EMBEDDING A RIEMANNIAN SURFACE IN R^3

SUN HYOUNG SONYA KIM

ADVISOR: DR. TOM LAGATTA

Abstract. This paper presents a wireframe algorithm that helps visualize embedded Riemannian surfaces in R^3. Using information provided by the induced metric, also known as the first fundamental form defined in the Euclidean space, the algorithm aims to find an isometric embedding of the intrinsic surface in the three-dimensional space. The fundamental idea is to use a variation of Simulated Annealing (SA) to yield an embedding whose geometric measurements correspond to those of the intrinsic surface.

1. Introduction

Isometric embedding is one of the most famous problems in differential geometry and is still being investigated by many. This classical problem is concerned with taking a two-dimensional Riemannian manifold characterized by its intrinsic metrics and realizing as an embedding into the three-dimensional Euclidean space.

Beyond the analysis of isometric embedding conducted by famous mathematicians such as John Nash and Louis Nirenberg, this problem has seen numerous applications in applied mathematics, computer science and engineering. Reconstructing surfaces in the three-dimensional space from geometric information defined in the two-dimensional space is a very powerful tool. For example, isometric embedding has been researched for building face recognition technology; given a two-dimensional picture, the theory implies that it is possible to reconstruct a facial surface that preserves the geodesic distances between vertices.

The reason for the problem's wide recognition is perhaps its applications as well as its complexity. In order to develop a thorough computational method for constructing an isometric embedding, many factors such as existence and uniqueness must be carefully studied. Furthermore, the advantages and disadvantages of many different approaches must be evaluated. Ingrid Hotz in [1] explains that local methods that are being tested have the risk of cluttering. Meanwhile, global methods such as the Newton-Raphson method seen in Nollert and Herold's paper [6] require good start values to iterate over. Without good start values, the convergence of the algorithm to correct positions of the discretized points of the given surface may fail.

With no known ideal and clean-cut algorithm in computational geometry for solving the isometric embedding problem, we are motivated to experiment with various methods in

Date: November 4, 2011.

hopes of obtaining and refining a general solution. This paper introduces a new idea of using a probabilistic algorithm inspired by Markov Chain Monte Carlo in order to arrive at the correct embedding in the three-dimensional space. When the isometric embedding problem is viewed as a global optimization problem of preserving the length of curves, the appropriateness of using a probabilistic model becomes quite clear. Probabilistic algorithms in optimization problems are known to be more robust than general iterative algorithms in that they have "stronger mechanisms for escaping local minima" in a large search space [7]. This indicates that in the context of our problem, the use of a probabilistic algorithms may yield a good approximation to the global optimum, which is essentially the embedding that preserves the metric.

2. The Problem

Mathematically, the isometric embedding problem can be summarized by the following. Let a surface parameterization $X : U \subset R^2 \to S \subset R^3$ be defined by $X(u,v) = (x(u,v), y(u,v), z(u,v))$. Then the properties of surface S can be expressed in terms of the metric tensor g, also known as the matrix containing the coefficients of the first fundamental form:

$$g = \begin{bmatrix} E & F \\ F & G \end{bmatrix} = \begin{bmatrix} X_u \cdot X_u & X_u \cdot X_v \\ X_u \cdot X_v & X_v \cdot X_v \end{bmatrix}$$

The first fundamental form represents the inner product of basis vectors of the tangent space of a surface. The basis vectors of the tangent space denoted as X_u and X_v as shown in Figure 1 allow for calculating the intrinsic geometric properties of the surface, such as lengths of curves on the surface, surface areas, and Gaussian curvature. These measurements pertain to just the intrinsic, not the extrinsic geometry of the surface and are invariant under isometries.

