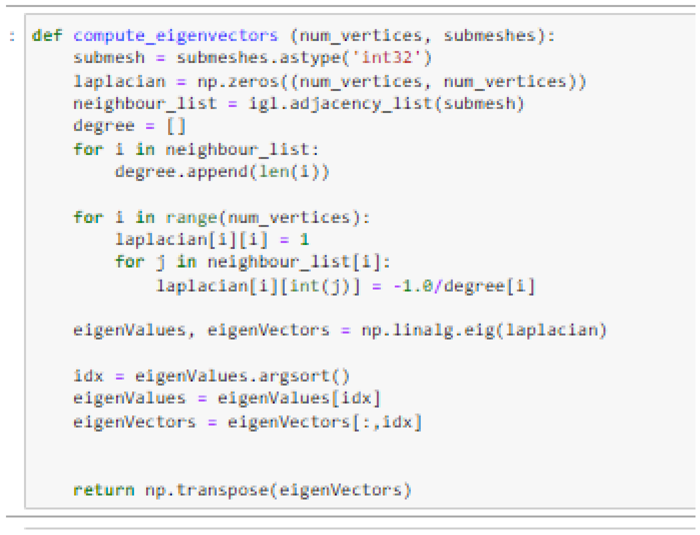
```
]: def embed(filename_in, filename_out, data, secret, strength, partitions):
       encrypted_data = encrypt(data, secret)

       mesh = igl.read_triangle_mesh(filename_in)
       vertices = mesh[0]
       faces = mesh[1]

       plist = partitioning.partition(mesh, partitions)
       submeshes, mapping = partitioning.mesh_partitioning(mesh, partitions,plist)

       bits_inserted = 0
       updated_vertices = []

       print('Inserting data in ',partitions,' submeshes')
       for i, submesh in enumerate(submeshes):
           submeshVertices = submesh[0]
           submeshFaces = submesh[1]

           B = compute_eigenvectors(len(submeshVertices), submeshFaces)

           SV = []
           TV = []
           UV = []

           for i in submeshVertices:
               SV.append(i[0])
               TV.append(i[1])
               UV.append(i[2])

           S = np.matmul(B, SV)
           T = np.matmul(B, TV)
           U = np.matmul(B, UV)

           S, T, U, count = embedding.embed(S, T, U, encrypted_data, strength)

           BM1 = sp.linalg.pinv(B)

           new_vertices = np.zeros((len(submeshVertices), 3))

           new_vertices[:, 0] = np.matmul(BM1, S)
           new_vertices[:, 1] = np.matmul(BM1, T)
           new_vertices[:, 2] = np.matmul(BM1, U)

           updated_vertices.append([new_vertices, submeshFaces])

           bits_inserted += count

       new_vertices = np.zeros((len(vertices), 3))


       for i in range(len(vertices)):
           for j in range(partitions):
               val = mapping[j][i]
               if val != -1:
                   new_vertices[i] = updated_vertices[j][0][int(val)]
                   continue

       igl.write_triangle_mesh(filename_out,new_vertices,faces)



       return bits_inserted
```

8.9 extract



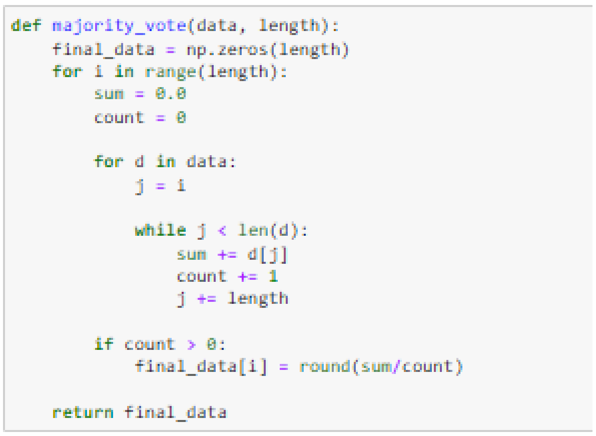
```
]: def extract(filename, secret, length, strength, partitions,original_filename,allign = False):

       mesh = igl.read_triangle_mesh(filename)
       original_mesh = igl.read_triangle_mesh(original_filename)
       vertices = mesh[0]
       faces = mesh[1]
       if allign == True:
           tri_mesh = trimesh.Trimesh(mesh[0],mesh[1])
           other_mesh = trimesh.Trimesh(original_mesh[0],original_mesh[1])

           new_v,cost = trimesh.registration.mesh_other(other_mesh,tri_mesh, samples=500, scale=True, icp_first=10, icp_final=50)

           ok = [[new_v[0][0],new_v[0][1],new_v[0][2]],[new_v[1][0],new_v[1][1],new_v[1][2]],[new_v[2][0],new_v[2][1],new_v[2][2]]]
           vertices = np.dot(mesh[0],ok)
       mp.plot(vertices,faces)

       plist = partitioning.partition((vertices,faces), partitions)
       submeshes, _ = partitioning.mesh_partitioning((vertices,faces), partitions,plist)

       data = []

       for i, submesh in enumerate(submeshes):

           submeshVertices = submesh[0]
           submeshFaces = submesh[1]
           if len(submeshFaces) != 0:
               B = compute_eigenvectors(len(submeshVertices), submeshFaces)

               S = np.matmul(B, submeshVertices[:, 0])
               T = np.matmul(B, submeshVertices[:, 1])
               U = np.matmul(B, submeshVertices[:, 2])

               data.append(extraction.extract(S, T, U, strength))

       final_data = decrypt(majority_vote(data, length), secret)

       return final_data
```

