Introducción al método de Elementos Finitos

Brayan Muñoz
John Osorio
Nicole Rivera

Repositorio:
https://github.com/jaosorioh/grupo2_burden12_4.git

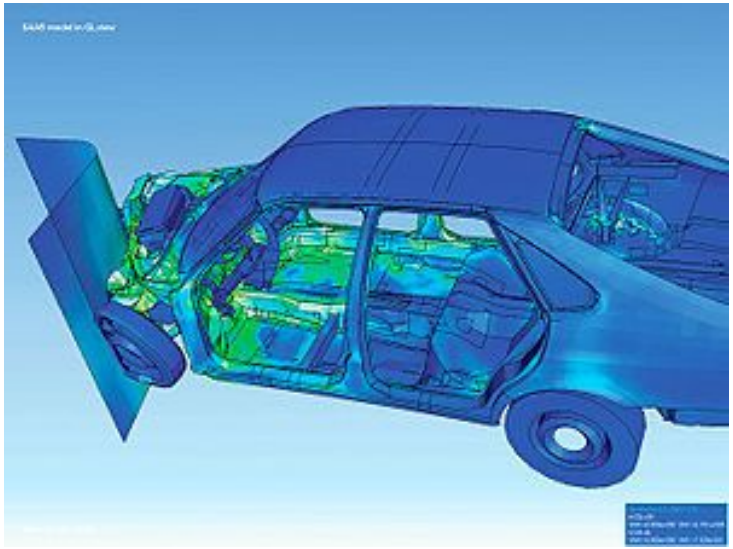
Física Computacional II - Parcial 3

Estudiantes del pregrado de Física

Tabla de contenido

1. Contextualización sobre el método

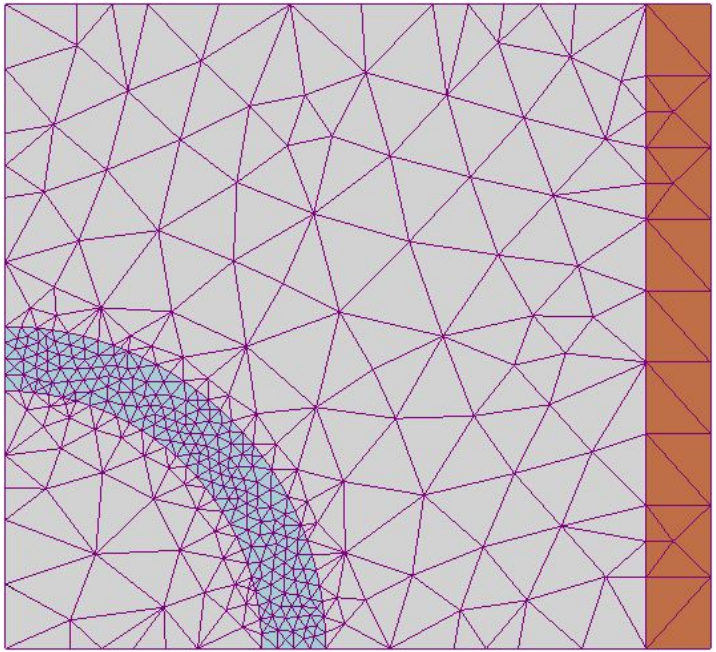
El método de los elementos finitos (en adelante FEM) permite obtener una solución numérica aproximada sobre un cuerpo, estructura o dominio sobre el que están definidas ciertas ecuaciones diferenciales que caracterizan el comportamiento físico del problema. Dicha solución la desarrolla al discretizar en un número elevado de subdominios que no se superpongan entre sí denominados «elementos finitos».



En la imagen: Simulación FEM de la deformación de un auto tras un choque frontal

1. Contextualización sobre el método

A partir de la discretización usando triángulos (Triangulación) se pueden distinguir una serie de puntos representativos llamados ***nodos*** que sirven de base para discretizar el dominio de los elementos finitos. Los cálculos se realizan sobre la malla formada por los nodos y, de esta forma la información del modelo se transmite entre los diferentes elementos a través de los nodos.



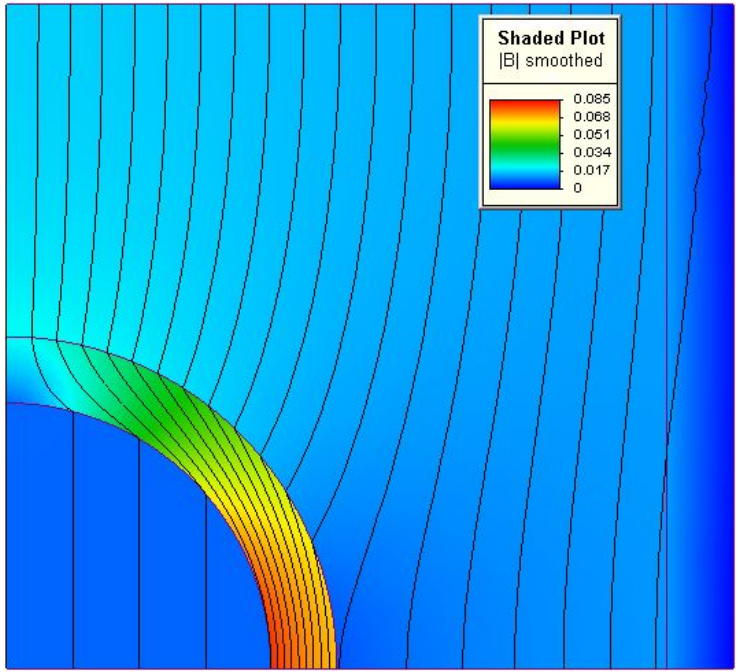
En la imagen: Malla 2D para una configuración de magnetostático

1. Contextualización sobre el método

Sobre cada nodo se define un valor para el conjunto de las variables incógnitas y, las relaciones de estas variables entre todos los nodos se puede escribir como un sistema de ecuaciones lineales

$$A\,c = b$$

donde la matriz A y el vector b contienen la información del problema y la solución c brindará el valor aproximado de la variable incógnita sobre los nodos.



En la imagen: Solución para la configuración de magnetostático. Las líneas muestran la dirección de la densidad de flujo calculada, y el color, su magnitud.

2. Desarrollo matemático del método FEM

FEM permite solucionar problemas físicos que tengan contornos irregulares o condiciones de frontera que involucran derivadas, pues incluye las condiciones de frontera como integrales en un funcional a minimizar. Así pues, la construcción del procedimiento es independiente de las condiciones particulares de cada problema.

Considerando una ecuación diferencial parcial de la forma

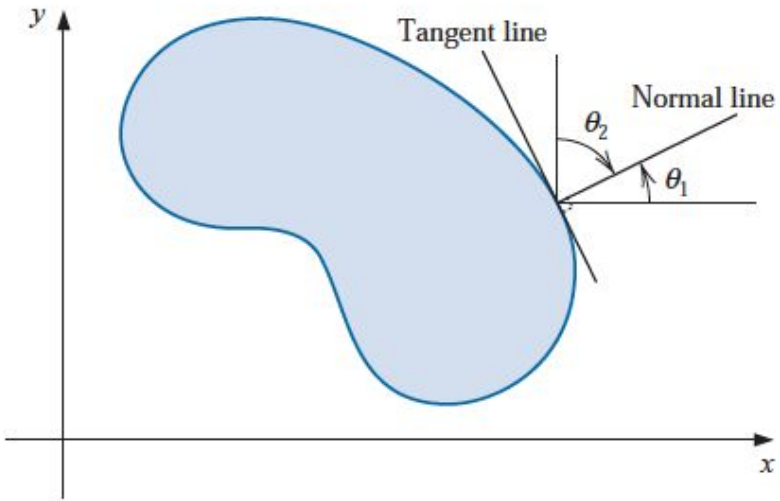
$$\frac{\partial}{\partial x}\left(p(x,y)\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right)+r(x,y)u(x,y)=f(x,y),$$

con $(x,y)\in\mathcal{D}$, donde $\mathcal{D}$ es una región plana con frontera $\mathcal{S}$.

