A Mathematical Analysis of Image Mesh Generation Using Delaunay Triangulation and Image Processing Techniques

Jose´ A. Rodrigues and Beatriz Vieira

***Abstract*—** In this paper, we explore the mathemati- cal foundations and algorithms behind the generation of triangular meshes from image data, focusing on binary images that contain distinct regions, such as a white zone. The approach involves image processing techniques to binarize the input, followed by Delaunay triangulation to create a mesh fitted to the detected region. Key concepts include grid generation, spatial partitioning, and Delaunay triangulation. We explore the mathematical theories supporting these techniques and their computational realization in Python.

***Keywords*—** Delaunay Triangulation, Mesh Genera- tion, Image Processing, Binary Image Segmentation, Grid Discretization, Computational Geometry.

I. Introduction

Mesh generation is a critical process in com- putational fields such as computer graphics, nu- merical simulations, and computer vision. A mesh subdivides a domain into smaller, simple geomet- ric shapes such as triangles, allowing for efficient computation and visualization. In this paper, we generate triangular meshes and fit them to a specific region within an image using Delaunay triangula- tion.

We detail the step-by-step implementation, dis- cussing how the mathematical concepts like grid discretization, point filtering, and spatial tessellation are used. The code is implemented in Python [1] with common libraries such as OpenCV, NumPy, SciPy, and Matplotlib.

A. Motivation

The triangulation and meshing of regions based on image data are essential in many applications like computer vision, image segmentation, finite element analysis (FEA), and physics simulations. This paper explores how Delaunay triangulation is employed to create a triangular mesh that conforms to a selected region in a binary image.

II. II. MATHEMATICAL FOUNDATIONS

A. Image Binarization

An image is represented as a 2D array of pixels, each acting as a point in a Cartesian plane. Image binarization transforms a grayscale image into a binary image, segmenting it into two regions based on pixel intensities. For any pixel intensity function $I(x, y)$, binarization is defined by a threshold function T:

$$I_{binary}(x,y) = \begin{cases} 255 & if\ I(x,y) > 255 \\ 0 & otherwise \end{cases}$$

In the code, the threshold $T = 240$ is applied to isolate the white zone of the image, resulting in a binary image.

B. Structured Grid Generation

Grid generation involves discretizing the image into a lattice of points. Given image dimensions (W, H), a grid of points is formed by spacing them evenly at intervals of 10 pixels, which is chosen for simplicity. The mathematical formulation for the grid is:

$$x_i = i \times \Delta x, \quad y_i = i \times \Delta y$$

where $\Delta x = \Delta y = 10$. This discretization di- vides the image into regularly spaced points, serving as candidate vertices for triangulation.

Once the grid is generated, we aim to select points that lie within the white zone. The binary to ensure a minimum distance between the interior points and the boundary.

Image is used as a mask to filter out points out- side the white region. Mathematically, for a point

p=(x_p,y_p), it is retained if:

I_binary (x_p,y_p)=255

This forms a subset of the grid points that fit the shape of the white zone. Using set notation, we can express this filtering as:

Filtered Points =

〖{p∈Grid Points |I〗 _binary (x_p,y_p)=255}

C. Unstructured Grid Generation

The structured mesh has limitations, as it cannot be used to create meshes for all types of geometries. To give us greater flexibility, we developed an un- structured mesh approach. This approach allows us to define meshes for any limited 2D connex geome- try and provides the freedom to choose the spacing between points. Additionally, it lets us control the number of points along the boundary and within the interior. This flexibility is especially beneficial, as the boundary typically requires a higher density of points compared to the interior.

Manuscript received October 14, 2024

Jose´ A. Rodrigues (Author), CIMA and Depart- ment of Mathematics of ISEL-Higher Institute of Engineering of Lisbon

Beatriz Vieira (Author), Master in Applied Math- ematics for Industry, Instituto Superior de Engen- haria de Lisboa, Portugal

To create a mesh for a segmented area in an im- age, a set of points was selected along the boundary of the area, followed by points within the interior. Let $B = \{b_1, b_2, \dots, b_n\}$ be an ordered set of points representing the boundary of the white area in the image. The objective is to distribute n points evenly along the boundary, ensuring that it is closed. For each segment along the boundary $[b_i, b_{i+1}]$, let $L_i = |b_i, b_{i+1}|$, represent the length of the segment. The position of a new point p_i, within the segment $[b_i, b_{i+1}]$, is given by:

$$p_i = b_i + t(b_{i+1} - b_i)$$

where $t = \frac{d - current\ length}{L_i}$, with d being the target spacing between consecutive points and current length representing the accumulated length along the boundary up to the start of this segment.