Given a surface parameterization, $X(u,v)$, the metric can be computed easily by taking the dot product of its partial derivatives. However, going the other way, i.e. computing the surface parameterization given the metric is not a trivial matter. The problem can be written down using differential equations

$$\begin{aligned} E &= x_u^2 + y_u^2 + z_u^2 \\ F &= x_u x_v + y_u y_v + z_u z_v \\ G &= x_v^2 + y_v^2 + z_v^2 \end{aligned}$$

The solutions of the above differential equations determine the parameterization $X(u,v)$ $= (x(u,v), y(u,v), z(u,v))$. Though the existence of a global embedding is not guaranteed, the Gauss-Bonnet theorem [3] states that any Riemannian manifold has a local embedding in the neighborhood of a given point. For the purpose of the paper, we do not want to solve the differential equation to obtain the parameterization. Rather, computing a solution of the each of the discretized points of the surface in the 3-dimensional Euclidean space via a MCMC inspired algorithm will help us visualize the intuitive meaning of the metric.

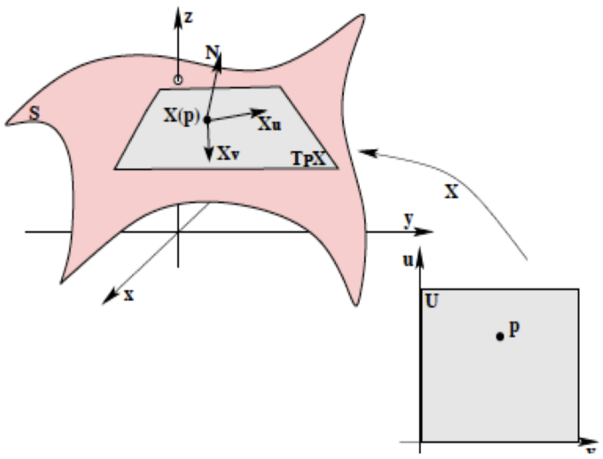

FIGURE 1. Embedded surface in R^3 with basis vectors of tangent space. (Image courtesy of [1])

3. Embedding Algorithm

The overall idea is to discretize a random grid in R^3 using a triangular mesh, and compute the candidate position of the points by minimizing the energy function. We measure the difference between the measurements of the candidate and the current position, and only accept the candidate as the updated position if it "passes" the condition defined a priori. We repeat this process until the geometric measurements of the surface match the intrinsic measurements computed by the metric tensor. The outline of the algorithm is the following:

– Create a triangular grid in sitting in R^2, the intrinsic grid whose lengths and surface area can be computed using the information provided by the metric tensor.

– Create a random triangular grid sitting in R^3 to simulate the embedding.

– Randomly pick a point on the intrinsic grid and index the six neighboring points.

– Use the metric to calculate the intrinsic quantities such as the lengths between the random point and each of the six neighboring points, the lengths between the neighboring points, and surface areas of six triangles formed by the random point and the neighboring points.

– Compute the corresponding lengths and areas on the embedding.

– Define energy function H as the sum of the differences of the intrinsic measurements and the measurements on the embedding squared.

– Minimize the energy function. The values of the neighboring points on the embedded surface that minimize H are candidates for update.

– Accept the candidate points as new neighboring points on the embedding with a predetermined probability. Iterate the process until H converges.

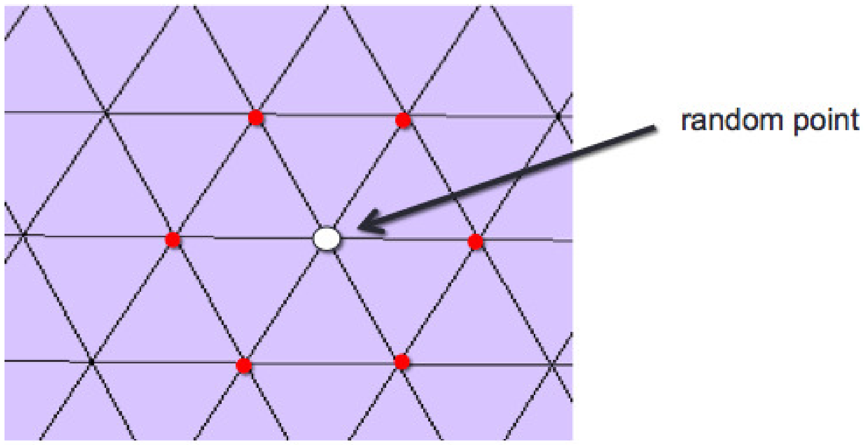

FIGURE 2. We pick a random point on the grid, index the 6 neighboring points to compute their geometric quantities.