2.

Las condiciones de frontera de la forma $u(x,y) = g(x,y)$ se imponen en una porción de la frontera y, en el resto de ella (denota $\mathcal{S}_2$, $\mathcal{S}_1$) la solución $u(x,y)$ debe satisfacer

$$p(x,y)\frac{\partial u}{\partial x}(x,y)\cos\theta_1 + q(x,y)\frac{\partial u}{\partial y}(x,y)\cos\theta_2 + g_1(x,y)u(x,y) = g_2(x,y),$$



Problemas de este tipo minimizan, típicamente, un funcional que incluye la clase de funciones determinada por el problema

En la imagen: Ángulos directores de la normal a la frontera en el punto (x, y).

2.

Suponga $p, q, r,$ y f continuas en $\mathcal{D} \cup \mathcal{S}$, p y q tienen primeras derivadas parciales co g_1 n g_2 y, , son co $\mathcal{S}_2$ uas en . $p(x,y) > 0,\ q(x,y) > 0, r(x,y) \leq 0, \quad g_1(x,y) > 0.$

Entonces la sc $u(x,y)$ a la ecuación diferencial parcial mostrada minimiza, únicamente, el funcional

$$I[w] = \iint_{\mathcal{D}} \left\{ \frac{1}{2}\left[p(x,y)\left(\frac{\partial w}{\partial x}\right)^2 + q(x,y)\left(\frac{\partial w}{\partial y}\right)^2 - r(x,y)w^2 \right] + f(x,y)w \right\} dx\, dy$$
$$+ \int_{\mathcal{S}_2} \left\{ -g_2(x,y)w + \frac{1}{2} g_1(x,y) w^2 \right\} dS \qquad (12.30)$$

sobre todas las funciones w diferenciables a segundo orden que cumplen $u(x,y) = g(x,y)$ $\mathcal{S}_1$ en

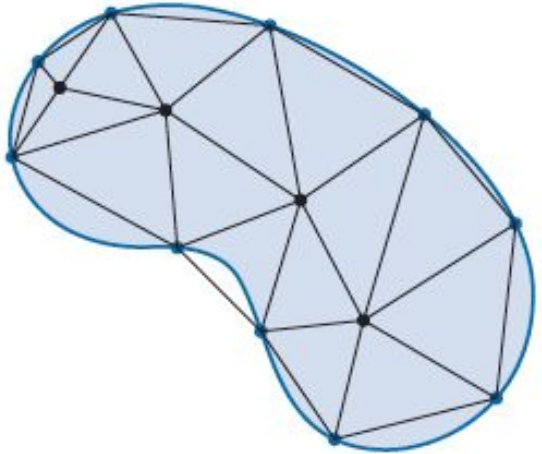
2.1 Definiendo los elementos

El primer paso para implementar FEM es dividir la región en un número finito de secciones o elementos, de un tamaño regular, usando cuadrados o triángulos (en nuestro caso solo usamos triángulos).

Y, las funciones usadas son polinomios a trozos para la aproximación sobre la región

$$\phi(x, y) = a + bx + cy,$$

tal que la función total y su integral sea continua.



En la imagen: Discretización de la región usando triángulos

Ahora, si uno divide la región en varios triángulos, podemos hacer la aproximación de la forma

$$\phi(x,y) = \sum_{i=1}^{m} \gamma_i \phi_i(x,y),$$

donde $\phi_1, \phi_2, \ldots, \phi_m$ son los polinomios a trozos linealmente independientes y $\gamma_1, \gamma_2, \ldots, \gamma_m$ son constantes, de las cuales, algunas $\gamma_{n+1}, \gamma_{n+2}, \ldots, \gamma_m$, son para asegurar que en $\mathcal{S}_1$ la condición

$$\phi(x,y) = g(x,y),$$

se satisfaga.

Dicha expansión usada como las funciones w en el funcional a minimizar, aplicando condiciones de mínimo

$$\frac{\partial I}{\partial \gamma_j} = 0, \quad \text{for each } j = 1, 2, \ldots, n.$$

Lleva, para cada $j = 1, 2, \ldots, n$, a la condición

$$0 = \sum_{i=1}^{m} \left[\iint_{\mathcal{D}} \left\{ p(x,y) \frac{\partial \phi_i}{\partial x}(x,y) \frac{\partial \phi_j}{\partial x}(x,y) + q(x,y) \frac{\partial \phi_i}{\partial y}(x,y) \frac{\partial \phi_j}{\partial y}(x,y) \right.\right.$$
$$\left.\left. - r(x,y)\phi_i(x,y)\phi_j(x,y) \right\} dx\, dy \right.$$
$$\left. + \int_{\mathcal{S}_2} g_1(x,y)\phi_i(x,y)\phi_j(x,y)\, dS \right] \gamma_i$$
$$+ \iint_{\mathcal{D}} f(x,y)\phi_j(x,y)\, dx\, dy - \int_{\mathcal{S}_2} g_2(x,y)\phi_j(x,y)\, dS,$$

Que puede reescribirse como un sistema lineal $A\mathbf{c} = \mathbf{b}$, donde $\mathbf{c} = (\gamma_1, \ldots, \gamma_n)^t$, $A = (\alpha_{ij})$ y $\mathbf{b} = (\beta_1, \ldots, \beta_n)^t$

con las constantes definidas por

$$\alpha_{ij} = \iint_{\mathcal{D}} \left[p(x,y) \frac{\partial \phi_i}{\partial x}(x,y) \frac{\partial \phi_j}{\partial x}(x,y) + q(x,y) \frac{\partial \phi_i}{\partial y}(x,y) \frac{\partial \phi_j}{\partial y}(x,y) - r(x,y)\phi_i(x,y)\phi_j(x,y) \right] dx\, dy + \int_{S_2} g_1(x,y)\phi_i(x,y)\phi_j(x,y)\, dS, \quad (12.33)$$

for each $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$, and

$$\beta_i = -\iint_{\mathcal{D}} f(x,y)\phi_i(x,y)\, dx\, dy + \int_{S_2} g_2(x,y)\phi_i(x,y)\, dS - \sum_{k=n+1}^{m} \alpha_{ik}\gamma_k, \quad (12.34)$$

for each $i = 1, \ldots, n$.

