Metric-Based Orthogonal Mesh Adaptation using Lp-Centroidal Voronoi Tesselation

Internship Report

Nihal Hegde

under the guidance of

DR. ARAVIND BALAN

DEPARTMENT OF AEROSPACE ENGINEERING INDIAN INSTITUTE OF SCIENCE BENGALURU - 560012

July 2024

Contents

1 Introduction

1.1 Mesh Adaptation

Advancements in computational fluid dynamics (CFD) have revolutionized aerospace design. HPC (High performance computing) hardware is changing rapidly, set to bring about a paradigm shift in the usage and implementation of algorithms and software in order to fully exploit the advancing hardware capabilities, as presented in the study by NASA [1]. However, despite the ever-increasing power of computers, reducing the complexity of numerical simulations remains a crucial issue. Mesh adaptation is one among the various methods available to reduce the simulation complexity. The aim is to control the numerical solution accuracy by modifying the discretization of the domain according to size and directional constraints.

In Computational Fluid Dynamics (CFD), unstructured mesh adaptation significantly reduces the mesh size—i.e., the number of degrees of freedom—while maintaining solution accuracy. This leads to gains in CPU time, memory usage, and storage space. Additionally, error estimates help detect physical phenomena, and meshes automatically adapt in critical regions without prior knowledge of the problem.

However, as more capable HPC hardware enables higher resolution simulations, fast, reliable mesh generation and adaptivity will become more problematic. Additionally, adaptive mesh techniques offer great potential, but have not seen widespread use due to issues related to software complexity, inadequate error estimation capabilities, and complex geometries. Therefore, there is a large scope for further improvement and research in this field, with the goal of overcoming the bottleneck it currently serves as.

2 Metrics, Definitions and Properties

Before proceeding to metric-field-based mesh adaptation, it is essential to clarify the notion of a metric and what metric spaces are. In case of meshing, the use of a metric mainly concerns the computation of lengths in different kinds of spaces: the Euclidean space, Euclidean metric spaces and Riemannian metric spaces.

Metrics are ever present in the following areas: curve and surface meshing construction is governed through control on curvature(s), meshing in general construction is governed through the control (at best) of the shape and size of elements and finally, adapted meshing construction is governed through the power, for example, of interpolation errors resulting in directives concerning shape, directionality and element size. In effect, one will explicitly rely on adequate metrics to orientate the creation or the modification of the meshing involved.

We use the following notations in the sequel. Boldface symbols, as $\mathbf{a}, \mathbf{b}, \mathbf{u}, \mathbf{v}, \mathbf{x}$,.. , denote vectors or points of $\mathbb{R}^3$. Vector coordinates are denoted by $x = (x_1, x_2, x_3)$. The natural dot product between two vectors u and v of $\mathbb{R}^3$ is:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=1}^{3} u_i v_i$$

2.1 Euclidean metric space

An Euclidean metric space $(\mathbb{R}^3, \mathcal{M})$ is a finite vector space where the dot-product is defined utilizing a symmetric definite positive tensor $\mathcal{M}$:

$$\langle \mathbf{u}, \mathbf{v} \rangle_{\mathcal{M}} = \langle \mathbf{u}, \mathcal{M}\mathbf{v} \rangle = {}^t\mathbf{u}\mathcal{M}\mathbf{v} \text{ for } \mathbf{u}, \mathbf{v} \in \mathbb{R}^3 X \mathbb{R}^3$$

The form $\mathcal{M}$ is usually written as a 3×3 matrix that is:

(i) (symmetric) $\forall (\mathbf{u}, \mathbf{v}) \in \mathbb{R}^3 X \mathbb{R}^3, \langle \mathbf{u}, \mathcal{M}\mathbf{v} \rangle = \langle \mathbf{v}, \mathcal{M}\mathbf{u} \rangle$

(ii) (positive) $\forall \mathbf{u} \in \mathbb{R}^3, \langle \mathbf{u}, \mathcal{M}\mathbf{u} \rangle \geqq 0$

(iii) (definite) $\langle \mathbf{u}, \mathcal{M}\mathbf{u} \rangle = 0 \Rightarrow \mathbf{u} = \mathbf{0}$

These properties ensure that $\mathcal{M}$ defines a dot product. And, the matrix $\mathcal{M}$ is simply called a metric tensor or a metric.

In these spaces, the length $\ell_{\mathcal{M}}$ of a segment $\mathbf{ab} = [\mathbf{a}, \mathbf{b}]$ is givenby the distance between its extremities:

$$\ell_{\mathcal{M}}(\mathbf{ab}) = d_{\mathcal{M}}(\mathbf{a}, \mathbf{b}) = \| \mathbf{ab} \|_{\mathcal{M}}$$

Note that this property is generally wrong for a general Riemannian metric space defined hereafter. In a Euclidean metric space, volumes and angles are still well represented. Given a bounded subset K of $\mathbb{R}^3$, the volume of K computed with respect to metric tensor $\mathcal{M}$ is:

$$|K|_{\mathcal{M}} = \sqrt{\det \mathcal{M}} |K|_{\mathcal{I}_3}$$

where $|K|_{\mathcal{I}_3}$ is the Euclidean volume of K. The angle between two non-zero vectors $\mathbf{u}$ and $\mathbf{v}$ is defined by the unique real-value $\theta_{\mathcal{M}} \in [0, \pi]$ verifying:

$$\cos(\theta_{\mathcal{M}}) = \frac{\langle \mathbf{u}, \mathbf{v} \rangle_{\mathcal{M}}}{\| \mathbf{u} \|_{\mathcal{M}} \|_{\mathcal{M}}}$$

We are also able to compute cross-product for metric tensor $\mathcal{M}$:

$$\mathbf{a} \times \mathcal{M}\mathbf{b} = \mathcal{M}^{\frac{1}{2}}\mathbf{a} \times \mathcal{M}^{\frac{1}{2}}\mathbf{b}$$

We deduce the following expression of the cross-product in 2D:

$$\mathbf{a} \times_{\mathcal{M}} \mathbf{b} = \sqrt{\det \mathcal{M}}(\mathbf{a} \times \mathbf{b})$$

Finally, as metric tensor $\mathcal{M}$ is symmetric, it is diagonalisable. It thus admits the following spectral decomposition:

$$\mathcal{M} = \mathcal{R}\Lambda^t\mathcal{R}$$

where $\mathcal{R}$ is an orthonormal matrix composed of the eigenvectors $(\mathbf{v}_i)_{i=1,3}$ of $\mathcal{M}$:

$$\mathcal{R} = (\mathbf{v}_1 \quad \mathbf{v}_2 \quad \mathbf{v}_3)$$

verifying ${}^t\mathcal{R}\mathcal{R} = \mathcal{R}^t\mathcal{R} = \mathcal{I}_3 . \Lambda = \mathbf{diag}\,(\lambda_i)$ is a diagonal matrix composed of the eigenvalues of $\mathcal{M}$ denoted $(\lambda_i)_{i=1,3}$, which are strictly positive.

2.1.1 Geometric interpretation

We will often refer to the geometric interpretation of a metric tensor. In the vicinity $\mathcal{V}(a)$ of point a, the set of points that are at a distance of $\mathbf{a}$, is given by:

$$\Phi_{\mathcal{M}}(\varepsilon) = \{\mathbf{x} \in \mathcal{V}(\mathbf{a}) | {}^t(\mathbf{x}-\mathbf{a})\mathcal{M}(\mathbf{x}-\mathbf{a}) = \varepsilon^2\}$$

Note that it is sufficient to describe $\Phi_{\mathcal{M}}(1)$ as $\Phi_{\mathcal{M}}(\varepsilon)$ can be deduced from $\Phi_{\mathcal{M}}(1)$ for all ε by homogeneity. Another description of $\Phi_{\mathcal{M}}(1)$ can be given using the spectral decomposition $\mathcal{M} = \mathcal{R}\Lambda^t\mathcal{R}$. In the eigenvectors frame, the initial quadratic form ${}^t(\mathbf{x}-\mathbf{a})\mathcal{M}(\mathbf{x}-\mathbf{a})$ becomes ${}^t(\tilde{\mathbf{x}}-\tilde{\mathbf{a}})\Lambda(\tilde{\mathbf{x}}-\tilde{\mathbf{a}})$. Consequently, rewriting $\Phi_{\mathcal{M}}(1)$ in this basis leads to:

$$\begin{aligned}\Phi_{\mathcal{M}}(1) &= \left\{\tilde{\mathbf{x}} \in \mathcal{V}(\tilde{\mathbf{a}}) \mid \sum_{i=1}^{n} \lambda_i(\tilde{x}_i - \tilde{a}_i)^2 = 1\right\} \\ &= \left\{\tilde{\mathbf{x}} \in \mathcal{V}(\tilde{\mathbf{a}}) \middle| \sum_{i=1}^{3} \left(\frac{\tilde{x}_i - \tilde{a}_i}{h_i}\right)^2 = 1\right\}\end{aligned}$$

The last relation defines an ellipsoid centered at a with its axes aligned with

the eigen directions of $\mathcal{M}$. Sizes along these directions are given by $h_i = \lambda_i^{-\frac{1}{2}}$. We denote by $\mathcal{B}_{\mathcal{M}}$ this ellipsoid depicted in Figure2.2. In the sequel, the set M(1) is called the unit ball of $\mathcal{M}$.

2.1.2 Natural metric mapping

The last information handled by a metric tensor M is the definition of an application that maps the unit ball $\mathcal{B}_{\mathcal{I}_3}$ of identity metric $\mathcal{I}_3$ onto the unit ball $\mathcal{B}_{\mathcal{M}}$ of $\mathcal{M}$. We introduce $\mathcal{M}^{-1/2}$ defined by the spectral decomposition:

$$\mathcal{M}^{-\frac{1}{2}} = \mathcal{R}\Lambda^{-\frac{1}{2}}{}^t\mathcal{R} \text{ where } \Lambda^{-\frac{1}{2}} = \operatorname{diag}\left(\lambda_i^{-\frac{1}{2}}\right) = \begin{pmatrix} \lambda_1^{-\frac{1}{2}} & 0 & 0 \\ 0 & \lambda_2^{-\frac{1}{2}} & 0 \\ 0 & 0 & \lambda_3^{-\frac{1}{2}} \end{pmatrix}$$

This decomposition exists because $\mathcal{M}$ is definite positive. Then this mapping is given by the application

$$\begin{aligned} \mathcal{M}^{-\frac{1}{2}}: \mathbb{R}^3 &\longmapsto \mathbb{R}^3 \\ \mathbf{x} &\longrightarrow \mathcal{M}^{-\frac{1}{2}}\mathbf{x} \end{aligned}$$

This mapping provides another description of the ellipsoid $\mathcal{B}_{\mathcal{M}}$:

$$\mathcal{B}_{\mathcal{M}} = \left\{\mathcal{M}^{-\frac{1}{2}}\mathbf{x} \middle| \ \| \mathbf{x} \|_2 = 1\right\}$$

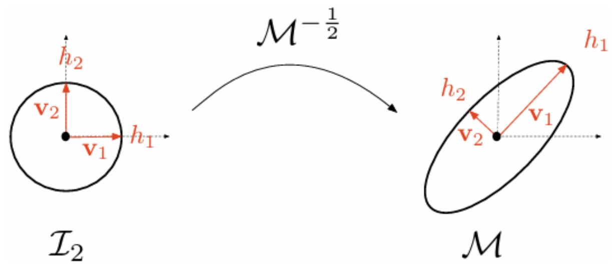


Figure 1: Metric mapping

2.2 Riemannian metric space

In differential geometry, a Riemannian manifold or Riemannian space $(\mathrm{M}, \mathcal{M})$ is a smooth manifold M in which each tangent space is equipped with a dot product defined by a metric tensor $\mathcal{M}$, a Riemannian metric, in a manner which varies smoothly from point to point. In other words, a Riemannian manifold is a smooth manifold in which the tangent space T_aM at each point a is a finite-dimensional Euclidean metric space $(T_aM, \mathcal{M}(\mathbf{a}))$.

