

SEP

SES

TNM

## INSTITUTO TECNOLÓGICO DE CHIHUAHUA II



### **“Desarrollo de librería para mallado 2D mediante redes neuronales”**

TESIS  
PARA OBTENER EL GRADO DE

**MAESTRO EN SISTEMAS COMPUTACIONALES**

PRESENTA  
ING. Claudio Iván Martínez Morfin.

**DIRECTOR DE TESIS**  
DR. HERNÁN DE LA GARZA GUTIÉRREZ

**CO-DIRECTOR DE TESIS**  
DR. ALBERTO DÍAZ DÍAZ

**CHIHUAHUA, CHIH. FEBRERO DEL 2017**

# Dictamen

Chihuahua, Chih., 09 de Febrero del 2017

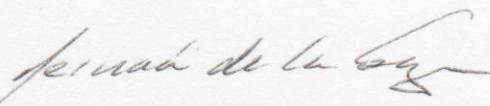
M.C. BLANCA MARICELA IBARRA MURRIETA  
JEFA DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN  
Presente.-

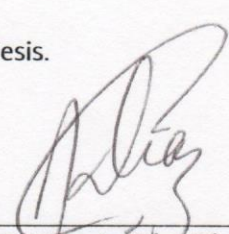
Por medio de este conducto el comité tutorial revisor de la tesis para obtención de grado de Maestro en Sistemas Computacionales, que lleva por nombre **"DESARROLLO DE LIBRERÍA PARA MALLADO 2D MEDIANTE REDES NEURONALES"**, que presenta el (la) **C. CLAUDIO IVÁN MARTÍNEZ MORFIN**, hace de su conocimiento que después de ser revisado ha dictaminado la **APROBACIÓN** del mismo.

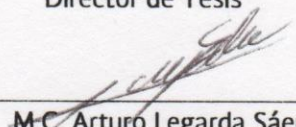
Sin otro particular de momento, queda de Usted.

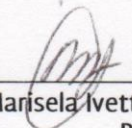
Atentamente


La Comisión de Revisión de Tesis.

  
\_\_\_\_\_  
Dr. Hernán de la Garza Gutiérrez  
Director de Tesis

  
\_\_\_\_\_  
Dr. Alberto Díaz Díaz  
Co-Director

  
\_\_\_\_\_  
M.C. Arturo Legarda Sáenz  
Revisor

  
\_\_\_\_\_  
M.C. Marisela Ivette Caldera Franco  
Revisor

  
SECRETARÍA DE  
EDUCACIÓN PÚBLICA  
INSTITUTO TECNOLÓGICO  
DE CHIHUAHUA II  
DIVISIÓN DE ESTUDIOS DE  
POSGRADO E INVESTIGACIÓN

## CONTENIDO

|  |    |
|--|----|
| ÍNDICE DE TABLAS.....                                | 9  |
| I. INTRODUCCIÓN .....                                | 10 |
| 1.1    Introducción.....                             | 10 |
| 1.2    Planteamiento del problema.....               | 14 |
| 1.3    Alcances y limitaciones. ....                 | 16 |
| 1.4    Justificación. ....                           | 17 |
| 1.5    Objetivos.....                                | 18 |
| 1.5.1    Objetivo General.....                       | 18 |
| 1.5.2    Objetivos específicos .....                 | 18 |
| II. ESTADO DEL ARTE .....                            | 20 |
| III. MARCO TEÓRICO .....                             | 23 |
| 3.1    Método del elemento finito.....               | 23 |
| 3.2    Consideración de un mallado de calidad .....  | 26 |
| 3.3    Triangulación Delaunay.....                   | 29 |
| 3.4    Inteligencia artificial .....                 | 37 |
| 3.5    Redes neuronales .....                        | 39 |
| IV. DESARROLLO .....                                 | 44 |
| 4.1    Modelo de negocio.....                        | 44 |
| 4.2    Delimitación de requerimientos generales..... | 45 |
| 4.3    Análisis y diseño general .....               | 50 |
| 4.4    Interfaz gráfica.....                         | 52 |
| 4.4.1    Determinación de requerimientos.....        | 52 |
| 4.4.2    Análisis y diseño.....                      | 53 |

|       |  |    |
|-------|--|----|
| 4.4.3 | Codificación. ....   | 54 |
| 4.5   | Módulo de mallado. ....  | 57 |
| 4.5.1 | Determinación de requerimientos.....                                   | 57 |
| 4.5.2 | Análisis y diseño.....   | 58 |
| 4.5.3 | Codificación. ....   | 58 |
| 4.6   | Módulo de red neuronal. ....   | 64 |
| 4.6.1 | Determinación de requerimientos.....                                   | 64 |
| 4.6.2 | Análisis y diseño.....   | 65 |
| 4.6.3 | Codificación. ....   | 67 |
| 4.7   | Implementación. ....   | 74 |
| V.    | RESULTADOS Y DISCUSIÓN .....   | 75 |
| 5.1   | Pruebas de rendimiento.....  | 75 |
| 5.2   | Prueba de experimentación .....  | 79 |
| VI.   | CONCLUSIONES .....   | 88 |
| VII.  | BIBLIOGRAFÍA .....   | 91 |
| VIII. | ANEXOS .....   | 93 |
| 8.1   | Exportar datos de dibujo. ....   | 93 |
| 8.2   | Crear elementos Point2D para área de dibujo. ....                      | 94 |
| 8.3   | Realizar trazos de segmentos en área de dibujo .....                   | 94 |
| 8.4   | Generar puntos en área de dibujo. ....                                 | 95 |
| 8.5   | Crear elementos Segment para área de dibujo.....                       | 96 |
| 8.6   | Importar datos del mallado. ....                                       | 97 |
| 8.7   | Inicialización de entrada de datos para librería de triangulación..... | 97 |
| 8.8   | Rutina onClockWise .....   | 98 |

|      |  |     |
|------|--|-----|
| 8.9  | Rutina pointInside.....  | 99  |
| 8.10 | Rutina linearMesh.....   | 100 |
| 8.11 | Conversión de datos de polígono a grafo para librería de triangulación. .... | 101 |
| 8.12 | Exportar malla.....  | 102 |
| 8.13 | Determinar continuidad en condiciones de frontera .....                      | 102 |
| 8.14 | Determinar ángulo interno .....  | 103 |
| 8.15 | Determinar proximidad con agujeros .....                                     | 104 |

## ÍNDICE DE FIGURAS

|   |    |
|---|----|
| Figura 1.1 Ejemplo de dominio 2D. ....  | 9  |
| Figura 1.2 Ejemplo de un dominio mallado. ....                                | 9  |
| Figura 1.3 Ejemplo de una deformación. ....                                   | 10 |
| Figura 1.4 Refinado de una malla. ....  | 11 |
| Figura 1.5 Diseño básico de una red neuronal. ....                            | 12 |
| Figura 3.1 Ejemplo de fronteras en dominio 2D. ....                           | 22 |
| Figura 3.2 Muestra de elementos inválido. ....                                | 22 |
| Figura 3.3 Nodos de interpolación presentes en los elementos. ....            | 23 |
| Figura 3.4 Elementos en contornos curvos. ....                                | 26 |
| Figura 3.5 Tipos de elementos clásicos. ....                                  | 27 |
| Figura 3.6 Mallados triangulares. ....  | 28 |
| Figura 3.7 Casos de triangulación Delaunay. ....                              | 29 |
| Figura 3.8 Grafo plano y Grafo no plano. ....                                 | 31 |
| Figura 3.9 Caso de triangulación compacta. ....                               | 32 |
| Figura 3.10 Arista inválida. ....   | 33 |
| Figura 3.11 Triángulo con propiedades poco favorables. ....                   | 34 |
| Figura 3.12 Triangulación de mala calidad. ....                               | 34 |
| Figura 3.13 Diagrama de una red neuronal clásica. ....                        | 38 |
| Figura 3.14 Diagrama general de un mapa de auto organizado. ....              | 41 |
| Figura 4.1 Esquema de plan de trabajo. ....                                   | 43 |
| Figura 4.2 Diagrama de casos de uso de funciones del sistema. ....            | 46 |
| Figura 4.3 Diagrama de actividades del sistema. ....                          | 48 |
| Figura 4.4 Diagrama de clases del sistema. ....                               | 49 |
| Figura 4.5 Diagrama de cascada con retroceso. ....                            | 50 |
| Figura 4.6 Interfaz gráfica. ....   | 52 |
| Figura 4.7 Demostración del sistema de coordenadas en el área de dibujo. .... | 53 |
| Figura 4.8 Muestreo de puntos para determinar sentido de los segmentos. ....  | 57 |
| Figura 4.9 Ejemplificación del método Ray Casting. ....                       | 58 |
| Figura 4.10 Representación de segmento como vector. ....                      | 59 |

|  |    |
|--|----|
| Figura 4.11 Diagrama de clases de red neuronal.....                        | 64 |
| Figura 4.12 Esquema de red neuronal para el proceso de entrenamiento. .... | 67 |
| Figura 4.13 Diagrama general de una red neuronal.....                      | 68 |
| Figura 4.14 Recreación de la red neuronal en Neuroph. ....                 | 69 |
| Figura 4.15 Ventana de entrenamiento de la red neuronal. ....              | 69 |
| Figura 4.16 Red neuronal resultante.....                                   | 70 |
| Figura 5.1 Geometría para prueba de tiempo. ....                           | 73 |
| Figura 5.2 Distintos tipos de mallado aplicados al mismo domino.....       | 74 |
| Figura 5.3 Juego de geometrias con condiciones. ....                       | 76 |
| Figura 5. 4 Malla homogenea refinada. ....                                 | 77 |
| Figura 5.5 Modelo de probeta original. ....                                | 78 |
| Figura 5.6 Modelo de probeta para experimeto.....                          | 79 |
| Figura 5.7 Tipos e mallado para la prueba ....                             | 82 |
| Figura 5.8 Gráfica de resultados en prueba de mecánica de fractura. ....   | 83 |
| Figura 5.9 Velocidad de convergencia en resultados. ....                   | 85 |



---

## ÍNDICE DE TABLAS

|   |    |
|---|----|
| Tabla 4.1 Diferencias de importancia entre combinaciones de criterios de interés. ....          | 64 |
| Tabla 4.2 Pesos sinápticos resultantes .....  | 71 |
| Tabla 5.1 Especificaciones del equipo empleado para las pruebas .....                           | 73 |
| Tabla 5.2 Comparativa entre número de elementos, tiempo de procesamiento y uso de memoria. .... | 75 |
| Tabla 5.3 Especificaciones de Workstation.....  | 81 |
| Tabla 5.4 Porcentajes de error obtenidos.....   | 82 |
| Tabla 5.5 Lectura de la energía de fractura para segundo experimento. ....                      | 84 |



## CAPÍTULO I. INTRODUCCIÓN

### 1.1 Introducción.

El avance de las ciencias de computación ha facilitado el trabajo en un sinnúmero de campos, entre ellos el de la simulación. Esto se debe principalmente al progreso en la tecnología de hardware y el desarrollo de software especializado, que permiten procesar enormes cantidades de datos y realizar un gran número de operaciones en poco tiempo.

Uno de los principales métodos numéricos empleados con el fin de simular y obtener una aproximación aceptable es el *Método del Elemento Finito* (mejor conocido como MEF en español y como FEM por sus siglas en inglés *Finite Element Method*). El MEF puede ser aplicado para obtener la solución a una gran variedad de problemas dentro de la ingeniería, por ejemplo: mecánica estructural, transferencia de calor, mecánica de fluidos, electromagnetismo, etc., por mencionar algunos (Moaveni, 1990), todo ello a través de la resolución de sistemas de ecuaciones diferenciales con derivadas parciales (PDE, siglas en inglés para *Partial Differential Equation*); los cuales son una manera común de modelar fenómenos físicos.