2.2 Triangulando la región

Para empezar el procedimiento se divide la región D en un conjunto de triángulos denotados $T_1, T_2, \ldots, T_M$ cada uno con 3 vértices o nodos denotados.

$$V_j^{(i)} = (x_j^{(i)}, y_j^{(i)}), \quad \text{for } j = 1, 2, 3.$$

Y, a cada vértice se le asocia el polinomio lineal

$$N_j^{(i)}(x, y) \equiv N_j(x, y) = a_j + b_j x + c_j y, \quad \text{where} \quad N_j^{(i)}(x_k, y_k) = \begin{cases} 1, & \text{if } j = k, \\ 0, & \text{if } j \neq k. \end{cases}$$

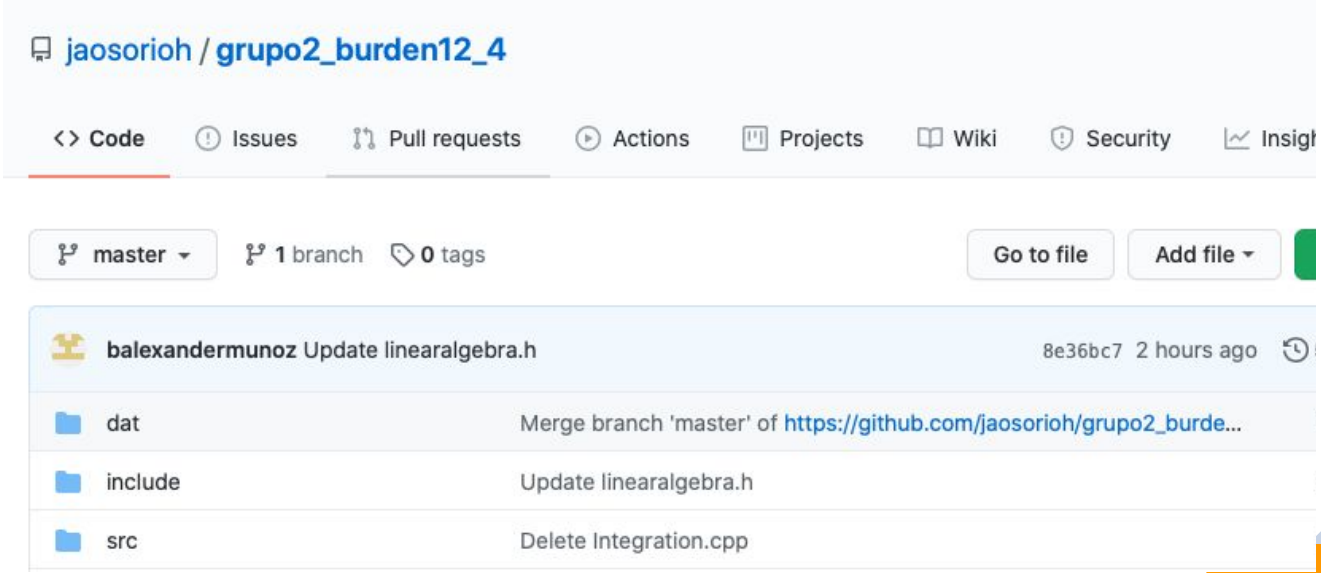
lo que produce un sistema lineal de la forma

$$\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{bmatrix} a_j \\ b_j \\ c_j \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},$$

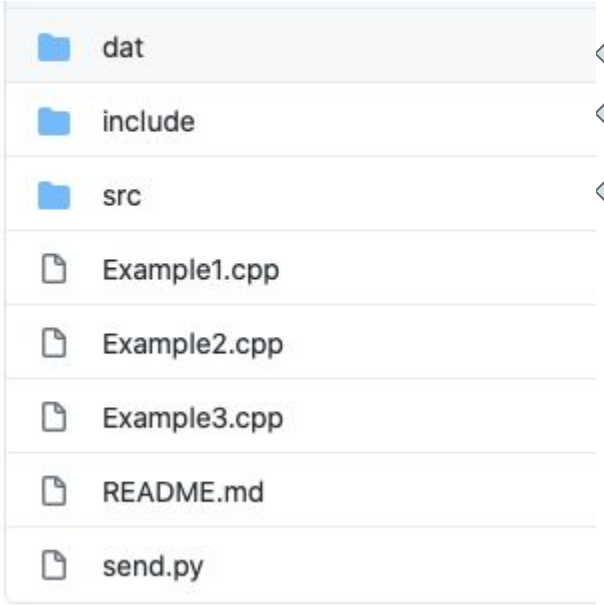
para cada triángulo.

3. Algoritmo desarrollado para aplicar FEM

Se escribió en lenguaje C++ un código que realiza los pasos del algoritmo 12.5 y se almacenó todo en el repositorio https://github.com/jaosorioh/grupo2_burden12_4.git



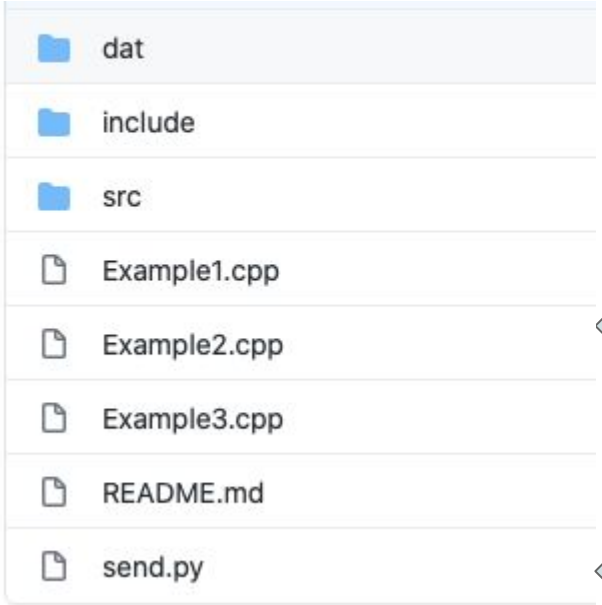
3.1. Estructura general del repositorio



Archivos de datos que se generan o se usan y las mallas

Prototipos de las clases

Implementación de las clases



Main de los ejemplos

Archivo para correr los códigos

3.2 Triangulación de la región

Para realizar la discretización de la región, se hizo una malla de triángulos que utiliza las siguientes clases

```
class Triangulation
```

Esta clase se usa para generar la malla completa sobre la región al acoplar los triángulos que fueron previamente usando la clase Triangle y son clasificados si pertenecen a S1, S2 o a la región D excluyendo la frontera.

```
class Triangle
```

Esta clase se usa para formar triángulos usando los puntos (nodos) que se definen con la clase Point

```
class Point
```

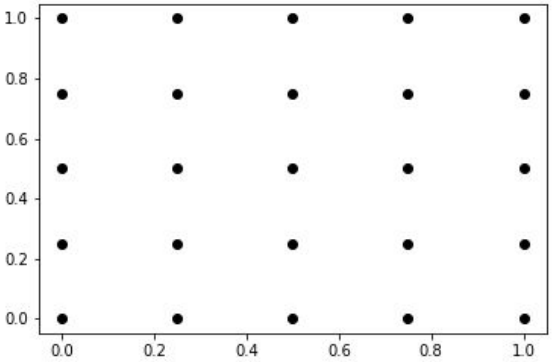
Esta clase fija y permite obtener cada uno de los nodos que se definen sobre la región D y su frontera S

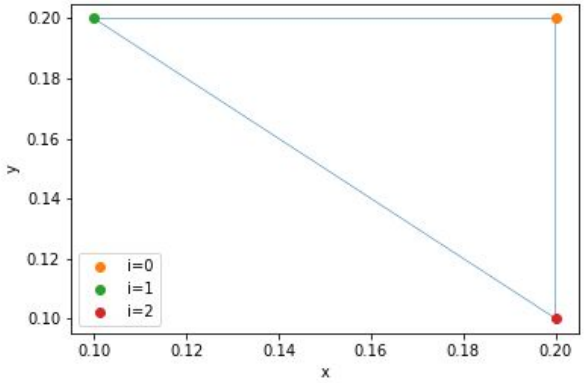
```
#ifndef POINT_H
#define POINT_H

#include <string>
#include <vector>

using namespace std;

class Point {
public:
    Point(double = 0.0, double = 0.0); //constructor
    void setX(double);
    double getX() const;

    void setY(double);
    double getY() const;

    void setXY(double, double);
    string toStr() const;

    double length() const; //para la derivada de curva parametrica

private:
    double x;
    double y;
};
#endif
```