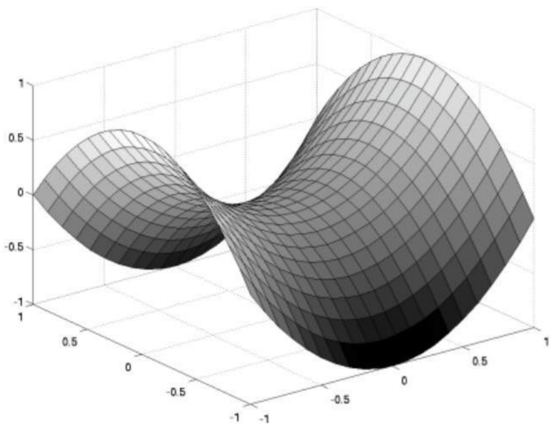
Even if no global definition of the scalar product exists, this allows one to define various geometric notions on a Riemannian manifold such as angles, lengths of curves, areas (or volumes), curvature, gradients of functions and divergence of vector fields. For instance, the distance between two points x and y is given by the length of the curve which locally joins these points along the shortest path: the geodesic. Indeed, Riemannian manifolds are usually curved.

Now let us extend the notions of length, volume and angle defined in Section 2.1 for Euclidean metric spaces to Riemannian metric spaces which will be the main operations performed by the mesher in such spaces. Fortunately, these notions can be easily derived in the context of meshing because we are not interested in evaluating these quantities on the Riemannian manifold. Indeed, as regards edge length computation, we do not want to compute the distance between two points which requires to find the shortest path on the curved manifold between these two points and to compute the length of the geodesic, but to compute the length of the path between these two points defined by the straight line parameterization, i.e., the segment representing the edge between these two points in the parametrization space which is our computational domain. To take into account the variation of the metric along the edge, the edge length is evaluated with an integral formula.

Definition 2.2.1. In Riemannian metric space $\mathbf{M} = (\mathcal{M}(\mathbf{x}))_{\mathbf{x}\in\Omega}$, the length of edge $\mathbf{ab}$ is computed using the straight line parameterization in domain $\Omega, \gamma(t) = \mathbf{a} + t\mathbf{ab}$, where $t \in [0,1]$:

$$\ell_{\mathcal{M}}(\mathbf{ab}) = \int_0^1 \|\gamma'(t)\|_{\mathcal{M}} \mathrm{d}t = \int_0^1 \sqrt{t_{\mathbf{ab}\mathcal{M}}(\mathbf{a} + t\mathbf{ab})\mathbf{ab}}\mathrm{d}t$$

Figure 4 depicts iso-values of segment length from the origin for different Riemannian metric spaces. The plotted function is $\ell_{\mathcal{M}}(\mathbf{ox})$ where $\mathbf{o}$ is the origin of the plane. The iso-values are isotropic for the Euclidean space. They are anisotropic in



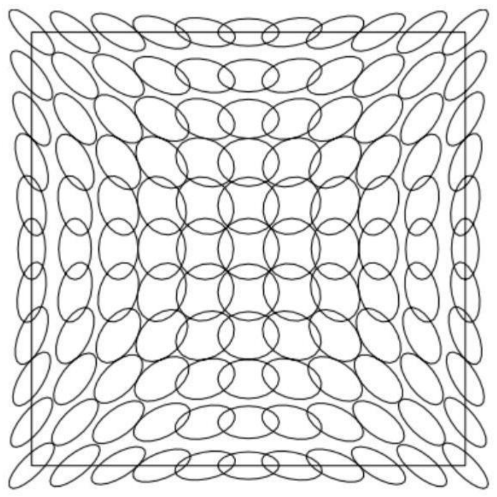


Figure 2: Left, example of a Cartesian surface embedded in $\mathbb{R}^3$. Right, geometric visualization of a Riemannian metric space $(\mathcal{M}(\mathbf{x}))_{\mathbf{x}\in[0,1]\times[0,1]}$ associated with this surface. At some points x of the domain, the unit ball of $\mathcal{M}(\mathrm{x})$ is drawn.

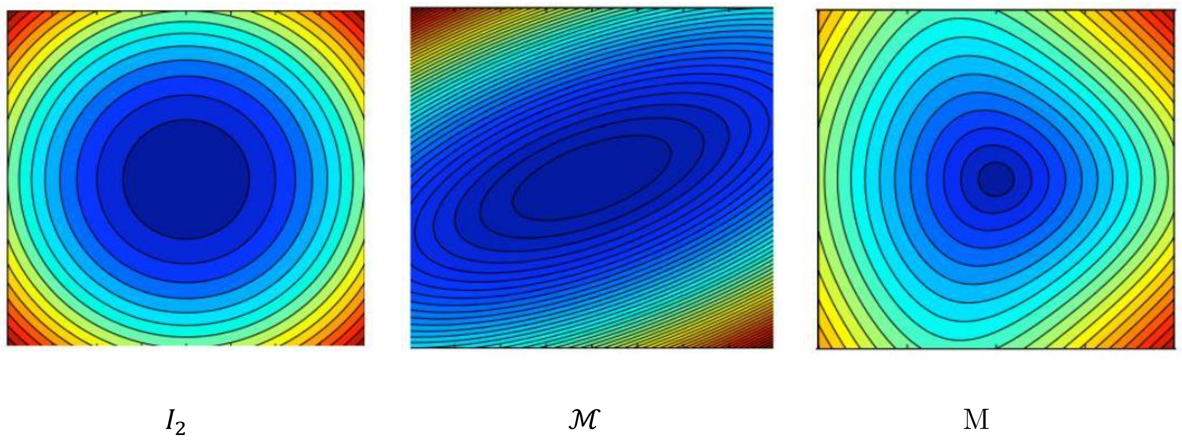


Figure 3: Iso-values of the function $f(\mathbf{x}) = \ell_{\mathcal{M}}(\mathbf{ox})$ where $\mathbf{o}$ is the origin, i.e., segment length issued from the origin, for different Riemannian metric spaces. Left, in the canonical Euclidean space $([-1,1]\times[-1,1], I_2)$, middle, in an Euclidean metric space $([-1,1]\times[-1,1], \mathcal{M})$ with $\mathcal{M}$ constant and, right, in a Riemannian metric space $(\mathcal{M}(\mathbf{x}))_{\mathbf{x}\in[-1,1]^2}$ with a varying metric tensor field.

the case of an Euclidean metric space defined by $\mathcal{M}$. The two principal directions of $\mathcal{M}$ clearly appear. In the case of a Riemannian metric space $(\mathcal{M}(\mathbf{x}))_{\mathbf{x}\in\Omega}$, all previous symmetries are lost.

Definition 2.2.2. The angle between two edges $\mathbf{pq}$ and $\mathbf{pr}$ of Ω in Riemannian metric space $(\mathcal{M}(\mathbf{x}))_{\mathbf{x}\in\Omega}$ is defined by the unique real-value $\theta \in [0,\pi]$ verifying:

$$\cos(\theta) = \frac{\langle \mathbf{pq}, \mathbf{pr} \rangle_{\mathcal{M}(\mathbf{p})}}{\| \mathbf{pq} \|_{\mathcal{M}(\mathbf{pp}}^{\|\mathbf{pr}\|_{\mathcal{M}(\mathbf{p})}}}$$

3 Lp – Centroidal Voronoi Tesselation

3.1 Voronoi diagrams and Delaunay triangulations

Given is a set of n unique points P in one level

$$P = (p_1, p_2, p_3, \cdots, p_n)$$

The Voronoi diagram of P is defined as subdividing the level into n cells. For each point in P, a cell is created with the property that a point q is only in the cell belonging to p_i if:

$$\text{dist}(q, p_i) < \text{dist}(q, p_j)$$

for every point $p_j \in P$ with $j \neq i$.
So, the Voronoi edges are constructed by creating the perpendicular bisector of the line between two points respectively and blending them.

The *Delaunay triangulation* of P is then created by connecting the points of all neighboring Voronoi cells. Therefore, Delaunay triangulations can be said to be the geometric dual of Voronoi diagrams.

3.2 Lloyd's relaxation

Llyod's algorithm is a useful algorithm related to Voronoi diagrams. The algorithm consists of repeatedly alternating between constructing Voronoi diagrams and finding the centroids (i.e. center of mass) of each cell [2]. At each iteration, the algorithm spaces the points apart and produces more homogeneous Voronoi cells.

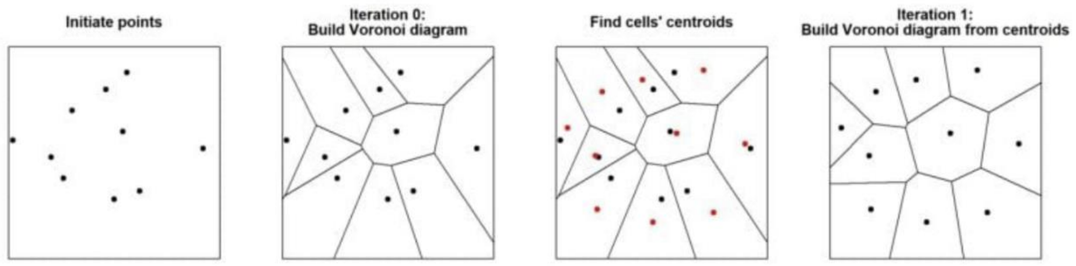


Figure 5: Steps in the Lloyd's relaxation process

The shapes of the Voronoi cells become as close as possible to circles. It can be shown that Lloyd's algorithm minimizes an energy functional equal to the sum of the moments of inertia of the Voronoi cells [3]. Some authors have used the limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) optimization procedure in order to minimize the energy functional [3]. LBFGS has the advantage of converging faster than the traditional fixed-point iteration process [3].

This minimization is performed in the Euclidean space, using a p-norm of 2, resulting in circular shaped Voronoi cells. Bruno and Levy [4] built on this concept, introducing optimization in a higher p norm, with the objective of obtaining quad-shaped elements.

3.3 Bruno and Levy's Lp – CVT

Bruno and Levy introduce an Lp-Centroidal Voronoi Tessellation (Lp-CVT), a generalization of CVT that minimizes a higher-order moment of the coordinates on the Voronoi cells. This generalization allows for aligning the axes of the Voronoi cells with a predefined background tensor field (anisotropy), which is basically the metric field, as explained earlier in the report. Lp-CVT is computed by a quasi-Newton optimization framework (LBFGS), based on closed-form derivations of the objective function and its gradient.

L_p Centroidal Voronoi Tesselation (L_p-CVT) is defined as the minimizer of the L_p-CVT objective function F_{L_p}, obtained by injecting both the anisotropy term and the L_p norm into CVT energy:

$$F_{L_p}(\mathbf{X}) = \sum_i \int_{\Omega_i \cap \boldsymbol{\Omega}} \|\mathbf{M}_\mathbf{y}[\mathbf{y} - \mathbf{x}_i]\|_p^p d\mathbf{y}$$

where $\|\cdot\|_p$ denotes the L_p norm $\left(\| \mathbf{V} \|_p = \sqrt[p]{|x|^p + |y|^p + |z|^p} \text{ and } \| \mathbf{V} \|_p^p = |x|^p + |y|^p + |z|^p\right)$. For an even value of p, $\| \mathbf{V} \|_p^p = x^p + y^p + z^p$. The domain $\boldsymbol{\Omega}$ is a surface $\mathcal{S}$.

If the anisotropy is defined as a symmetric tensor field $\mathbf{G}_\mathbf{y}$, the matrix $\mathbf{M}_\mathbf{y}$ is obtained from the SVD of $\mathbf{G}_\mathbf{y}$: $\mathbf{M}_\mathbf{y} = \boldsymbol{\Sigma}^{1/2}\mathbf{W}$ and $\mathbf{G}_\mathbf{y} = \mathbf{W}^t\boldsymbol{\Sigma}\mathbf{W} = \mathbf{M}_\mathbf{y}^t\mathbf{M}_\mathbf{y}$ (the columns of $\mathbf{M}_\mathbf{y}$ are the axes of the anisotropy ellipsoid).

