Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Matemática

MAT0309 – Álgebra Linear para Computação – Turma 1 – 2016.2

Prof. Marcelo Ferreira Siqueira

Projeto 2A

POR FAVOR, LEIA AS SEGUINTES INSTRUÇÕES:

- Este trabalho deve ser realizado individualmente ou por grupos de apenas dois discentes.
- O projeto envolve programação e, desta vez, a linguagem de programação tem de ser C++, pois a sua solução será construída a partir de um código já existente.
- O trabalho deverá ser entregue **até o dia 11 de novembro de 2016**, através do envio de um arquivo compactado contendo o diretório com o código correspondente à sua solução.
- O diretório deve possuir um arquivo `README` com o nome completo do(s) autor(es) do trabalho.
- O arquivo compactado deve conter, também, um arquivo PDF com as soluções para as questões da lista de exercícios 2A, que trata de transformações lineares. A pontuação do Projeto 2A leva em conta o código (Questão 1) e as respostas para a lista de exercícios 2A (Questões 2-11).

Questão	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Max	80	2	2	2	2	2	2	2	2	2	2
Pontos											
TOTAL											

1 Introduction

In this project, you are required to implement an algorithm for sampling a parametric surface using a technique called *anisotropic Poisson disk sampling*. The input of the algorithm is a positive real number α and a surface $S \subset \mathbb{E}^3$ such that $S = \Phi(\Omega)$, where $\Phi : \Omega \subset \mathbb{E}^2 \to \mathbb{E}^3$ is a function known as the *parametrization* of S. The parameter *domain*, Ω, is a rectangular region, $[u_0, u_1] \times [v_0, v_1]$ of $\mathbb{E}^2$. The output of the algorithm is a point set P randomly sampled from S and such that the "geodesic" distance between any two points in P are no smaller than λ.

The algorithm must be implemented in `C++` using the GNU g++ compiler, Visual Studio or XCode. You will be provided with empty projects for Linux, Windows, and OSX plataforms. The output of the algorithm (i.e, the set P) must be written to a ASCII file in the format specified in Section 8. To help you visualize the output of the algorithm, you will also be provided with code for generating a *Delaunay triangulation* of the set P. This code will read the output file your code generates, and then will produce another output file with the triangulation. The latter file can be opened with a publicly and freely available tool, with a nice GUI, called `Meshlab`.

2 Well-Spaced Points

The sampling algorithm you are required to implement is a typical algorithm for generating a set of *well-spaced* points. Given a *compact*[1] domain Ω in $\mathbb{E}^2$, and a positive number α, well-spaced points can be defined with the help of two auxiliary disks in $\mathbb{E}^2$: a *point disk* and a *trial disk*. Each point q in Ω is associated with a point disk, $D(q, \alpha)$, centered at q and of radius α. We say that a finite subset P of points from Ω satisfies the *minimum distance property* if and only if the interior of the point disks associated with the points of P are pairwise disjoint. Note that P satisfies the minimum distance property if and only if no two points of P are closer than 2α.

Suppose that P satisfies the minimum distance property. We say that P is *maximal* if and only if

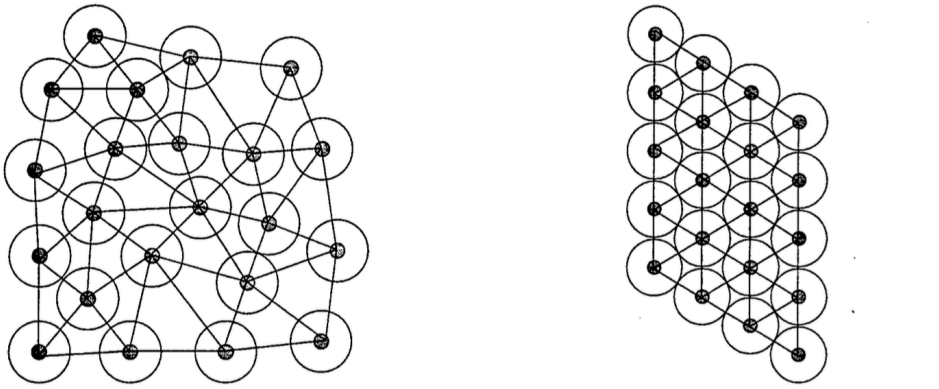
$$D(q, \alpha) \cap \bigcup_{p \in P} D(p, \alpha) \neq \emptyset \,,$$

