

# Surface Smoothing

Curve and surface smoothing without shrinkage  
Gabriel Taubin

**IGR202**

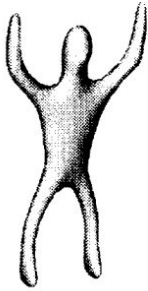
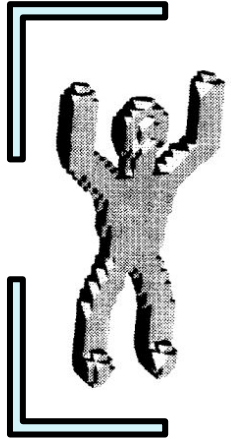
**Professors:** Amal DEV PARRAKAT  
Kiwon UM

**Student:** Giulia MANNAIOLI



**IP PARIS**

# Table of Contents



**01**

**Introduction**

**02**

**Comparison**

**03**

**Result**

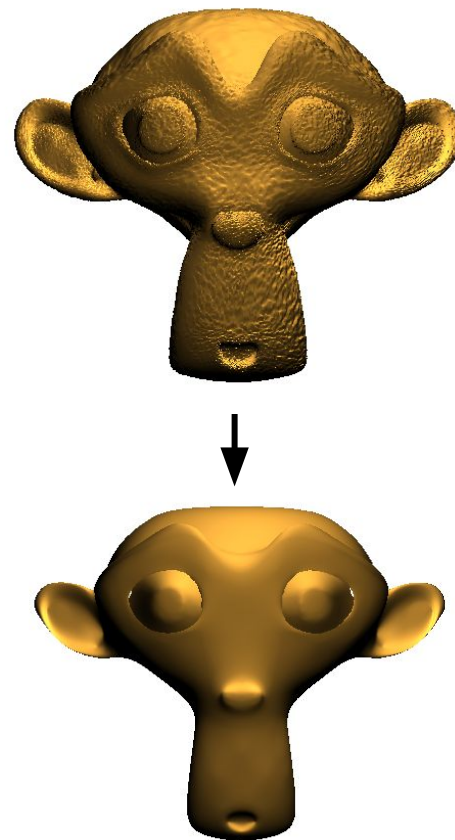
**04**

**Conclusion**

# Introduction

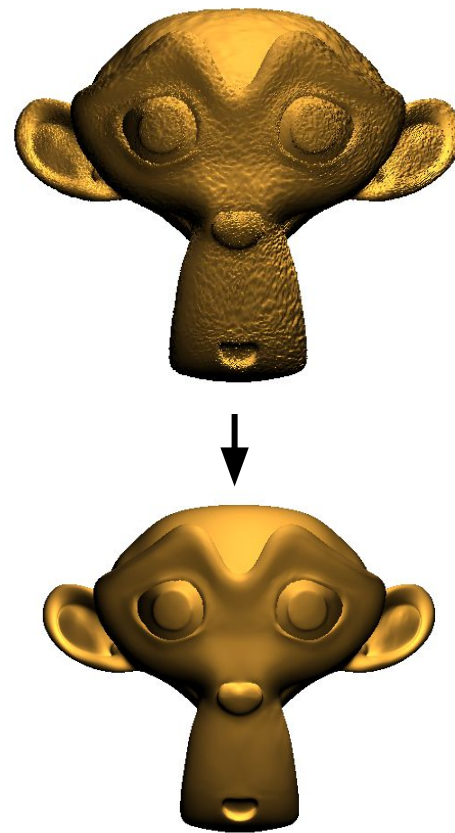
Previous smoothing methods, such as Fourier descriptors and Gaussian smoothing, suffer from shrinkage issues.

The paper shows how to define Gaussian smoothing on polyhedral surfaces of arbitrary topology, but also points out its **shrinkage problem**.



# Introduction

The **new smoothing algorithm** consists of two consecutive passes of Gaussian smoothing with alternating positive and negative scale factors to avoid the shrinkage.



# Comparison

## Gaussian

Gaussian smoothing uses a Gaussian-type filter, which has a **symmetric and continuous** weighting function.

This filter calculates the weighted average of neighboring points for each point in the mesh, **with greater weight given to closer points**.

In general, it is better suited for removing uniform and subtle noise.

## Laplacian

Laplacian smoothing uses a Laplacian-type filter, which has an **asymmetric and discontinuous** weighting function.

This filter calculates the difference between the position of the mesh point and the average of its neighbors, and then updates the position based on a weight that **is the same for all the neighbours** points.

Laplacian smoothing is better suited for removing irregular and coarser noise.

# Comparison

## Gaussian

```
int N = 25;
for (int n = 0; n < N; ++n) {

    for (unsigned int v = 0; v < _vertexPositions.size(); ++v) {

        for (auto it = trianglesOnEdge[v].begin(); it != trianglesOnEdge[v].end(); ++it) {

            if (it->second == 2) {

                float sum = 0.f;
                glm::vec3 newPosition(0.0f);

                for (auto i = neighboringVertices[v].begin(); i != neighboringVertices[v].end(); ++i) {
                    L = _vertexPositions[v] - _vertexPositions[*i];
                    float wij = exp(-(glm::dot(L, L)) / (2.0f * sigma * sigma));

                    newPosition += wij * _vertexPositions[*i];
                    sum += wij;
                }

                newPosition /= sum;
                _vertexPositions[v] = newPosition;
            }
        }
    }

    recomputePerVertexNormals();
    recomputePerVertexTextureCoordinates();
}
```

# Comparison

## Laplacian

It can cause significant shrinkage on meshes with small geometric features.

## Cotangent Laplacian

Generally, it also reduces the shrinkage problem compared to the Normal Laplacian and preserves the mesh details.