The advantage of using a higher p-norm with the anisotropic term is that:

- The elements are aligned with the metric field
- Right angled corners in elements are forced due to energy function distances being measured in a higher p-norm

The right-angled corners of the elements are forced as distances in L_p-norm are determined by

$$\| \mathbf{y} - \mathbf{x} \|_p^p = |y_1 - x_1|^p + |y_2 - x_2|^p$$

and unit circles in distance measurements of different norms are given by:

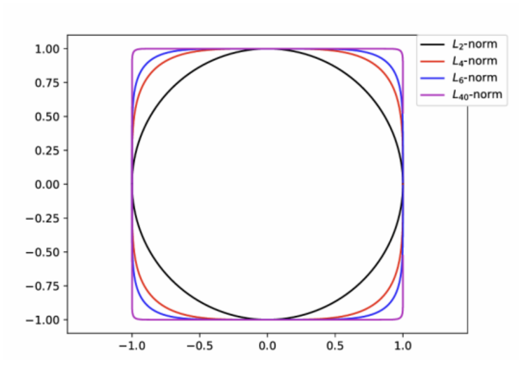


Figure 6: Unit circles for a range of p-norms

Therefore, in a higher p – norm, during energy minimization, the energy function will be satisfactorily minimized only when the Voronoi diagrams approach the shape of a square, naturally resulting in right angled elements.

3.4 Quasi – Newton methods (L-BFGS)

Lloyd's method is inefficient for minimizing CVT energy due to its linear convergence, making it slow for large numbers of sites, as given in the paper [5]. Newton-like methods are faster but computationally expensive, as they require calculating the Hessian at each step. Quasi-Newton methods like L-BFGS are preferred since they only need the energy and gradient, approximating the Hessian for better computational efficiency. However, BFGS has high space requirements for storing the dense Hessian matrix. This issue is addressed by the limited-memory BFGS method (L-BFGS), which is more space-efficient.

The user specifies the number M of BFGS corrections that are to be kept, and provides a sparse symmetric and positive definite $\widetilde{H}_0$, which approximates the inverse Hessian of f. During the first M iterations the method is identical to the BFGS method. For $k > M, \widetilde{H}_k$ is obtained by applying M BFGS updates to $\widetilde{H}_0$ using information from M previous iterations."

Here $f(x)$ is the objective function and g is the gradient of f. Let x_k denote the iterate at the k-th iteration and $s_k = x_{k+1} - x_k, y_k = g_{k+1} - g_k$. The inverse BFGS formula is $\widetilde{H}_{k+1} = V_k^T \widetilde{H}_k V_k + \rho_k s_k s_k^T$, where $\rho_k = 1/\left(y_k^T s_k\right)$ and $V_k = \mathrm{I} - \rho_k y_k s_k^T$. Typically M is set as $3 \sim 20$. The explicit formula of $\widetilde{H}_{k+1}$ is:

$$\begin{aligned}
\widetilde{H}_{k+1} =& \left(V_k^T \cdots V_{k-\widehat{M}}^T\right) \widetilde{H}_0 (V_{k-\widehat{M}} \cdots V_k) \\
&+ \rho_{k-\widehat{M}} \left(V_k^T \cdots V_{k-\widehat{M}+1}^T\right) s_{k-\widehat{M}} s_{k-\widehat{M}}^T (V_{k-\widehat{M}+1} \cdots V_k) \\
&+ \rho_{k-\widehat{M}+1} \left(V_k^T \cdots V_{k-\widehat{M}+2}^T\right) s_{k-\widehat{M}+1} s_{k-\widehat{M}+1}^T (V_{k-\widehat{M}+2} \cdots V_k) \\
&\ \vdots \\
&+ \rho_k s_k s_k^T,
\end{aligned}$$

4 Implementation

4.1 Brief idea

1. Discretize the boundary edges of the domain such that each edge is unit in the metric field and obtain the metric values at each vertex of the domain

2. Reconstruct the Voronoi cells at the boundaries as it is proved to provide better results than the clipped case. Cells are reconstructed by placing a point at a unit distance from the circumcenter of the boundary element, under the metric field, as represented in Figure 8.

3. Split each Voronoi cell into simplices and calculate the energy of each Voronoi cell by summing the simplex energies.

4. Find the energy gradient values at each Voronoi cell.

5. Input the energy and gradient values into the LBFGS algorithm to minimize it and specify a tolerance for the LBFGS loop.

Note: Once metric values are found at each of the vertices, the current work uses linear interpolation through barycentric coordinates to find the metric at any other point (assuming the point in question lies inside a triangle or on an edge whose vertex metrics are known).

4.2 Equations and computations

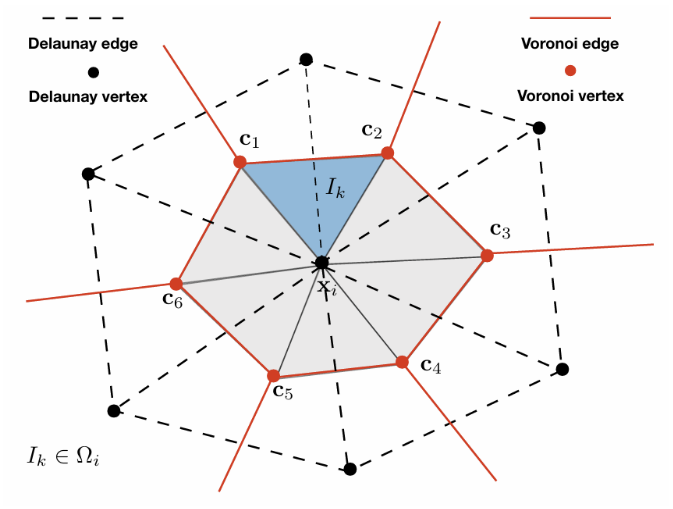


Figure 7: Sketch of a Voronoi cell Ω_i that corresponds to Delaunay vertex $\mathbf{x}_i$

Energy computation:

Each Voronoi cell is split into simplices, and the energy at each simplex (shaded region in Figure 7) is computed using the following equation, where the energy integral I_S, can be written as

$$I_S(\mathbf{x}_i) = \int_S \|M(\mathbf{y})(\mathbf{y} - \mathbf{x_i})\|_p^p d\mathbf{y} = \int_{S'} \|M(\mathbf{T}(\xi))(\mathbf{T}(\xi) - \mathbf{x_i})\|_p^p J d\xi$$

where $\mathbf{y} = \mathbf{T}(\boldsymbol{\xi})$ represents the linear transformation of local element coordinates to standard element coordinates (ξ) and J represents the determinant of the Jacobian $\left(J = \left|\frac{\partial \mathbf{y}}{\partial \xi}\right|\right)$ of the linear transformation [21].

Gradient computation:

The gradient equation computing the gradient of the energy of each Voronoi cell with respect to the node $\mathbf{x}_i$[21] in Figure 7 can be given by:

$$\frac{d}{d\mathbf{x}_k}E(\mathbf{x}_1,\dots,\mathbf{x}_n) = \frac{\partial I^{\Omega_k}(\mathbf{x}_k)}{\partial \mathbf{x}_k} + \sum_{i=1}^{n}\sum_{j=1}^{m_i}\frac{dI^{\Omega_i}(\mathbf{x}_i)}{d\mathbf{c}_{i,j}}\frac{d\mathbf{c}_{i,j}}{d\mathbf{x}_k} \qquad j = 1,\dots,n$$

The symbol m_i represents the number of Voronoi vertices that surround the i^{th} Delaunay vertex. For the Voronoi cell that is shown in Figure 6, we have that $m_i = 6$ but this may vary for each Delaunay vertex. I^{Ω_i} represents the energy of each Voronoi cell surrounding $\mathbf{x}_i$, computed by the energy equation. n is the number of vertices in the mesh. The first term on the right-hand side of the gradient equation can be written as

$$\frac{\partial I_S(\mathbf{x}_i)}{\partial \mathbf{x}_i} = \int_{S'} \frac{\partial \parallel F \parallel_p^p}{\partial F}\left[\frac{\partial M}{\partial \mathbf{T}}\frac{\partial \mathbf{T}}{\partial \mathbf{x}_i}(\mathbf{T}(\xi) - \mathbf{x}_i) + M\left(\frac{\partial \mathbf{T}}{\partial \mathbf{x_i}} - 1\right)\right]J + \parallel F \parallel_p^p \frac{\partial J}{\partial \mathbf{x}_i}\,d\xi$$

where $F = M(\mathbf{y})(\mathbf{y} - \mathbf{x_i})$. Taking again simplex S (shaded blue) in Figure 7 as an example, we see that it has two Voronoi vertices, $\mathbf{c}_{i,1}$ and $\mathbf{c}_{i,2}$. To calculate their contributions to the gradient of the energy with respect to the Delaunay vertex $\mathbf{x}_i$, we use

$$\frac{\partial I_S(\mathbf{x}_i)}{\partial \mathbf{c}_{i,j}} = \int_{S'} \frac{\partial \parallel F \parallel_p^p}{\partial F}\left[\frac{\partial M}{\partial \mathbf{T}}\frac{\partial \mathbf{T}}{\partial \mathbf{c}_{i,j}}(\mathbf{T}(\boldsymbol{\xi}) - \mathbf{x}_i) + M\left(\frac{\partial \mathbf{T}}{\partial \mathbf{c}_{i,j}}\right)\right]J + \parallel F \parallel_p^p \frac{\partial J}{\partial \mathbf{c}_{i,j}}\,d\boldsymbol{\xi}$$

$\mathbf{c}_{i,j}$ are circumcenters (Voronoi vertices), whose value depends on the coordinates of the triangle is lies inside. Therefore, we can see that most terms on the right-hand side of the gradient equation becomes zero, as for a given $\mathbf{x}_i$, only few $\frac{d\mathbf{c}_{i,j}}{d\mathbf{x}_k}$ exist. Thus, computation becomes easier. This is explained in further detail below.

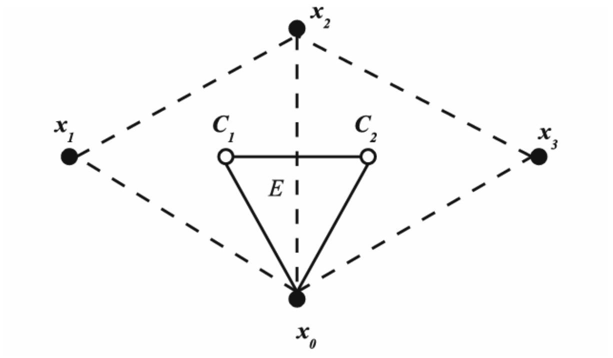


Figure 8: C_1 is the center of the circle circumscribing the Delaunay element $\boldsymbol{x}_0 - \boldsymbol{x}_1 - \boldsymbol{x}_2$

Figure 8 shows one simplex of a Voronoi cell, along with its dependent elements. C_1 and C_2 are the circumcenters of the elements (Delaunay triangles). For gradient calculations, a for loop iterates over each simplex, calculating the contribution of each simplex to the gradients of each vertex in the mesh. For the simplex E given in Figure 8, it's contributions to the vertex it belongs to, and the surrounding vertices are given below. It is assumed that $\boldsymbol{x_0}, \boldsymbol{x}_1, \boldsymbol{x}_2$ and $\boldsymbol{x}_3$ are non-boundary mesh vertices. A $+=$ sign is used because there will be contributions from other elements as well.