3.1. **Calculation of Intrinsic Length and Surface Area.** The intrinsic length and surface area are well-known and easily computable given the metric. The length of a curve on a surface is

$$l = \int_0^h \sqrt{E\left(\frac{du}{dt}\right)^2 + 2F\left(\frac{du}{dt}\right)\left(\frac{dv}{dt}\right) + G\left(\frac{du}{dt}\right)^2}\, dt \tag{1}$$

where h is the time step, and E, F and G are coefficients of the first fundamental form. Using the above equation, we compute the length between the randomly chosen point and its neighbors on the intrinsic surface.

Similarly, the area of a triangle on a surface is well-defined and can be computed by simply using the coefficients of the first fundamental form. The formula is given by:

$$\alpha = \iint_{\Delta} \sqrt{EG - F^2}\, du dv \tag{2}$$

Using the two formulas above, we are able to find the intrinsic lengths between the randomly chosen point and each of the six neighbors, the intrinsic lengths between the neighbors, as well as the surface area of six triangles created by the random point and its neighbors.

3.2. **Calculation of Length and Surface Area on the Embedding.** In order to compute the geometry of the embedding, we use the Euclidean distance formula for length between two points and Heron's formula for the area of a triangle. They are respectively defined by:

$$d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + (q_3 - p_3)^2} \tag{3}$$

where (q_1, q_2, q_3) and (p_1, p_2, p_3) are two points on the embedding in R^3, and

$$A = \frac{1}{4}\sqrt{(a^2 + b^2 + c^2)^2 - 2(a^4 + b^4 + c^4)} \tag{4}$$

where a, b, c are the edges of a triangle formed by three points on the mesh of embedding. Analogous to computing the intrinsic quantities, using the above formulas, we are able to find the distance between the randomly chosen point and its six neighbors, the distance between each of the neighbors, and the area of the six triangles formed by these relevant points.

3.3. **Definition of Energy Function.** The definition of energy function is crucial in making the algorithm converge to the correct embedding. For our purpose, the energy of the system, i.e. the embedding, is expressed by the sum of the difference between the intrinsic and extrinsic measurements squared. Intuitively, this makes sense because we want the embedding to match the geometric quantities of the intrinsic surface as the energy of the system nears 0. That is, the optimization of the energy function should reflect the embedding evolving to become more and more like the intrinsic surface.

Combining the information obtained from section 2.1 and 2.2, define the energy function H as

$$H = \sum_{i=0}^{6} ((L_i - l_i)^2 + (P_i - \rho_i)^2 + (A_i - \alpha_i)^2)^2 \tag{5}$$

where,

L_i = Euclidean distance between a random point and its neighbor
l_i = Intrinsic length between a random point and its neighbor
P_i = Euclidean distance between two neighbors
ρ_i = Intrinsic length between two neighbors
A_i = Area of each of the six triangles surrounding the random point
α_i = Intrinsic surface area of each of the triangles surrounding the random point

3.4. **Minimization of the Energy Function.** Using the definition of the energy function, the next step is to compute the change in energy at each iteration. In order to do so, plug in the known values of lengths and area directly into equation (5) to calculate the energy at the current step; call this value $H_{current}$. Then leaving L_i, P_i, and A_i as variables in equation (5), minimize the function H. The values which minimize H tell us the candidate points of the six neighboring points that are to be updated. Call the minimized value of H H_{new}. Then the change in energy from the current step to the ideal next step can now be defined by

$$\Delta H = H_{current} - H_{new} \tag{6}$$

3.5. **Simulated Annealing.** Simulated Annealing (SA) is a variation of the Markov Chain Monte Carlo method which strives to find the global optimum of a function in a large search space. The algorithm provides iterative improvements of the solution by considering a random solution. The random solution may be then accepted as the new solution with a certain probability defined a priori.