Existen diversos textos donde se trata como tema principal el MEF, como por ejemplo, una publicación literaria titulada “*Finite Element Method*” (Gouri Dhatt, 2012), en el cual se describen las etapas básicas que se deben de realizar para el proceso de análisis del MEF, las cuales son:

- *Definir un dominio: Generalmente el dominio es una representación gráfica del medio en el que se desarrolla el modelo que se quiere analizar. En la Figura 1.1 se ejemplifica con una placa en dos dimensiones.*



Figura 1.1 Ejemplo de dominio 2D.

- *Definir un mallado: El proceso de mallado 2D consiste en discretizar el dominio bidimensional en elementos de geometría simple, por lo general suelen ser triángulos o cuadriláteros, en los cuales se interpola la solución de las ecuaciones mediante polinomios de las variables de espacio  $x$  y  $y$ .*

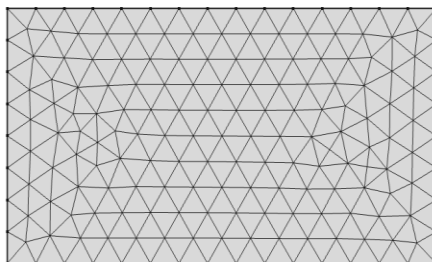
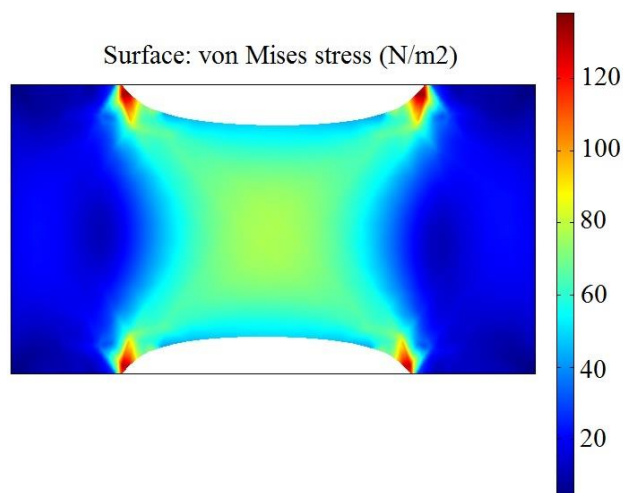


Figura 1.2 Ejemplo de un dominio mallado.

- *Calcular: Mediante el método numérico, el sistema de ecuaciones diferenciales es transformado en un sistema de ecuaciones algebraicas donde las incógnitas son los valores de las funciones incógnitas en los nodos (los vértices de los elementos).*
- *Procesar el resultado: Se interpola la solución en cada elemento y al sumar todos los resultados de cada elemento obtendremos el resultado de todo el dominio. También, se pueden tratar los resultados para obtener otros valores numéricos como integrales de la solución en un dominio o en un contorno.*



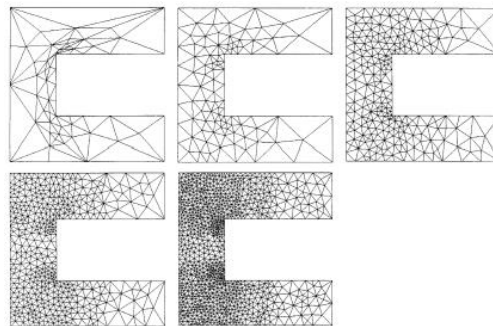
*Figura 1.3 Ejemplo de una deformación (resultado de aplicar fuerza en el borde de un dominio), el color permite interpretar el nivel de esfuerzos.*

Es innegable que todas las etapas presentan un nivel de complejidad muy elevado que hacen necesario el uso de software de análisis y simulación. Estos software emplean los conocimientos generados por diversos estudios efectuados en varios campos de la ciencia como las matemáticas, la física, la química, entre otros, como base para asegurar que la información que utilizan es correcta, afirmando con ello que el resultado obtenido es válido.

Del mismo modo es preciso que la herramienta que genera el mallado tenga una base en metodologías y teoremas comprobados, debido a que la malla tiene una particular repercusión en la precisión del método; es decir, a peor calidad en la malla peor será su aproximación; caso contrario, una malla de calidad permite una aproximación más cercana a la realidad (Larry Manevitz, 1997). Como base para ello, se emplea el análisis según la triangulación *Delaunay* con algunas variantes como lo son el algoritmo de *Ruppert* y *Chew*, entre otros (Alfonzetti, 1998 ) (Shewchuk, 1997).

Se dice que, un mallado óptimo es aquél que coloca una mayor cantidad de elementos en las zonas donde son más importantes para el estudio y menos población de elementos donde éstos no aportan precisión o mayor información a la aproximación (A. Bahreininejad, 1997).

A continuación en la Figura 1.4 se observa el proceso de refinar (incrementar la cantidad de elementos presentes en una malla) la malla dentro del dominio y cómo los elementos se van organizando y concentrando en las áreas donde son de mayor relevancia (Labridis, 2002).



*Figura 1.4 Refinado de una malla, en cada paso se incrementa la cantidad de elementos.*

Al realizar una técnica de mallado adecuada buscando la optimización del mismo (el mallado), da como resultado un mejor aprovechamiento de los recursos de la computadora junto con una excelente aproximación en el resultado.

Para determinar un buen control sobre la posición de los puntos que se emplean para crear la triangulación requerida en una malla, se han utilizado las Redes Neuronales (en lo sucesivo RN, uno de los campos de estudio de la inteligencia artificial). Una red neuronal en la naturaleza, adquiere un impulso eléctrico proveniente de la corteza cerebral (equivalente a una entrada de datos), posteriormente los núcleos de cada neurona atenúan o aumentan el impulso, reaccionando según su intensidad, después de esto, cada neurona emplea una ramificación de sí misma llamada Axom, de la cual saldrá el pulso eléctrico a otra neurona pasando por una resistencia llamada brecha sináptica. Este proceso es utilizado por el cerebro para procesar la información, reconocer patrones y definir procesos de reconstrucción.

Computacionalmente hablando, las redes neuronales tienen la capacidad de clasificar nuevos problemas a partir de soluciones ya encontradas con anterioridad, de tal modo que es posible catalogar una serie de posibles soluciones eficientes a un problema (A. Bahreininejad, 1997).

El funcionamiento de una RN consiste en una serie de entradas de datos, los cuales se procesan en capas hasta llegar a un resultado final. Cada capa (entrada, proceso y salida) está constituida por nodos llamados neuronas, las cuales se conectan unas con otras pasando la información entre las capas. En este proceso la salida de cada neurona se convierte en la entrada de otra después de haber sido multiplicado por un “peso sináptico”.

El modelo clásico de la red neuronal se expone en la Figura 1.5 donde cada círculo es una neurona, cada línea que los conecta tiene un valor de peso neuronal equivalente a la sinapsis.

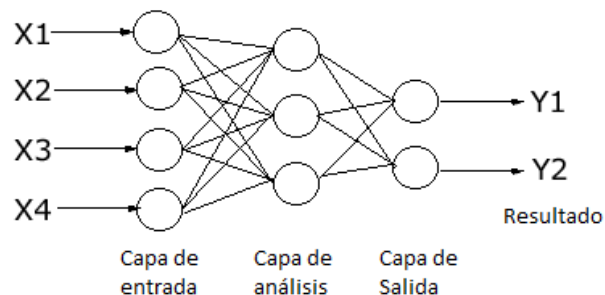


Figura 1.5 Diseño básico de una red neuronal.

Por lo tanto, si se considera como entradas de datos para una red neuronal los parámetros de una geometría 2D, es posible determinar la ubicación de los puntos que conforman la triangulación y de esta manera optimizar el mallado.

La presente tesis muestra el desarrollo de una librería de mallado para geometrías 2D utilizando técnicas de inteligencia artificial, en concreto redes neuronales, para la optimización de la misma. Este estudio está enfocado a una malla con fines de simulación mediante el MEF, sin embargo, es posible que este mismo principio sea viable para generar una malla con un fin distinto (un mallado aplicado a telecomunicaciones, coloreado y modelado 3D, etc.).

## 1.2 Planteamiento del problema.

Todo software de simulación está constituido por etapas, las cuales están relacionadas directamente entre sí, esto quiere decir que la falla en la etapa del mallado puede llevarnos a la no simulación (resolución) del modelo o a la obtención de resultados poco confiables.

Centrados en los problemas presentes en la no simulación, podemos decir que casi todos pueden derivarse de los siguientes dos escenarios:

- Trazos geométricos incongruentes: En donde no se puede aplicar mallado alguno al dominio originado en la etapa de diseño CAD(diseño asistido por ordenador en inglés computer-aided design) en 2D, debido a que algunos puntos o líneas no han sido conectados entre sí provocando polígonos abiertos.
- Limitaciones de memoria y errores de programación: Otro posible escenario conflictivo ocurre cuando los algoritmos que generan el mallado simplemente no son capaces de procesar de manera correcta los datos de entrada, produciéndose errores como por ejemplo: zonas sin cubrir de la geometría original o generación de elementos triangulares inválidos.

Es necesario mencionar que la propuesta de solución presente en este documento no pretende remediar todos los problemas presentes en la no simulación del MEF. Se pretende solucionar únicamente los problemas del mallado que fueron presentados anteriormente; para lograrlo se plantea la implementación de un método riguroso y confiable de teselado que genere polígonos sin defectos para el mallado.

Por otro lado, centrados en la problemática de obtener resultados poco confiables nos enfrentamos con la premisa de que la mayoría del software que se encarga de realizar un mallado de manera automática por lo general produce una malla uniforme, lo cual si bien no es un problema, es posible que se genere un gasto de memoria y procesamiento de datos en zonas donde no hay una aportación tan relevante al resultado. En estos casos si el usuario

tiene conocimientos del problema que plantea, así como de los parámetros para generar la malla, éste puede ser capaz de configurar el software para generar una cantidad mayor de elementos en las áreas donde tiene un mayor interés y de este modo aprovechar mejor la capacidad del hardware.

Debido a la complejidad que tiene crear automáticamente una malla de calidad, se plantea desarrollar una librería de software que con base en las disposiciones geométricas dadas por el usuario, así como otros datos especificados por el mismo (datos del MEF), procese un mallado intentando que la concentración de elementos esté distribuida en las áreas que puedan ser de mayor utilidad. De este modo, el mallado estará optimizado para que el resultado del proceso de simulación sea lo más fiable posible, simplificando el manejo de software y reduciendo el tiempo de trabajo.

### **1.3 Alcances y limitaciones.**

#### **Alcances**

- La librería crea una malla óptima de cualquier geometría 2D cerrada.
- La librería es capaz de mallar analizando parámetros de la geometría y condicionamiento del problema.
- El tiempo total del procesamiento es aceptable en comparación de un mallado no optimizado.
- La librería cuenta con un diseño de clases que permite su fácil implementación.
- La librería genera información referente a los elementos que conforman el mallado, como el número y posición de nodos, la cantidad de elementos generados y los puntos que los componen.

#### **Limitaciones**

- El sistema está limitado a figuras 2D que estén cerradas.
- El proceso de mallado está sujeto a las características que demande el usuario y a los recursos de hardware que tenga el equipo que ejecute la librería.