To uniformly distribute points within the region defined by the boundary, Algorithm 1 is applied. In this method, Poisson disk sampling is employed

Algorithm 1 Constrained Poisson Disk Sampling for Interior Points *Input*:
Let R be the region of interest defined by the previously defined boundary, $d_{\min}$ the minimum distance between points in the interior of the region, d_{boundary} the minimum distance between points on the boundary and those in the interior, and num_samples the desired number of samples.

Output: A set of uniformly distributed points within R that satisfy the distance constraints.

1. Compute the bounding box of R to limit the sampling area.
2. Initialize an empty set of points, points= ∅:

while size of points< num_samples
Generate a candidate point $q = (x, y)$ ran- domly within the bounding box.

if $q \in R$ **and** $\min_{p \in \text{points}} |q - p| > d_{\min}$
if $\min_{b \in \text{boundary}} |q - b| > d_{\text{boundary}}$
Add q to points

end if end if
end while
return points

D. Delaunay Triangulation

Delaunay triangulation [2] is the process of defin- ing triangles such that no point lies inside the circumcircle of any triangle. For a set of points $P = \{p_1, p_2, \dots, p_n\}$on the segmented area, the Delaunay triangulation ensures that for any triangle T, its circumcircle does not enclose any point from P.

Given three points p_i, p_j, p_k, the circumcircle condition is satisfied if:

$$|p_i - p_j| \cdot |p_j - p_k| \cdot |p_k - p_i|$$

This is the primary criterion that Delaunay tri- angulation algorithms use to form well-conditioned triangles, avoiding slivers.

E. Laplacian Smoothing

Given a set of points B that define the boundary of a segmented region in an image, the set Polygon(B) is defined as follows: Supposing, as above, the finite set of points ordered such that each consecutive pair (*bi, bi (i+1)*) defines an edge of the polygon. Each edge e_i of the polygon is the line segment connecting points *bi* and *(bi+1)*, that is,

$$e_i = \{(1-t)b_i + tb_{i+1} | t \in [0,1]\},$$
$$i = 1,2, \dots, n,$$

where $b_{n+1} = b_1$ to close the loop. The polygon Polygon(B) is the region enclosed by the edges $e_1, e_2, \dots, e_n$.

Our goal is to define a set $P = \{p_1, p_2, \dots, p_n\}$ such that $P \subseteq$ Polygon(B) subject to certain restrictions to be defined.

Using the points P, we aim to define a Delaunay triangulation, ensuring that all triangles lie within Polygon(B). This will be achieved by imposing the condition that the centroid of each triangle belongs to Polygon(B):

For each triangle $t = \{p_1, p_2, p_3\}$ where p_i are the previously established points, the centroid is given by

$$centroid = \frac{p_1 + p_2 + p_3}{3}$$

The triangle is accepted if

centroid ∈ Polygon(B).

With a valid mesh in hand, the quality and appear- ance are ensured by applying Laplacian Smoothing
[3] to each triangle.
The set of points P, represents the mesh nodes and the set of triangles is denoted by

$$T = \{t_1, t_2, \dots, t_n\}$$

Additionally, a subset of these points is designated as boundary points, which should remain fixed and not be smoothed. Given a number of smoothing iterations k, the objective of Laplacian smoothing is to update each interior point's position to the average position of its neighboring points over k iterations.

For each iteration, consider each interior point *pi (i.e.*, points that are not boundary points) and identify the set of neighboring points *pi:j∈N(i),*

where $N(i)$ denotes the set of indices of neighbors of point i. Then, update the position of p_i to the average of its neighboring points as follows:

$$p_i^{new} = \frac{1}{|N(i)|} \sum_{j \in N(i)} p_j$$

Repeat this process for k iterations. At the end of each iteration, an updated set of points p_i^{new} is obtained, while boundary points remain unchanged.

To find the neighboring points of a given point p_i in the mesh, it is sufficient to examine the triangles that include as p_i a vertex. Any points that share a triangle with p_i are

considered its neighbors, as they form edges with p_i within the triangle

III. Computational Implementation

We now discuss how the mathematical concepts are implemented in the Python code. The process can be divided into four key functions:

A. Image Reading and Binarization

The function read_image reads the image, converts it to grayscale, and applies a threshold to generate a binary image. This step identifies the white region in the image:

```
def read_image(image_path):
image = cv2.imread(image_path) gray
= cv2.cvtColor(image,
cv2.COLOR_BGR2GRAY)
_, binary = cv2.threshold(gray, 240,
255, cv2.THRESH_BINARY)
return image, binary
```

B. Structured Grid Generation

```
def
    generate_mesh(image_sha
    pe): height, width =
    image_shape x =
    np.arange(0, width, 10)
    y = np.arange(0, height,
    10) xv, yv =
    np.meshgrid(x, y)
    points = np.vstack([xv.ravel(),
    yv.ravel()]).T
return points
```