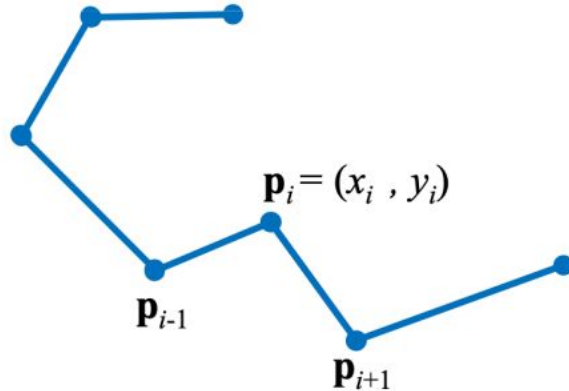
## Taubin

It is designed to reduce the shrinkage effect compared to other mesh smoothing algorithms but it may still occur.

# Comparison

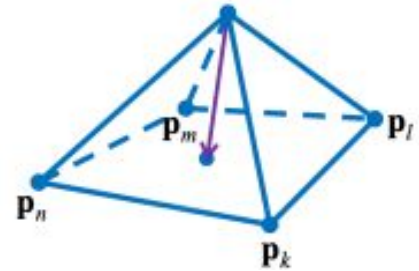
## Laplacian

$$L(\mathbf{p}_i) = \frac{1}{2}(\mathbf{p}_{i-1} - \mathbf{p}_i) + \frac{1}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i)$$



## Laplacian smoothing

$$\mathbf{p}_i \leftarrow \mathbf{p}_i + \lambda L(\mathbf{p}_i)$$

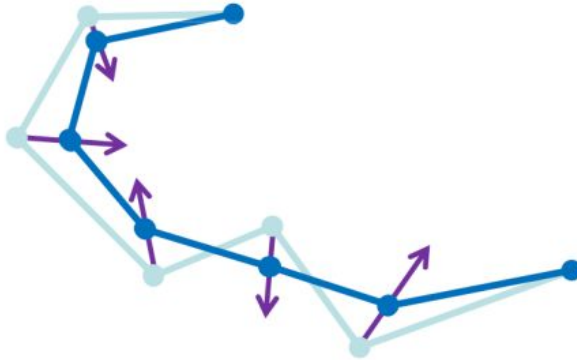




# Comparison

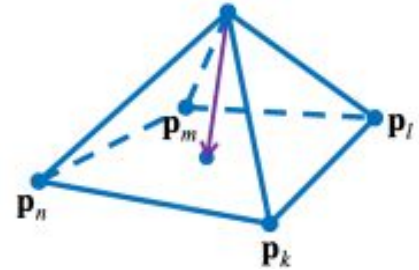
## Laplacian

$$L(\mathbf{p}_i) = \frac{1}{2}(\mathbf{p}_{i-1} - \mathbf{p}_i) + \frac{1}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i)$$



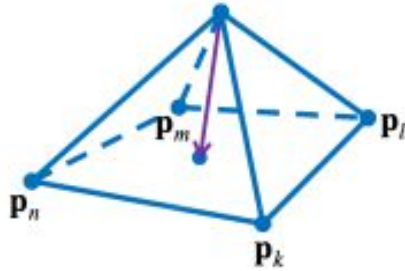
## Laplacian smoothing

$$\mathbf{p}_i \leftarrow \mathbf{p}_i + \lambda L(\mathbf{p}_i)$$



# Comparison

## Laplacian smoothing



$$\frac{1}{2}(\mathbf{p}_{i+1} + \mathbf{p}_{i-1}) - \mathbf{p}_i$$

$$\frac{1}{|N_i|} \left( \sum_{j \in N_i} \mathbf{p}_j \right) - \mathbf{p}_i$$

$$\mathbf{p}_i \leftarrow \mathbf{p}_i + \lambda L(\mathbf{p}_i)$$

# Comparison

```
int N = 50;
for (int n = 0; n < N; ++n) {

    for (unsigned int v = 0; v < _vertexPositions.size(); ++v) {

        for (auto it = trianglesOnEdge[v].begin(); it != trianglesOnEdge[v].end(); ++it) {
            unsigned int neighbor_vertex = (it->first.a == v) ? it->first.b : it->first.a;

            if (it->second == 2) {
                for (auto i = neighboringVertices[v].begin(); i != neighboringVertices[v].end(); ++i) {
                    L += _vertexPositions[*i] - _vertexPositions[v];
                }
                Ni = neighboringVertices[v].size();
                L /= Ni;

                _vertexPositions[v] += lambda * L;
            }
        }
    }

    recomputePerVertexNormals();
    recomputePerVertexTextureCoordinates();
}
```


$$\frac{1}{|N_i|} \left( \sum_{j \in N_i} \mathbf{p}_j \right) - \mathbf{p}_i$$

$$\mathbf{p}_i \leftarrow \mathbf{p}_i + \lambda L(\mathbf{p}_i)$$

# Comparison

## Laplacian

### PROBLEM


$$\frac{1}{2}(\mathbf{p}_{i+1} + \mathbf{p}_{i-1}) - \mathbf{p}_i$$


Same weight for both neighbors,  
although one is closer.

# Comparison

## Laplacian

### PROBLEM

$$\frac{1}{2}(\mathbf{p}_{i+1} + \mathbf{p}_{i-1}) - \mathbf{p}_i$$


Same weight for both neighbors,  
although one is closer.