for every point $q \in \Omega \setminus P$. In other words, P is *maximal* if and only if we cannot add a point q in $\Omega \setminus P$ to P without violating the minimum distance property. Intuitively, this means that the union of the point disks associated with all points from P has no "gaps" of radius equal to or greater than α. A disk $D(q, \beta)$ of radius β, for any radius $\beta > 0$, centered at $q \in \Omega \setminus P$ is called a *trial disk*. Figure 1 shows two point sets in $\mathbb{E}^2$ satisfying the minimum distance property. Both of them are maximal well-spaced, but one is more tightly packed than the other.

For any given set $P \subset \Omega$ satisfying the minimum distance property with respect to a radius

[1] You can think of "compact" as some domain that is contained in a rectangular region of $\mathbb{E}^2$.

α, let $R_{\max}$ be the largest radius of a trial disk. Point set P is said to be *well-spaced* if and only if $R_{\max}$ is not much larger than α, i.e., if and only if for some constant C, $R_{\max} < C \cdot \alpha$. The smaller C is, the better the "quality" of P for some important applications, such as mesh generation. At the very extreme, when $C \leq 1$, set P is said to be a *maximal well-spaced point set*.



(a) maximal point set (b) maximal point set, more tightly packed

Figura 1: Two maximal well-spaced point sets. Figure taken from [2].

A technique called *Poisson Disk Sampling* (or PDS, for short) can be used to generate a maximal well-spaced point set P. The sampling algorithm you are required to implement is based on PDS. It ensures the minimum distance property. Since maximality is not easy to implement, it won't be required in this project. However, you are free to incorporate maximality if you want to.

Algorithm 1 is a generic algorithm for generating a (non-maximal) well-spaced set. Note that the input has one more parameter, MT, which is a constant corresponding to the maximum number of attempts to insert a new point into set P. This constant is used to ensure that the algorithm always terminates, even though the output may not be a maximal well-spaced point set.

3 Anisotropic Sampling

Well-spaced sets, as defined in Section 2, are point sets of a "flat" region such as the rectangular domain $\Omega \subset \mathbb{E}^2$ of parametrization Φ. In this project we are interested in sampling a surface in $\mathbb{E}^3$, which is a not necessarily flat object. This means that P is actually a subset of points from S rather than from Ω. However, one can sample S by using an indirect approach, which also draws points from Ω. Indeed, let q be any point in Ω. Then, we can define a *disk* in S as follows:

$$\Phi(D(q,\alpha)) := \bigcup_{p \in D(q,\alpha)} \Phi(p) \subset S\,,$$

which can be viewed as the *point disk* in S associated with q. Now, we say that *a point set P satisfies the minimum distance property* if and only if, for any two points q and r in Ω, the interior of their associated point disks in S are disjoint; that is, if and only if the intersection of the interior of their associated disks,

$$int(\Phi(D(q,\alpha))) \cap int(\Phi(D(r,\alpha)))\,,$$

is empty. Unfortunately, it is not easy to compute the above intersection for any given parametrization, Φ. So, here, we resort to an approximation to the distance between $\Phi(q)$ and $\Phi(r)$ over S.

Algorithm 1 Algorithm for sampling a surface by the PDS technique.

```
 1: function PDS(Ω, α, MT)
 2:     // Ω: a compact subset of 𝔼²
 3:     // α: a positive real number
 4:     // MT: maximum number of trials
 5:     P ← ∅
 6:     trials ← 0
 7:     while trials < MT do
 8:         q ← uniformly random drawn from Ω
 9:         conflicted ← false
10:         for all q′ ∈ P do
11:             if d(q, q′) < 2 · α then
12:                 conflicted ← true
13:             end if
14:         end for
15:         if not conflicted then
16:             P ← P ∪ {q}
17:             trials ← 0
18:         else
19:             trials ← trials + 1
20:         end if
21:     end while
22:     return P
23: end function
```

What does it mean the *distance between $\Phi(q)$ and $\Phi(r)$ over S*?