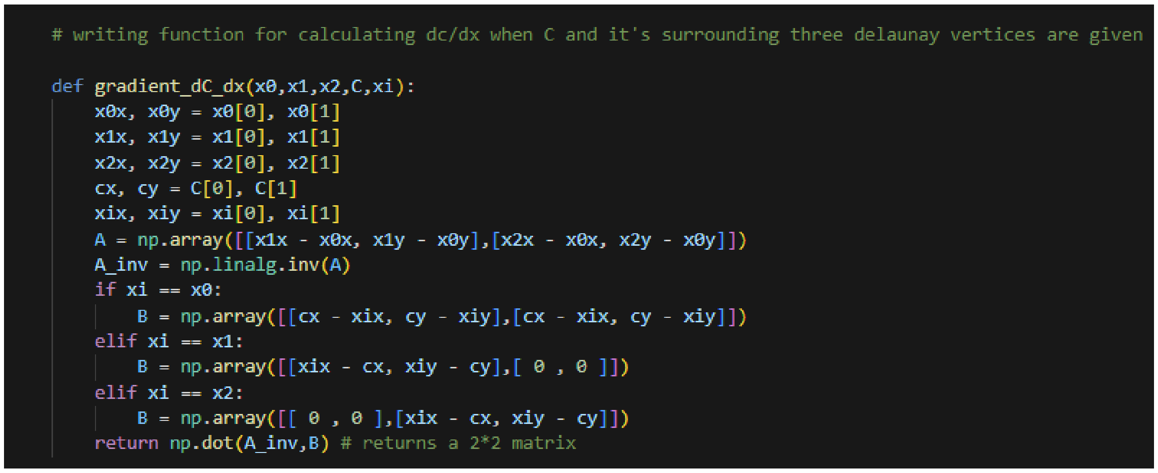
$$\frac{\mathrm{d}F}{\mathrm{d}\boldsymbol{x}_0} += \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{x}_0} + \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_1}\frac{\mathrm{d}\boldsymbol{C}_1}{\mathrm{d}\boldsymbol{x}_0} + \frac{\partial \boldsymbol{I}^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_2}\frac{\mathrm{d}\boldsymbol{C}_2}{\mathrm{d}\boldsymbol{x}_0}$$
$$\frac{\mathrm{d}F}{\mathrm{d}\boldsymbol{x}_1} += \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_1}\frac{\mathrm{d}\boldsymbol{C}_1}{\mathrm{d}\boldsymbol{x}_1}$$
$$\frac{\mathrm{d}F}{\mathrm{d}\boldsymbol{x}_2} += \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_1}\frac{\mathrm{d}\boldsymbol{C}_1}{\mathrm{d}\boldsymbol{x}_2} + \frac{\partial \boldsymbol{I}^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_2}\frac{\mathrm{d}\boldsymbol{C}_2}{\mathrm{d}\boldsymbol{x}_2}$$
$$\frac{\mathrm{d}F}{\mathrm{d}\boldsymbol{x}_3} += \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_2}\frac{\mathrm{d}\boldsymbol{C}_2}{\mathrm{d}\boldsymbol{x}_3}$$

If $\mathbf{C}$ represents the general circumcenter, and $\mathbf{x}_0,\mathbf{x}_1,\mathbf{x}_2$ represent the mesh vertices (Delaunay triangles vertices whose circumcenter is $\mathbf{C}$) then the relation between them can be given as follows:

$$\mathbf{C} = \mathbf{A}^{-1}\mathbf{B} \text{ where } \mathbf{A} = \begin{pmatrix} [\mathbf{x}_1 - \mathbf{x}_0]^t \\ [\mathbf{x}_2 - \mathbf{x}_0]^t \end{pmatrix} ; \mathbf{B} = \frac{1}{2}\begin{pmatrix} \mathbf{x}_1^2 - \mathbf{x}_0^2 \\ \mathbf{x}_2^2 - \mathbf{x}_0^2 \end{pmatrix}$$

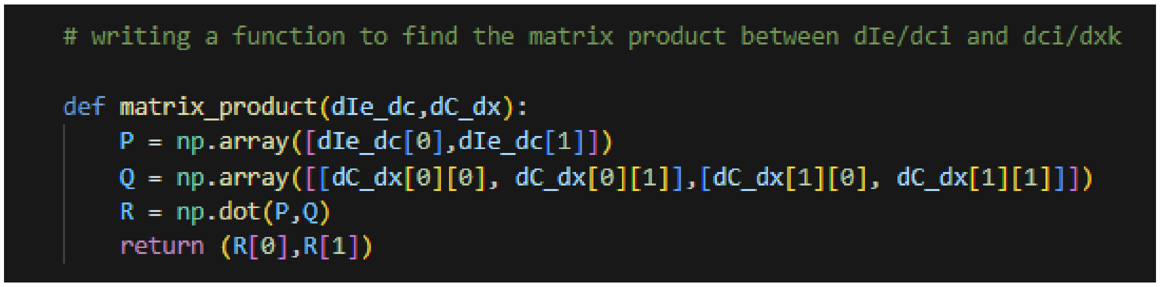
$\frac{\mathrm{d}\boldsymbol{C}_{\square}}{\mathrm{d}\boldsymbol{x}_{\square}}$ can thereby be found by differentiating the above matrices.

A Code snippet pertaining to the above:



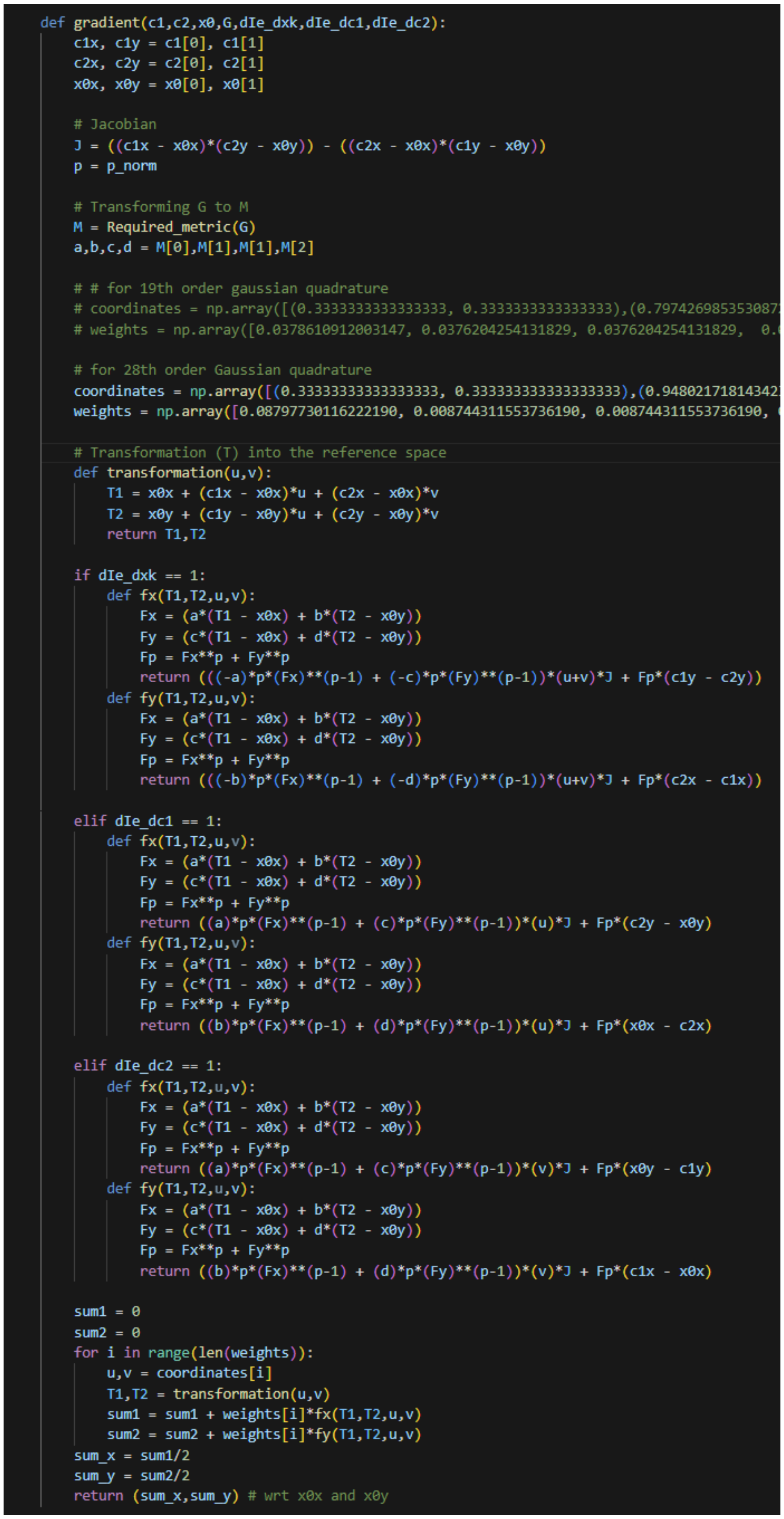
```
# writing function for calculating dc/dx when C and it's surrounding three delaunay vertices are given

def gradient_dC_dx(x0,x1,x2,C,xi):
    x0x, x0y = x0[0], x0[1]
    x1x, x1y = x1[0], x1[1]
    x2x, x2y = x2[0], x2[1]
    cx, cy = C[0], C[1]
    xix, xiy = xi[0], xi[1]
    A = np.array([[x1x - x0x, x1y - x0y],[x2x - x0x, x2y - x0y]])
    A_inv = np.linalg.inv(A)
    if xi == x0:
        B = np.array([[cx - xix, cy - xiy],[cx - xix, cy - xiy]])
    elif xi == x1:
        B = np.array([[xix - cx, xiy - cy],[ 0 , 0 ]])
    elif xi == x2:
        B = np.array([[ 0 , 0 ],[xix - cx, xiy - cy]])
    return np.dot(A_inv,B) # returns a 2*2 matrix
```

A code snippet finding the matrix product between the gradients:



```
# writing a function to find the matrix product between dIe/dci and dci/dxk

def matrix_product(dIe_dc,dC_dx):
    P = np.array([dIe_dc[0],dIe_dc[1]])
    Q = np.array([[dC_dx[0][0], dC_dx[0][1]],[dC_dx[1][0], dC_dx[1][1]]])
    R = np.dot(P,Q)
    return (R[0],R[1])
```

A code snippet finding the rest of gradients:



```python
def gradient(c1,c2,x0,G,dIe_dxk,dIe_dc1,dIe_dc2):
    c1x, c1y = c1[0], c1[1]
    c2x, c2y = c2[0], c2[1]
    x0x, x0y = x0[0], x0[1]

    # Jacobian
    J = ((c1x - x0x)*(c2y - x0y)) - ((c2x - x0x)*(c1y - x0y))
    p = p_norm

    # Transforming G to M
    M = Required_metric(G)
    a,b,c,d = M[0],M[1],M[1],M[2]

    # # for 19th order gaussian quadrature
    # coordinates = np.array([(0.3333333333333333, 0.3333333333333333),(0.797426985353087
    # weights = np.array([0.0378610912003147, 0.0376204254131829, 0.0376204254131829,  0.

    # for 28th order Gaussian quadrature
    coordinates = np.array([(0.33333333333333333, 0.333333333333333333),(0.94802171814342
    weights = np.array([0.08797730116222190, 0.008744311553736190, 0.008744311553736190,

    # Transformation (T) into the reference space
    def transformation(u,v):
        T1 = x0x + (c1x - x0x)*u + (c2x - x0x)*v
        T2 = x0y + (c1y - x0y)*u + (c2y - x0y)*v
        return T1,T2

    if dIe_dxk == 1:
        def fx(T1,T2,u,v):
            Fx = (a*(T1 - x0x) + b*(T2 - x0y))
            Fy = (c*(T1 - x0x) + d*(T2 - x0y))
            Fp = Fx**p + Fy**p
            return (((-a)*p*(Fx)**(p-1) + (-c)*p*(Fy)**(p-1))*(u+v)*J + Fp*(c1y - c2y))
        def fy(T1,T2,u,v):
            Fx = (a*(T1 - x0x) + b*(T2 - x0y))
            Fy = (c*(T1 - x0x) + d*(T2 - x0y))
            Fp = Fx**p + Fy**p
            return (((-b)*p*(Fx)**(p-1) + (-d)*p*(Fy)**(p-1))*(u+v)*J + Fp*(c2x - c1x))

    elif dIe_dc1 == 1:
        def fx(T1,T2,u,v):
            Fx = (a*(T1 - x0x) + b*(T2 - x0y))
            Fy = (c*(T1 - x0x) + d*(T2 - x0y))
            Fp = Fx**p + Fy**p
            return ((a)*p*(Fx)**(p-1) + (c)*p*(Fy)**(p-1))*(u)*J + Fp*(c2y - x0y)
        def fy(T1,T2,u,v):
            Fx = (a*(T1 - x0x) + b*(T2 - x0y))
            Fy = (c*(T1 - x0x) + d*(T2 - x0y))
            Fp = Fx**p + Fy**p
            return ((b)*p*(Fx)**(p-1) + (d)*p*(Fy)**(p-1))*(u)*J + Fp*(x0x - c2x)

    elif dIe_dc2 == 1:
        def fx(T1,T2,u,v):
            Fx = (a*(T1 - x0x) + b*(T2 - x0y))
            Fy = (c*(T1 - x0x) + d*(T2 - x0y))
            Fp = Fx**p + Fy**p
            return ((a)*p*(Fx)**(p-1) + (c)*p*(Fy)**(p-1))*(v)*J + Fp*(x0y - c1y)
        def fy(T1,T2,u,v):
            Fx = (a*(T1 - x0x) + b*(T2 - x0y))
            Fy = (c*(T1 - x0x) + d*(T2 - x0y))
            Fp = Fx**p + Fy**p
            return ((b)*p*(Fx)**(p-1) + (d)*p*(Fy)**(p-1))*(v)*J + Fp*(c1x - x0x)

    sum1 = 0
    sum2 = 0
    for i in range(len(weights)):
        u,v = coordinates[i]
        T1,T2 = transformation(u,v)
        sum1 = sum1 + weights[i]*fx(T1,T2,u,v)
        sum2 = sum2 + weights[i]*fy(T1,T2,u,v)
    sum_x = sum1/2
    sum_y = sum2/2
    return (sum_x,sum_y) # wrt x0x and x0y
```