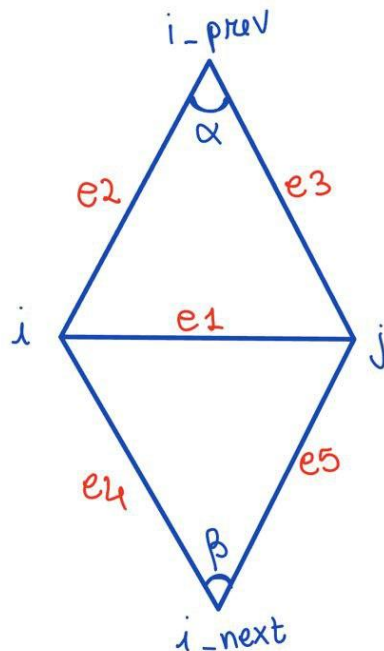
### SOLUTION

Use a weighted average to define  $\Delta$   
**Cotangent laplacian**

# Comparison

## Cotangent Laplacian

$$w_{ij} = \frac{h_{ij}^1 + h_{ij}^2}{l_{ij}} = \frac{1}{2} (\cot \alpha_{ij} + \cot \beta_{ij})$$

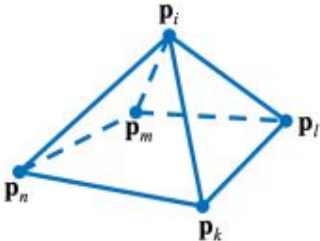


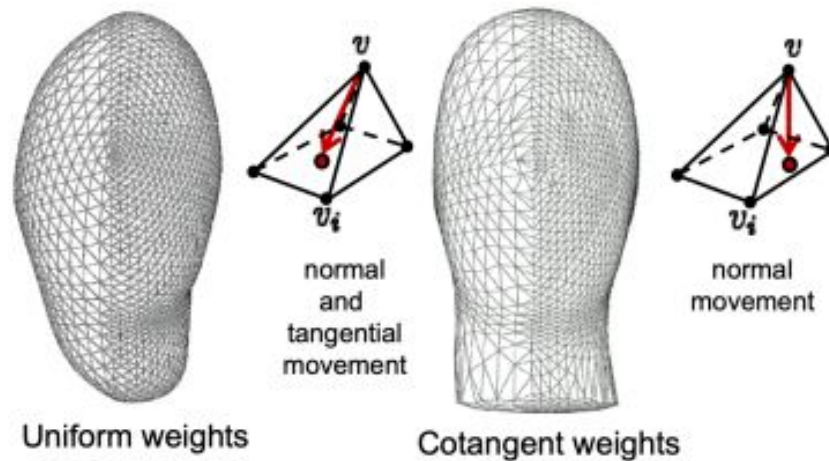
$$\cos \alpha = \frac{e_3^2 + e_2^2 - e_1^2}{2 \cdot e_3 \cdot e_2}$$

$$\cot \alpha = \frac{\cos \alpha}{\sin \alpha} = \frac{\cos \alpha}{\sqrt{1 - \cos^2 \alpha}}$$

# Comparison

## Cotangent Laplacian


$$L(\mathbf{p}_i) = \frac{1}{\sum_{j \in N_i} w_{ij}} \left( \sum_{j \in N_i} w_{ij} \mathbf{p}_j \right) - \mathbf{p}_i$$



# Comparison

```

for (unsigned int v = 0; v < _vertexPositions.size(); ++v) {
    glm::vec3 L(0.f);
    float wij = 0.f;

    for (auto it = trianglesOnEdge[v].begin(); it != trianglesOnEdge[v].end(); ++it) {
        unsigned int neighbor_vertex = (it->first.a == v) ? it->first.b : it->first.a;

        if (it->second == 2) {
            for (auto i = neighboringVertices[v].begin(); i != neighboringVertices[v].end(); ++i) {

                unsigned int i_next = (std::next(i) == neighboringVertices[v].end()) ? *(neighboringVertices[v].begin()) : *(std::next(i));
                unsigned int i_prev = (i == neighboringVertices[v].begin()) ? *(std::prev(neighboringVertices[v].end())) : *(std::prev(i));
                glm::vec3 e_ijk = _vertexPositions[i_prev] - _vertexPositions[v]; //i,j-1
                glm::vec3 e_ijm = _vertexPositions[i_next] - _vertexPositions[v]; //i,j+1
                glm::vec3 e_ij = _vertexPositions[*i] - _vertexPositions[v]; //i,j
                glm::vec3 e_jjp = _vertexPositions[*i] - _vertexPositions[i_prev]; //j,j-1
                glm::vec3 e_jjm = _vertexPositions[i_next] - _vertexPositions[i_prev]; //j,j+1
                float cot_beta = 1.f / tan(glm::angle(e_ijk, e_jjp));
                float cot_alpha = 1.f / tan(glm::angle(e_ijm, e_jjm));

                wij += 0.5f * (cot_alpha + cot_beta);
                L += wij * _vertexPositions[*i];
            }

            L = (1/wij) * L - _vertexPositions[v];

            _vertexPositions[v] += lambda * L;
        }
    }
}

recomputePerVertexNormals();
recomputePerVertexTextureCoordinates();

```

$$w_{ij} = \frac{1}{2} (\cot \alpha_{ij} + \cot \beta_{ij})$$

$$L(\mathbf{p}_i) = \frac{1}{\sum_{j \in N_i} w_{ij}} \left( \sum_{j \in N_i} w_{ij} \mathbf{p}_j \right) - \mathbf{p}_i$$

$$\mathbf{p}_i \leftarrow \mathbf{p}_i + \lambda L(\mathbf{p}_i)$$

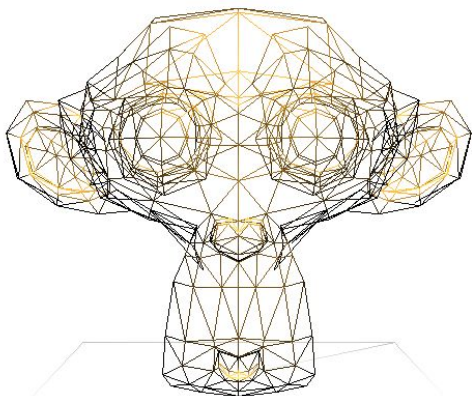


# Comparison

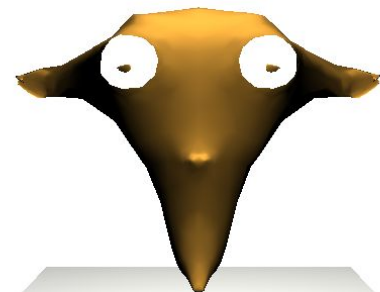
$$w_{ij} = \frac{1}{2}(\cot \alpha_{ij} + \cot \beta_{ij})$$

# Comparison

If there are triangles with a very large angle near a large region of the mesh, then the Laplace filter may have a greater impact on that region, causing a loss of detail. In general, the Laplace filter can cause a loss of detail in regions of the mesh with high curvature, but this depends on the choice of weights.