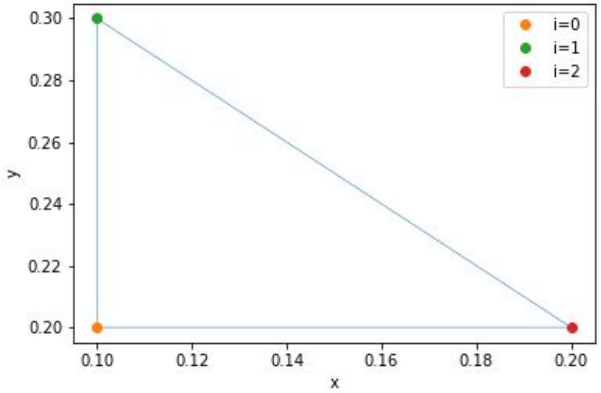
```
class Triangle {
public:
    Triangle(Point &, Point &, Point &);
    Triangle();

    void setV1(Point &);
    Point getV1() const;

    void setV2(Point & );
    Point getV2() const;

    void setV3(Point & );
    Point getV3() const;

    string toStr() const;

    vector<Point> getVertices();

    double getArea() const;
    void setArea(double);//se usa en la integracion

private://los 3 vertices del triangulo, V = vertex
    Point V1;
    Point V2;
    Point V3;
    double Area;

};
```

```
#include "Triangle.h"

using namespace std;

class Triangulation {
public:
    void loadNodes(vector<Point> &, const string &);
    void saveNodes(vector<Point> &, const string &);
    void saveTriangles(vector<Triangle> &, const string &);
    void buildTrianglesAndNodes(vector<Triangle> &, vector<Triangle> &
private:
    bool isTriangleAdded(Triangle&, vector<Triangle> &);
    bool inside(const Point&, double* (*)(const double&, double&));
    bool inside(const Triangle&, double* (*)(const double&, double&));
    bool onS(const Point&, double* (*)(const double&, double&), double
    bool onS(const Triangle&, double* (*)(const double&, double&), dou

};
```

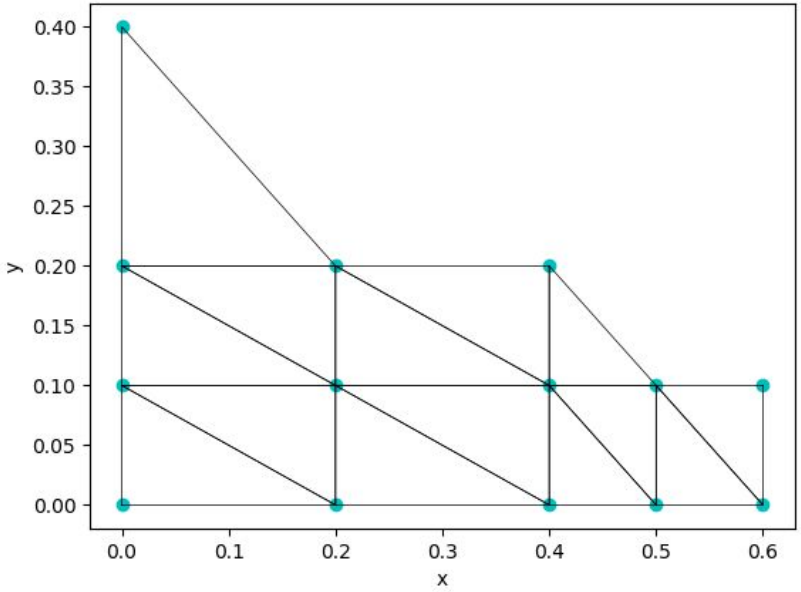
Ilustrando cómo funciona la triangulación







Resultado: se rechazan los puntos por fuera



3.3 Solución de los sistemas de ecuaciones

Para los pasos que requerían de algún método numérico de álgebra lineal se creó una clase llamada linearalgebra que incluye solución de determinantes, de sistemas de ecuaciones lineales como el sistema Ac = b y una función similar al linspace en Python.

```
class linearalgebra {
    // ######### DECLARACIÓN DE FUNCIONES ############## //
public:
    //#### Funcion para resolver determinante:
    double det(vector<vector<double> >& matrix);

    //#### Funciones para resolver sistema lineal (Solo hay que implementar gaussianElimination):

    // Función principal, soluciona AC = B: para obtener el contenido de la matriz y llamar las otras funciones
    //INPUT: Matriz A NxN y vector B de tamaño N
    //OUTPUT: Vector solución C de tamaño N
    vector<double> gaussianElimination(vector<vector<double> > A, vector<double> B);

    void multi_linspace(vector<double>&, int&, vector<double>&);
    void linspace(double&, double&, int&, bool&, vector<double>&);
```

```
private:
    // función para reducir la matriz a r.e.f. Devuelve un valor para indicar si la matriz es singular o no(r.e.f = reductio
    int forwardElim(vector<vector<double> > mat);

    // Funcion para calcular el valor de las incognitas
    vector<double> backSub(vector<vector<double> > mat);

    //Muestra las matrices

    // Función para la operación de elementaria de intercambiar filas
    void swap_row(vector<vector<double> > mat, int i, int j);

    // función para llenar matriz del sistema AC = B con los vectores A y B (el sistema es una matriz con la última columna
    vector<vector<double> > fillSystem(vector<vector<double> > A, vector<double> B);

    //########################################
```

```
vector<double> linearalgebra::gaussianElimination(vector<vector<double> > A, vector<double> B)
{
    vector<vector<double> > mat = fillSystem(A, B);
    int N = mat.size();

    // Reducción
    int singular_flag = forwardElim(mat);

    // Si la matriz es singular
    if (singular_flag != -1) {
        printf("Matriz Singular.\n");

        // si el RHS de la ecuación correspondiente a la fila cero es 0, el sistema tiene infinitas soluciones, de lo contrario es inconsistente
        if (mat[singular_flag][N])
            printf("Sistema inconsistente");
        else
            printf("May have infinitely many "
                   "solutions.");
    }

    // Obtener la solución y guardarla en un vector
    vector<double> Solutions = backSub(mat);
    return Solutions;
}
```

3.3

Ejemplo de los resultados para el algoritmo implementado en los sistemas lineales

```
Determinante a resolver:
|1.5 0 0 |
|0 1.5 0 |
|0 0 1.5 |
El determinante es: 3.375
```

```
Sistema a resolver:
    1    2   -1    3|   -8
    2    0    2   -1|   13
   -1    1    1   -1|    8
    3    3   -1    2|   -1

matriz triangular:
    3    3   -1    2|   -1
    0    2 0.67-0.33|  7.7
    0    0  3.3 -2.7|   21
    0    0    0  1.7| -5.1


Solución del sistema:
x0 = 1
x1 = 2
x2 = 4
x3 = -3
```