Note: A 19-point Gaussian quadrature is used for all integral calculations by transforming the required element into a reference element of (1,0), (0,1), (0,0), in order to solve this integral numerically. The transformation and representation are given below:

$$\boldsymbol{y} = \boldsymbol{T}(u, v) = \boldsymbol{x}_0(1 - u - v) + \boldsymbol{C}_1 u + \boldsymbol{C}_2 v$$

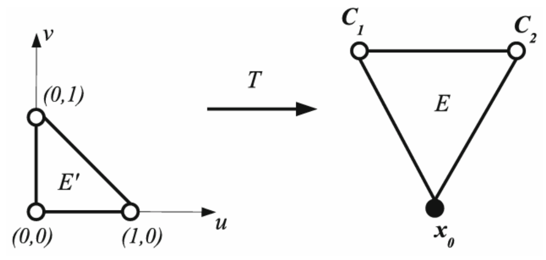


Figure 8: The linear transformation $\boldsymbol{T}$

Metric interpolation:

A linear interpolation framework is used to find the metric at any given point. Assuming the point lies inside a triangle whose metrics at each vertex is known, the barycentric weights of the point are found, and a linear summation of the weight and metric product is performed.

Reconstruction of the Voronoi cell:

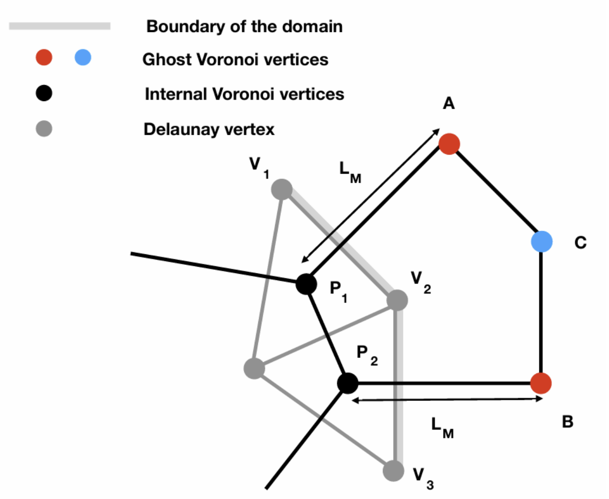


Figure 9: Voronoi cell reconstruction at the boundary

The reconstruction of the Voronoi cell on the boundary is implemented considering the underlying metric and the assumption that for a sufficiently high p-norm, the corners of the Voronoi cells are going to be right-angled at a minimum energy state. The reconstruction procedure is graphically presented in Figure 9. The location of point A is found by reconstructing a line that is in the direction perpendicular to boundary edge V1-V2 and has unit length under the metric (LM = 1 in Figure 9). The same is done to find the location of point B but now we use the boundary edge V2-V3 to determine the direction with respect to P2. Point C is found by taking the tangential directions from point A and point B and we find the intersection between those edges.

The gradient and energy computation formulas are obtained from [6] and [7]. The reconstruction is implemented as shown in [7].

4.3 Implementation in the code

Note:

- NETGEN software and it's associated functions in python are used for all mesh visualization and code – geometry interactions.
- Metric value for each simplex is held constant. It is obtained by averaging the value of the metric at the three vertices of the simplex.

List of NETGEN/NGSolve functions used:

- Mesh(file_name) – reads the .vol file and prepares it for further operations. In the present work: ma = Mesh(file_name).
- ma.nv and ma.ne returns number of vertices and elements.
- ma.Elements(VOL) stores all element information.
- For each el in ma.Elements(VOL), el.vertices returns the vertex information for that element.
- For each vertex v, in el.vertices, ma[v].point returns the coordinates of the vertex.

Implementation steps:

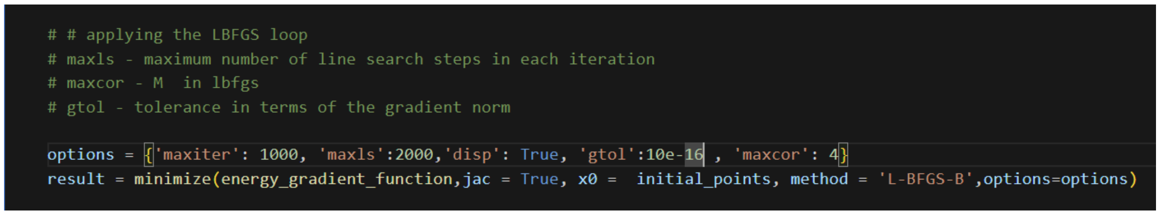
1) Discrete metric field

- Import all the required python libraries – primary ones being ngsolve, numpy, matplotlib, and scipy.
- Define all the required functions.
- Read the .vol file and store information about the vertices, the elements, and vertex coordinates in separate lists.
- Store the metric information at each mesh vertex. There can be two cases:
  > Implied metric – Obtaining metric from the mesh assuming it is already adapted in the metric field. This is carried out by assuming each element is already unit in the metric space and thus, should have an edge length of one in the metric space.
  > Target/Background metric – assuming a file with metric information at each vertex is already provided, and the mesh needs to be adapted according to this target metric.

- Store the obtained metric information for each vertex in dictionaries and define a function get_metric() such that it returns the metric values for each vertex coordinate given as an argument.
- Discretize the boundary segments such that each edge on the segment is unit in the prescribed metric field and store the newly generated vertex coordinates in a separate list.
- Define a function(energy_gradient_function) which returns the energy and gradient values, that is called by LBFGS loop at each iteration.
- Flatten the list of interior points read from the initial .vol file into a 1×n(number of points) array and input it into the energy_gradient_function.
- Within the energy_gradient_function:
  > Convert the flattened array of interior coordinates into a list of tuples of interior coordinates.
  > Add the discretized boundary points to the above list and generate a new triangulation using the scipy Delaunay library.
  > Read and write the newly generated triangulation into a .vol file using the function write_vol_file represented in Figure 10.
  > Store information about the elements and vertex coordinates of the newly generated triangulation in lists(as a list of tuples).
  > Find the metric values at the interior coordinates of the newly generated triangulation by defining a for loop which checks whether the new vertex lies:
  1)Inside a triangle of the initial triangulation
  2)On the vertex of the initial triangulation
  3)On one of the edges of the initial triangulation
  4)Outside the domain (in which case, the metric value for the vertex is defined to be unit)
  and respectively interpolates to find the metric.
  The code snippet is given in Figure 12.
  > A get_metric_f() function is defined which plays the same role as get_metric().
  > Voronoi cell information is stored as a list in which each element of the list contains the coordinates of the Voronoi cell vertices and Voronoi cell center(mesh vertex), arranged in anticlockwise direction.
  > Voronoi cells are reconstructed at the boundaries and are assigned metric values (metric of the closest boundary vertex).
  > From the Voronoi cell, simplices are obtained and each simplex data is stored in a list
  > Energy values is found by iterating over all the simplices, finding their energy using simplex_energy(), defined in Figure 11.
  > Summation of the energy values of the interior Voronoi cells is returned in the energy_gradient_function().
  > Gradient values are found similarly, by iterating over each simplex, finding the their gradient contribution to all the surrounding Voronoi cells (mesh vertices) and to the Voronoi cell it belongs to, and adding the gradients to the respective

dependent mesh vertices (Voronoi cell centres).
> Summation of the gradient values of the interior Voronoi cell is returned in the energy_gradient_function() function.

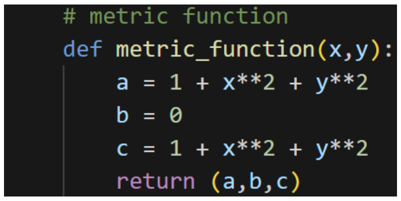
- LBFGS optimization is implemented using the scipy.optimise package.
- Maximum iterations are initialized to 1000 and maximum line searches per iteration is also initialized to 1000. M is given to equal to 4.
  A snippet from the code is given below:



```
# # applying the LBFGS loop
# maxls - maximum number of line search steps in each iteration
# maxcor - M  in lbfgs
# gtol - tolerance in terms of the gradient norm

options = {'maxiter': 1000, 'maxls':2000,'disp': True, 'gtol':10e-16 , 'maxcor': 4}
result = minimize(energy_gradient_function,jac = True, x0 =  initial_points, method = 'L-BFGS-B',options=options)
```

2) Analytical metric field

- Define the analytical metric field through metric_function()



```
# metric function
def metric_function(x,y):
    a = 1 + x**2 + y**2
    b = 0
    c = 1 + x**2 + y**2
    return (a,b,c)
```