Intuitively, you can think of this distance as the length of the shortest curve arc on S that connects $\Phi(q)$ to $\Phi(r)$. This curve (and its length) may also be difficult and expensive to compute, but we can easily approximate the distance. The approximation relies on the Jacobian matrix of Φ. More specifically, let p be any point in Ω. Then, the *Jacobian matrix*, $J_p(\Phi)$, of $D\Phi$ at p is the matrix associated with the linear transformation, $D\Phi : \mathbb{R}^2 \to \mathbb{R}^3$, with respect to the canonical basis $\mathcal{E}_2$ and $\mathcal{E}_3$. The linear transformation $D\Phi$ is the well-known *differential*

of Φ, and

$$J_p(\Phi) := \begin{bmatrix} \frac{\partial \Phi_x}{\partial u}(p) & \frac{\partial \Phi_x}{\partial v}(p) \\ \frac{\partial \Phi_y}{\partial u}(p) & \frac{\partial \Phi_y}{\partial v}(p) \\ \frac{\partial \Phi_z}{\partial u}(p) & \frac{\partial \Phi_z}{\partial v}(p) \end{bmatrix} .$$

Using $J_p(\Phi)$, we can estimate the distance from $\Phi(p)$ to $\Phi(t)$, for every $t \in \Omega$, as follows:

$$d(\Phi(p), \Phi(t)) := \sqrt{\boldsymbol{pt}^t J_p(\Phi)^t J_p(\Phi) \boldsymbol{pt}} \,,$$

where $\boldsymbol{pt}$ is the vector of $\mathbb{R}^2$ defined from p to t. The estimate above is not symmetric, as

$$d(\Phi(t), \Phi(p)) := \sqrt{\boldsymbol{tp}^t J_p(\Phi)^t J_p(\Phi) \boldsymbol{tp}}$$

is not necessarily equal to $d(\Phi(p), \Phi(t))$. This means that function d is not actually a *distance function*. Nevertheless, we can still use it, as we can replace the test on line 11 of Algorithm 1 by

$$d(\Phi(p), \Phi(t)) < 2 \cdot \alpha \quad \textbf{or} \quad d(\Phi(t), \Phi(p)) < 2 \cdot \alpha \,.$$

You may wonder why $d(\Phi(p), \Phi(t))$ is an estimate to the distance between $\Phi(q)$ and $\Phi(r)$ over S. A little bit of Linear Algebra and Calculus can help you understand the reason. Let p be any point in Ω, and let $\boldsymbol{h}$ be a vector in $\mathbb{R}^2$. We have seen in our review of derivatives that $J_p(\Phi)$ can be used to estimate the coordinates of the point $\Phi(p + \boldsymbol{h})$ using a linear approximation, namely

$$\Phi(p + \boldsymbol{h}) \approx \Phi(p) + J_p(\Phi) \cdot \boldsymbol{h} \,.$$

So,

$$\Phi(p + \boldsymbol{h}) - \Phi(p) \approx J_p(\Phi) \cdot \boldsymbol{h} \,,$$

and hence the length of the vector $\Phi(p + \boldsymbol{h}) - \Phi(p)$ is an approximation to the (Euclidean) distance between the points. We have just seen that the length of $\Phi(p + \boldsymbol{h}) - \Phi(p)$ can be computed by the Euclidean norm, $\|\Phi(p+\boldsymbol{h}) - \Phi(p)\|$, of $\Phi(p+\boldsymbol{h}) - \Phi(p)$, which is approximated by

$$\|J_p(\Phi) \cdot \boldsymbol{h}\| = \sqrt{\langle J_p(\Phi) \cdot \boldsymbol{h}, J_p(\Phi) \cdot \boldsymbol{h} \rangle} = \sqrt{(J_p(\Phi) \cdot \boldsymbol{h})^t \cdot J_p(\Phi) \cdot \boldsymbol{h}} = \sqrt{\boldsymbol{h}^t \cdot J_p(\Phi)^t \cdot J_p(\Phi) \cdot \boldsymbol{h}} \,.$$

Matrix

$$J_p(\Phi)^t \cdot J_p(\Phi)$$

is known as the *First Fundamental Form* of S. It is a square, symmetric, *positive definite* matrix. We shall see that this type of matrix can be *diagonalized*, and it has two real positive eigenvalues.

The only problem with the above "discovery" is that we are *not* interested in the *Euclidean* distance between $\Phi(q)$ and $\Phi(r)$, but in the distance between $\Phi(q)$ and $\Phi(r)$ *over* S. By being

more precise about the meaning of such a distance, we can finally understand our distance formula.