- La librería no garantiza que los resultados sean los mejores para cada problema particular, o que supere un mallado efectuado por un experto, sin embargo, el resultado obtenido tiene mejor eficiencia en comparación con una malla uniforme.
- La librería verifica la existencia o no de un criterio de interés en un punto específico de la geometría, sin embargo, no se considera el valor(o magnitud) que toma el criterio dentro del estudio.

#### **1.4 Justificación.**

En la actualidad existen herramientas de software para el mallado de figuras 2D pero en ocasiones presentan inconvenientes. Cuando suceden situaciones como figuras abiertas, la posibilidad de generar una malla es nula, sin embargo, para cualquier otro caso existe la posibilidad de implementar técnicas para mejorar la malla resultante. Una de ellas es cambiar los parámetros de mallado manualmente, a fin de buscar un conjunto de datos que produzcan un resultado adecuado, por otro lado esto implica un conocimiento avanzado en el tema.

Una propuesta para optimizar el proceso de mallado es analizar el dominio y las propiedades del fenómeno a simular de manera automática, es decir, contar con una herramienta que evalúe la figura para optimizar este paso del MEF y garantizar un mallado de calidad independientemente del fenómeno físico, para eso se propone emplear redes neuronales.

Las redes neuronales presentan la capacidad de recrear una actividad específica para la cual se les entrena; si se emplean buenas técnicas de análisis para reconocer zonas de interés en la geometría del dominio, es posible enseñar a la red qué patrones tiene que identificar para determinar una densidad de elementos aceptable para cada área del dominio, todo basado en las características que se le presenten.

Con base en las características de análisis presentadas en las redes neuronales es posible crear una herramienta que proporcione una malla adaptada a la geometría (2D) y al condicionamiento del problema que se le plantee.

## 1.5 Objetivos

### 1.5.1 Objetivo General

Desarrollar una herramienta de software libre en el lenguaje de programación C++ capaz de realizar un mallado optimizado sobre figuras 2D empleando las características de la geometría y el condicionamiento del problema. La herramienta logra su objetivo mediante el uso de una red neuronal para determinar las zonas en donde se debe presentar una mayor concentración de elementos, para ello, se evalúan los puntos que conforman la geometría, sobre los cuales, la red neuronal analiza la siguiente lista de consideraciones en cada uno de ellos (los puntos de la geometría):

- El ángulo interno generado por los segmentos de la geometría que se unen en el punto evaluado debe ser menor o igual a  $90^\circ$ .
- La longitud del segmento más corto conectado al punto que se está evaluando por la red.
- La continuidad sobre las condiciones de frontera entre los segmentos que intersectan en el punto evaluado.
- La proximidad del punto analizado con orificios dentro de la geometría según un criterio definido por el usuario.

### 1.5.2 Objetivos específicos

- *Desarrollar la herramienta de dibujo:* Desarrollar una interfaz de dibujo para figuras 2D en la cual se realicen las pruebas de la librería. Funciona como entrada y salida de datos.
- *Implementar la triangulación Delaunay:* Implementar una librería externa que comparta el mismo tipo de licencia y filosofía de software que el buscado para este trabajo. Dicha librería será la base sobre la cual trabajen los algoritmos de inteligencia artificial.

- *Crear un módulo de análisis a figuras 2D por medio de redes neuronales:* Optimizar el proceso de mallado desarrollando una red neuronal mediante la cual se procesen los datos de la geometría y el condicionamiento del problema.
- *Definir entrenamiento de la red neuronal:* Aplicar un esquema de entrenamiento para la red neuronal.
- *Validar la eficiencia de la malla:* Evaluar los resultados obtenidos en comparación con otro tipo de malla.

## CAPÍTULO II. ESTADO DEL ARTE

Con el fin de buscar una manera de optimizar el proceso de mallado se ha explorado por diversos estudios la posibilidad de implementar inteligencia artificial para realizar en cierta medida esta tarea. Al tener una gran variedad de disciplinas, la inteligencia artificial aporta diferentes mecánicas para solucionar este problema.

En el año de 1988 Alfonzetti y Coco aportaron diversos estudios en su publicación *“Elfin: an n-dimensional finite-element code for the computation of electromagnetic fields”* (S.Alfonzetti S. , 1988) en donde exponen un software de simulación para fenómenos de electromagnetismo, planteando el uso de redes neuronales para la elaboración del mallado, tomando en cuenta como entradas de datos los parámetros del fenómeno físico para determinar las zonas con mayor densidad de malla. Posteriormente estos mismos autores publican *“Automatic Mesh Generation by the Let-It-Grow Neural Network”* (S.Alfonzetti S. S., 1996) en colaboración con Cavalieri y Malgeri replantean el estilo de red neuronal que emplearon originalmente y proponen la implementación de uno nuevo.

Otro enfoque es presentado en 1991 por Chang Ahn en su artículo *“A self-organizing neural network approach for automatic mesh generation”* (Ahn, 1991) en el cual muestra una aplicación a los mapas de auto organizado de Kohonen para recrear una malla con base en la mecánica empleada por esta red.

Los estudios realizados en el año 1992 por Dyck, Lowther, y McFee en su publicación *“Determining an approximate finite element mesh density using neural network techniques”* (D. N. Dyck, 1992) plantean un sistema para determinar la densidad de elementos en una malla para problemas de electromagnetismo.

Algunas otras aplicaciones de inteligencia artificial al proceso de mallado no solo utilizan las redes neuronales, sino que también se buscan la implementación de sistemas expertos, ejemplo de ello es el caso de Kang y Haghighi que generaron un sistema que tenía la

posibilidad de ser retroalimentado, el cual fue presentado en su publicación *“Intelligent Finite Element Mesh Generation”*. (E. Kang, 1995)

En 1996 Chedid y Najjar retoman la idea de implementar redes neuronales y publicaron su estudio titulado *“Automatic Finite-Element Mesh Generation Using Artificial Neural Networks”* (R. Chedid, 1996) en donde en un primer planteamiento proponen un modelo capaz de mallar cualquier figura 2D considerando únicamente la geometría del dominio.

Por otro lado Bahreininejad plantea una manera en la cual puede procesar la malla con ayuda de redes neuronales las cuales evalúan para su entrenamiento el mallado que fue generado previamente por un sistema experto. (A. Bahreininejad, 1997). En ese mismo año Manevitz, Yousef y Givoli presentan una comparativa de eficiencia entre un mallado clásico y uno generado por mapas de auto organizado, los resultados de este estudio fueron publicados en el artículo *“Finite-Element Mesh Generation Using Self-Organizing Neural Networks”* (Larry Manevitz, 1997).

Para el año 2002, Triantafyllidis y Labridis publicaron una metodología en la cual se plantea un primer mallado burdo y después lo refinan con base en la densidad de elementos generada por una red que se basa en el trabajo de Chedid y Najjar. En adición al trabajo anterior presentaron el algoritmo que sigue su software y la comparativa contra un mallado genérico. (Labridis, 2002). Bojan Dolsak posteriormente se retomó el planteamiento de emplear sistemas expertos con el fin de asistir al usuario durante todo el proceso de simulación incluido el mallado, en el cual se le solicita al usuario defina parámetros que serán empleados para elaborar la malla considerando el condicionamiento del problema que ha planteado antes de llegar a esta etapa. (Dolska, 2002).

En el año 2009 Srasuay, Chumthong y Ruangsinchaiwanich publicaron un estudio *“Mesh Generation of FEM by ANN on Iron”* (K. Srasuay, 2009) en el cual, basados en los resultados de Dyck y Lowther, replantean diferentes parámetros de evaluación que una red neuronal puede evaluar a modo de mejorar la estimación de la densidad para un problema de

electromagnetismo. Para esto especializaron más el caso de estudio ya que su sistema está pensado exclusivamente para transformadores con un núcleo de hierro.

En el transcurso de ese mismo año, sobre la base de los mapas de auto organizado y sus diferentes presentaciones Jilani, Bahreininejad y Ahmadi exploran cómo diferentes mapas de auto organizado pueden generar mallas con mejor desempeño para diferentes problemas, además de realizar la comparativa con modelos clásicos de mallado. Según los autores las estrategias que se han propuesto con mejores resultados son aquellas que integran mapas auto-organizados, ya que por medio de éstas se logra un mallado adaptativo. (H. Jilani, 2009).

En particular los trabajos que se han realizado con el fin de implementar inteligencia artificial para elaborar mallas de calidad están abocados en solventar casos muy específicos, como por ejemplo: electromagnetismo, mecánica de fluidos, etc. Por otro lado los estudios que se basan en analizar el dominio y sus dimensiones generalizan la importancia de sus criterios esperando que sean válidos para el caso de uso que se plantea.

## CAPÍTULO III. MARCO TEÓRICO

### 3.1 Método del elemento finito

El MEF fue planteado desde 1943 por Richard Courant, sin embargo, a lo largo del tiempo diferentes personas aportaron cambios dentro de la teoría hasta formar las bases de lo que se tiene hoy en día. El MEF está diseñado para solucionar sistemas de ecuaciones diferenciales con derivadas parciales respecto a las variables de espacio. (Gouri Dhatt, 2012)

Las ecuaciones con derivadas parciales son comunes para la representación matemática de fenómenos físicos ya que es común el plantear una relación de espacio y tiempo en la definición del problema.

En el caso particular de la solución de problemas en dos dimensiones, las ecuaciones del problema tienen parámetros de evaluación en el eje de las ordenadas, abscisas y en algunos casos el tiempo. (Gouri Dhatt, 2012)

El MEF opera evaluando el sistema de ecuaciones en diferentes puntos del dominio y acumulando el cambio que se genera entre un punto y otro. Este cambio está dado por los parámetros del problema que varían con frecuencia dependiendo del fenómeno físico. Sin embargo, existen características forzosas que se deben especificar, como las *condiciones iniciales* (que son las condiciones en las que se encuentra el dominio antes de iniciar el análisis en caso que sea dependiente del tiempo) y las *condiciones de frontera*.

Las condiciones de frontera representan valores que se le imponen a las incógnitas de las ecuaciones que el MEF resolverá o a sus derivadas espaciales. Gráficamente, en un dominio 2D las condiciones de frontera se aplican en los contornos del dominio y los puntos que forman la geometría. Esto se puede observar en la Figura 3.1



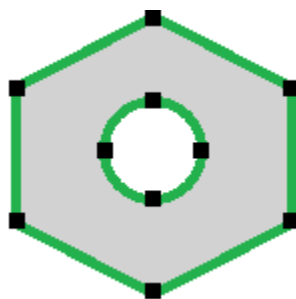


Figura 3.1 Ejemplo de fronteras en dominio 2D. Los segmentos de color verde representan las fronteras.