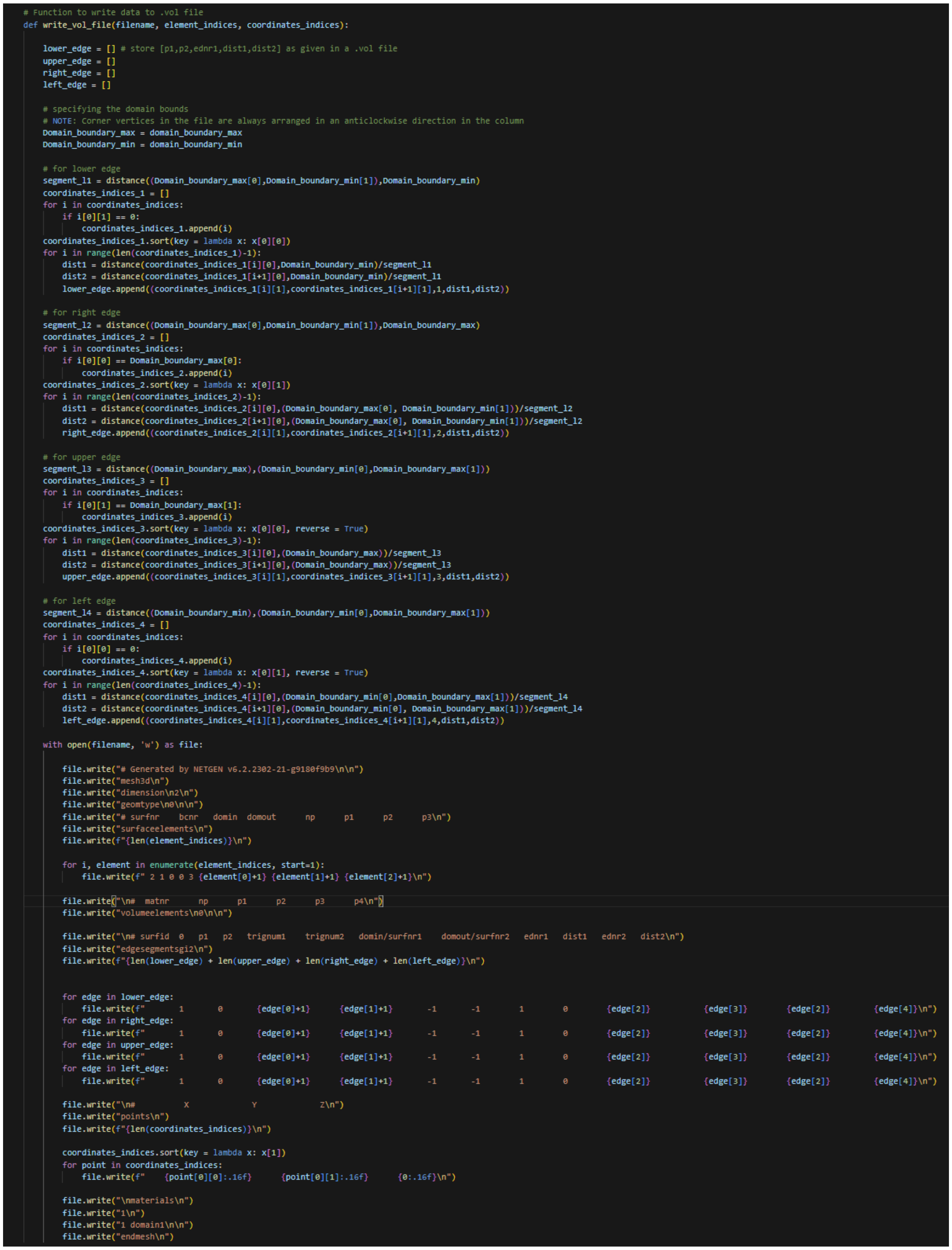
- Instead of reading a .vol file in the beginning of the program, as done in the discrete metric field case, the boundary is discretized and a random set of interior points are generated.
- A triangulation is generated with these random set of interior points, and Lp – CVT optimization is then implemented.

Calculating orthogonal quality:

The variable τ is used to measure how close to perfect square triangles the elements are:

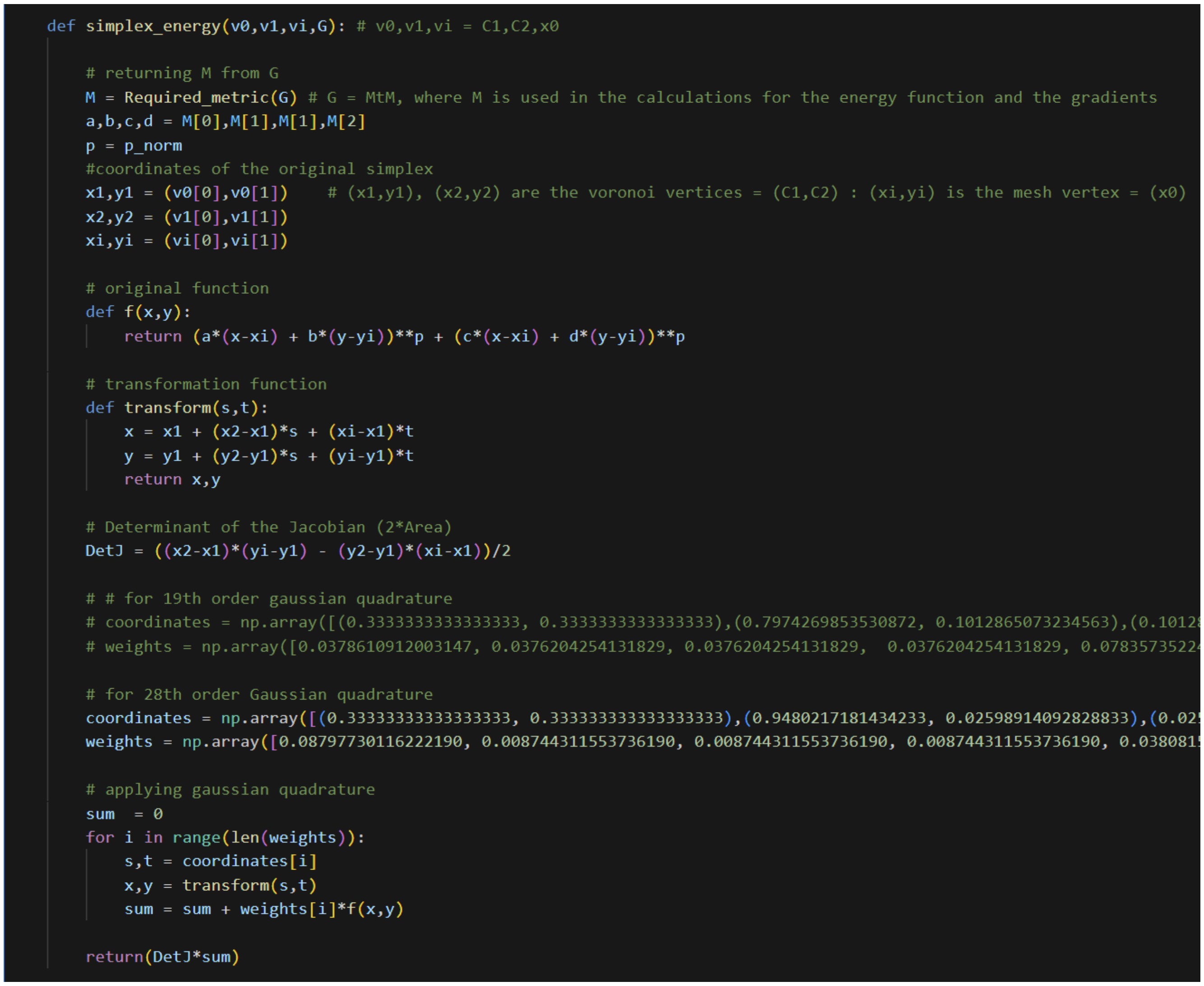
$$\tau = \frac{1}{N}\sum_{i=1}^{N} \max_{j=1,2,3} \left(\text{abs}\left(\sin\ \theta_{ij}\right)\right)$$

N is the total number of triangular elements in the mesh. θ_{i1} is the first interior angle of triangular element i, θ_{i2} is the second one and θ_{i3} is the third one. τ is equal to 1 for a mesh containing only perfect square triangles. It is equal to 0.866 for a mesh containing only equilateral triangles.



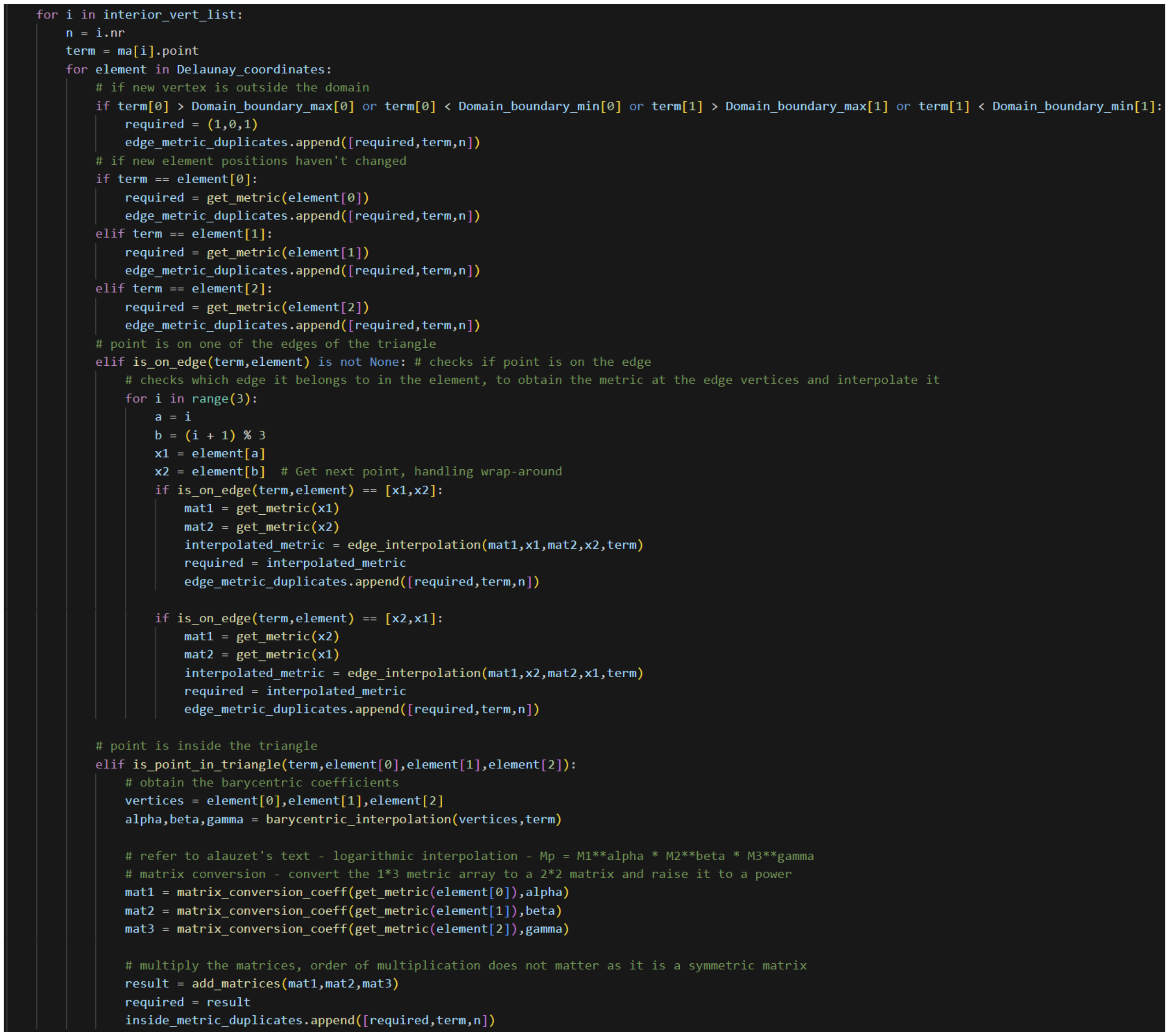
```python
# Function to write data to .vol file
def write_vol_file(filename, element_indices, coordinates_indices):

    lower_edge = [] # store [p1,p2,ednr1,dist1,dist2] as given in a .vol file
    upper_edge = []
    right_edge = []
    left_edge = []

    # specifying the domain bounds
    # NOTE: Corner vertices in the file are always arranged in an anticlockwise direction in the column
    Domain_boundary_max = domain_boundary_max
    Domain_boundary_min = domain_boundary_min

    # for lower edge
    segment_l1 = distance((Domain_boundary_max[0],Domain_boundary_min[1]),Domain_boundary_min)
    coordinates_indices_1 = []
    for i in coordinates_indices:
        if i[0][1] == 0:
            coordinates_indices_1.append(i)
    coordinates_indices_1.sort(key = lambda x: x[0][0])
    for i in range(len(coordinates_indices_1)-1):
        dist1 = distance(coordinates_indices_1[i][0],Domain_boundary_min)/segment_l1
        dist2 = distance(coordinates_indices_1[i+1][0],Domain_boundary_min)/segment_l1
        lower_edge.append((coordinates_indices_1[i][1],coordinates_indices_1[i+1][1],1,dist1,dist2))

    # for right edge
    segment_l2 = distance((Domain_boundary_max[0],Domain_boundary_min[1]),Domain_boundary_max)
    coordinates_indices_2 = []
    for i in coordinates_indices:
        if i[0][0] == Domain_boundary_max[0]:
            coordinates_indices_2.append(i)
    coordinates_indices_2.sort(key = lambda x: x[0][1])
    for i in range(len(coordinates_indices_2)-1):
        dist1 = distance(coordinates_indices_2[i][0],(Domain_boundary_max[0], Domain_boundary_min[1]))/segment_l2
        dist2 = distance(coordinates_indices_2[i+1][0],(Domain_boundary_max[0], Domain_boundary_min[1]))/segment_l2
        right_edge.append((coordinates_indices_2[i][1],coordinates_indices_2[i+1][1],2,dist1,dist2))

    # for upper edge
    segment_l3 = distance((Domain_boundary_max),(Domain_boundary_min[0],Domain_boundary_max[1]))
    coordinates_indices_3 = []
    for i in coordinates_indices:
        if i[0][1] == Domain_boundary_max[1]:
            coordinates_indices_3.append(i)
    coordinates_indices_3.sort(key = lambda x: x[0][0], reverse = True)
    for i in range(len(coordinates_indices_3)-1):
        dist1 = distance(coordinates_indices_3[i][0],(Domain_boundary_max))/segment_l3
        dist2 = distance(coordinates_indices_3[i+1][0],(Domain_boundary_max))/segment_l3
        upper_edge.append((coordinates_indices_3[i][1],coordinates_indices_3[i+1][1],3,dist1,dist2))

    # for left edge
    segment_l4 = distance((Domain_boundary_min),(Domain_boundary_min[0],Domain_boundary_max[1]))
    coordinates_indices_4 = []
    for i in coordinates_indices:
        if i[0][0] == 0:
            coordinates_indices_4.append(i)
    coordinates_indices_4.sort(key = lambda x: x[0][1], reverse = True)
    for i in range(len(coordinates_indices_4)-1):
        dist1 = distance(coordinates_indices_4[i][0],(Domain_boundary_min[0],Domain_boundary_max[1]))/segment_l4
        dist2 = distance(coordinates_indices_4[i+1][0],(Domain_boundary_min[0], Domain_boundary_max[1]))/segment_l4
        left_edge.append((coordinates_indices_4[i][1],coordinates_indices_4[i+1][1],4,dist1,dist2))

    with open(filename, 'w') as file:

        file.write("# Generated by NETGEN v6.2.2302-21-g9180f9b9\n\n")
        file.write("mesh3d\n")
        file.write("dimension\n2\n")
        file.write("geomtype\n0\n\n")
        file.write("# surfnr    bcnr   domin  domout      np      p1      p2      p3\n")
        file.write("surfaceelements\n")
        file.write(f"{len(element_indices)}\n")

        for i, element in enumerate(element_indices, start=1):
            file.write(f" 2 1 0 0 3 {element[0]+1} {element[1]+1} {element[2]+1}\n")

        file.write("\n#  matnr      np      p1      p2      p3      p4\n")
        file.write("volumeelements\n0\n\n")

        file.write("\n# surfid  0   p1   p2   trignum1    trignum2   domin/surfnr1    domout/surfnr2   ednr1   dist1   ednr2   dist2\n")
        file.write("edgesegmentsgi2\n")
        file.write(f"{len(lower_edge) + len(upper_edge) + len(right_edge) + len(left_edge)}\n")


        for edge in lower_edge:
            file.write(f"       1       0       {edge[0]+1}      {edge[1]+1}       -1       -1       1       0       {edge[2]}          {edge[3]}        {edge[2]}         {edge[4]}\n")
        for edge in right_edge:
            file.write(f"       1       0       {edge[0]+1}      {edge[1]+1}       -1       -1       1       0       {edge[2]}          {edge[3]}        {edge[2]}         {edge[4]}\n")
        for edge in upper_edge:
            file.write(f"       1       0       {edge[0]+1}      {edge[1]+1}       -1       -1       1       0       {edge[2]}          {edge[3]}        {edge[2]}         {edge[4]}\n")
        for edge in left_edge:
            file.write(f"       1       0       {edge[0]+1}      {edge[1]+1}       -1       -1       1       0       {edge[2]}          {edge[3]}        {edge[2]}         {edge[4]}\n")

        file.write("\n#          X             Y             Z\n")
        file.write("points\n")
        file.write(f"{len(coordinates_indices)}\n")

        coordinates_indices.sort(key = lambda x: x[1])
        for point in coordinates_indices:
            file.write(f"    {point[0][0]:.16f}      {point[0][1]:.16f}      {0:.16f}\n")

        file.write("\nmaterials\n")
        file.write("1\n")
        file.write("1 domain1\n\n")
        file.write("endmesh\n")
```