The following function adjust_mesh adjusts the grid points by filtering out those that do not lie within the white zone, using the binary image as a mask.

```
        def adjust_mesh(points,
        binary_image): y, x =
        np.where(binary_image == 255) mask =
        np.zeros(binary_image.shape,
        dtype=bool) mask[y, x] = True
        adjusted_points =
        [point for point in points if
        mask[int(point[1]), int(point[0])]]
        return np.array(adjusted_points)
```

C. Unstructured Grid Generation

The first function presented here, *distribute_boundary_points*, is designed to evenly distribute a specified number of points along the perimeter of a closed boundary path. To achieve this, the function first calculates the total perimeter

```
        t = (spacing -
            current_length)/segment_length
        new_point = start + t * (end - start)
        boundary_points.append(new_point)
        current_length = 0
        start = new_point
        segment_length = np.
            linalg.norm(end - start)
    current_length += segment_length
    i += 1
    if i >= len(boundary) - 1:
        break
if np.linalg.norm(
        boundary_points[0]-
        boundary_points[-1])>
        spacing / 2:
    boundary_points.append
    (boundary_points[0])
return np.array
    (boundary_points)
```

length, then determines the spacing between points, and finally distributes the points along the boundary. The point distribution is accomplished by iterating over each boundary segment to check if the next spaced point fits within the remaining segment. If it does, the function uses linear interpolation between the start and end points of the segment to place the new point. This process repeats until the desired number of points is placed. Finally, the function checks if the last point in the list is close enough to the first point based on the calculated spacing. If not, it adds the first point again to ensure the boundary is fully closed.

```
def distribute_boundary_points
(boundary, n_points):
    boundary = np.vstack([boundary,
        boundary[0]])
    total_length = np.sum(np.sqrt
        (np.sum(np.diff
        (boundary,axis=0)**2, axis=1)))
    spacing = total_length/n_points
    boundary_points = [] current_length =
    0
    i = 0
    while len(boundary_points)<n_points:
        start = boundary[i]
        end = boundary[i + 1] segment_length
        = np.linalg.
 norm(end - start) while current_length +
segment_length >= spacing:
```

The function `poisson_disk_sampling`, which takes as arguments the minimum distance between points in the interior of the region, the minimum distance between boundary points and interior points, and the desired number of samples, will apply Algorithm 1 to set the points within the interior region

```
def poisson_disk_sampling
        (boundary, boundary_points,
        min_dist, num_samples,
        min_dist_to_boundary):
    polygon =
    Polygon(boundary) bbox
    = polygon.bounds points
    = []
    while
        len(points)<num_sample
        s: x =
        np.random.uniform
            (bbox[0],
        bbox[2]) y =
        np.random.uniform
            (bbox[1],
        bbox[3]) candidate
        = Point(x, y) if
        polygon.contains
            (candidate):
            if len(points)
            == 0 or
```

```
            np.min(cdist(
            [candidate.
            coords[0]],points))>
                min_dist: if
                np.min(cdist
                    ([candidate.coords[0]],
                    boundary_points))
                > min_dist_to_boundary:
                    points.append([x, y])
 return np.array(points)
```

Given the image processed by the function `read_image`, the `findContours` function, from the library `cv2`,identifies the contours of the white region within the image. From these contours, the boundary points are extracted, followed by the interior points. The boundary and interior points are then combined to form the final set of points representing the region of interest

```
 def put_together(image,
     n_boundary_points, n_interior_points,
     min_distance, min_dist_to_boundary):
     contours, _ = cv2.findContours
         (binary_image, cv2.RETR_EXTERNAL,
         cv2.CHAIN_APPROX_SIMPLE)
     for contour in contours: boundary =
         contour.
             reshape(-1, 2) boundary_points =
             distribute_uniform_
             boundary_points (boundary,
             n_boundary_points)
         interior_points =
             poisson_disk_sampling
             (boundary, boundary_points,
             min_distance,
             n_interior_points,
             min_dist_to_boundary)
 all_points = np.vstack ([boundary_points,
interior_points])
```

D. Delaunay Triangulation

Delaunay triangulation is performed on the fil- tered points, and the resulting triangles are plotted on the original image:

```
 def create_mesh(tri_points):
  return Delaunay(tri_points)
```

E. Laplacian Smoothing

To ensure all triangles remain within the bound- ary of the mesh, `valid_triangles` is applied

```
 valid_triangles = []
     for simplex in tri.simplices: vertices =
 all_points[simplex]

         triangle_center =
             vertices.mean(axis=0)
         if polygon.contains
             (Point(triangle_center))
             : valid_triangles.append
 (simplex)
```

The function `find_neighbors` identifies the neighboring points of each previously obtained point.

```
 def
     find_neighbors(point_i
     ndex, triangles):
     neighbors = set()
     for tri in triangles:
         if point_index in tri:
             neighbors.update(tri)
     if point_index in neighbors:
         neighbors.remove(point_index)
 return list(neighbors)
```