Dentro de la operación del MEF utilizar únicamente los puntos propios de la geometría para evaluar el dominio produce aproximaciones erróneas. Esto es debido a que el cambio producido de punto a punto es demasiado abrupto, en un dominio bidimensional es necesario discretizar el dominio en elementos de geometría simple, en los cuales se interpola la solución de las ecuaciones mediante polinomios de las variables de espacio  $x$  y  $y$ . Al proceso que genera estos elementos se le denomina mallado y se nombra como malla al conjunto de los mismos. Los elementos no deben tener espacios vacíos entre ellos (a no ser que claro, que exista un orificio en el dominio) y no se traslapan entre sí (ver Figura 3.2);

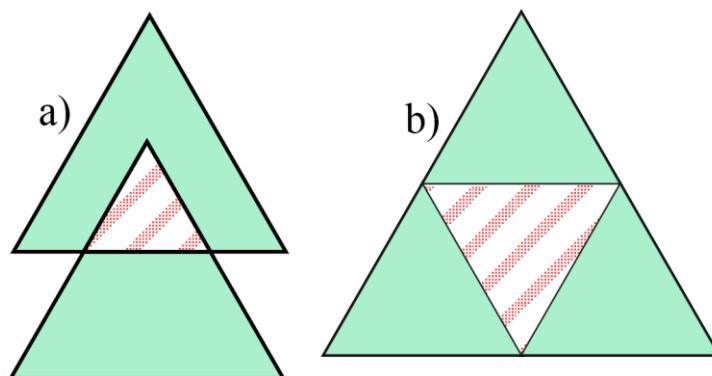
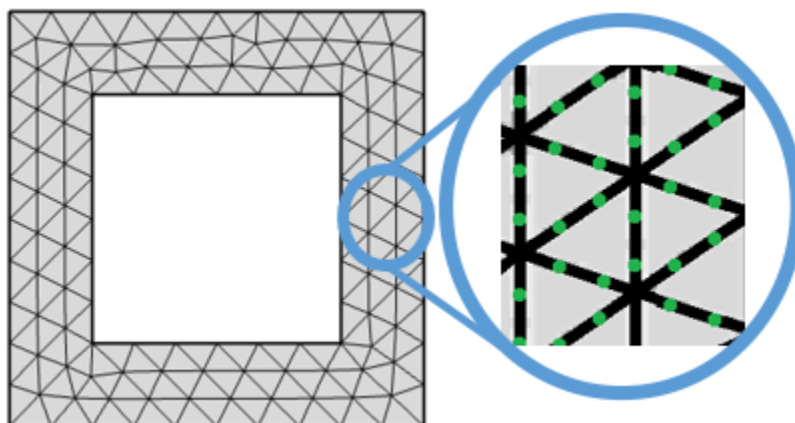


Figura 3.2 Muestra de elementos inválidos, las áreas inválidas se representan por franjas punteadas rojas, a) traslape entre elementos, b) espacio hueco entre dos elementos.

En cada elemento se resuelven las ecuaciones considerando sus propias fronteras y el resultado de cada elemento se va almacenando. Los elementos respetan las condiciones de frontera que se plantearon originalmente acarreando las propiedades con las cuales el problema fue delimitado originalmente.

Los elementos de un mallado durante el proceso del MEF son interpolados con un polinomio de grado  $N$ . En la siguiente imagen (ver Figura 3.3) se puede observar una representación de la interpolación sobre los elementos de una malla:



*Figura 3.3 Nodos de interpolación presentes en los elementos.*

Al contar una gran cantidad de elementos en una malla y un polinomio de interpolación relativamente elevado se produce una gran cantidad de puntos. En ellos el MEF evalúa las ecuaciones que rigen el problema o fenómeno que se plantea estudiar; una gran cantidad de puntos propicia una buena aproximación en el resultado obtenido por el MEF. Sin embargo, el tener una gran cantidad de puntos implica que el tiempo de procesamiento al igual que el uso de memoria de un equipo de cómputo sea elevado. Para disminuir los costos computacionales existe la posibilidad de incrementar la cantidad de elementos exclusivamente alrededor de zonas donde se sabe que al evaluar las variables del problema se obtendrá un cambio considerable, dejando con una cantidad de elementos menor aquellas áreas donde las variables no tiene un cambio relevante. Al tipo de malla que cumple este comportamiento se le denomina como mallado adaptativo (S.H. Lo, 2015); por otro lado las mallas estructuradas u homogéneas (definidas como *Grid* en inglés si la simetría es perfecta) son aquellas que presentan simetría dentro del dominio y son un ejemplo de los puntos mencionados anteriormente como se puede apreciar en la Figura 1.2 o en la Figura 3.3

### 3.2 Consideración de un mallado de calidad

Para realizar el mallado no existe un estándar o modelo a seguir; aunque comúnmente la metodología de mallado va en relación al uso que se planteó para el mismo y al nivel de exactitud que se requiera. (Gouri Dhatt, Finite Element Method, 2012)

Como ya se dijo con anterioridad, el proceso de mallado es importante dentro de un estudio por medio de MEF ya que la calidad obtenida en la malla repercute directamente sobre la aproximación obtenida por el método una vez completado. La calidad de la malla está sujeta a diferentes variables como lo son el tipo de problema a solucionar, las dimensiones del dominio y el tipo de elemento que se planea utilizar. Es por esto que no existe una mecánica que produzca siempre un mallado ideal para cualquier caso de manera general. Sin embargo, una manera para determinar la calidad que tiene un mallado (planteada para el MEF aplicado en dominios 2D) es evaluando las propiedades geométricas de los elementos que lo conforman y según el porcentaje de elementos presentes cerca de áreas de mayor relevancia (para el MEF). (Larry Manevitz, 1997)

La siguiente lista engloba algunas consideraciones para determinar si los elementos de una malla son de calidad:

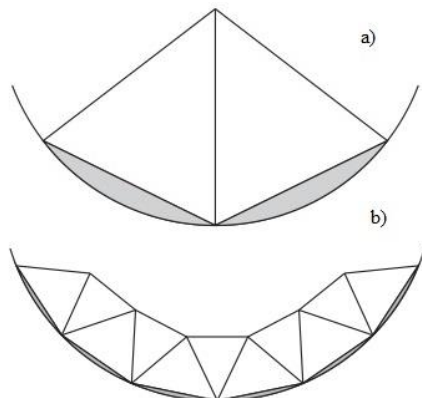
- Elementos proporcionales, esto implica que los elementos no deben de tener ángulos internos muy dispares entre ellos, es decir, que los elementos sean polígonos lo más equiláteros posible.
- Considerando el área de los elementos, debe existir una transición entre elementos pequeños hacia los más grandes de manera gradual, una recomendación sería que los elementos no llegaran a ser 1.5 o 2 veces más pequeños o grandes en comparación de sus vecinos. (Larry Manevitz, 1997)

- Los elementos generados por la malla deben de estar conectados mutuamente por vértices y por intersección en aristas de otros elementos, esto es, que mantenga una continuidad en sus conexiones. (Pascal Jean Frey, 2000)

Como se mencionó anteriormente la malla es usada en el MEF para agregar nodos en los cuales se evalúan las ecuaciones que se definieron para modelar el problema (Gouri Dhatt, Finite Element Method, 2012), sin embargo, según la naturaleza de dicho problema (o fenómeno físico) pueden existir zonas donde conocer el resultado y comportamiento de las variables que se emplean sea de gran interés.

Si bien en cada fenómeno físico se consideran diferentes variables y no hay manera alguna de generalizarlos, es posible sugerir zonas que presenten mayor relevancia al problema que otras según las siguientes consideraciones:

- La malla debería ser fina en regiones donde existen cambios abruptos en la geometría, debido a que eso propicia gradientes importantes de la función solución. Es por ello que zonas como esquinas, quiebres, agujeros y cambios pequeños en la frontera del dominio, son consideradas de interés. (Larry Manevitz, 1997)
- Cambios en las condiciones de frontera son también zonas de interés ya que en éstas se establece un alto gradiente de la función solución. (Gouri Dhatt, Finite Element Method, 2012)
- Adicionalmente un mallado más fino aplicado a zonas con curvas ayuda a evitar la pérdida de precisión en esas áreas, como se ve en la siguiente imagen.



*Figura 3.4 Elementos en contornos curvos. a) Muestra un mallado burdo en una circunferencia, b) Ejemplifica un mallado más fino sobre el mismo dominio. El área gris en ambos incisos representa el área que se perderá en la experimentación.*

Con base en las cualidades que deben presentar los elementos y las zonas de interés planteadas se puede suponer entonces que un mallado óptimo es aquél que tiene elementos de calidad con proporciones correctas y que además genera una mayor concentración de elementos en zonas donde hay más relevancia.

Ahora que se tiene una definición más formal de qué se busca en una malla óptima, es preciso buscar una metodología que genere elementos con propiedades como las mencionadas en párrafos anteriores. En cuanto al tipo de elemento no existe una norma a seguir, por lo cual se podría emplear cualquier tipo de elemento (triangular o cuadrilátero) como se ve en la siguiente imagen Figura 3.5.

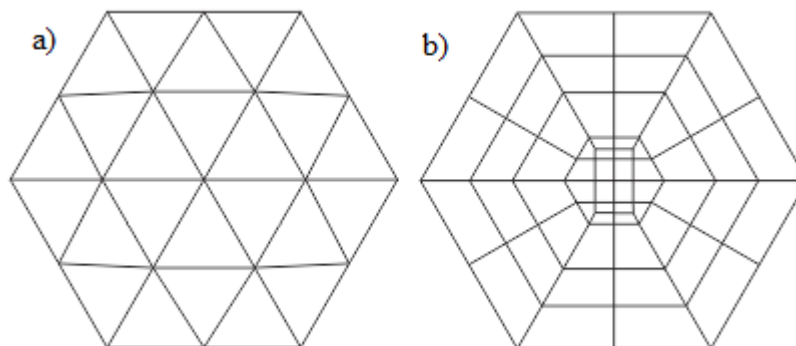


Figura 3.5 Tipos de elementos clásicos. a) Malla con elementos triangulares, b) Elementos cuadriláteros.

Sin embargo existe una predilección por utilizar figuras simples que permitan implementar un orden sencillo para manejar los vértices por elemento. Debido a esto último, el tipo de elemento seleccionado para esta herramienta de software es el triángulo, para generar elementos de este tipo existen diferentes técnicas a las cuales se les llama *Métodos de triangulación*. (Pascal Jean Frey, 2000)

### 3.3 Triangulación Delaunay

Existen múltiples técnicas de triangulación, pero para fines de simulación las principales son la triangulación *Delaunay* y el conocido como *avance frontal*. Estas dos técnicas permiten obtener elementos de buena calidad debido a las mecánicas que emplean para generarlos. En la Figura 3.6 se muestran ejemplos de mallado por ambos métodos.

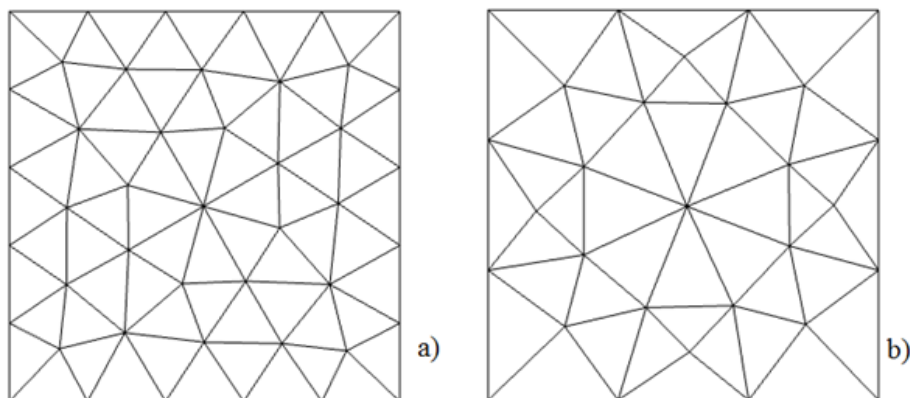


Figura 3.6 Mallados triangulares. a) Triangulación avance frontal, b) Triangulación Delaunay.