3.4 Métodos de integración

Para realizar las integrales, se crea una clase que contiene un método para integración 2D y otro para integrales de línea 2D

```
class Integrals {
public:
    Integrals(void); //constructor

    double Integration2D(Triangle&, function<double(const double&, const double&)>&);

    double lineIntegration(function<double(const double&, const double&)>&, function<Point(const d

    double Simpsons_comp1D(function<double(const double&)>&, double, double);
```

```
private:
    Point derivativeS(function<Point(const double&)>, const double&);
    double Gaussian_quad2D(function<double(const double&, const double&)>&, double, double, double, fu

    //revisar como declarar esto sin que se queje el compilador
    const double r_ij[5][5] = {
        { 0.0, 0.0, 0.0, 0.0, 0.0 },
        // Roots n=2
        { 0.5773502692, -0.5773502692, 0.0, 0.0, 0.0 },
        // Roots n=3
        { 0.7745966692, 0.0000000000, -0.7745966692, 0.0, 0.0 },
        // Roots n=4
        { 0.8611363116, 0.3399810436, -0.3399810436, -0.8611363116, 0.0 },
        // Roots n=5
        { 0.9061798459, 0.5384693101, 0.0000000000, -0.5384693101, -0.9061798459 }
    };

    const double C_ij[5][5] = {
        { 0.0, 0.0, 0.0, 0.0, 0.0 },
        // Roots n=2
        { 1.0000000000, 1.0000000000, 0.0, 0.0, 0.0 },
        // Roots n=3
        { 0.5555555556, 0.8888888889, 0.5555555556, 0.0, 0.0 },
        // Roots n=4
        { 0.3478548451, 0.6521451549, 0.6521451549, 0.3478548451, 0.0 },
        // Roots n=5
        { 0.2369268850, 0.4786286705, 0.5688888889, 0.4786286705, 0.2369268850 }
    };
```

```
double Integrals::Integration2D(Triangle& Tr, function<double(const double& _x, const double& _y)>& toInt)
{
    double x1, y1, x2, y2, x3, y3;
    x1 = Tr.getV1().getX();
    y1 = Tr.getV1().getY();

    x2 = Tr.getV2().getX();

    y2 = Tr.getV2().getY();

    x3 = Tr.getV3().getX();
    y3 = Tr.getV3().getY();


    function<double(const double&, const double&)> x = [&x1, &x2, &x3](const double& s, const double&t) { return x1 + (x2-x1)*s+(x3-x1)*t;};

    function<double(const double&, const double&)> y = [&y1, &y2, &y3](const double& s, const double&t) { return y1 + (y2-y1)*s+(y3-y1)*t;};

    function<double(const double&, const double&)> fst = [&toInt, &x, &y](const double & s, const double &t){
        return toInt(x(s, t), y(s, t));
    };

    function<double(const double&)> t_s = [](const double & s){
        return 1-s;
    };

    double result = Tr.getArea()*Gaussian_quad2D(fst, 0, 1, 0, t_s);

    return result;
}
```

```
double Integrals::lineIntegration(function<double(const double& _x, const double& _y)>& toInt,
                function<Point(const double& t_)>& SE, function<Point(const double& t_)>& DSE, const double ta, const double tb)
{
    //Ingresar al integrador
    double result;
    function<double(const double&)> f = [this, &toInt, &SE, &DSE](const double& t) {
        double al = DSE(t).length();
        Point p = SE(t);
        return toInt(p.getX(), p.getY()) * al;
    };

    result = Simpsons_comp1D(f, ta, tb);

    return result;
}
```

3.5 Ensamble de todos los pasos

Para acoplar los métodos anteriores teniendo en cuenta los pasos a seguir en el algoritmo del libro, se creó una clase llamada FEM

```
#include "Triangulation.h"
#include <algorithm>
#include <iostream>
#include <functional>
#include "linearalgebra.h"
#include "Integrals.h"

using namespace std;

class FEM {
public:
    FEM(double* (*)(const double&, double&), double* (*)(double&, const double&), function<double(const double&, const double&)>&,
    void solve();
```

```
private:
    void coefABC(Triangle&, vector<double>&, vector<double>&, vector<double>&);
    void coefABC();
    void doubleIntegrals();
    void lineIntegrals();
    int findNodeIndex(Point&);
    int findNodeIndex(Point&, vector<Point>);
    void assembleDoubleIntegrals();
    void assembleLineIntegrals();
    vector<Triangle> Triangles;
    vector<Point> Nodes;
    int K, N, M, n, m;

    vector<vector<double> > alpha;
    vector<double> beta;
    vector<double> gamma;
    vector<vector<double> > A;
    vector<vector<double> > B;
    vector<vector<double> > C;
    vector<double> xx;
    vector<double> yy;
    vector<vector<vector<double> > > Z;
    vector<vector<double> > H;
    vector<vector<vector<double> > > J;
    vector<vector<double> > I;

    function<double(const double&, const double&)> p;
    function<double(const double&, const double&)> q;
    function<double(const double&, const double&)> r;
    function<double(const double&, const double&)> f;
    function<double(const double&, const double&)> g;
    function<double(const double&, const double&)> g1;
    function<double(const double&, const double&)> g2;
    function<Point(const double&)> SE;
    function<Point(const double&)> DSE;
    double ta;
    double tb;

    Integrals Integrator;
    linearalgebra la;
    Triangulation triang;

};
```

Pasos 0, 1 y 2 se realizan en el constructor de la clase FEM

Step 0 Divide the region D into triangles $T_1, \ldots, T_M$ such that:
$T_1, \ldots, T_K$ are the triangles with no edges on S_1 or S_2;
(*Note:* $K = 0$ *implies that no triangle is interior to* D.)
$T_{K+1}, \ldots, T_N$ are the triangles with at least one edge on S_2;
$T_{N+1}, \ldots, T_M$ are the remaining triangles.
(*Note:* $M = N$ *implies that all triangles have edges on* S_2.)
Label the three vertices of the triangle T_i by
$\left(x_1^{(i)}, y_1^{(i)}\right), \left(x_2^{(i)}, y_2^{(i)}\right)$, and $\left(x_3^{(i)}, y_3^{(i)}\right)$.
Label the nodes (vertices) $E_1, \ldots, E_m$ where
$E_1, \ldots, E_n$ are in $D \cup S_2$ and $E_{n+1}, \ldots, E_m$ are on S_1.
(*Note:* $n = m$ *implies that* S_1 *contains no nodes.*)

Step 1 For $l = n + 1, \ldots, m$ set $\gamma_l = g(x_l, y_l)$. (*Note:* $E_l = (x_l, y_l)$.)

Step 2 For $i = 1, \ldots, n$
set $\beta_i = 0$;
for $j = 1, \ldots, n$ set $\alpha_{i,j} = 0$.

```
FEM::FEM(double* (*S2Fx_)(const double& x, double& y), double* (*S2Fy_)(double& x, const double& y),
         function<double(const double& x_, const double& y_)>& p_, function<double(const double& x_, const double& y_)>& q_,
         function<double(const double& x_, const double& _y)>& r_, function<double(const double& x_, const double& y_)>& f_,
         function<double(const double& x_, const double& y_)>& g_, function<double(const double& x_, const double& y_)>& g1_,
         function<double(const double& x_, const double& y_)>& g2_, vector<double>& x_i, vector<double>& y_i, int& N_, int& M_,
         function<Point(const double& t_)>& SE_, function<Point(const double& t_)>& DSE_, const double& ta_, const double& tb_)
{
    //---------------STEP # 1 ---------------
    vector<Triangle> trianglesnotS1S2;
    vector<Triangle> trianglesS2;
    vector<Triangle> trianglesS1;

    la.multi_linspace(x_i, N_, xx);
    la.multi_linspace(y_i, M_, yy);

    triang.buildTrianglesAndNodes(trianglesS1, trianglesS2, trianglesnotS1S2, S2Fx_, S2Fy_, xx, yy);

    Triangles = trianglesnotS1S2;
    K = Triangles.size();
```

```
Triangles.insert(Triangles.end(), trianglesS2.begin(), trianglesS2.end());
N = Triangles.size();
Triangles.insert(Triangles.end(), trianglesS1.begin(), trianglesS1.end());

M = Triangles.size();
for (int i = 0; i < N; i++) {
    vector<Point> nodes = Triangles[i].getVertices();
    for (int j = 0; j < nodes.size(); j++) {
        int l = findNodeIndex(nodes[j]);
        if (Nodes.size() == 0 || l >= Nodes.size()) {
            Nodes.push_back(nodes[j]);
        }
    }
}
n = Nodes.size();
```

```
for (int i = N; i < M; i++) {
    vector<Point> nodes = Triangles[i].getVertices();
    for (int j = 0; j < nodes.size(); j++) {
        int l = findNodeIndex(nodes[j]);
        if (Nodes.size() == 0 || l >= Nodes.size()) {
            Nodes.push_back(nodes[j]);
        }
    }
}

m = Nodes.size();
//---------------VALORES K, N, M, n, m---------------
cout << "n: " << n << ", m: " << m << endl;
cout << "K: " << K << ", N: " << N << ", M: " << M << endl;
//---------------STEP 1---------------
gamma = vector<double>(m, 0);
for (int i = n; i < m; i++) {

    gamma[i] = g_(Nodes[i].getX(), Nodes[i].getY());
}
```

```
    //---------------STEP 2---------------
    alpha = vector<vector<double> >(n, vector<double>(n, 0));

    beta = vector<double>(n, 0);

    Z = vector<vector<vector<double> > >(M, vector<vector<double> >(3, vector<double>(3, 0)));
    H = vector<vector<double> >(M, vector<double>(3, 0));

    J = vector<vector<vector<double> > >(N - K, vector<vector<double> >(3, vector<double>(3, 0)));
    I = vector<vector<double> >(N - K, vector<double>(3, 0));

    p = p_;
    q = q_;
    r = r_;
    f = f_;
    g = g_;
    g1 = g1_;
    g2 = g2_;
    SE = SE_;
    DSE = DSE_;
    ta = ta_;
    tb = tb_;
}
```