Let $\gamma : [s_a, s_b] \subset \mathbb{R} \to \mathbb{E}^2$ be a regular, parametric curve whose trace is a simple, open curve connecting the points $a = \gamma(s_a)$ and $b = \gamma(s_b)$ of $\mathbb{E}^2$. From Calculus, you learned that the *length* of γ, denoted by $\mathcal{L}(\gamma, [s_a, s_b])$, is given by the following integral (refer to your Calculus textbook):

$$\mathcal{L}(\gamma, [s_a, s_b]) := \int_{s_a}^{s_b} \|\gamma'(s)\| ds \,,$$

where

$$\|\gamma'(s)\| = \langle \gamma'(s), \gamma'(s) \rangle \,,$$

is the Euclidean norm (i.e., $\langle \cdot, \cdot \rangle$ is the canonical inner product in $\mathbb{R}^2$), and $\gamma'(s)$ is the derivative vector of γ at s. Now, let us consider the curve $\Gamma : [s_a, s_b] \subset \mathbb{R} \to \mathbb{E}^3$ over S such that $\Gamma(s) = \Phi(\gamma(s))$, for every $s \in [s_a, s_b]$. The length $\mathcal{L}(\Gamma, [s_a, s_b])$ of Γ is also given by an integral, namely,

$$\mathcal{L}(\Gamma, [s_a, s_b]) := \int_{s_a}^{s_b} \|\Gamma'(s)\| ds \,.$$

But,

$$\begin{aligned}
\|\Gamma'(s)\|^2 &= \langle \Gamma'(s), \Gamma'(s) \rangle \\
&= \langle (\Phi(\gamma(s)))', (\Phi(\gamma(s)))' \rangle \\
&= \langle J_{\gamma(s)}(\Phi) \cdot \gamma'(s), J_{\gamma(s)}(\Phi) \cdot \gamma'(s) \rangle \\
&= \langle J_{\gamma(s)}(\Phi) \cdot \gamma'(t), J_{\gamma(s)}(\Phi) \cdot \gamma'(s) \rangle \\
&= (J_{\gamma(s)}(\Phi) \cdot \gamma'(s))^t \cdot (J_{\gamma(s)}(\Phi) \cdot \gamma'(s)) \\
&= \gamma'(s)^t \cdot J_{\gamma(s)}(\Phi)^t \cdot J_{\gamma(s)}(\Phi) \cdot \gamma'(s) \,.
\end{aligned}$$

So,

$$\|\Gamma'(s)\| = \sqrt{\gamma'(s)^t \cdot J_{\gamma(s)}(\Phi)^t \cdot J_{\gamma(s)}(\Phi) \cdot \gamma'(s)} \,,$$

and thus

$$\mathcal{L}(\Gamma, [s_a, s_b]) = \int_{s_a}^{s_b} \left(\sqrt{\gamma'(s)^t \cdot J_{\gamma(s)}(\Phi)^t \cdot J_{\gamma(s)}(\Phi) \cdot \gamma'(s)} \right) ds \,.$$

We can see that the integrand of the above integral looks familiar! Suppose that γ is actually a *line segment*; that is, $\gamma(s) = p + s\boldsymbol{u}$, where $p \in \mathbb{E}^2$ and $\boldsymbol{u}$ is a *unit* vector. This means that $\gamma'(s) = \boldsymbol{u}$, and hence $\|\gamma'(s)\| = 1$, for every $s \in [s_a, s_b]$. Why this assumption on γ? Well, note that

$$\|\gamma(s_b) - \gamma(s_a)\| = \|s_b\boldsymbol{u} - s_a\boldsymbol{u}\| = \|(s_b - s_a)\boldsymbol{u}\| = |s_b - s_a|\|\boldsymbol{u}\| = s_b - s_a \,.$$