Considering the interior points of the mesh and the valid triangles, Laplacian Smoothing is applied to these points.

```
 def
     laplacian_smoothing(po
     ints,        triangles,
     boundary_points,
     iterations=10):
     for _ in range(iterations):
         new_points =
         np.copy(points) for i in
         range(len(points)):
             if i <
                 len(boundary_points)
                 : continue
             neighbors =
                 find_neighbors (i,
                 triangles)
             if neighbors:
                 new_points[i] =
                     np.mean(point
                     s
                     [neighbors],
                     axis=0)
 points[:] = new_points return points
```

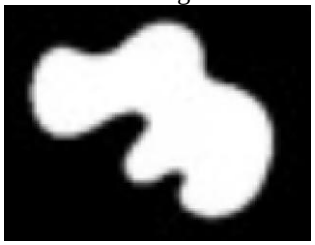
IV. RESULTS AND DISCUSSION

The result is a triangular mesh that conforms to the white zone of the binary image. Delaunay tri- angulation ensures well-shaped triangles that avoid sharp angles, which are advantageous for numerical simulations and graphical modeling.

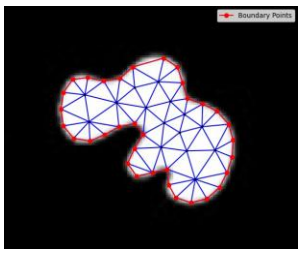
For structured grids, the grid spacing and the threshold used in binarization both influence the quality of the mesh. Structured grids have limita- tions, as they cannot be used to create meshes for all types of geometries. Furthermore, the placement of points is largely predetermined, which can result in a denser mesh. This yields finer triangulations but comes at a higher computational cost.

For unstructured grids, we can manipulate the distance between the points and the number of points, providing better control. This allows us to achieve a compromise between finer triangulations and computational cost.

Figures 1 and 2 show the results of applying Delaunay triangulation to images with irregular shapes, represented in red. The meshes provide good approximations of the shapes, successfully delineating their boundaries. Delaunay triangulation succeeds in avoiding sharp angles, resulting in well- formed meshes for both images.

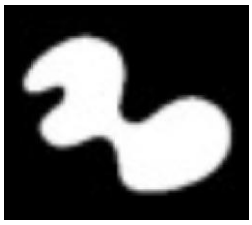


(a) Segmented image illustrating an irregular white shape

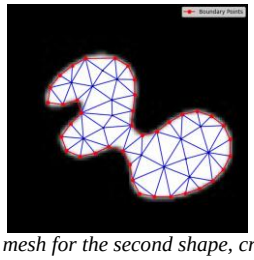


(b) Mesh generated for the irregular shape us- ing Delaunay triangula- tion.

Fig. 1: Results obtained from applying Delaunay triangulation to segmented images with irregular shapes.



(a) Segmented im- age of a second dis- tinct irregular white shape.



(b) Resulting mesh for the second shape, created with Delaunay triangulation.

Fig. 2: Additional results showcasing Delaunay triangulation applied to different irregular shapes.

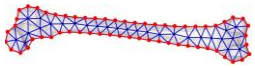


Fig. 3: Generated mesh overlay for a segmented image of a femur bone, illustrating boundary and interior triangulation.

Figure 3 shows an example of applying mesh generation to a practical scenario. A mesh was created for a human femur, making it possible to apply the finite element method (FEM) to calculate deformations.

V. CONCLUSION

This paper presents a mathematical and compu- tational approach to generating triangular meshes based on image data, focusing on Delaunay tri- angulation. The method effectively generates well- shaped triangles and fits them to a region of interest in an image. This work is applicable in fields such as computer vision, image-based modeling, and finite element analysis. There are other methods that segment and generate a grid for an image, such as `FreeFem` [4], where the grid is generated by dif- ferential equations, which is more computationally.

ACKNOWLEDGMENT

This research was partially sponsored with na- tional funds through the Fundac¸a˜o Nacional para a Cieˆncia e Tecnologia, Portugal-FCT, under projects UIDB/04674/2020 (CIMA).

https://doi.org/10.54499/UIDB/04674/2020

REFERENCES

[1] Python Software Foundation. Python Language Reference version 3.x. Available at: https://www.python.org/.

[2] Delaunay, *Sur la sphe`re vide. A la me´moire de Georges Voronoï*, Bulletin de l'Acade´mie des Sciences de l'URSS. Classe des sciences mathe´matiques et na, no. 6, 793–800, 1934

[3] Botsch, Mario, et al. *Polygon Mesh Processing*. AK Pe- ters/CRC Press, 2010. ISBN: 978-1568814261 https://doi.org/10.1201/b10688

[4] FreeFem++ Team. FreeFem++: A software for solving par- tial differential equations. Available at: https://www.freefem. org/.