8.9 computer eigenvectors

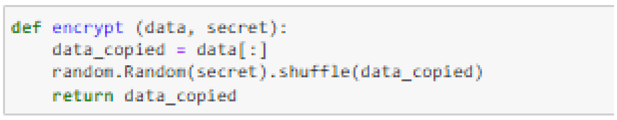


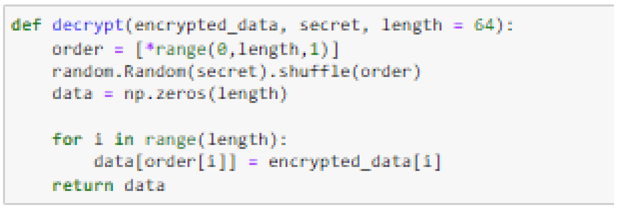
```
: def compute_eigenvectors (num_vertices, submeshes):
      submesh = submeshes.astype('int32')
      laplacian = np.zeros((num_vertices, num_vertices))
      neighbour_list = igl.adjacency_list(submesh)
      degree = []
      for i in neighbour_list:
          degree.append(len(i))

      for i in range(num_vertices):
          laplacian[i][i] = 1
          for j in neighbour_list[i]:
              laplacian[i][int(j)] = -1.0/degree[i]

      eigenValues, eigenVectors = np.linalg.eig(laplacian)

      idx = eigenValues.argsort()
      eigenValues = eigenValues[idx]
      eigenVectors = eigenVectors[:,idx]


      return np.transpose(eigenVectors)
```

8.9.1 majority vote

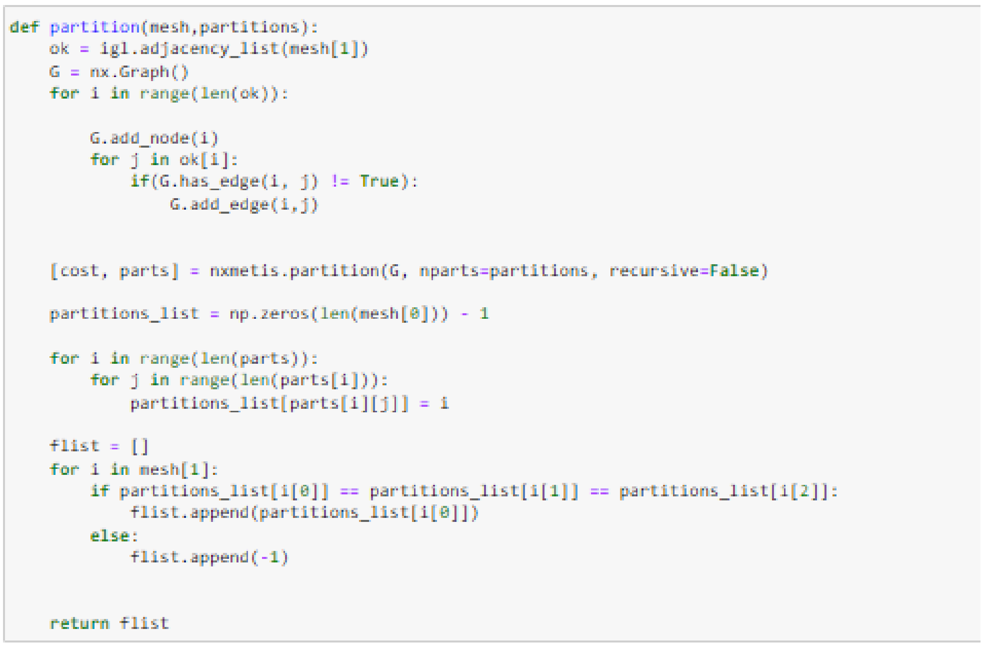

```python
def majority_vote(data, length):
    final_data = np.zeros(length)
    for i in range(length):
        sum = 0.0
        count = 0

        for d in data:
            j = i

            while j < len(d):
                sum += d[j]
                count += 1
                j += length

        if count > 0:
            final_data[i] = round(sum/count)

    return final_data
```