Aunque ambos métodos son eficientes, el estilo de Delaunay va más acorde a las necesidades que se pretenden para este trabajo. La triangulación fue propuesta en el año 1934 por Boris Delaunay, que toma sus bases de los diagramas de Voronói y la teselación de Dirichlet. La mecánica que se emplea para realizar la triangulación Delaunay no ha cambiado mucho a lo largo de los años, pero sí ha servido de base para muchos otros estilos de triangulación que siguen la misma idea.

La triangulación Delaunay se puede alcanzar mediante las siguientes dos maneras:

1. La triangulación Delaunay se logra si para cada uno de sus elementos (triángulos) se puede dibujar un círculo circunscrito que no contenga en su interior ningún otro punto que conforme los vértices del triángulo que generaron el círculo.
2. Si sobre un conjunto de puntos se puede dibujar un diagrama de Voronói, automáticamente se puede producir una triangulación Delaunay, esto por la manera en que se generan los polígonos para el diagrama de Voronoy.

A continuación se muestra un par de imágenes donde se presentan ambos casos:



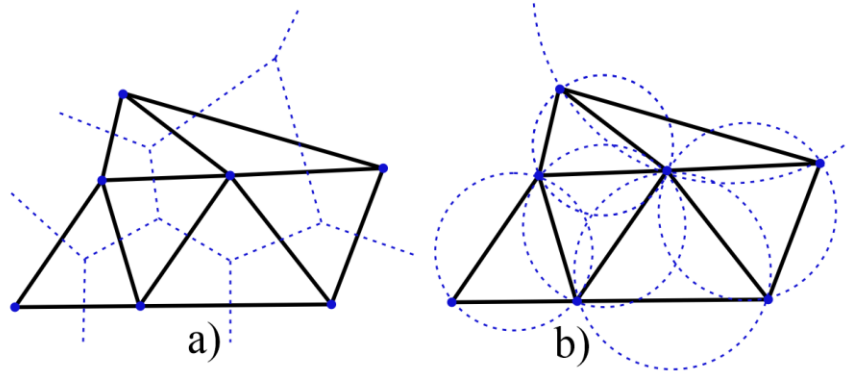


Figura 3.7 Casos de triangulación Delaunay. a) Diagrama de Voronói (línea punteada), y b) Círculos inscritos de cada elemento para la misma triangulación.

En la ciencia de la computación generalmente la segunda definición es empleada para evaluar la triangulación completa, mientras que la primera es aplicada comúnmente para el refinado de algún elemento.

Los diagramas de Voronói se pueden interpretar como un conjunto de polígonos (conocidos como polígonos de Voronói) cuyas propiedades son expuestas a continuación. Se considera un conjunto (o lista) de puntos  $\mathbb{P} = \{p_i, i = 1, n\}$  que forman parte de  $n$  distintos elementos en un plano de  $\mathbb{R}^2$ . Se cuenta con una serie de polígonos definidos en un conjunto  $V = \{V_i, i = 1, n\}$

$$V_i = \{x \in \mathbb{R}^2: ||x - p_i|| < ||x - p_j||, \forall j \neq i\}$$

donde  $V_i$  es una región de  $\mathbb{R}^2$  cuyos puntos se encuentran más próximos al punto  $p_i$  que a cualquier otro punto y  $||x - p||$  define la distancia entre dos puntos. Cada punto  $p$  está únicamente en una región. El punto  $x$  es un punto que está en el mismo plano  $\mathbb{R}^2$  pero que no forma parte del conjunto  $\mathbb{P}$ .  $V_i$  es un polígono convexo cuyos vértices son porciones perpendiculares de las bisectrices entre los puntos  $p_i$  y  $p_j$ . Estos nodos pertenecen a los polígonos  $V_i$  y  $V_j$  los cuales son (de seguir la definición anterior) polígonos de Voronoy. La colección de los polígonos anteriores, es decir,  $V$  es entonces un diagrama de Voronoy (o dicho de otro modo una teselación de Dirichlet del conjunto de puntos  $\mathbb{P}$ ). (S.H. Lo, 2015)

Una vez planteadas las dos definiciones de lo que es una triangulación Delaunay, surge la necesidad de generar un conjunto de puntos y relacionarlos para generar los elementos triangulares que constituirán la malla para el MEF.

El modelo de Delaunay no contempla la generación de la población de puntos iniciales (conocida como *nube de puntos*), ello propicia que es posible tener desviaciones en referencia a la forma y figura geométrica original utilizando el modelo como tal. Para solventar el problema y no modificar las fronteras del dominio, es conveniente implementar en un inicio una variante de Delaunay conocida como *triangulación comprimida o compacta*.

La triangulación compacta es un modelo en el cual es posible representar la forma original del dominio. A su vez se genera una malla compuesta por los puntos originales de la figura, que servirá como base para ser refinada hasta obtener una malla de calidad según los criterios planteados originalmente por Delaunay. De este modo al mantener los nodos originales, es posible mantener el contorno del dominio sin cambios y producir un primer mallado sin necesidad de una nube de puntos como tal, dichas mecánicas fueron presentadas por L. Paul Chew. (Chew, 1989)

El primer paso de la triangulación compacta es crear una representación del dominio en términos de grafos; específicamente grafos planos. Los grafos planos son aquellos que no tienen aristas que se traslapan, es decir, las aristas van de nodo a nodo sin cruzarse como se puede ver a continuación. (T. Nishizeki, 1991).

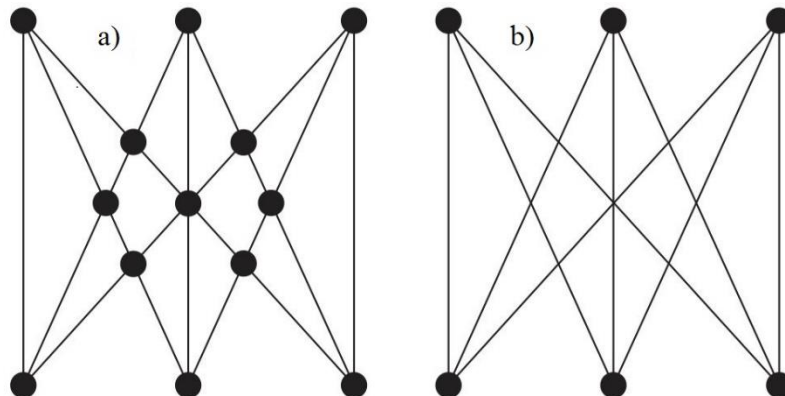
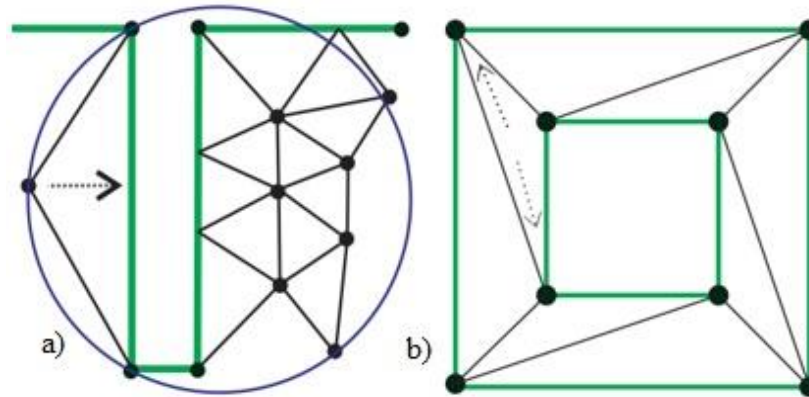


Figura 3.8 a) Grafo plano y b) Grafo no plano. Véase como el grafo plano tiene más nodos que permiten la separación de diferentes segmentos y evitando así el traslape entre ellos.

Posteriormente se definen los segmentos del grafo como *aristas restringidas*, las cuales cambian las reglas impuestas por Delaunay en sus definiciones originales y “relajan” las condiciones que determinan si un elemento es válido o no:

- La primera definición de Delaunay aplica como fue planteada originalmente y además se considera que un elemento es válido incluso si en el interior del círculo inscrito del elemento existen puntos adicionales, siempre y cuando el elemento tenga una arista restringida y el nodo o punto extra está del otro lado de la arista restringida, por lo cual no es “visible” por los nodos que conforman el elemento (ver Figura 3.9). (Chew, 1989)

Como se puede apreciar en Figura 3.9, elementos con dos ángulos muy agudos y un ángulo muy abierto son generados regularmente con este cambio a la definición de la triangulación. Al acomodar estos elementos en el dominio, da la apariencia que los elementos están apretados en el interior de la figura, de aquí que también se le llame a esta clase de triangulación como “*triangulación compacta*” o “*compresa*”.



*Figura 3.9 Caso de triangulación compacta. a) Las líneas verdes indican las aristas originales del grafo a mallar y se les denomina como Arista restringida, además no pueden ser alteradas y b) Triangulación compacta conformada por los nodos originales de un grafo.*

Posteriormente se aplicarían métodos de *refinado*, los cuales consisten en dividir elementos existentes y generar nuevos elementos con dimensiones más pequeñas que los originales. Existen diversos procedimientos de refinado propuestos para una primera triangulación de tipo compacto, los sugeridos y mayormente implementados al usar una triangulación compacta son los métodos de Ruppert.

El método de Ruppert tiene como base las siguientes dos estrategias:

1. El círculo generado por un segmento (el círculo diametral) no debe de tener ningún punto extra en su interior, si existe alguno se considera que el segmento es una arista *inválida*; en caso de tener una arista de este tipo es preciso agregar un nuevo nodo a la mitad del segmento y se vuelve a plantear la triangulación como se puede observar en la Figura 3.10. (Ruppert, 1995)

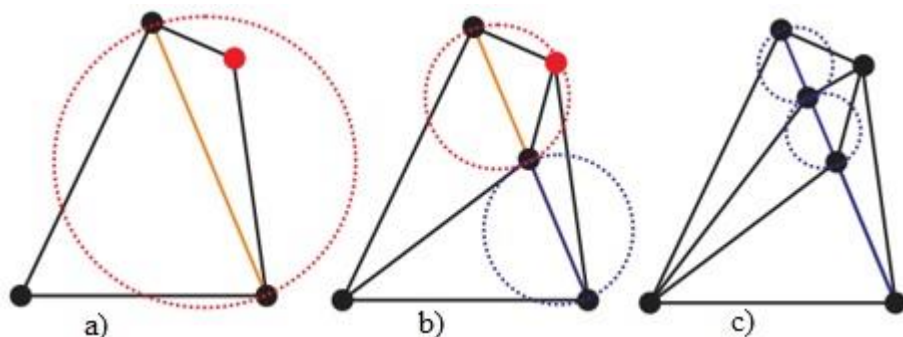


Figura 3.10 a) La arista naranja es inválida y pasa a ser dividida, b) Los nuevos segmentos se consideran con el mismo criterio y uno de ellos es inválido, c) Finaliza el proceso de corte en los segmentos inválidos, al insertar nuevos nodos se generan más elementos.