Cotangent Laplacian



Laplacian

# Comparison

## Laplacian

### PROBLEM


$$\frac{1}{2}(\mathbf{p}_{i+1} + \mathbf{p}_{i-1}) - \mathbf{p}_i$$

The laplacian smoothing is a weighted average of vertex positions to shrink the mesh **towards its center**. This produces **shrinkage**.

# Comparison

## Laplacian

### PROBLEM

$$\frac{1}{2}(\mathbf{p}_{i+1} + \mathbf{p}_{i-1}) - \mathbf{p}_i$$


The laplacian smoothing is a weighted average of vertex positions to shrink the mesh towards its center. This produces **shrinkage**.

### SOLUTION

We add a second step which is a weighted average that expands the mesh back out to its original size:

**TAUBIN SMOOTHING.**

# Comparison

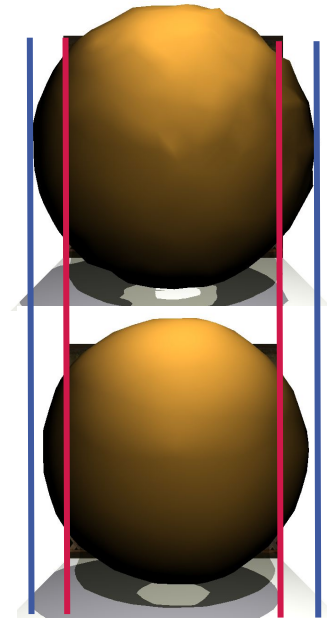
## Taubin smoothing

01

$$\mathbf{p}_i \leftarrow \mathbf{p}_i + \lambda \Delta \mathbf{p}_i$$

**Shrink**

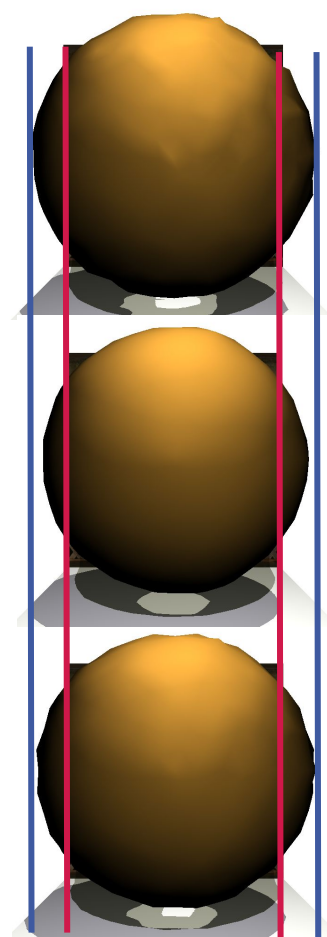
$$0 < \lambda < 1$$



# Comparison

## Taubin smoothing

- 01**     $\mathbf{p}_i \leftarrow \mathbf{p}_i + \lambda \Delta \mathbf{p}_i$     **Shrink**     $0 < \lambda < 1$
- 02**     $\mathbf{p}_i \leftarrow \mathbf{p}_i + \mu \Delta \mathbf{p}_i$     **Inflate**     $0 < \lambda < -\mu$



# Comparison

```
int N = 2;
for (int n = 0; n < N; ++n) {

    for (unsigned int v = 0; v < _vertexPositions.size(); ++v) {
        for (auto it = trianglesOnEdge[v].begin(); it != trianglesOnEdge[v].end(); ++it) {
            // it->first is an Edge object, so it->first.a returns the a vertex of the Edge
            //and it->first.b returns the b vertex
            unsigned int neighbor_vertex = (it->first.a == v) ? it->first.b : it->first.a;

            if (it->second == 2) {

                glm::vec3 delta(0.0f);
                float wij = 0.f;

                for (auto i = neighboringVertices[v].begin(); i != neighboringVertices[v].end(); ++i) {

                    wij = 1.f / neighboringVertices[*i].size();
                    delta += wij * (_vertexPositions[*i] - _vertexPositions[v]);
                }

                // shrink step
                _vertexPositions[v] += (lambda * delta);
            }
        }
    }

    recomputePerVertexNormals();
    recomputePerVertexTextureCoordinates();
}
```

$$v'_i = v_i + \lambda \Delta v_i$$

$$\Delta v_i = \sum_{j \in i^*} w_{ij} (v_j - v_i)$$

In this case, as recommended in the article we use  $w_{ij}$  equal to the inversed of the number of neighbours.

We repeat the same algorithm for the second time using  $\mu =$

$$0 < \lambda < -\mu$$

# Comparison

## Taubin smoothing

The scaling factor  $\lambda$  controls the speed at which the position of the vertices is shifted towards the weighted average of vertex positions to shrink the mesh towards its center.

The scaling factor  $\mu$  controls the speed at which the position of the vertices is shifted in the opposite direction to expand the mesh and make it back out to its original size.

A larger value of  $\lambda$  will result in a stronger smoothing effect.

A larger value of  $\mu$  will privilege the preservation of the shape.