8.9.2 encrypt and decrypt

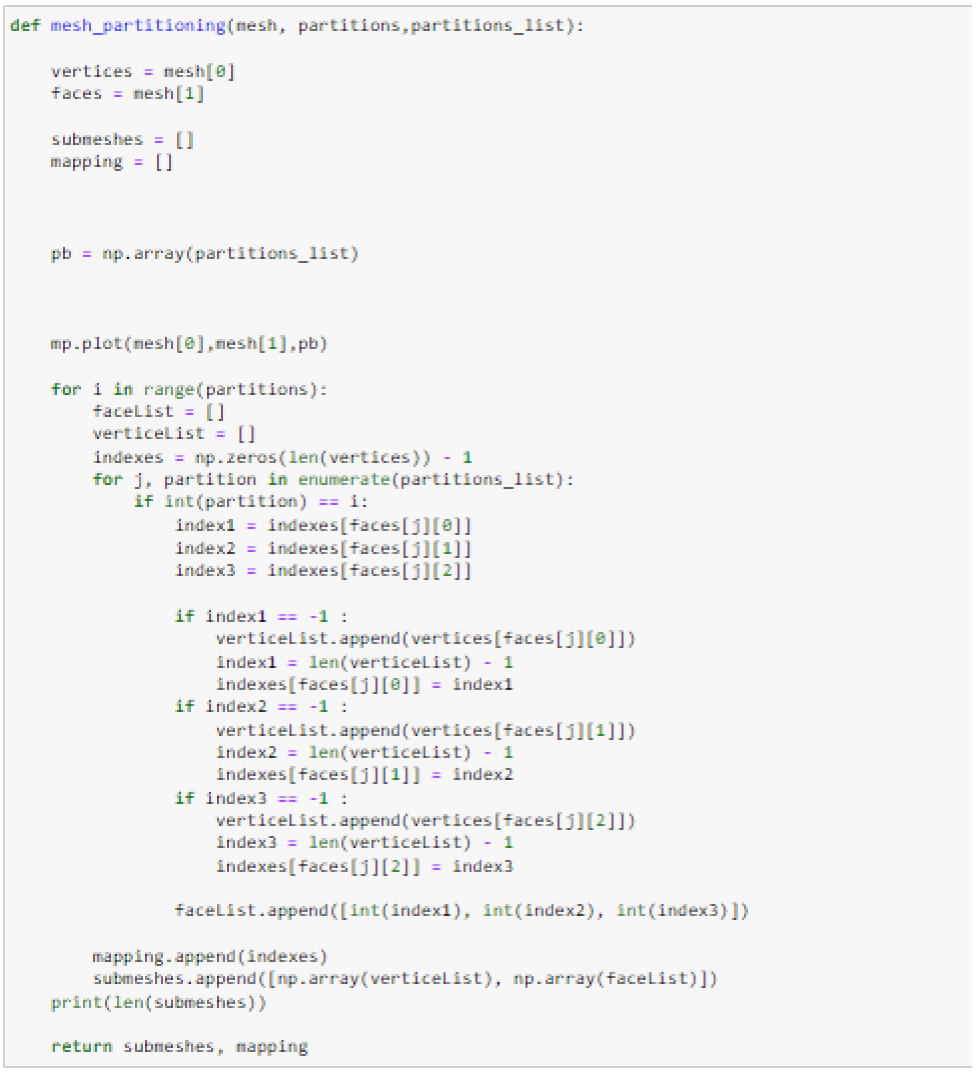

```python
def encrypt (data, secret):
    data_copied = data[:]
    random.Random(secret).shuffle(data_copied)
    return data_copied
```


```python
def decrypt(encrypted_data, secret, length = 64):
    order = [*range(0,length,1)]
    random.Random(secret).shuffle(order)
    data = np.zeros(length)

    for i in range(length):
        data[order[i]] = encrypted_data[i]
    return data
```