Figure 10: Function to write in .vol format.



```python
def simplex_energy(v0,v1,vi,G): # v0,v1,vi = C1,C2,x0

    # returning M from G
    M = Required_metric(G) # G = MtM, where M is used in the calculations for the energy function and the gradients
    a,b,c,d = M[0],M[1],M[1],M[2]
    p = p_norm
    #coordinates of the original simplex
    x1,y1 = (v0[0],v0[1])    # (x1,y1), (x2,y2) are the voronoi vertices = (C1,C2) : (xi,yi) is the mesh vertex = (x0)
    x2,y2 = (v1[0],v1[1])
    xi,yi = (vi[0],vi[1])

    # original function
    def f(x,y):
        return (a*(x-xi) + b*(y-yi))**p + (c*(x-xi) + d*(y-yi))**p

    # transformation function
    def transform(s,t):
        x = x1 + (x2-x1)*s + (xi-x1)*t
        y = y1 + (y2-y1)*s + (yi-y1)*t
        return x,y

    # Determinant of the Jacobian (2*Area)
    DetJ = ((x2-x1)*(yi-y1) - (y2-y1)*(xi-x1))/2

    # # for 19th order gaussian quadrature
    # coordinates = np.array([(0.3333333333333333, 0.3333333333333333),(0.7974269853530872, 0.1012865073234563),(0.1012
    # weights = np.array([0.0378610912003147, 0.0376204254131829, 0.0376204254131829,  0.0376204254131829, 0.0783573522

    # for 28th order Gaussian quadrature
    coordinates = np.array([(0.33333333333333333, 0.33333333333333333),(0.9480217181434233, 0.02598914092828833),(0.02
    weights = np.array([0.08797730116222190, 0.008744311553736190, 0.008744311553736190, 0.008744311553736190, 0.038081

    # applying gaussian quadrature
    sum  = 0
    for i in range(len(weights)):
        s,t = coordinates[i]
        x,y = transform(s,t)
        sum = sum + weights[i]*f(x,y)

    return(DetJ*sum)
```

Figure 11: Function to calculate the simplex energy



```
    for i in interior_vert_list:
        n = i.nr
        term = ma[i].point
        for element in Delaunay_coordinates:
            # if new vertex is outside the domain
            if term[0] > Domain_boundary_max[0] or term[0] < Domain_boundary_min[0] or term[1] > Domain_boundary_max[1] or term[1] < Domain_boundary_min[1]:
                required = (1,0,1)
                edge_metric_duplicates.append([required,term,n])
            # if new element positions haven't changed
            if term == element[0]:
                required = get_metric(element[0])
                edge_metric_duplicates.append([required,term,n])
            elif term == element[1]:
                required = get_metric(element[1])
                edge_metric_duplicates.append([required,term,n])
            elif term == element[2]:
                required = get_metric(element[2])
                edge_metric_duplicates.append([required,term,n])
            # point is on one of the edges of the triangle
            elif is_on_edge(term,element) is not None: # checks if point is on the edge
                # checks which edge it belongs to in the element, to obtain the metric at the edge vertices and interpolate it
                for i in range(3):
                    a = i
                    b = (i + 1) % 3
                    x1 = element[a]
                    x2 = element[b]  # Get next point, handling wrap-around
                    if is_on_edge(term,element) == [x1,x2]:
                        mat1 = get_metric(x1)
                        mat2 = get_metric(x2)
                        interpolated_metric = edge_interpolation(mat1,x1,mat2,x2,term)
                        required = interpolated_metric
                        edge_metric_duplicates.append([required,term,n])

                    if is_on_edge(term,element) == [x2,x1]:
                        mat1 = get_metric(x2)
                        mat2 = get_metric(x1)
                        interpolated_metric = edge_interpolation(mat1,x2,mat2,x1,term)
                        required = interpolated_metric
                        edge_metric_duplicates.append([required,term,n])

            # point is inside the triangle
            elif is_point_in_triangle(term,element[0],element[1],element[2]):
                # obtain the barycentric coefficients
                vertices = element[0],element[1],element[2]
                alpha,beta,gamma = barycentric_interpolation(vertices,term)

                # refer to alauzet's text - logarithmic interpolation - Mp = M1**alpha * M2**beta * M3**gamma
                # matrix conversion - convert the 1*3 metric array to a 2*2 matrix and raise it to a power
                mat1 = matrix_conversion_coeff(get_metric(element[0]),alpha)
                mat2 = matrix_conversion_coeff(get_metric(element[1]),beta)
                mat3 = matrix_conversion_coeff(get_metric(element[2]),gamma)

                # multiply the matrices, order of multiplication does not matter as it is a symmetric matrix
                result = add_matrices(mat1,mat2,mat3)
                required = result
                inside_metric_duplicates.append([required,term,n])
```

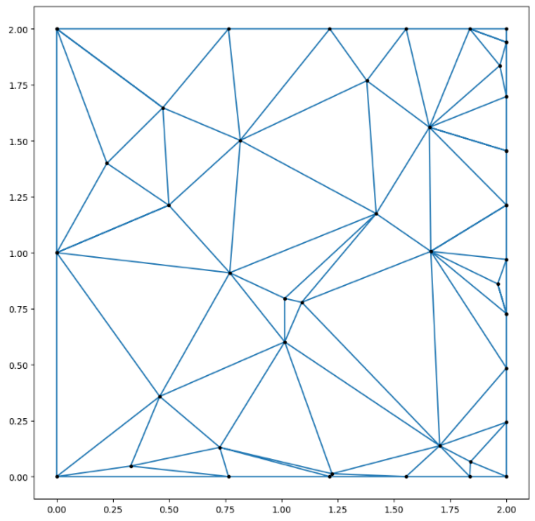
Figure 12: Loop calculating the metric at the new coordinates.

5 Results

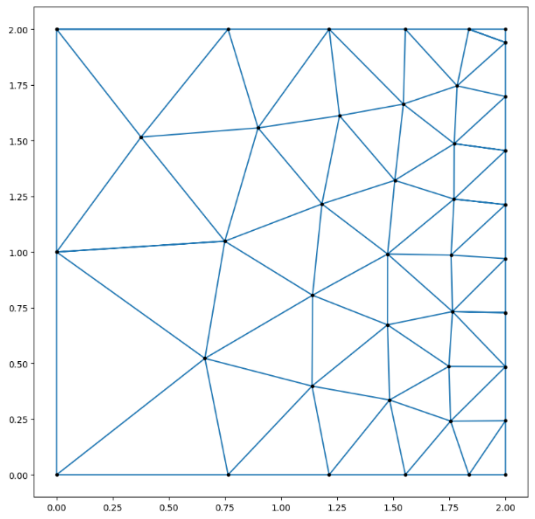
5.1 Analytical (isotropic) metric

5.1.1 Metric 1 - $M = \begin{pmatrix} 1+4x^2 & 0 \\ 0 & 1+4x^2 \end{pmatrix}$

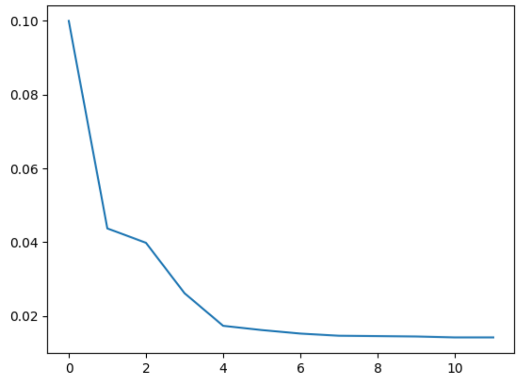
Number of interior nodes – 20; Domain – [0,2] square.