In the context of the problem, the SA algorithm will strive to find the positions of each of the grid points that minimize the total energy of the embedding. At each iteration, it will pick a random point on the grid, compute the necessary measurements to yield the random solution and the change in energy, ΔH. Then, with probability

$$\frac{e^{-\beta \Delta H}}{e^{\beta \Delta H} + e^{-\beta \Delta H}}$$

where,

$\beta = 1/$"Temperature"
$\Delta H = H_{current}$ - H_{new}

we accept the random solution as the new solution of the system. If the random solution fails to pass the test, then the solution is ignored and a new random solution is calculated for testing. The new solution represents the updated points of the 6 points neighboring the randomly selected point. As the algorithm iterates and the energy of the system cools down, the idea is that the embedding will converge to the desired surface whose discrete measurements match those of the intrinsic surface.

4. Results

For simple testing, let

$$g = \begin{bmatrix} E & F \\ F & G \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

This is the Euclidean metric. Since the Euclidean metric corresponds to planes, the expected result of the embedding is a plane. The evolution of the embedding, which starts out as a slightly perturbed surface is shown in Figure 3 and Figure 4.

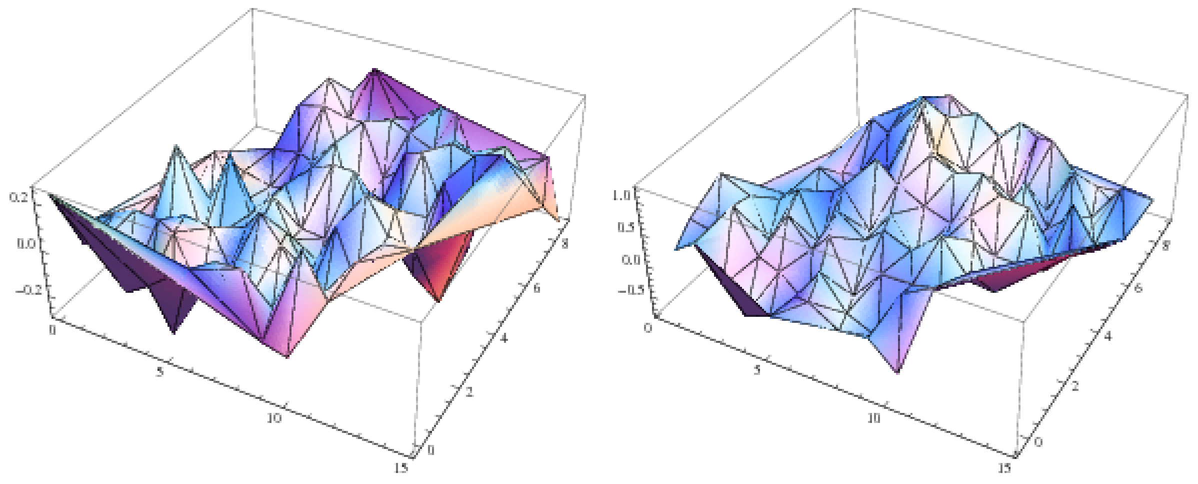

FIGURE 3. Left shows the embedding at the first iteration and the right shows the embedding at the 50th iteration