Paso 3 se realiza en la función coefABC

Step 3 For $i = 1, \ldots, M$

$$\text{set } \Delta_i = \det \begin{vmatrix} 1 & x_1^{(i)} & y_1^{(i)} \\ 1 & x_2^{(i)} & y_2^{(i)} \\ 1 & x_3^{(i)} & y_3^{(i)} \end{vmatrix};$$

$$a_1^{(i)} = \frac{x_2^{(i)} y_3^{(i)} - y_2^{(i)} x_3^{(i)}}{\Delta_i}; \qquad b_1^{(i)} = \frac{y_2^{(i)} - y_3^{(i)}}{\Delta_i}; \qquad c_1^{(i)} = \frac{x_3^{(i)} - x_2^{(i)}}{\Delta_i};$$

$$a_2^{(i)} = \frac{x_3^{(i)} y_1^{(i)} - y_3^{(i)} x_1^{(i)}}{\Delta_i}; \qquad b_2^{(i)} = \frac{y_3^{(i)} - y_1^{(i)}}{\Delta_i}; \qquad c_2^{(i)} = \frac{x_1^{(i)} - x_3^{(i)}}{\Delta_i};$$

$$a_3^{(i)} = \frac{x_1^{(i)} y_2^{(i)} - y_1^{(i)} x_2^{(i)}}{\Delta_i}; \qquad b_3^{(i)} = \frac{y_1^{(i)} - y_2^{(i)}}{\Delta_i}; \qquad c_3^{(i)} = \frac{x_2^{(i)} - x_1^{(i)}}{\Delta_i};$$

for $j = 1, 2, 3$

define $N_j^{(i)}(x, y) = a_j^{(i)} + b_j^{(i)} x + c_j^{(i)} y.$

```
//---------------STEP 3---------------
//Calcula los vectores a, b, c del triángulo i
void FEM::coefABC(Triangle& T, vector<double>& a, vector<double>& b, vector<double>& c)
{
    double x1, x2, x3, y1, y2, y3;
    x1 = T.getV1().getX();
    y1 = T.getV1().getY();
    x2 = T.getV2().getX();
    y2 = T.getV2().getY();
    x3 = T.getV3().getX();
    y3 = T.getV3().getY();

    vector<vector<double> > matrix = { { 1.0, x1, y1 },
        { 1.0, x2, y2 },
        { 1.0, x3, y3 } };
    double delta = la.det(matrix);
    double a1, a2, a3;
    a1 = (x2 * y3 - y2 * x3) / delta;
    a2 = (x3 * y1 - y3 * x1) / delta;
    a3 = (x1 * y2 - y1 * x2) / delta;
    a.insert(a.end(), { a1, a2, a3 });

    double b1, b2, b3;
    b1 = (y2 - y3) / delta;
    b2 = (y3 - y1) / delta;
    b3 = (y1 - y2) / delta;
    b.insert(b.end(), { b1, b2, b3 });

    double c1, c2, c3;
    c1 = (x3 - x2) / delta;
    c2 = (x1 - x3) / delta;
    c3 = (x2 - x1) / delta;

    c.insert(c.end(), { c1, c2, c3 });
}
```

Paso 4 se realiza en la función doubleIntegrals

Step 4 For $i = 1, \ldots, M$ *(The integrals in Steps 4 and 5 can be evaluated using numerical integration.)*

for $j = 1, 2, 3$

for $k = 1, \ldots, j$ *(Compute all double integrals over the triangles.)*

$$\text{set } z_{j,k}^{(i)} = b_j^{(i)} b_k^{(i)} \iint T_i p(x,y)\, dx\, dy + c_j^{(i)} c_k^{(i)} \iint T_i q(x,y)\, dx\, dy - \iint T_i r(x,y) N_j^{(i)}(x,y) N_k^{(i)}(x,y)\, dx\, dy;$$

$$\text{set } H_j^{(i)} = -\iint T_i f(x,y) N_j^{(i)}(x,y)\, dx\, dy.$$

Y el paso 5 usando la función lineIntegrals

Step 5 For $i = K + 1, \ldots, N$ *(Compute all line integrals.)*

for $j = 1, 2, 3$

for $k = 1, \ldots, j$

$$\text{set } J_{j,k}^{(i)} = \int_{S_2} g_1(x,y) N_j^{(i)}(x,y) N_k^{(i)}(x,y)\, dS;$$

$$\text{set } I_j^{(i)} = \int_{S_2} g_2(x,y) N_j^{(i)}(x,y)\, dS.$$

```
//---------------STEP 4---------------
void FEM::doubleIntegrals()
{
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k <= j; k++) {
                function<double(const double&, const double&)> inte = [this, &i, &j, &k](const double& x, const double& y) {
                    return this->r(x, y) * (this->A[i][j] + this->B[i][j] * x + this->C[i][j] * y) * (this->A[i][k] + this->B[i][k] * x + this->C[i][k] * y);
                };
                Z[i][j][k] = B[i][j] * B[i][k] * Integrator.Integration2D(Triangles[i], p) + C[i][j] * C[i][k] * Integrator.Integration2D(Triangles[i], q) - Integrato
                                                                                                                    r.Integration2D(Triangles[i], inte);

            }

            function<double(const double&, const double&)> inte2 = [this, &i, &j](const double& x, const double& y) {
                return this->f(x, y) * (this->A[i][j] + this->B[i][j] * x + this->C[i][j] * y);
            };

            H[i][j] = -Integrator.Integration2D(Triangles[i], inte2);
        }
    }
}
```

```
//---------------STEP 5---------------
void FEM::lineIntegrals()
{
    for (int i = K; i < N; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k <= j; k++) {
                function<double(const double&, const double&)> inte = [this, &i, &j, &k](const double& x, const double& y) {
                    return this->g1(x, y) * (this->A[i][j] + this->B[i][j] * x + this->C[i][j] * y) * (this->A[i][k] + this->B[i][k] * x + this->C[i][k] * y);
                };

                J[i - K][j][k] = Integrator.lineIntegration(inte, SE, DSE, ta, tb);
            }

            function<double(const double&, const double&)> inte2 = [this, &i, &j](const double& x, const double& y) {
                return this->g2(x, y) * (this->A[i][j] + this->B[i][j] * x + this->C[i][j] * y);
            };

            I[i - K][j] = Integrator.lineIntegration(inte2, SE, DSE, ta, tb);
        }
    }
}
```

El paso 6, el cual incluye los pasos 7-12 usa dos funciones: findNodeIndex y assembleDoubleIntegrals

Step 6 For $i = 1, \ldots, M$ do Steps 7–12. (*Assembling the integrals over each triangle into the linear system.*)

Step 7 For $k = 1, 2, 3$ do Steps 8–12.