8.9.3 partition

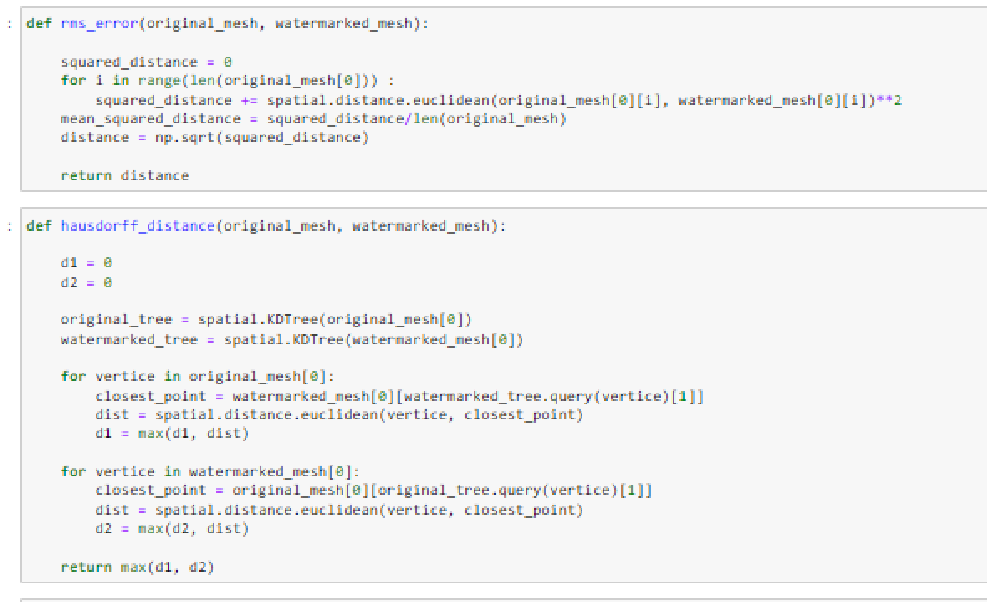

```python
def partition(mesh,partitions):
    ok = igl.adjacency_list(mesh[1])
    G = nx.Graph()
    for i in range(len(ok)):

        G.add_node(i)
        for j in ok[i]:
            if(G.has_edge(i, j) != True):
                G.add_edge(i,j)


    [cost, parts] = nxmetis.partition(G, nparts=partitions, recursive=False)

    partitions_list = np.zeros(len(mesh[0])) - 1

    for i in range(len(parts)):
        for j in range(len(parts[i])):
            partitions_list[parts[i][j]] = i

    flist = []
    for i in mesh[1]:
        if partitions_list[i[0]] == partitions_list[i[1]] == partitions_list[i[2]]:
            flist.append(partitions_list[i[0]])
        else:
            flist.append(-1)


    return flist
```

8.9.4 map partitions



```python
def mesh_partitioning(mesh, partitions,partitions_list):

    vertices = mesh[0]
    faces = mesh[1]

    submeshes = []
    mapping = []



    pb = np.array(partitions_list)



    mp.plot(mesh[0],mesh[1],pb)

    for i in range(partitions):
        faceList = []
        verticeList = []
        indexes = np.zeros(len(vertices)) - 1
        for j, partition in enumerate(partitions_list):
            if int(partition) == i:
                index1 = indexes[faces[j][0]]
                index2 = indexes[faces[j][1]]
                index3 = indexes[faces[j][2]]

                if index1 == -1 :
                    verticeList.append(vertices[faces[j][0]])
                    index1 = len(verticeList) - 1
                    indexes[faces[j][0]] = index1
                if index2 == -1 :
                    verticeList.append(vertices[faces[j][1]])
                    index2 = len(verticeList) - 1
                    indexes[faces[j][1]] = index2
                if index3 == -1 :
                    verticeList.append(vertices[faces[j][2]])
                    index3 = len(verticeList) - 1
                    indexes[faces[j][2]] = index3

                faceList.append([int(index1), int(index2), int(index3)])

        mapping.append(indexes)
        submeshes.append([np.array(verticeList), np.array(faceList)])
    print(len(submeshes))

    return submeshes, mapping
```