So, if we let $\boldsymbol{h} := \gamma(s_b) - \gamma(s_a)$, we only have to choose $s_b = s_a + \|\boldsymbol{h}\|$. Finally, assume that $J_{\gamma(s)}(\Phi) = J_{\gamma(s_a)}(\Phi)$, for every $s \in [s_a, s_b]$. Together, the above two assumptions make the integrand

$$\sqrt{\gamma'(s)^t \cdot J_{\gamma(s)}(\Phi)^t \cdot J_{\gamma(s)}(\Phi) \cdot \gamma'(s)}$$

independent of s. If the two assumptions hold, then we can easily see that length $\mathcal{L}(\Gamma, [s_a, s_b])$ is

$$\begin{aligned}\int_{s_a}^{t_b} \left(\sqrt{\gamma'(s)^t \cdot J_{\gamma(s)}(\Phi)^t \cdot J_{\gamma(s)}(\Phi) \cdot \gamma'(t)}\right) ds &\approx \left(\sqrt{\boldsymbol{u}^t \cdot J_{\gamma(s_a)}(\Phi)^t \cdot J_{\gamma(s_a)}(\Phi) \cdot \boldsymbol{u}}\right) \int_{s_a}^{t_b} dt \\ &= \left(\sqrt{\boldsymbol{u}^t \cdot J_{\gamma(s_a)}(\Phi)^t \cdot J_{\gamma(s_a)}(\Phi) \cdot \boldsymbol{u}}\right) (s_b - s_a) \\ &= \left(\sqrt{\boldsymbol{v}^t \cdot J_{\gamma(s_a)}(\Phi)^t \cdot J_{\gamma(s_a)}(\Phi) \cdot \boldsymbol{v}}\right) ,\end{aligned}$$

where

$$\boldsymbol{v} = (s_b - s_a)\boldsymbol{u} .$$

Now, imagine that $p = \gamma(s_a)$ and that $p+\boldsymbol{h} = \gamma(s_b)$. In this case, we have that $J_{\gamma(s_a)}(\Phi) = J_p(\Phi)$ and $\boldsymbol{v} = \boldsymbol{h}$. Can you see the implications of the two assumptions? Can you explain them to yourself intuitively? Obviously, I am loosely approximating $\mathcal{L}(\Gamma, [s_a, s_b])$. But, what are the issues?

4 Random Points

Line 8 of Algorithm 1 randomly draws a point q from the domain Ω of Φ. This operation was explained to all of you who attend our lecture of October 7, 2016, given by Professor Roberto Teodoro (DMAT-UFRN). For this project, I will provide you with a class named `Random` which can be used to obtain a (pseudo)random number n in the interval $[0, 1]$ of the real line. As Professor Teodoro explained to you, you can use n to compute a (pseudo)random number m in an arbitrary real line interval $[a, b]$, with $b > a$. In fact, all you have to do is to define m as follows:

$$m := a + n \cdot (b - a) . \tag{1}$$

To obtain point q in Line 8 of Algorithm 1, you must obtain (pseudo)random values for the coordinates u and v of q. Since $\Omega = [u_0, v_0] \times [u_1, v_1]$, we let $u = u_0 + n_u \cdot (u_1 - u_0)$ and $v = v_0 + n_v \cdot (v_1 - v_0)$, where n_u and n_v are the two (pseudo)random numbers we obtain by invoking the (pseudo)number generator twice. Your C++ code to obtain (u, v) will look like this

```
// Define a generator of pseudo-random numbers in [0,1].
Random CoordinateValues();

...

// Choose a point (u,v) in the parameter domain.
double u = domain.m_uMin + CoordinateValues() * (domain.m_uMax - domain.m_uMin);
double v = domain.m_vMin + CoordinateValues() * (domain.m_vMax - domain.m_vMin);
```

5 Algorithm

Based on the discussion in Section 3, the algorithm you will implement is Algorithm 2.

Algorithm 2 Algorithm for sampling a surface by the anisotropic PDS technique.

```
1: function PDS(Ω, α, Φ, MT)
2:     // Ω: a compact subset of 𝔼²
3:     // α: a positive real number
4:     // MT: maximum number of trials
5:     P ← ∅
6:     trials ← 0
7:     while trials < MT do
8:         q ← uniformly random drawn from Ω
9:         conflicted ← false
10:        for all q′ ∈ P do
11:            if d(Φ(q), Φ(q′)) < 2 · α or d(Φ(q′), Φ(q)) < 2 · α then
12:                conflicted ← true
13:            end if
14:        end for
15:        if not conflicted then
16:            P ← P ∪ {(q, Φ(q))}
17:            trials ← 0
18:        else
19:            trials ← trials + 1
20:        end if
21:    end while
22:    return P
23: end function
```