Step 8 Find l so that $E_l = \left(x_k^{(i)}, y_k^{(i)}\right)$.

Step 9 If $k > 1$ then for $j = 1, \ldots, k-1$ do Steps 10, 11.

Step 10 Find t so that $E_t = \left(x_j^{(i)}, y_j^{(i)}\right)$.

Step 11 If $l \leq n$ then

if $t \leq n$ then set $\alpha_{lt} = \alpha_{lt} + z_{k,j}^{(i)}$;

$\alpha_{tl} = \alpha_{tl} + z_{k,j}^{(i)}$

else set $\beta_l = \beta_l - \gamma_t z_{k,j}^{(i)}$

else

if $t \leq n$ then set $\beta_t = \beta_t - \gamma_l z_{k,j}^{(i)}$.

Step 12 If $l \leq n$ then set $a_{ll} = \alpha_{ll} + z_{k,k}^{(i)}$;

$\beta_l = \beta_l + H_k^{(i)}$.

```
//---------------STEP 6---------------

int FEM::findNodeIndex(Point& p)
{
    int i = 0;
    while (i < Nodes.size()) {
        //cout << i <<" - (" << Nodes[i].toStr() <<") and (" << p.toStr() <<") are " << ((p
        if (p.getX() == Nodes[i].getX() && p.getY() == Nodes[i].getY())
            break;
        i++;
    }
    return i;
}

int FEM::findNodeIndex(Point& p, vector<Point> nd)
{
    int i = 0;
    while (i < nd.size()) {
        if (p.getX() == nd[i].getX() && p.getY() == nd[i].getY())
            break;
        i++;
    }
    return i;
}
```

```
void FEM::assembleDoubleIntegrals()
{
    for (int i = 0; i < M; i++) {
        vector<Point> Nodesi = Triangles[i].getVertices(); //Los 3 Nodos del triángulo i

        //Paso 7: correr k = 1,2,3 (En nuestro caso 0,1,2)
        for (int k = 0; k < 3; k++) {
            //Paso 8: Encontramos el indice l en la lista de todos los nodos en S1US2
            int l = findNodeIndex(Nodesi[k]);
            //Paso 9:
            if (k > 0) { //If k != 0/
                for (int j = 0; j < k - 1; j++) { // j =0,1...k-1

                    //Paso 10:
                    int t = findNodeIndex(Nodesi[j]);
                    //Paso 11:

                    if (l < n) {
                        if (t < n) {
                            alpha[l][t] = alpha[l][t] + Z[i][k][j];
                            alpha[t][l] = alpha[t][l] + Z[i][k][j];
                        }
                        else {
                            beta[l] = beta[l] - gamma[t] * Z[i][k][j];
                        }
                    }
                    else if (t < n) {
                        beta[t] = beta[t] - gamma[l] * Z[i][k][j];
                    }
                }
            }
            //Paso 12:

            if (l < n) {
                alpha[l][l] = alpha[l][l] + Z[i][k][k];
                beta[l] = beta[l] + H[i][k];
            }
        }
    }

} //Fin :D
```

El paso 13 que incluye los pasos 14-19 usa la función assembleLineIntegrals

Step 13 For $i = K + 1, \ldots, N$ do Steps 14–19. (*Assembling the line integrals into the linear system.*)

Step 14 For $k = 1, 2, 3$ do Steps 15–19.

Step 15 Find l so that $E_l = \left(x_k^{(i)}, y_k^{(i)}\right)$.

Step 16 If $k > 1$ then for $j = 1, \ldots, k-1$ do Steps 17, 18.

Step 17 Find t so that $E_t = \left(x_j^{(i)}, y_j^{(i)}\right)$.

Step 18 If $l \leq n$ then

if $t \leq n$ then set $\alpha_{lt} = \alpha_{lt} + J_{k,j}^{(i)}$;

$\alpha_{tl} = \alpha_{tl} + J_{k,j}^{(i)}$

else set $\beta_l = \beta_l - \gamma_t J_{k,j}^{(i)}$

else

if $t \leq n$ then set $\beta_t = \beta_t - \gamma_l J_{k,j}^{(i)}$.

Step 19 If $l \leq n$ then set $\alpha_{ll} = \alpha_{ll} + J_{k,k}^{(i)}$;

$\beta_l = \beta_l + I_k^{(i)}$.

```
void FEM::assembleLineIntegrals()
{
    //Paso 13: correr sobre los triangulos con al menos un edge en S2
    //Triángulos con al menos un edge en S2, sobre esto corre el for
    for (int i = K; i < N; i++) {
        vector<Point> Nodesi = Triangles[i].getVertices();
        //Los 3 Nodos del triángulo i

        //Paso 14: correr sobre k = 1,2,3 (0,1,2 en nuestro caso)
        for (int k = 0; k < 3; k++) {
            //Paso 15: encontrar l en la lista de nodos

            int l = findNodeIndex(Nodesi[k]); //Encontramos el indice l
            //Paso 16:
            if (k > 0) { //If k != 0
                for (int j = 0; j < k - 1; j++) { // j =0,1...k-1

                    //Paso 17: encontrar t
                    int t = findNodeIndex(Nodesi[j]);

                    //Paso 18:
                    if (l < n) {
                        if (t < n) {
                            alpha[l][t] = alpha[l][t] + J[i - K][k][j];
                            alpha[t][l] = alpha[t][l] + J[i - K][k][j];
                        }
                        else {
                            beta[l] = beta[l] - gamma[t] * J[i - K][k][j];
                        }
                    }
                    else if (t < n) {
                        beta[t] = beta[t] - gamma[l] * J[i - K][k][j];
                    }
                }
            }
            //Paso 19:
            if (l < n) {
                alpha[l][l] = alpha[l][l] + J[i - K][k][k];
                beta[l] = beta[l] + I[i - K][k];
            }
        }
    }

} //Fin :D
```

Y, finalmente, la función solve que ejecuta todos los pasos anteriores y los restantes 20-23

Step 20 Solve the linear system $A\mathbf{c} = \mathbf{b}$ where $A = (\alpha_{l,t}), \mathbf{b} = (\beta_l)$ and $\mathbf{c} = (\gamma_t)$ for $1 \le l \le n$ and $1 \le t \le n$.

Step 21 OUTPUT $(\gamma_1, \ldots, \gamma_m)$.
(*For each* $k = 1, \ldots, m$ *let* $\phi_k = N_j^{(i)}$ *on* T_i *if* $E_k = \left(x_j^{(i)}, y_j^{(i)}\right)$.
Then $\phi(x, y) = \sum_{k=1}^{m} \gamma_k \phi_k(x, y)$ *approximates* $u(x, y)$ *on* $D \cup S_1 \cup S_2$.)

Step 22 For $i = 1, \ldots, M$
for $j = 1, 2, 3$ OUTPUT $\left(a_j^{(i)}, b_j^{(i)}, c_j^{(i)}\right)$.