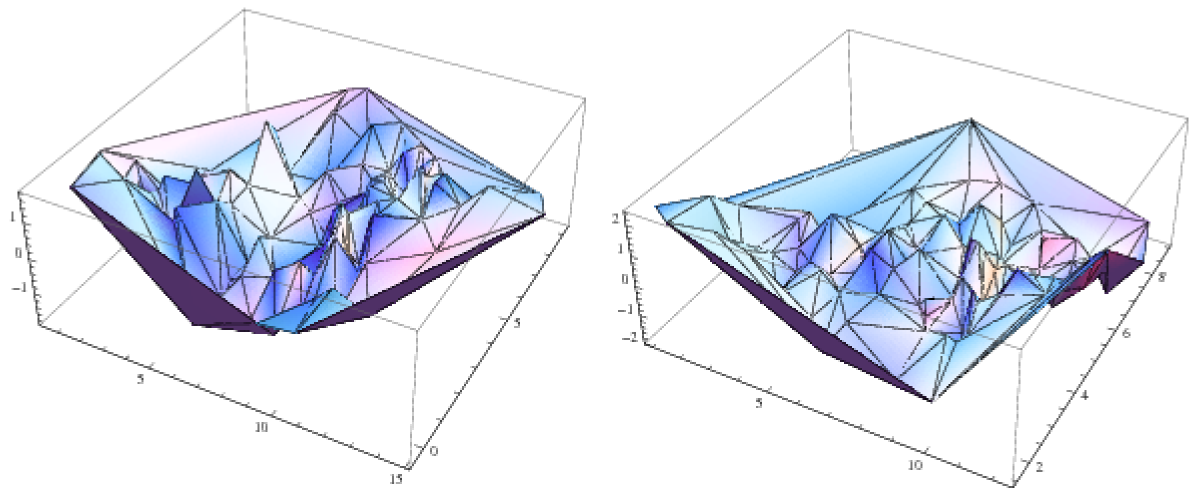

FIGURE 4. Left shows the embedding at the 500th iteration and the right shows the embedding at the 1000th iteration

A qualitative glance at the images of the embedding at various iterations unfortunately does not confirm the effectiveness of the algorithm. Namely, the embedding does not converge to a plane as desired, and seems to wander in a random manner. In fact, the height of the images change from 0.2 at the first iteration to 2 at the 1000th iteration, which indicate that the embedding is slowly growing and is not being contained. However, as desired, the change in energy seems to be converging to a smaller value as the SA algorithm is iterated. For this particular example, ΔH corresponding to the first, 50th, 500th and 1000th iterations were respectively -1.4134, -1.04868, -0.926808 and -0.125748.

5. Conclusion

Simple testing using the Euclidean metric resulted in the algorithm not converging to a desired embedding, namely a plane. However, observing the behavior of the embedding's updates at each iteration, we can predict how the algorithm can be improved in order to reach convergence.

While the random walk on the large search space of all possible surfaces creates fluctuations of the points on the embedding, the optimization problem is insufficiently defined to yield convergence. The random updates of the points on the embedding in Figures 3 and 4 proved this. Currently, the energy function H only includes to minimize the difference between the intrinsic and extrinsic lengths and the intrinsic and extrinsic areas. These constraints were clearly not enough to force the algorithm to find an optimal solution for the embedding. This implies that the algorithm must include further optimization constraints. For example, amending the energy function to include other geometric measurements such as curvature would narrow down the search space. However, the reader should be advised that approximating discrete curvature on a mesh is not a trivial task and would require a good amount of consideration.

References

[1] Ingrid Hotz, *Isometric Embedding By Surface Reconstruction From Distances*, IEEE Visualization 2002, page 252.

[2] Alfred Gray *Modern Differential Geometry of Curves and Surfaces with Mathematica, Second Edition*, CRC-Press, 1997.

[3] Manfredo P. Do Carmo *Differential Geometry of Curves and Surfaces*, Prentice Hall, 1976.

[4] P. Laarhoven, E. Aarts. *Simulated Annealing: Theory and Applications*, Kluwer Academic Publishers, 1989.

[5] Barrett O'Neil: *Some Basic Mathematica: http://www.math.ucla.edu/∼bon/mma.html.*

[6] Hans-Peter Nollert, Heinz Herold (1996): *Visualization in Curves Spacetimes - Visualization of Surfaces via Embedding* F.W.Hehl, R.A.Puntigam, H.Ruder (Eds.) Relativity and Scientific Computing. Springer Berlin Heidelberg, pages 330-351.

[7] Andreas Kuehlmann *The Best of Iccad: 20 Years of Excellence in Computer-Aided Design*, Kluwer Academic Publishers, 2003.

Acknowledgements

I would like to express my gratitude to my advisor, Dr. Tom LaGatta, for his generosity and patience throughout my summer research experience. He dedicated so much of his personal time and energy kindly supervising and mentoring me throughout this difficult experience. I am extremely fortunate to have had the opportunity to learn from him. I would also like to thank Alfredo Hubard for his advisement on the project. Finally, thank you Courant Institute for its financial support and allowing me to pursue undergraduate research.