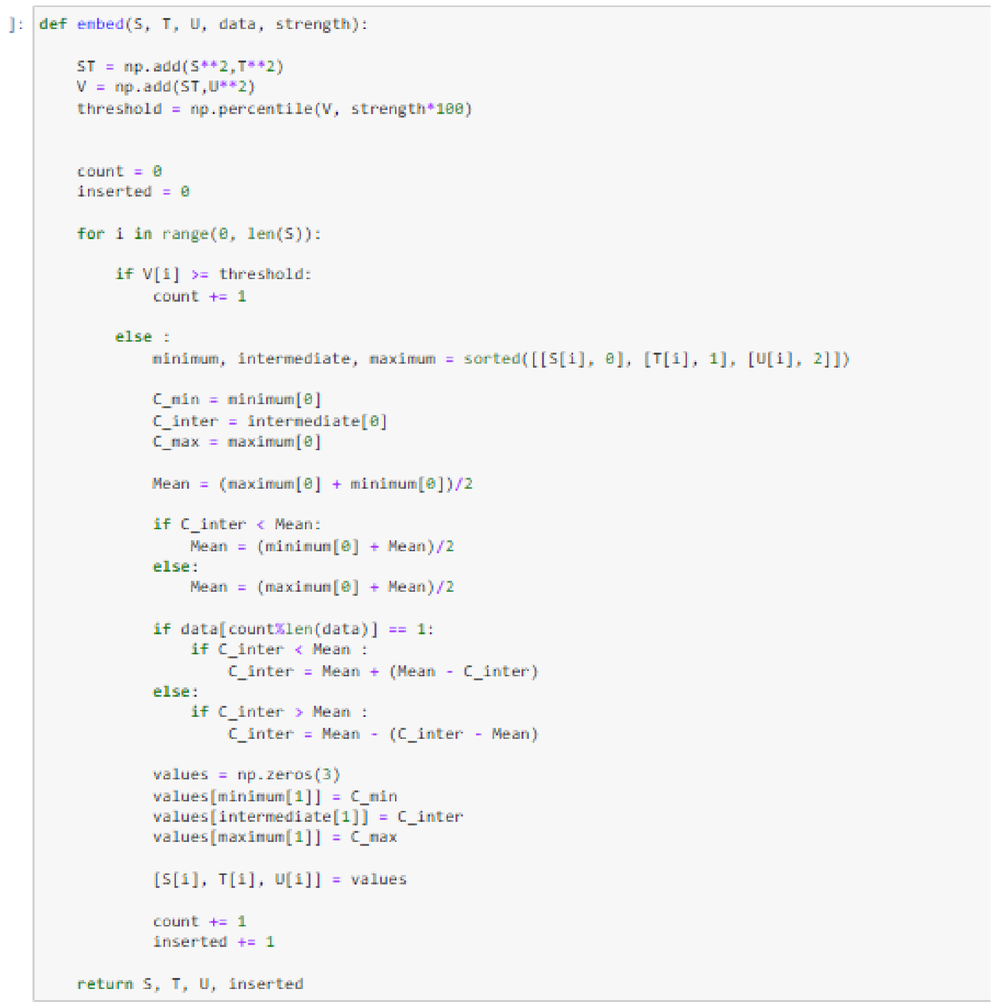
8.9.5



```python
def rms_error(original_mesh, watermarked_mesh):

    squared_distance = 0
    for i in range(len(original_mesh[0])) :
        squared_distance += spatial.distance.euclidean(original_mesh[0][i], watermarked_mesh[0][i])**2
    mean_squared_distance = squared_distance/len(original_mesh)
    distance = np.sqrt(squared_distance)

    return distance
```