# Comparison

## Taubin smoothing

So the normals of the vertices have to go in one direction first, and then in the opposite direction.

```
for (unsigned int v = 0; v < _vertexPositions.size(); ++v) {  
  
    float dotProduct = glm::dot(_vertexNormals[v], _lambdaVertexNormals[v]);  
  
    // confronta il prodotto interno con 0.1  
    if (dotProduct < 0.001) {  
        sum+=1;  
    }  
    else {  
        notIn += 1;  
    }  
}
```

To check this it is needed to calculate the dot product between the original vertex normal and the new vertex normal after applying the smoothing operation, and checking that it falls within a small range.

# Comparison

## Taubin smoothing

### PROBLEM

The results depends a lot on the parameters  $\lambda$  and  $\mu$  and on the  $N$  iterations.

**How to choose the parameters?**

# Comparison

## Taubin smoothing

### PROBLEM

The results depends a lot on the parameters  $\lambda$  and  $\mu$  and on the  $N$  iterations.

### How to choose the parameters?

### SOLUTION

Try and try again

OR

Use a metric to analyse the different results and choose the best one.

# Comparison

How to choose the parameters?

**SOLUTION**

```
float findOptimalSmoothingParameters() {
    float bestMSE = std::numeric_limits<float>::max();
    float optimalLambda = 0.f;
    float optimalmu = 0.f;

    // Try different smoothing parameters
    for (float lambda = 0.2f; lambda <= .9f; lambda += 0.1f) {
        for (float mu = lambda+0.05f; mu <= lambda+0.2f; mu += 0.05f) {
            // Apply smoothing with the current parameter
            taubinSmoothing(lambda, mu);

            // Calculate MSE between the smoothed mesh and the original mesh
            float mse = calculateMSE();

            // If the MSE is better than the current best, update the best MSE and the optimal smoothing parameter
            if (mse < bestMSE) {
                bestMSE = mse;
                optimalLambda = lambda;
                optimalmu = mu;
            }

            std::cout << "lambda: " << lambda << std::endl;
            std::cout << "mu: " << mu << std::endl;
            std::cout << "MSE: " << mse << std::endl;

            // Reset the mesh to the original state for the next iteration
            _vertexPositions = _noisyVertexPositions;

            recomputePerVertexNormals();
            recomputePerVertexTextureCoordinates();
        }
    }

    std::cout << "OPTILambda: " << optimalLambda << std::endl;
    std::cout << "OPTImu: " << optimalmu << std::endl;
    std::cout << "OPTIMSE: " << bestMSE << std::endl;

    return optimalLambda, optimalmu;
}
```

# Comparison

How to choose the parameters?

**SOLUTION**

```
float findOptimalSmoothingParameters() {
    float bestMSE = std::numeric_limits<float>::max();
    float optimalLambda = 0.f;
    float optimalmu = 0.f;

    // Try different smoothing parameters
    for (float lambda = 0.2f; lambda <= .9f; lambda += 0.1f) {
        for (float mu = lambda+0.05f; mu <= lambda+0.2f; mu += 0.05f) {
            // Apply smoothing with the current parameter
            taublinSmoothing(lambda, mu);

            // Calculate MSE between the smoothed mesh and the original mesh
            float mse = calculateMSE();

            // If the MSE is better than the current best, update the best MSE and the optimal smoothing parameter
            if (mse < bestMSE) {
                bestMSE = mse;
                optimalLambda = lambda;
                optimalmu = mu;
            }

            std::cout << "lambda: " << lambda << std::endl;
            std::cout << "mu: " << mu << std::endl;
            std::cout << "MSE: " << mse << std::endl;

            // Reset the mesh to the original state for the next iteration
            _vertexPositions = _noisyVertexPositions;

            recomputePerVertexNormals();
            recomputePerVertexTextureCoordinates();
        }
    }

    std::cout << "OPTILambda: " << optimalLambda << std::endl;
    std::cout << "OPTImu: " << optimalmu << std::endl;
    std::cout << "OPTIMSE: " << bestMSE << std::endl;

    return optimalLambda, optimalmu;
}
```

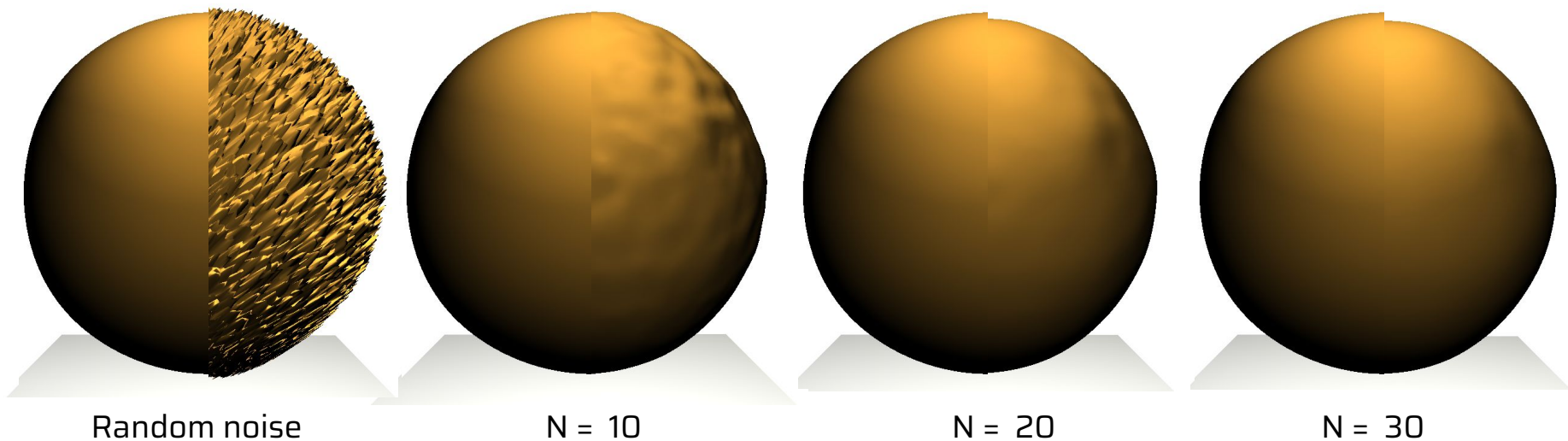
```
float calculateMSE() {
    float mse = 0.f;

    for (unsigned int v = 0; v < _vertexPositions.size(); ++v) {
        glm::vec3 diff = _originalVertexPositions[v] - _vertexPositions[v];
        mse += glm::dot(diff, diff);
    }

    return mse / _vertexPositions.size();
}
```