2. Si un elemento presenta malas características, se agrega al conjunto de puntos el centro del círculo circunscrito que pasa por los vértices del elemento con malas propiedades y se vuelve a evaluar la triangulación buscando mejores elementos. Si el nuevo nodo invade un segmento, el nuevo nodo es eliminado y los vértices invadidos son divididos a la mitad. (Ruppert, 1995)

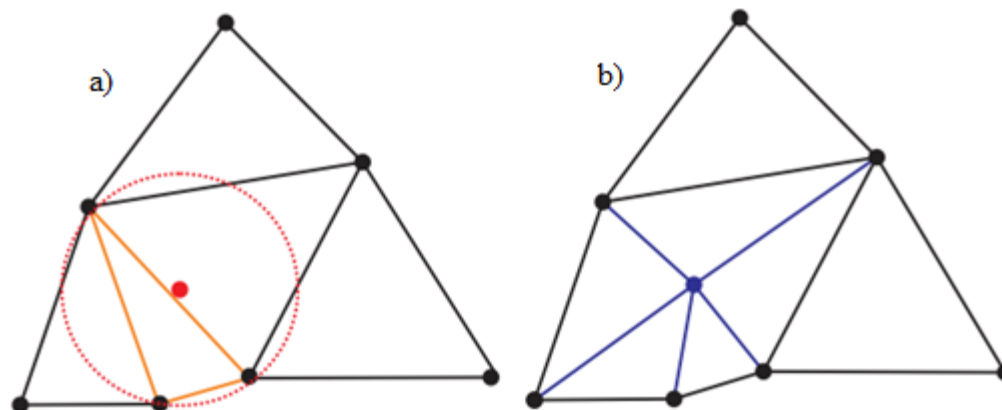


Figura 3.11. a) Triángulo con propiedades poco favorables, b) Inserción de nuevo nodo y generación de nuevos elementos.

A partir de la aplicación de estos dos criterios es posible volver a las definiciones originales de Delaunay. En la siguiente figura se presenta una triangulación de mala calidad, los cuales se buscan solucionar también mediante el refinado:

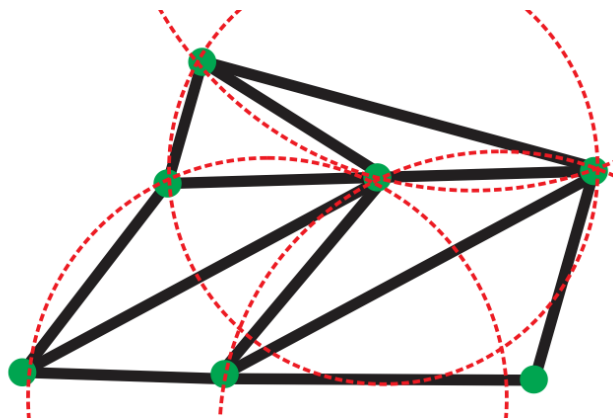


Figura 3.12 Los círculos rojos son generados por una mala triangulación, ya que contiene otros vértices en su interior.

Una vez que la triangulación ha alcanzado los parámetros que se solicitaron, el proceso finaliza. El mallado será empleado entonces para el proceso del MEF.

Existen librerías de software como el desarrollado por Jonathan Shewchuk donde se implementan la triangulación Delaunay y el refinado de Ruppert para generar una malla con elementos de calidad.

Hasta este punto sólo se cuentan con elementos de calidad; ahora es preciso encontrar una manera de evaluar los parámetros con los cuales se define el problema a modo de que sea posible determinar de manera automática las zonas que pueden ser de interés a refinar.

### 3.4 Inteligencia artificial

Para realizar el análisis de datos según ciertas situaciones y/o parámetros, la computación aporta prácticas mediante las cuales es posible programar sistemas que presenten cualidades de evaluación semejantes a la inteligencia, es decir, existen métodos para crear sistemas que evalúen de manera inteligente los parámetros de un problema. A estos criterios o técnicas de programación se les conoce como *Inteligencia artificial*. (Kriesel, 2005). A continuación se enlistan y describen algunas de las técnicas propias de la inteligencia artificial:

1. **Sistemas expertos:** Los sistemas expertos consisten en programar los criterios que evalúa una persona especializada al realizar una tarea, en este caso, la elaboración de la malla. Se le llama *heurística* al criterio o recomendación que el usuario impone según su experiencia (Stuart J. Russell, 1995). Por ejemplo un especialista en mecánica es capaz de establecer ciertos criterios o puntos de interés respecto a un problema de esta índole, sin embargo un experto en electromagnetismo pudiera despreciar estos mismos puntos por no considerarlos relevantes dentro de su campo de acción. Por lo tanto se puede decir que los sistemas expertos en general están más orientados en guiar al usuario acerca de cómo llevar a cabo una tarea en vez de realizarla automáticamente y debido a que no son generalizables, no son una solución exacta para la elaboración de una herramienta de mallado genérico.
2. **Algoritmos genéticos:** Los algoritmos genéticos plantean recrear la evolución que se da en el medio natural pero enfocado a un problema. Se proponen una serie de posibles soluciones y se codifican en cromosomas que almacenan la información de cada solución, los cromosomas son evaluados y se les asigna una pareja con la que se



“aparean” y generan un nuevo cromosoma que simboliza una nueva solución. El proceso se itera hasta obtener una población con soluciones mejor adaptadas al problema y por lo tanto más óptimas. Este proceso puede encontrar la solución más óptima, o al menos, una solución que se ajuste mejor al problema que las planteadas originalmente (Stuart J. Russell, 1995). El detalle negativo considerando su aplicación al proceso de mallado está en el tiempo de respuesta; el mallado es un proceso que al ser realizado por una computadora se espera que se ejecute en el menor tiempo posible, sin embargo, el analizar posibles soluciones de mallas para evaluar cuál es la más adecuada para el caso de estudio, es equivalente a realizar el mallado tantas veces como soluciones se evalúen, en lugar de generar un solo mallado óptimo en el primer intento.

3. Redes neuronales: Las redes neuronales son sistemas que emulan la estructura del cerebro. La estructura de una red neuronal se compone de una serie de nodos conocidos como *neuronas* las cuales se conectan entre sí. Cuando una neurona transfiere la información a otra se multiplica por un parámetro que aumenta o disminuye la intensidad de la señal, al que se le llama *peso sináptico*. Las redes neuronales son una opción ampliamente viable para el desarrollo de aplicaciones en donde se requiera detectar cierto comportamiento o patrón en un conjunto de datos (Kriesel, 2005). Para lograr esta tarea es preciso que la red neuronal reciba un entrenamiento, en el cual se le configura para que sea capaz de evaluar los datos que manejará. Considerando las características que el MEF contempla como zonas críticas en donde el mallado debería ser más fino, es posible entonces aplicar una red neuronal que busque y evalúe dichas zonas.

Después de hablar un poco de los campos de la inteligencia artificial, se torna un poco obvio que las redes neuronales son la solución más atractiva para la implementación de un software que determine la densidad de elementos alrededor de un punto.

### 3.5 Redes neuronales

Las redes neuronales son programas diseñados para realizar una tarea en específico para la cual son entrenadas; funcionan imitando a las neuronas del cerebro. En el cerebro de los seres vivos existen neuronas las cuales intercambian impulsos eléctricos mediante apéndices llamados *dendritas*. Las neuronas no se conectan directamente, existe un espacio entre las dendritas de las neuronas llamado *espacio sináptico* o *brecha sináptica*, en el cual existen diferentes compuestos químicos. Es en la brecha sináptica donde se lleva a cabo el proceso de la *sinapsis*, el cual consiste en que una neurona envía un impulso a otra, sin embargo, en la sinapsis, la presencia de diversos químicos que están entre las neuronas determinan una oposición al paso del impulso, lo cual afecta la percepción de la neurona receptora.

La computación ha tomado esta mecánica y con base en ella, se han creado diversas estructuras de datos para la interpretación y búsqueda de patrones o asociación de datos. A dichas estructuras de datos se le denominan *redes neuronales artificiales* o simplemente redes neuronales (Kriesel, 2005).

La abstracción más simple de una red neuronal se presenta como la interconexión de objetos *neuronas* por medio de pesos sinápticos; donde las neuronas representan las neuronas reales y los pesos sinápticos representan todo el proceso de la sinapsis.

Las neuronas se agrupan en conjuntos conocidos como *capas* (*layers* en inglés). La información pasa de una capa de neuronas a otra hasta llegar a la capa final en donde se obtiene un resultado. En la siguiente imagen se muestra un diagrama clásico de una red neuronal, en él se observa la representación típica de una neurona, los pesos sinápticos que las conectan, así como también las diferentes capas que puede tener la red .

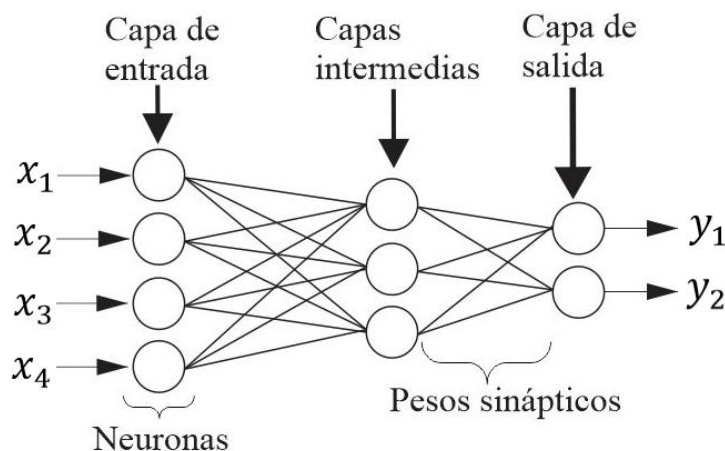


Figura 3.13 Diagrama de una red neuronal clásica. La lista de valores  $X$  representan el vector de entrada de la red, mientras que la lista de  $Y$  representa la salida de datos.

Los pesos sinápticos presentes en la red neuronal no son iguales para la conexión de cada par de neuronas y esto se debe precisamente a que en la naturaleza esto no es así; dicho de otro modo, los pesos sinápticos entre las neuronas presentan diferentes valores de peso.

Cada red neuronal se caracteriza por las siguientes consideraciones (Fauseet, 1994):

- La manera en que se conectan las neuronas: La manera en que las conexiones se dan es libre según el modelo de la red neuronal. Las neuronas no están limitadas a conectarse exclusivamente con las neuronas de otra capa, las conexiones se pueden dar entre neuronas de una misma capa, incluso en algunos casos es posible que exista una conexión que retroalimente a una misma neurona. Algunos modelos de red neuronal definen un comportamiento general para las conexiones, sin embargo, esto al final depende del diseñador de la red.
- La mecánica empleada para determinar los valores que tienen los pesos sinápticos: A este proceso se le llama también entrenamiento y al igual que el punto anterior, esto también depende en gran parte del modelo de red. Típicamente los métodos de entrenamiento se dividen en dos grupos, los supervisados (con un patrón guía) y los no supervisados (sin patrón guía), cada uno con sus propias fortalezas y desventajas.

- La función de activación: La función de activación es el proceso que se lleva a cabo dentro de la neurona. La sumatoria de las entradas percibidas por una neurona son evaluadas por esta función, el resultado obtenido será la salida que propagará la neurona. La función de activación puede ser desde una simple condición de falso y verdadero, hasta una compleja función matemática, todo depende del diseño de la red neuronal y del propósito que se le dé a la misma.

Dependiendo del uso y objetivo, cada red neuronal tiene varias capas internas donde se procesa la información, por lo general en los diagramas, las capas internas no son mostradas y solo se hace énfasis en la capa de entrada y la de salida que son las de mayor importancia.

Como se mencionó en párrafos anteriores, las redes neuronales son empleadas para el análisis de la información, siendo la detección de patrones y la asociación de grupos una de las aplicaciones más típicas para las redes.

En el campo del mallado así como en el desarrollo de triangulación, existen diversos estudios que se enfocan en la implementación de redes neuronales en los cuales se presentan diferentes ideas para detectar “patrones” sobre un dominio y con base en esto, determinar la triangulación o la densidad de elementos sobre el área que evaluó la red.