Mathematica Code

```
Needs["VectorAnalysis`"]

(* First Testing with Euclidean metric *)
ee[u_, v_] := 1;
ff[u_, v_] := 0;
gg[u_, v_] := 1;

gij[ee_, ff_, gg_][u_, v_] := {{ee[u, v], ff[u, v]}, {ff[u, v], gg[u, v]}};
metricDet[ee_, ff_, gg_][u_, v_] := (ee[u, v] * gg[u, v]) - (ff[u, v])^2;

m = 10;
h = 1;

(* Create triangular lattice *)
e1 = {1, 0, 0};
e2 = {Cos[60 Degree], Sin[60 Degree], 0};
flatGrid = Table[i * h * e1 + j * h * e2, {i, 0, m}, {j, 0, m}];

(*Adding noise to the grid*)
gridNoise = 0.1 * h * RandomVariate[NormalDistribution[], {m + 1, m + 1, 3}];
grid = flatGrid + gridNoise;
(*grid // MatrixForm*)


(* Plotting the lattice *)
gridFlat = Partition[Flatten[grid], 3];
ListPlot3D[gridFlat, Mesh → All]

(* Define a coordinate/intrinsic map using first fundamental form *)

ubasis = {1, 0};
vbasis = {Cos[60 Degree], Sin[60 Degree]};
intrinsicGrid = Table[i * h * ubasis + j * h * vbasis, {i, 0, m}, {j, 0, m}];
(* latticepoints = ξ *)
intrinsicFlat = Partition[Flatten[intrinsicGrid], 2];
(*intrinsicGrid//MatrixForm*)
(*ListPlot[intrinsicFlat]*)

iterations = 0;
maxiteration = 1000;

While[ iterations < maxiteration,

 (* Ranomly pick a point x on the intrinsic grid *)

 xI = RandomInteger [{2, m}];
 xJ = RandomInteger [{2, m}];
 x = intrinsicGrid [[xI, xJ]];
```

```
(* Assign the neighboring points to x *)
x1 = intrinsicGrid[[xI + 1, xJ]];
x2 = intrinsicGrid [[xI, xJ + 1]];
x3 = intrinsicGrid [[ xI - 1, xJ + 1]];
x4 = intrinsicGrid [[xI - 1, xJ]];
x5 = intrinsicGrid[[xI, xJ - 1]];
x6 = intrinsicGrid [[xI + 1, xJ - 1]];

NeighborMatrix = {x1, x2, x3, x4, x5, x6};
NeighborMatrix // MatrixForm;
Dimensions[NeighborMatrix];


(* Calculate intrinsic quantities: length (λ) from x to xi,
side length (σ) from xi to xj, area (α) of 6 triangles *)
λ = Table[Integrate[Sqrt[(NeighborMatrix[[i]] - x).
       gij[ee, ff, gg][(x + t * (NeighborMatrix[[i]] - x))[[1]],
        (x + t * (NeighborMatrix[[i]] - x))[[2]]].