6 Code

Once you download the `zip` file associated with Project 2A from `SIGAA` and uncompress this file, make sure you get a directory named `Project2A` with the following subdirectories and files:

```
bin
cdt
common
compile.sh
data
doc
install.sh
lib
pdsampling.pdf
```

```
run.sh
src
```

Subdirectory `src` contains a `main.cpp` file, which should not be modified by you, except for the lines that define the surface to be sampled. Your job is mostly add code (your classes, functions, etc.) to subdirectory `src`. Inside `src` you will see the files `PDSampler.h` and `PDSampler.cpp`. Those are the files you are supposed to edit to implement Algorithm 2. In particular, those two files define class `PDSampler`, which contains a public method, `sample()`, that should implement Algorithm 2. You can always compile your code by using script `compile.sh`, which basically invokes the makefile inside subdirectory `src`. However, the first thing you should do, after downloading the code, is to execute script `install.sh` to make sure library `CDT` is compiled and copied to subdirectory `lib`. Script `install.sh` will also do the job of script `compile.sh`: it will compile the entire code and copy the executable to subdirectory `bin`. Library `CDT` is used by function `main()` in `main.cpp` to create a Delaunay triangulation from the point set your code produces. This triangulation is then written to an output file, which can be opened by `Meshlab` (see details in Section 8). You must download and install `Meshlab`,

`http://meshlab.sourceforge.net/`

to see the triangulations, of course. I will show you, in class, how to use `Meshlab` to render the triangulations you will generate with your own code. `Meshlab` is easy to use and above all, it's fun!

Another important class is `SamplePoint`. This class represents a *sample point*. In this project, a sample point is a pair of points, $(q, \Phi(q))$, where $q \in \Omega$ and $\Phi(q) \in S$. Class `SamplePoint` inherits from class `DtPoint`, which is an abstract class of the `CDT` library. An instance of class `SamplePoint` stores the (u, v) coordinates of a point in Ω and the Cartesian coordinates (x, y, z) of the corresponding point $\Phi(u, v)$ on S. As you can see in the provided code, function `sample()` receives a reference to a list of *shared pointers* to instances of class `SamplePoint`. Your goal is to add points to this list. Make sure you know how to use *shared pointers* in C++. This new feature of C++ free us from having to worry about deallocation of dynamically allocated memory. The `CDT` library takes in the list returned by function `sample()` (in the way it is).

I am providing code for Linux-based systems. If you use Visual Studio or XCode, please ask me for project files to those systems. I can easily generate Visual Studio and XCode projects for you!

7 Input Surface

You will run your code on some parametric surface. So, the code I provide contains another abstract class, called `Surface`, that represents a generic surface S. This class provides us with several virtual methods, which are used to get the limits u_0, u_1, v_0, and v_1 of the intervals $[u_0, u_1]$ and $[v_0, v_1]$ defining Ω, to calculate the coordinates x, y, and z of point $\Phi(u, v)$, to obtain the entries of the Jacobian matrix $J_q(\Phi)$, for $q = (u, v)$, etc. I provide you with a concrete class,

`Cylinder`, that inherits from `Surface` and can be used as your first test case. However, you are encouraged to create your own surface classes, using `Cylinder` as a guide, to test your code.

I also suggest that your implementation of `sample()` include the four corners (u_0, v_0), (u_1, v_0), (u_1, v_1), and (u_1, v_0) of Ω into set P. You can actually initialize set P in Line 5 of Algorithm 2 with

$$\{((u_0, v_0), \Phi(u_0, v_0)), ((u_0, v_1), \Phi(u_0, v_1)), ((u_1, v_0), \Phi(u_1, v_0)), ((u_1, v_1), \Phi(u_1, v_1))\}.$$

Ideally, you should sample the four boundary curves of S using an algorithm similar to Algorithm 2. For that, you could create a (private) function, which is then invoked by `sample()`. Recall that a boundary curve of S is one of the following four curves: $\Phi(u_0, v)$, $\Phi(u_1, v)$, $\Phi(u, v_0)$, and $\Phi(u, v_1)$, for every $u \in [u_0, u_1]$ and for every $v \in [v_0, v_1]$. That is, one of u or v is fixed, while the other varies. You can sample the boundary curves first, and then let P be initialized in Line 5 of Algorithm 2 with the four corners of Ω and the points you sampled on the boundary curves. I guarantee that you will produce points sets that yield "better-looking" triangulations.