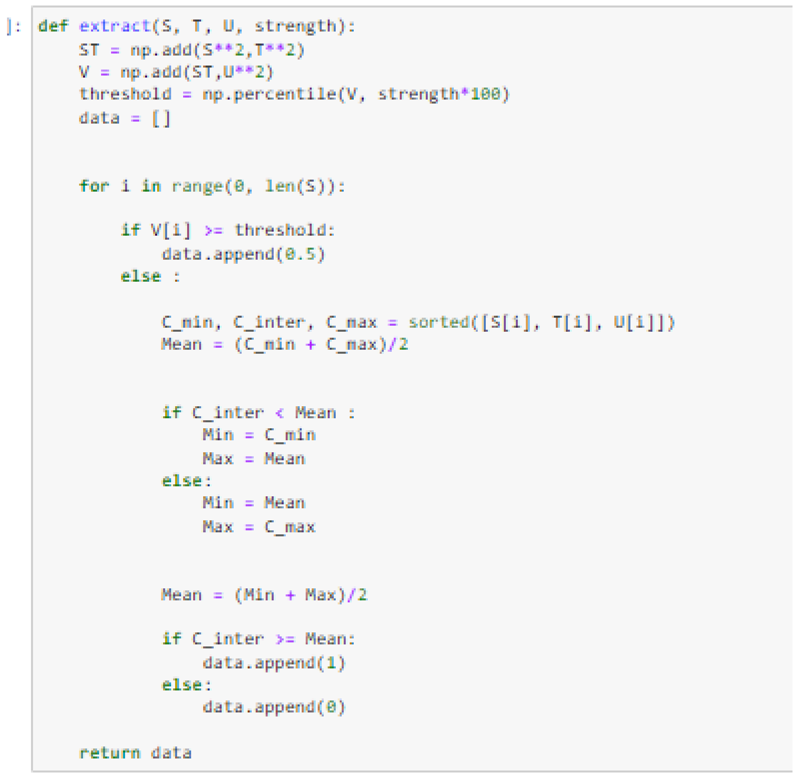
```python
def hausdorff_distance(original_mesh, watermarked_mesh):

    d1 = 0
    d2 = 0

    original_tree = spatial.KDTree(original_mesh[0])
    watermarked_tree = spatial.KDTree(watermarked_mesh[0])

    for vertice in original_mesh[0]:
        closest_point = watermarked_mesh[0][watermarked_tree.query(vertice)[1]]
        dist = spatial.distance.euclidean(vertice, closest_point)
        d1 = max(d1, dist)

    for vertice in watermarked_mesh[0]:
        closest_point = original_mesh[0][original_tree.query(vertice)[1]]
        dist = spatial.distance.euclidean(vertice, closest_point)
        d2 = max(d2, dist)

    return max(d1, d2)
```

8.9.6 embedding algorithm



```
]: def embed(S, T, U, data, strength):

       ST = np.add(S**2,T**2)
       V = np.add(ST,U**2)
       threshold = np.percentile(V, strength*100)


       count = 0
       inserted = 0

       for i in range(0, len(S)):

           if V[i] >= threshold:
               count += 1

           else :
               minimum, intermediate, maximum = sorted([[S[i], 0], [T[i], 1], [U[i], 2]])

               C_min = minimum[0]
               C_inter = intermediate[0]
               C_max = maximum[0]

               Mean = (maximum[0] + minimum[0])/2

               if C_inter < Mean:
                   Mean = (minimum[0] + Mean)/2
               else:
                   Mean = (maximum[0] + Mean)/2

               if data[count%len(data)] == 1:
                   if C_inter < Mean :
                       C_inter = Mean + (Mean - C_inter)
               else:
                   if C_inter > Mean :
                       C_inter = Mean - (C_inter - Mean)

               values = np.zeros(3)
               values[minimum[1]] = C_min
               values[intermediate[1]] = C_inter
               values[maximum[1]] = C_max

               [S[i], T[i], U[i]] = values

               count += 1
               inserted += 1

       return S, T, U, inserted
```

8.9.6 extracting algorithm



```
def extract(S, T, U, strength):
    ST = np.add(S**2,T**2)
    V = np.add(ST,U**2)
    threshold = np.percentile(V, strength*100)
    data = []


    for i in range(0, len(S)):

        if V[i] >= threshold:
            data.append(0.5)
        else :

            C_min, C_inter, C_max = sorted([S[i], T[i], U[i]])
            Mean = (C_min + C_max)/2


            if C_inter < Mean :
                Min = C_min
                Max = Mean
            else:
                Min = Mean
                Max = C_max


            Mean = (Min + Max)/2

            if C_inter >= Mean:
                data.append(1)
            else:
                data.append(0)

    return data
```