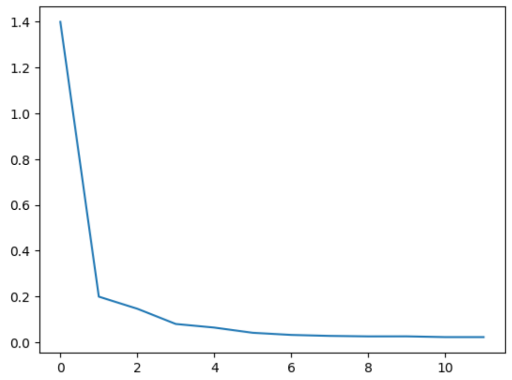
Before optimization



After optimization



Energy v/s iterations



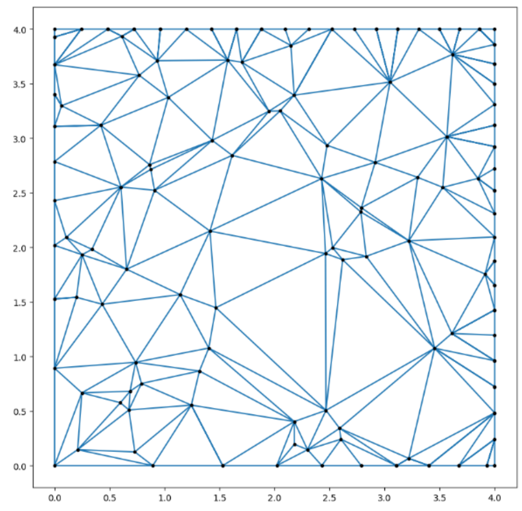
Gradient v/s iterations

Orthogonal quality : τ before - 0.9240252305055181

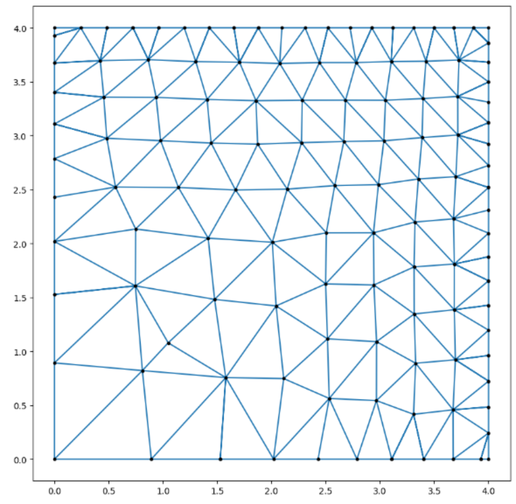
τ after - 0.960626622540224

5.1.2 Metric 2 - $M = \begin{pmatrix} 1+x^2+y^2 & 0 \\ 0 & 1+x^2+y^2 \end{pmatrix}$

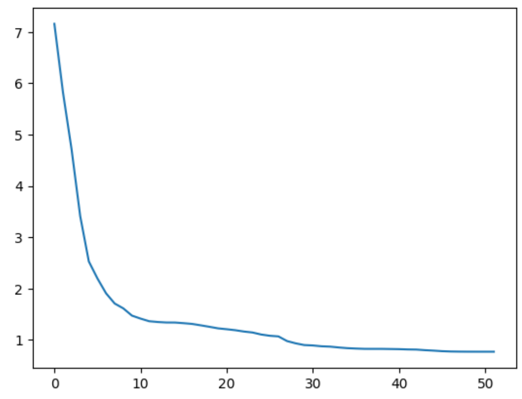
Number of interior nodes – 64; Domain – [0,4] square.



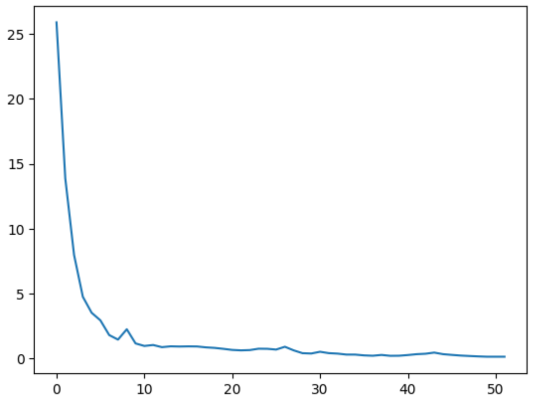
Before optimization



After optimization



Energy v/s iterations



Gradient v/s iterations

Orthogonal quality: τ before - 0.9465041294340213

τ after - 0.9638856927220493

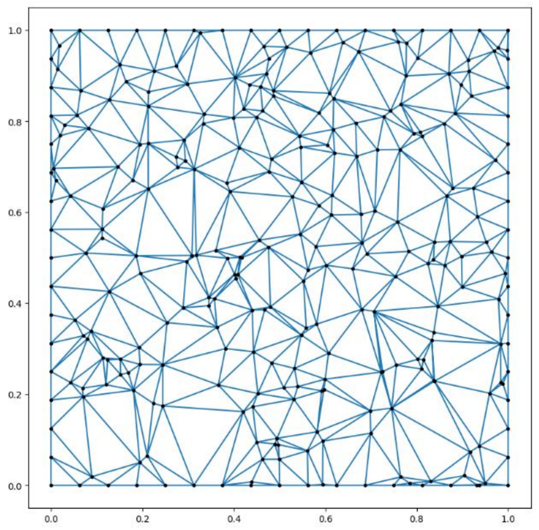
Energy: Before – 7.24

After – 0.076

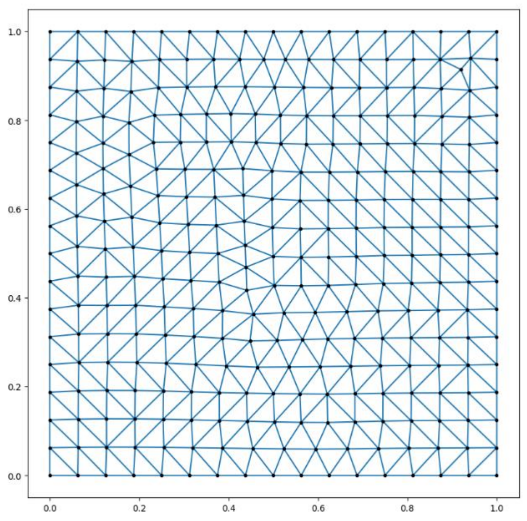
5.2 Discrete metric

5.2.1 Target metric - $M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

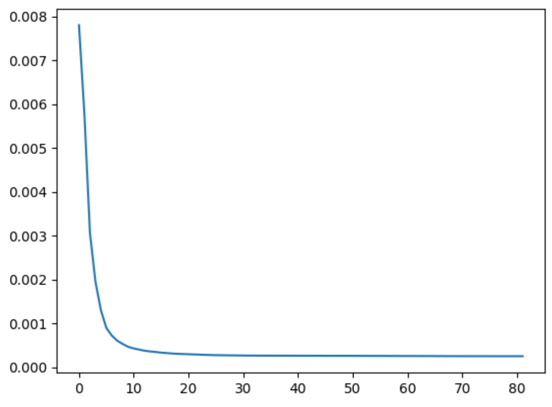
Number of interior nodes – 225; Domain – [0,1] square.



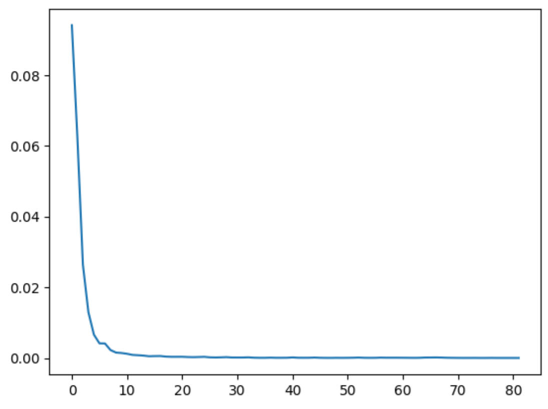
Before optimization



After optimization



Energy v/s iterations

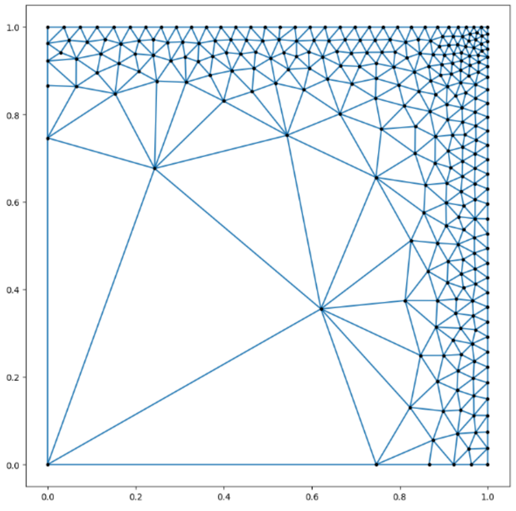


Gradient v/s iterations

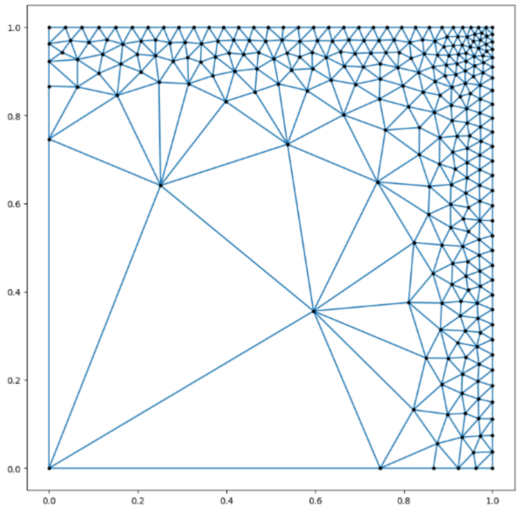
Energy: Before – 0.00798653

After – 1.646495470692687e-05

5.2.2 Implied metric



Before optimization



After optimization

Orthogonal quality: τ before - 0.9467728810827933

τ after - 0.9472958588088273

Energy: Before – 0.032

After – 0.028

Obstacles faced:

The derivative of the matrix in the gradient equation could not be calculated due to mathematical inadequacies. Because of this, wrong results were obtained while adapting a given initial mesh with respect to a target, varying, discrete mesh.

7 References

[1] Slotnick JP, Khodadoust A, Alonso J, Darmofal D, Gropp W, Lurie E, Mavriplis DJ. CFD vision 2030 study: a path to revolutionary computational aerosciences. 2014 Mar 1.

[2] Du Q, Faber V, Gunzburger M (1999) Centroidal voronoi tes sellations: applications and algorithms. Siam Rev 41:637–676

[3] Liu Y, Wang W, Levy B, Sun F, Yan DM, Lu L, Yang C (2009) On centroidal voronoi tessellation-energy smoothness and fast computation. ACM Trans Graphic 28:1–17

[4] Lévy B, Liu Y. L p centroidal voronoi tessellation and its applications. ACM Transactions on Graphics (TOG). 2010 Jul 26;29(4):1-1.

[5] Liu Y, Wang W, Lévy B, Sun F, Yan DM, Lu L, Yang C. On centroidal Voronoi tessellation—energy smoothness and fast computation. ACM Transactions on Graphics (ToG). 2009 Sep 8;28(4):1-7.

[6] MacLean K, Nadarajah S. Anisotropic mesh generation and adaptation for quads using the Lp-CVT method. Journal of Computational Physics. 2022 Dec 1;470:111578.

[7] Ekelschot D, Ceze M, Garai A, Murman SM. Robust metric aligned quad-dominant meshing using Lp centroidal voronoi tessellation. In2018 AIAA Aerospace Sciences Meeting 2018 (p. 1501).