8 Output

Function `main()` in `main.cpp` calls a function from the `CDT` library to create a Delaunay triangulation of the point set your `sample()` function generates. The triangulation data is then written to a file with extension `OFF`. This type of file contains the Cartesian coordinates of the points *on the surface* and a description of the triangles of the triangulation, i.e., a sequence of triples, each of which contains the identifiers of the three points that correspond to the vertices of a triangle. Since an `OFF` is a text file, you can easily open and inspect its contents. There are several file formats to represent triangulations. The `OFF` format is just one of them, but it is one of the easiest to work with. `Meshlab` can read several of these type of files, and it can also convert a format into the other. It is indeed a very useful piece of software. I hope you like it!

9 Conclusion

This project has a lot to do with programming, of course. However, the code you are supposed to write is somewhat short. My intention was to give you an application for the Jacobian matrix of a multivariate function. The fact that this matrix is the matrix associated with a linear transformation is the only connection between this project and linear algebra. However, in my opinion, it is a **very important** connection. Moreover, the importance of this connection would be much more appreciated by yourself if you already knew about eigenvalues and eigenvectors.

Why?

If I asked you to analyze the complexity of Algorithm 2, then you can easily conclude that the runtime of the algoritm is in $\Theta(|P|^2)$. I am assuming that the constant MT is much smaller

than $|P|$. Observe that I am stating the complexity in terms of the size of the *output* point set P, which may seem awkward to you. For practical purposes, this complexity is actually pretty high. However, if you had already seen the last topic of our course (Eigenvectors and Eigenvalues), you would have seen a theorem that allows us to lower that complexity to linear in $|P|$ when combined with a standard data structure (i.e., a hash table). The idea behind this code optimization relies on the fact that matrix $J_p(\Phi)^t J_p(\Phi)$ is symmetric, and positive definite. I will discuss this optimization when I lecture on matrix diagonalization and quadratic forms. I strongly believe that this optimization is a typical example where *good theory promotes better practice*. In some sense, it shows us how useful mathematics can be for solving real problems!

I also would like to take this opportunity to comment on the context of this project. First, you should wonder why we care for well-spaced point sets. It turns out that well-spaced point sets yields *high-quality* Delaunay triangulations [2]. What do I mean by that? A high-quality triangulation is a triangulation in which all triangles have very good *aspect-ratio*. Roughly speaking, this means that the internal angles of every triangle are bounded from below by 30^o. Why is this important? Well, several engineering applications must solve very hard differential equations for which there are no known closed formuli. In this case, we resort to numerical methods, such as the Finite Element Method, to compute a (numerical) solution. In turn, those methods require a *mesh* of the problem domain to start with. For two-dimensional problems, a triangulation is a typical mesh. The number and shape quality of the triangles have a huge impact on the accuracy of the numerical solution as well as on the time taken to compute the solution.

A very gentle introduction to the topic "mesh generation" can be found in [1].

Poisson-Disk Sampling (PDS) is currently a hot-topic in the graphics community. You should recall that Algorithm 2 does not ensure that the point set P is *maximal*. Actually, nobody has been able to come up with a simple way of ensuring maximality for points sets obtained *on surfaces*. In other words, this is still an open problem whose solution would be greatly appreciated by the geometry processing and graphics communities. A starting point is the algorithm to generate PDS for *planar* domains given by Yan and Wonka [3]. A colleague of mine and myself were recently able to compute maximal well-spaced point sets on a restricted class of surfaces, but our algorithm relies on notions that are not covered in basic undergraduate courses.

Referências

[1] Marcelo Siqueira, Paulo Pagliosa, and Afonso Paiva. *Geração de Malhas por Refinamento de Delaunay*. 30º Colóquio Brasileiro de Matemática. IMPA, Rio de Janeiro, RJ, Brasil, 2015.

[2] Dafna Talmor. *Well-Spaced Points for Numerical Methods*. PhD thesis, Carnegie Mellon University, Pittsburg, PA, United States of America, August 1997.

[3] Dong-Ming Yan and Peter Wonka. Gap processing for adaptive maximal poisson-disk sampling. *ACM Transactions on Graphics*, 32(5):148:1–148:15, October 2013.