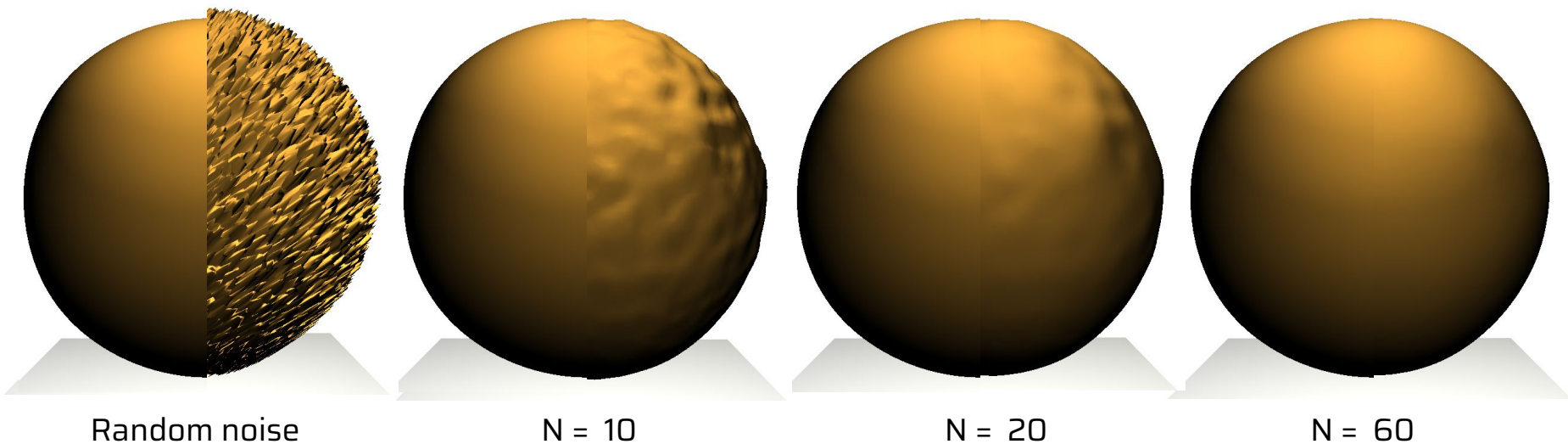
# Results

## Laplacian smoothing

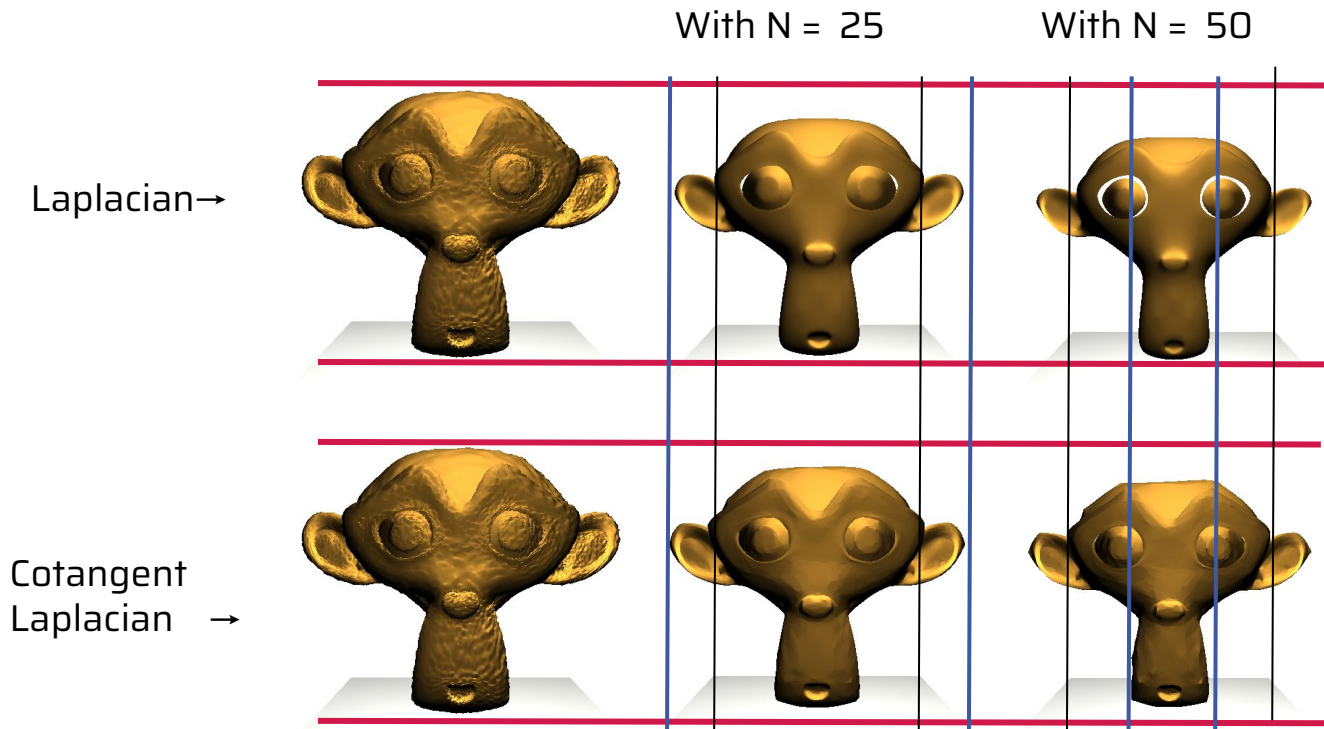


# Results

## Taubin smoothing

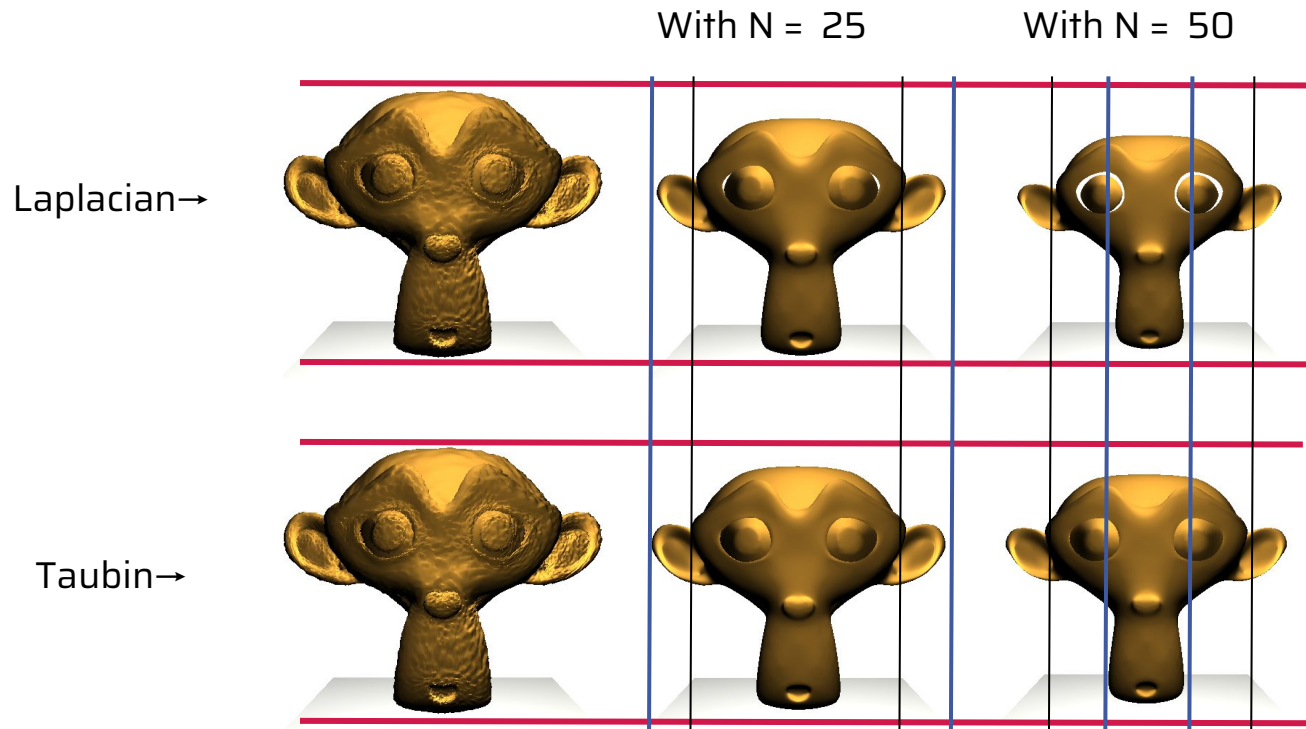


# Results





# Results

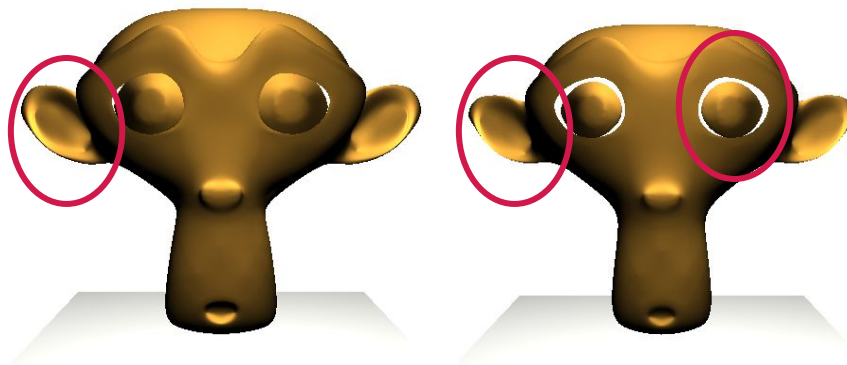


# Results

With  $N = 25$

With  $N = 50$

Laplacian →

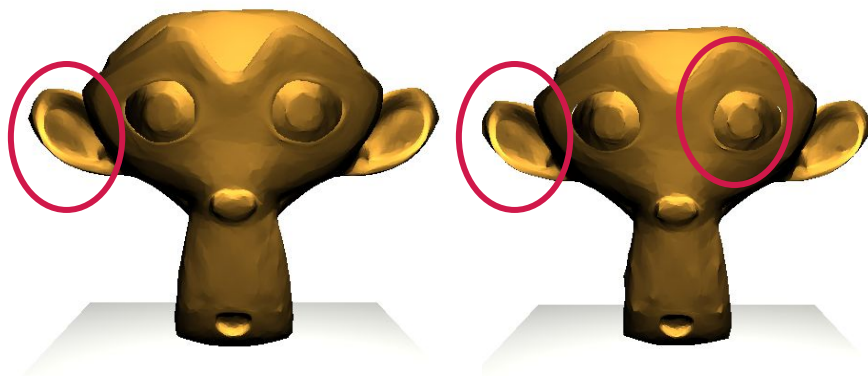


# Results

With  $N = 25$

With  $N = 50$

Cotangent  
Laplacian→

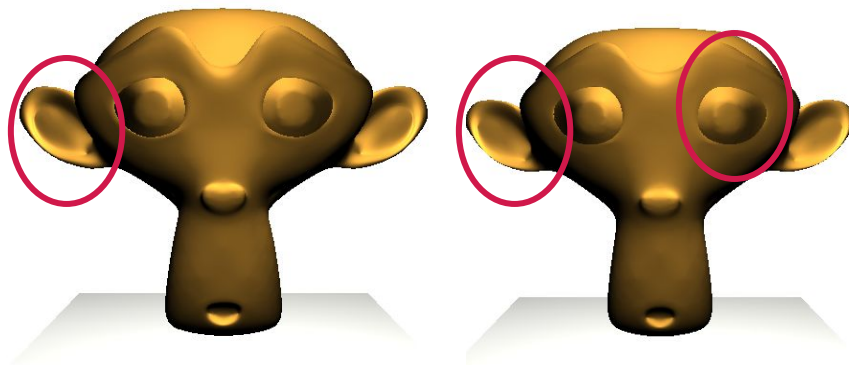


# Results

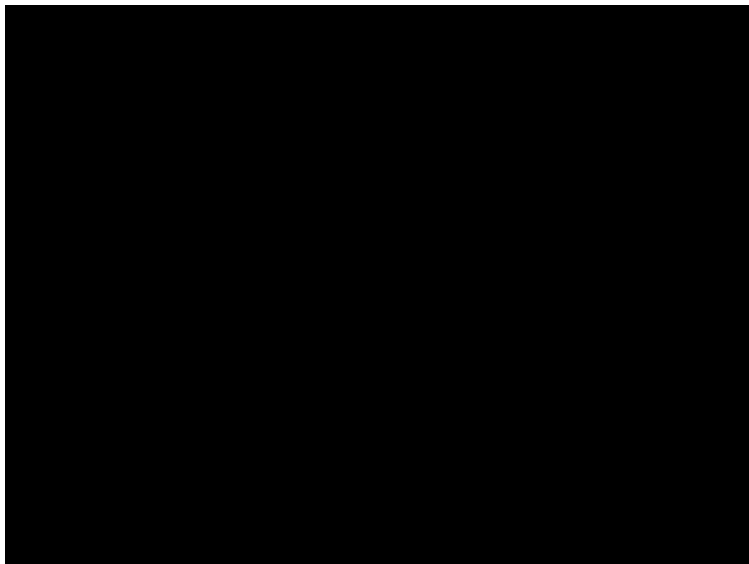
With  $N = 25$

With  $N = 50$

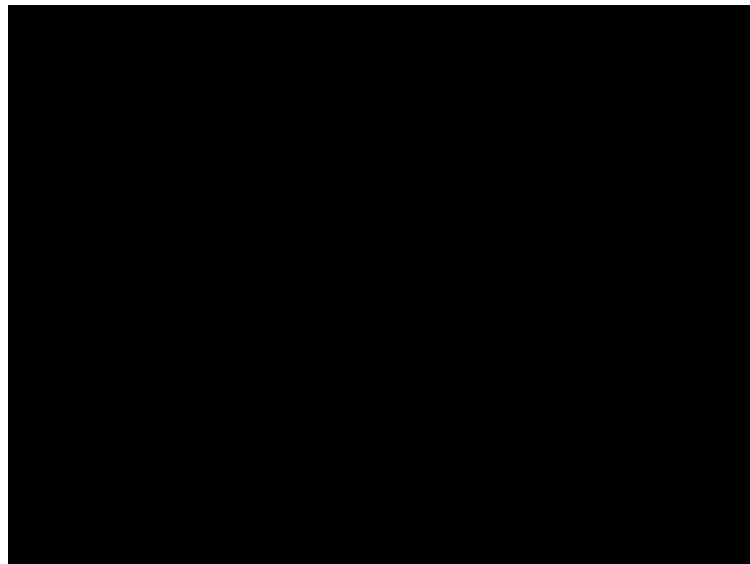
Taubin→