Step 23 STOP. (*The procedure is complete.*)

```
void FEM::solve()
{
    coefABC();
    doubleIntegrals();
    assembleDoubleIntegrals();
    lineIntegrals();
    assembleLineIntegrals();
    //---------------STEP 20---------------
    vector<double> gamma2 = la.gaussianElimination(alpha, beta);

    for (int i = 0; i < n; i++) {
        gamma[i] = gamma2[i];
    }

    ofstream resultsFile("dat/Results.dat");

    if (resultsFile.fail()) {
        cout << "Results file cannot be opened." << endl;
        exit(1);
    }
    //---------------STEP 21-22---------------
    for (int i = 0; i < M; i++) {
        vector<Point> vtx = Triangles[i].getVertices();
        for (int j = 0; j < 3; j++) {
            int m = findNodeIndex(vtx[j]);
            resultsFile << A[i][j] << " " << B[i][j] << " " << C[i][j] << " " << gamma[m] << endl;
        }
    }
    triang.saveTriangles(Triangles, "dat/Triangles.dat");

    triang.saveNodes(Nodes, "dat/Nodes.dat");
}
```

4.2 4.1 Ecuación de Poisson

El ejemplo es la ecuación de Poisson (*ver Example2.cpp)*:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \sin(\pi x)\sin(\pi y)$$

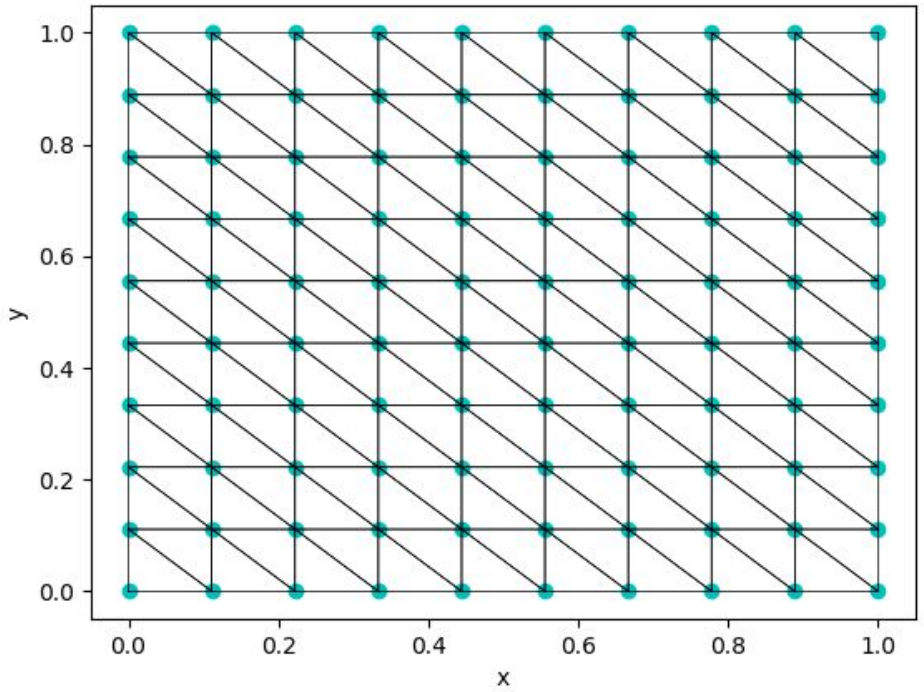
Con las condiciones

$$u(x, y) = 0 \text{ along the boundaries}, \quad 0 \le x, y \le 1$$

$$u_x(0, y) = -\frac{\sin(y\pi)}{2\pi}.$$

Este problema representa un modelo simple para la distribución de temperatura (x, y) en una placa cuadrada. La fuente específica de temperatura modela el calentamiento uniforme de la placa, y la condición de contorno modela el borde de la placa que se mantiene a una temperatura baja. La forma simple del dominio permite que la solución se representa explícitamente.

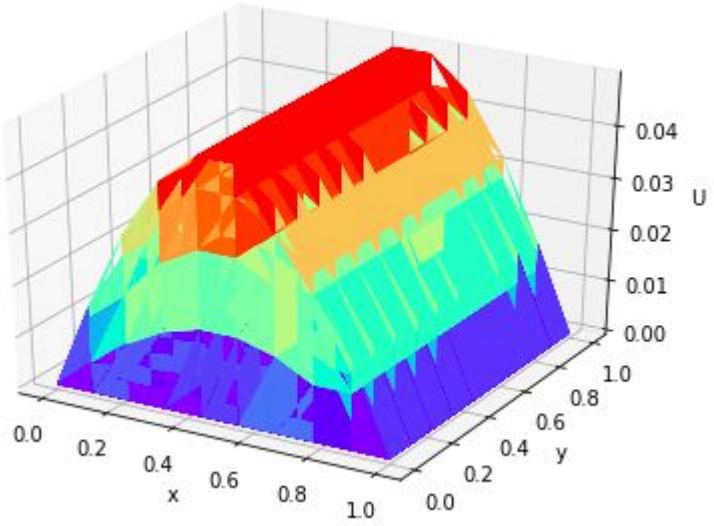
Entonces se crea la siguiente malla



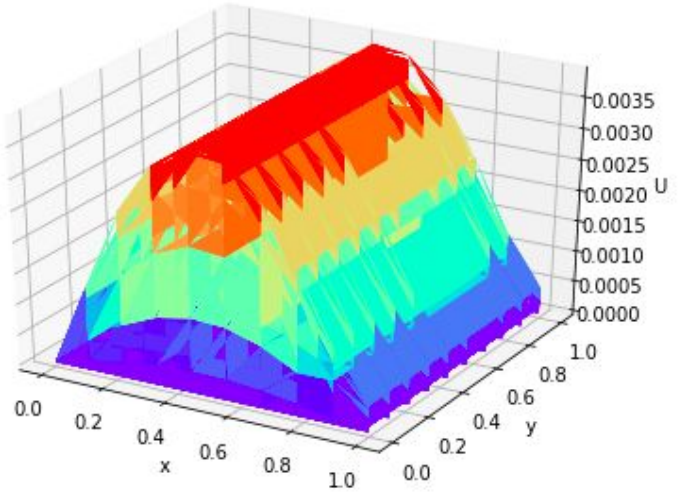
Y, obtenemos las siguientes gráficas

Solución exacta del ejemplo (evaluada en los nodos)

$$u(x, y) = \frac{\sin(x\pi)\sin(\pi y)}{-2\pi^2}.$$



Solución obtenida (aproximada)



5. Discusión de los errores

- El error del método es debido a varios factores, entre ellos los integradores numéricos usados en repetidas ocasiones y la solución del sistema de ecuaciones.
- El método en sí mismo es imperfecto, Normalmente, el error para problemas elípticos de segundo orden con funciones de coeficientes suaves es O(h^2), donde h es el diámetro máximo de los elementos triangulares o la diagonal máxima de los elementos rectangulares.
- Se pueden usar otras funciones base para dar resultados O(h^4), pero la construcción es más compleja.
- Los teoremas de error eficientes para métodos de elementos finitos son difíciles de enunciar y aplicar porque la precisión de la aproximación depende de la regularidad de la frontera, así como de las propiedades de continuidad de la solución.

5. Conclusión

- A pesar que se logra un código bien estructurado, el algoritmo desarrollado logra una forma similar a las soluciones esperadas pero no es exactamente igual al valor numérico de las mismas.
- El código es de una complejidad bastante alta, y pese a esto se logra una buena aproximación con el método.
- Se crea una tanto un repositorio como unas clases completamente organizadas y funcionales.

Llegamos al final