       (NeighborMatrix[[i]] - x)], {t, 0, h}], {i, 1, 6}];

σPartial = Table[
   Integrate[Sqrt[(NeighborMatrix[[i + 1]] - NeighborMatrix[[i]]).gij[ee, ff, gg][
        (NeighborMatrix[[i]] + t * (NeighborMatrix[[i + 1]] - NeighborMatrix[[i]]))[[
         1]], (NeighborMatrix[[i]] +
           t * (NeighborMatrix[[i + 1]] - NeighborMatrix[[i]]))[[2]]].
       (NeighborMatrix[[i + 1]] - NeighborMatrix[[i]])], {t, 0, h}], {i, 1, 5}];

σLast = Integrate[Sqrt[(NeighborMatrix[[1]] - NeighborMatrix[[6]]).gij[ee, ff, gg][
       (NeighborMatrix[[6]] + t * (NeighborMatrix[[1]] - NeighborMatrix[[6]]))[[1]],
       (NeighborMatrix[[6]] + t * (NeighborMatrix[[1]] - NeighborMatrix[[6]]))[[2]]].
      (NeighborMatrix[[1]] - NeighborMatrix[[6]])], {t, 0, h}];
σ = Append[σPartial, σLast];

αPartial = Table[Sqrt[3]/2 * Sqrt[metricDet[ee, ff, gg][
        (1/3 * (x + NeighborMatrix[[i]] + NeighborMatrix[[i + 1]]))[[1]],
        (1/3 * (x + NeighborMatrix[[i]] + NeighborMatrix[[i + 1]]))[[2]]]], {i, 1, 5}];
αLast = Sqrt[3]/2 * Sqrt[metricDet[ee, ff, gg][
       (1/3 * (x + NeighborMatrix[[6]] + NeighborMatrix[[1]]))[[1]],
       (1/3 * (x + NeighborMatrix[[6]] + NeighborMatrix[[1]]))[[2]]]];

α = Append[αPartial, αLast];
```

```
(* Corresponding random points on the grid *)

r = grid[[xI, xJ]];
r1 = grid[[xI + 1, xJ]];
r2 = grid [[xI, xJ + 1]];
r3 = grid [[ xI - 1, xJ + 1]];
r4 = grid [[xI - 1, xJ]];
r5 = grid[[xI, xJ - 1]];
r6 = grid [[xI + 1, xJ - 1]];
NeighborMatrixGrid = {r1, r2, r3, r4, r5, r6};

(* Calculate extrinsic quantities: length (λr)from x to xi,
side length (σr) from xi to xj, area (αr )of 6 triangles *)

λr = Table[EuclideanDistance[r, NeighborMatrixGrid [[j]]], {j, 1, 6}];

σrPartial = Table[EuclideanDistance[
    NeighborMatrixGrid[[j]], NeighborMatrixGrid[[j + 1]]], {j, 1, 5}];
σrLast = EuclideanDistance[NeighborMatrixGrid[[6]], NeighborMatrixGrid[[1]]];
σr = Append[σrPartial, σrLast ];

(* Use Heron's Formula to compute the area *)
```