El proceso de mallado está sujeto a diferentes variables como lo son las dimensiones del dominio, la tasa de crecimiento entre los elementos, el tamaño máximo y mínimo de los elementos, entre otros, por lo cual el resultado obtenido del proceso es una malla que puede variar mucho si se realiza algún cambio en cada una de dichas variables. Debido a esto, se requerirá de mucho esfuerzo para generar un registro que relacione los cambios de las variables con el resultado obtenido; por ello no es una sencilla tarea el determinar una muestra de patrones que se pueda emplear en el entrenamiento de una red. Por lo planteado anteriormente, la implementación de redes neuronales en estos casos está guiada por métodos

de entrenamiento no supervisado, debido a que no se cuenta con un patrón o registro entre datos de entrada conocidos y salida esperada.

Los modelos de red neuronal que implementan estrategias de aprendizaje no supervisado se caracterizan por la búsqueda de un comportamiento en los datos que analizan. Para simplificar el proceso de entrenamiento no supervisado existe la posibilidad de imponer reglas o tendencias que deben presentar los datos, esto a través de heurísticas.

La manera tradicional para entrenar una red sin un conjunto de datos guía, es comparando los valores resultantes de su función de activación; cuando existe una neurona con un valor muy grande (o muy bajo según se requiera) se reducen los pesos sinápticos de sus neuronas vecinas y de sí misma, pero aumenta el valor del peso sináptico de las neuronas que no son vecinos inmediatos, de esta manera puede decirse que la neurona “atrae” a las neuronas que son vecinas inmediatas. Con el paso de las iteraciones de entrenamiento las neuronas que generaron valores altos alteran considerablemente sus pesos sinápticos y el de las neuronas vecinas. Esto se traduce en el medio natural como una asociación de neuronas, lo que significa que un grupo de neuronas de manera natural tiende a relacionarse y facilitar el paso de la información.

Esta mecánica de acercar las neuronas a una en particular o a un grupo de éstas se le denomina como auto organizado, ya que las neuronas automáticamente se ordenan en grupos. Cada grupo representa la identificación de una clase de patrón entre los datos de entrada.

Dentro de los modelos de entrenamiento no supervisados se encuentran los *mapas de auto organizado* (conocidos también como *SOM* por Self-Organizing Map en inglés), para ser más preciso los *Mapas de Kohonen* (ver Figura 3.14). Al igual que la mayoría de modelos de redes neuronales, SOM emplea capas para procesar los datos. (Kohonen, 1982)

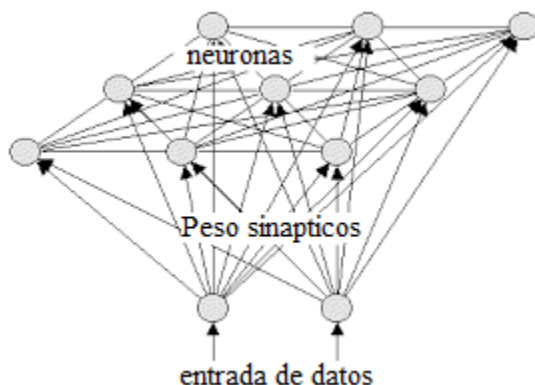


Figura 3.14 Diagrama general de un mapa de auto organizado.

Tomando los SOM como fundamento, diversos trabajos se han realizado a lo largo del mundo buscando aprovechar sus características en el proceso de diseño y desarrollo del mallado, convirtiéndose en la principal implementación de agentes inteligentes en la generación de mallas. Típicamente cada estudio define una serie de parámetros los cuales son denominados como *criterios de interés*. Los SOM evalúan la ubicación y existencia de dichos criterios en un área o zona del dominio; si existe la presencia de algún patrón, dato, o criterio que la red neuronal busque en la zona que evalúa, entonces esa zona gana un nivel de importancia superior en comparación a las zonas donde no existe presencia de criterios.

Cabe mencionar que la mayoría de los trabajos donde se emplean SOM están orientados a la simulación y estudio de un fenómeno físico en específico, por lo cual los criterios de interés están orientados a un fenómeno en particular. Se quiere adaptar el SOM a un nuevo caso de estudio, es preciso volver a entrenar la red para evaluar nuevos criterios.

## CAPÍTULO IV. DESARROLLO

### 4.1 Modelo de negocio

El objetivo principal de esta tesis, es la creación de una herramienta que permita la creación de una malla para un dominio bidimensional. Dicha herramienta de software formará parte de un sistema de simulación, el cual se encuentra en desarrollo dentro del Centro de Investigación en Materiales Avanzados S.C. (CIMAV por sus siglas).

El software de simulación se encuentra compuesto por diferentes módulos, los cuales aportan características que van desde la interfaz gráfica, hasta el procesamiento matemático. A la fecha de desarrollo de esta tesis, el software cuenta con módulos para realizar estudios 1D. La inclusión del proyecto de software presentado en este documento, dotará al software de simulación de un módulo que le permitirá realizar estudios en 2D y por lo tanto le permitirá aumentar sus alcances.

El enfoque contemplado para ambos software (tanto la herramienta de mallado, así como el sistema de simulación) es presentarlos como software libre y que esté a disposición del público en general. La intención de esta distribución es permitir el acceso a una herramienta de software confiable a diferentes sectores de la sociedad que no pueden costear la compra de un programa profesional de este mismo estilo, como lo pueden llegar a ser las instituciones educativas públicas y los alumnos de nivel superior y posgrado en general.

En el apartado de marco teórico se mostró una metodología que es seguida por diversos estudios presentados en el estado del arte, los cuales descomponen el proceso de mallado en diversos pasos. Buscando seguir la misma mecánica en el proceso de desarrollo y la lógica del sistema, se planteó el siguiente esquema de trabajo que contempla el proceso de investigación previo al desarrollo de la herramienta como tal (ver Figura 4.1).

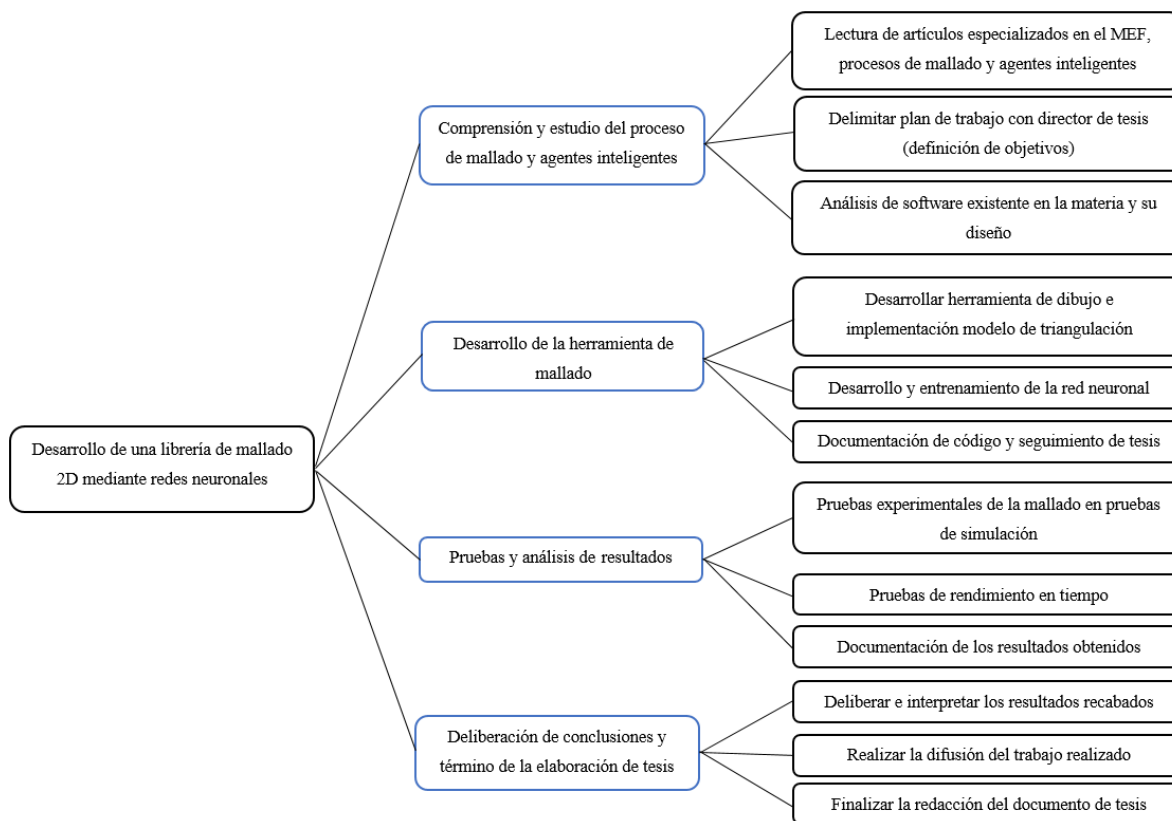


Figura 4.1 Esquema de plan de trabajo.

## 4.2 Delimitación de requerimientos generales

A lo largo de esta etapa se definen los puntos clave que se deben de tener previo al proceso de desarrollo y la metodología que se seguirá a lo largo del desarrollo, contemplando en todo momento los objetivos que el software debe de cubrir.

Con base en los datos que se establecen en *objetivos específicos, alcances y limitaciones* se procede a determinar los requisitos para el desarrollo y especificaciones de funcionalidad de la librería a desarrollar, los cuales se enlistan a continuación:

- *Entorno de desarrollo:* Es preciso contar con un entorno de desarrollo que facilite la implementación de un lenguaje en el cual se pueda aplicar el paradigma orientado a objetos, ya que en dicho paradigma la representación de una estructura de datos es



una tarea que se realiza de manera sencilla y eficaz, permitiendo además la representación ordenada de la información que se empleará tanto para el procesamiento como para la interacción con el usuario.

- *Zona de dibujo:* Aún y cuando el producto final sea una librería de software, es necesario el desarrollo de ambiente de dibujo en el cual sea posible mostrar gráficamente el resultado del mallado, además el mismo puede ser tomado como una guía que permita a un usuario comprender la manera en que se implementa y se hace uso de la librería.
- *Estructuración de los datos:* La nomenclatura de las variables, así como de los tipos de datos creados para la librería deberán ser claros e intuitivos, de manera en que para un usuario no experimentado en programación sea capaz de comprender la representación y uso de cada tipo de dato o rutina de código. Adicionalmente se contempla la implementación de un esquema de nomenclatura *camel Case* para este propósito.
- *Validación geométrica:* Se requiere que el software sea “seguro” en cuanto a la validación de la información previa al procesamiento de la misma, esto es, determinar qué características tales como si el dominio se encuentra abierto, si los tamaños máximos y mínimos de los elementos no son contradictorios etc., así como que sean datos coherentes.
- *Procesamiento:* El tiempo de respuesta de la herramienta debe ser veloz o al menos aceptable; este punto es fundamental pues el mallado conforma un paso de un proceso más grande (el MEF) y por lo tanto no puede ser lento, ya que esto perjudica el tiempo de análisis del proceso de simulación completo.

- *Puntos de interés:* Para determinar puntos con datos de interés, la herramienta presentará una estructura de datos tal que el usuario pueda realizar ajustes sobre los parámetros de cada punto de manera individual con facilidad.
- *Importación/Exportación:* La herramienta presenta rutinas que permiten realizar la importación y exportación de los datos del dominio así como su respectiva malla.
- *Proceso de refinado:* La librería permite realizar un proceso de refinado de la malla partiendo del mallado anterior.