# Results



Applying just the first  
gaussian filter (with  $\lambda$ )



Applying the two gaussian filters  
(1st with  $\lambda$  and 2nd with  $\mu$ )

# Comparison : Conclusion

## Laplacian

Simple and fast algorithm that can cause mesh **deformation** and **loss of details**.

It can cause **significant shrinkage** on meshes with small geometric features.

**$O(n)$** , where  $n$  is the number of vertices in the mesh.

## Cotangent Laplacian

Considers the local geometry of the mesh and **reduces unwanted deformations**.

Generally, it also **reduces a bit the shrinkage problem** compared to the Normal Laplacian and preserves the mesh details.

**$O(m \log n)$** , where  $m$  is the number of edges and  $n$  is the number of vertices in the mesh.

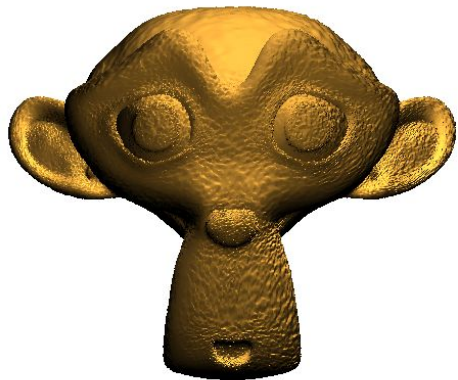
## Taubin

Iterative algorithm that improves mesh quality by **reducing irregularities**, but may cause slight deformation of the original shape.

It is designed to **reduce the shrinkage effect** but it may still occur.

**$O(kn)$** , where  $k$  is the number of iterations and  $n$  is the number of vertices in the mesh.

# Comparison : Conclusion



Laplacian



Cotangent Laplacian



Taubin