$$\alpha rPartial = Table\left[\frac{1}{4} * Sqrt\left[\left(\lambda r\,[[j]]^2 + \sigma r[[j]]^2 + \lambda r\,[[j+1]]^2\right)^2 - 2 * \left(\lambda r\,[[j]]^4 + \sigma r[[j]]^4 + \lambda r\,[[j+1]]^4\right)\right], \{j, 1, 5\}\right];$$

$$\alpha rLast = \frac{1}{4} * Sqrt\left[\left(\lambda r\,[[6]]^2 + \sigma r[[6]]^2 + \lambda r\,[[1]]^2\right)^2 - 2 * \left(\lambda r\,[[6]]^4 + \sigma r[[6]]^4 + \lambda r\,[[1]]^4\right)\right];$$

```
αr = Append[αrPartial, αrLast];

(* Utility/Energy Function to minimize *)
```

$$HCurrent = Sum\left[E\,\hat{}\,\left((\lambda r[[i]] - \lambda[[i]])^2\right) + E\,\hat{}\,\left((\sigma r[[i]] - \sigma[[i]])^2\right) + E\,\hat{}\,\left(\left(\alpha r[[i]]^2 - \alpha[[i]]^2\right)^2\right) \text{(* + boundary term*)}, \{i, 1, 6\}\right];$$

```
newNeighborMatrix = Array[a, {6, 3}];

λrNew = Table[EuclideanDistance[r, newNeighborMatrix [[j]]], {j, 1, 6}];

σrNewPartial = Table[EuclideanDistance[
    newNeighborMatrix[[j]], newNeighborMatrix[[j + 1]]], {j, 1, 5}];
σrNewLast = EuclideanDistance[newNeighborMatrix[[6]], newNeighborMatrix[[1]]];
σrNew = Append[σrNewPartial, σrNewLast ];
```

```
(* Finding the minimum of energy function *)

Energy = Quiet[FindMinimum[
    Sum[E^((EuclideanDistance[r, newNeighborMatrix[[i]]] - λ[[i]])^2), {i, 1, 6}] +
     Sum[E^((EuclideanDistance[newNeighborMatrix[[j]],
             newNeighborMatrix[[j + 1]]] - σ[[j]])^2), {j, 1, 5}] +
     E^((EuclideanDistance[newNeighborMatrix[[6]], newNeighborMatrix[[1]]] -
          σ[[6]])^2) + Sum[
      E^((((1/4 * Sqrt[(λrNew [[j]]^2 + σrNew[[j]]^2 + λrNew[[j + 1]]^2)^2 - 2 * (λrNew [[j]]^4 +
                    σrNew[[j]]^4 + λrNew [[j + 1]]^4)]))^2 - α[[j]]^2)^2), {j, 1, 5}] +

     E^((((1/4 * Sqrt[((λrNew [[6]]^2 + σrNew[[6]]^2 + λrNew[[1]]^2)^2 -
                  2 * (λrNew[[6]]^4 + σrNew[[6]]^4 + λrNew[[1]]^4))]))^2 - α[[6]]^2)^2),
    {a[1, 1], NeighborMatrixGrid[[1, 1]]}, {a[1, 2], NeighborMatrixGrid[[1, 2]]},
    {a[1, 3], NeighborMatrixGrid[[1, 3]]},
    {a[2, 1], NeighborMatrixGrid[[2, 1]]},
    {a[2, 2], NeighborMatrixGrid[[2, 2]]},
    {a[2, 3], NeighborMatrixGrid[[2, 3]]},
    {a[3, 1], NeighborMatrixGrid[[3, 1]]},
    {a[3, 2], NeighborMatrixGrid[[3, 2]]},
    {a[3, 3], NeighborMatrixGrid[[3, 3]]}, {a[4, 1], NeighborMatrixGrid[[4, 1]]},
    {a[4, 2], NeighborMatrixGrid[[4, 2]]}, {a[4, 3], NeighborMatrixGrid[[4, 3]]},
    {a[5, 1], NeighborMatrixGrid[[5, 1]]}, {a[5, 2], NeighborMatrixGrid[[5, 2]]},
    {a[5, 3], NeighborMatrixGrid[[5, 3]]}, {a[6, 1], NeighborMatrixGrid[[6, 1]]},
    {a[6, 2], NeighborMatrixGrid[[6, 2]]}, {a[6, 3], NeighborMatrixGrid[[6, 3]]}]];

(* Storing the result in a new Matrix called newSol *)
HMin = First[Energy];
newPointMatrix = Partition[Energy[[2]], 3];

newSol = Array[b, {6, 3}];

For[i = 1, i ≤ 6, i++,
  For[j = 1, j ≤ 3, j++,
   newSol[[i, j]] = Last[newPointMatrix[[i, j]]]]];

(* MCMC *)

β = 100;
ΔH = HMin - HCurrent;
prob = E^(-β*ΔH) / (E^(β*ΔH) + E^(-β*ΔH));
If[RandomReal[] < prob , x = 1, x = 0];
```

```
If [x == 1, {grid[[xI + 1, xJ]] = newSol[[1]], grid[[xI, xJ + 1]] = newSol[[2]],
    grid [[xI - 1, xJ + 1]] = newSol[[3]], grid [[xI - 1, xJ]] = newSol[[4]],
    grid[[xI, xJ - 1]] = newSol[[5]], grid [[xI + 1, xJ - 1]] = newSol[[6]] }];

 grid // MatrixForm;
 gridFlat2 = Partition[Flatten[grid], 3] ;
 gridFlatNoEnd = Delete[gridFlat2, 1];
 gridFlatNoEnd2 = Delete[gridFlatNoEnd, (m + 1)^2 - 1];

 If[x == 1, Print[ListPlot3D[gridFlatNoEnd2, Mesh → All]];
   Print[{ΔH, Max[newSol - NeighborMatrixGrid]}]];

 iterations ++
 ]
```