El siguiente diagrama de casos de uso (ver Figura 4.2) representa el funcionamiento de la librería, en el cual se contempla que los puntos anteriores deben ser cumplidos.

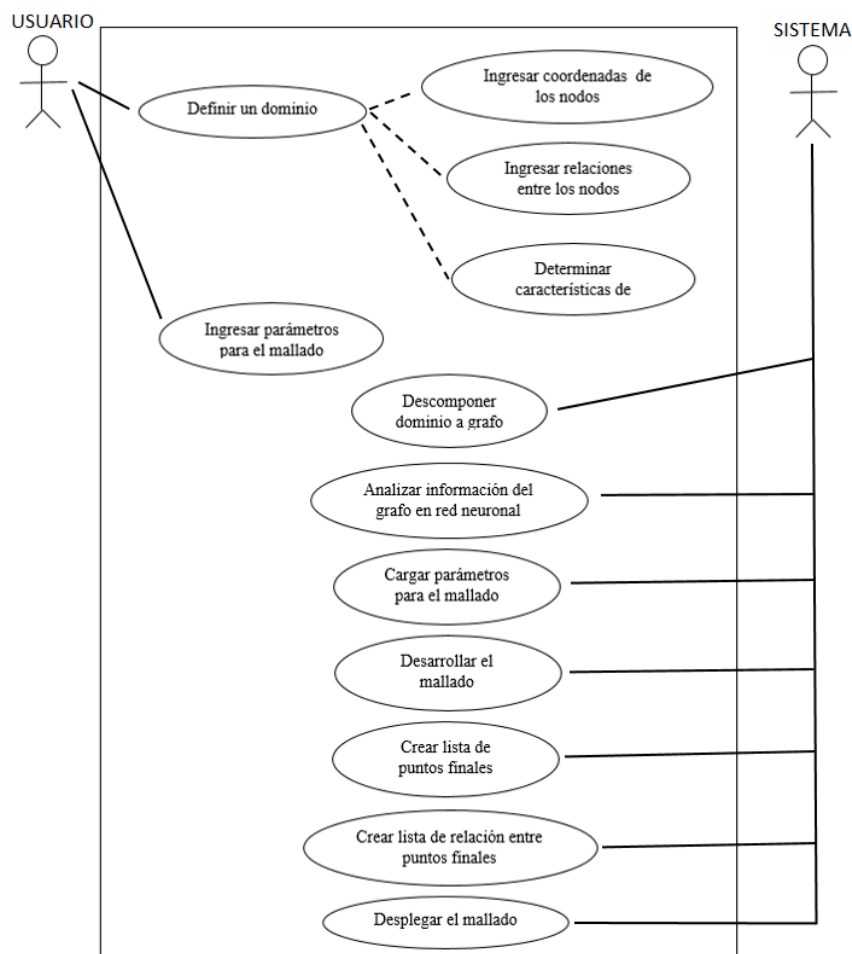


Figura 4.2 Diagrama de casos de uso de funciones del sistema.

A partir de la información anterior se puede determinar que el sistema después de recibir la información proporcionada por el usuario, divide el procesamiento en 3 etapas:

- Preparar la información para crear la malla: Esto paso consiste en descomponer el dominio a grafo, analizar el grafo en la red neuronal y cargar parámetros para el mallado.
- Crear y procesar la malla con los datos preparados.

- Generar datos de salida: Su función es crear lista de puntos finales, crear lista de relación entre puntos finales y desplegar el mallado.

El sistema en los últimos pasos del caso de uso generará una serie de listas, las cuales contienen información de los puntos, como lo son las aristas de los elementos, así como la relación que debe existir entre ellos para generar la malla. Con estas listas se puede crear una representación visual del mallado permitiendo verificar la congruencia entre las entradas de datos y el resultado.

Para finalizar la etapa de requerimientos, se presenta el diagrama de actividades en la siguiente figura (ver Figura 4.3), dicho diagrama permite establecer de manera visual el flujo de información dentro del sistema de acuerdo a los requerimientos y al diagrama de casos de uso antes expuesto.

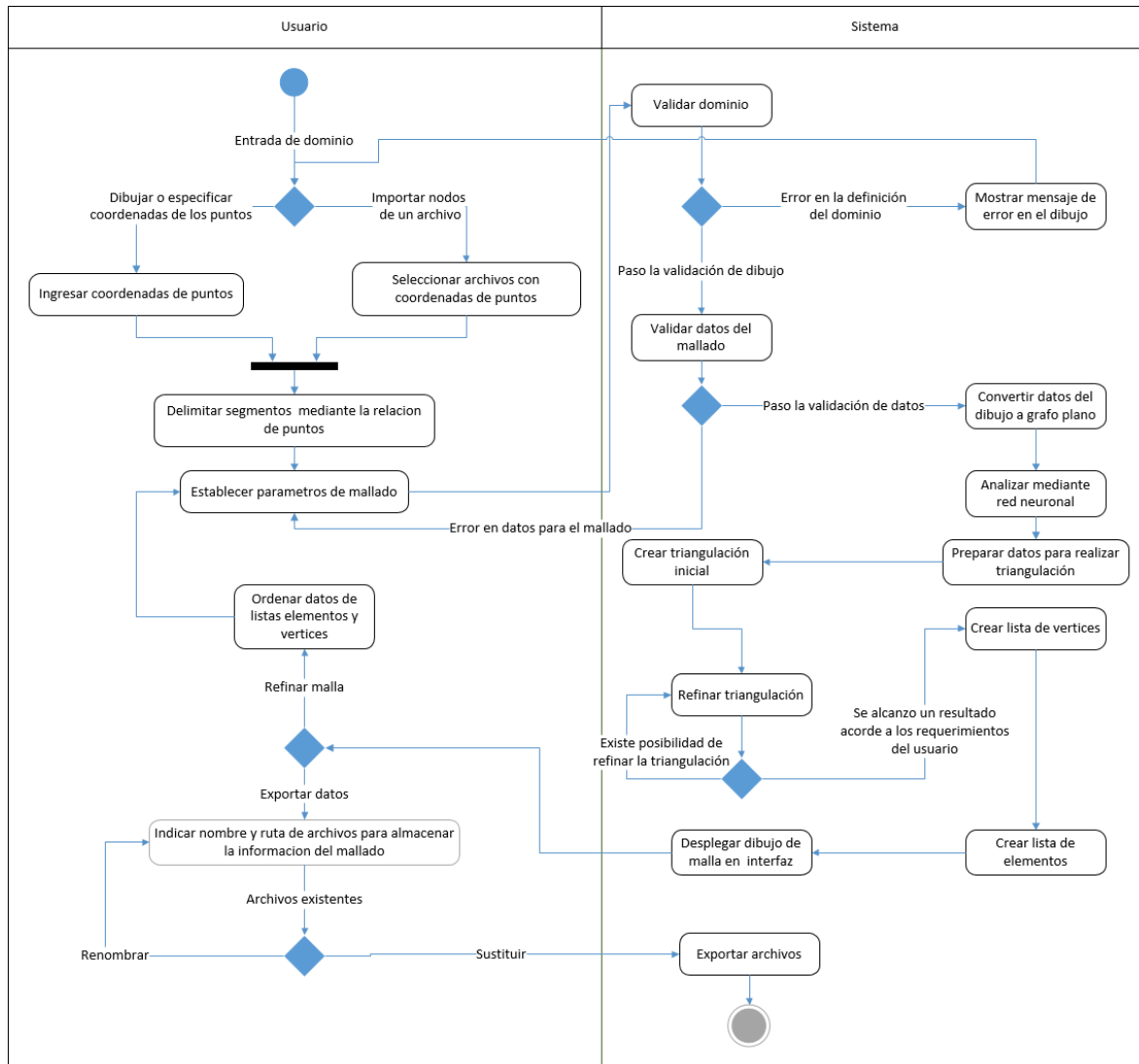


Figura 4.3 Diagrama de actividades del sistema.

### 4.3 Análisis y diseño general

La etapa de análisis y diseño consiste en traducir los requisitos a una especificación o modelado que describe cómo implementar el sistema. Para el desarrollo de la herramienta se decidió implementar el paradigma de programación orientado a objetos mediante la cual es posible definir estructuras de datos llamadas *clases*, las cuales representan un conjunto de datos que describen a un objeto específico así como su comportamiento.

Los diagramas de clases exponen los elementos que conforman una clase, es decir, el modelado que se le dio a la clase que en el diagrama se representa, permitiendo ver de manera clara los componentes que tendrá cada objeto de dicha clase.

A continuación se muestran los diagramas de clase definidos y empleados para este desarrollo (ver Figura 4.4), los cuales están divididos en dos bloques: las clases propias para el análisis de los trazos y las clases para el desarrollo de la red neuronal.

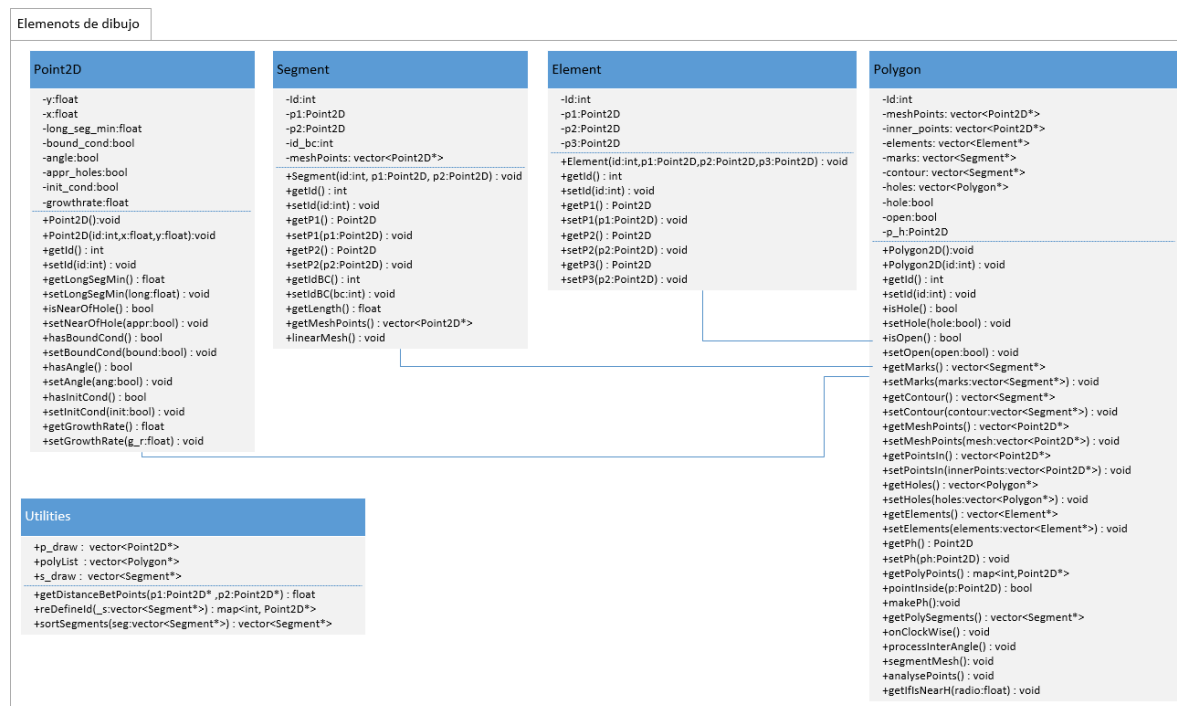
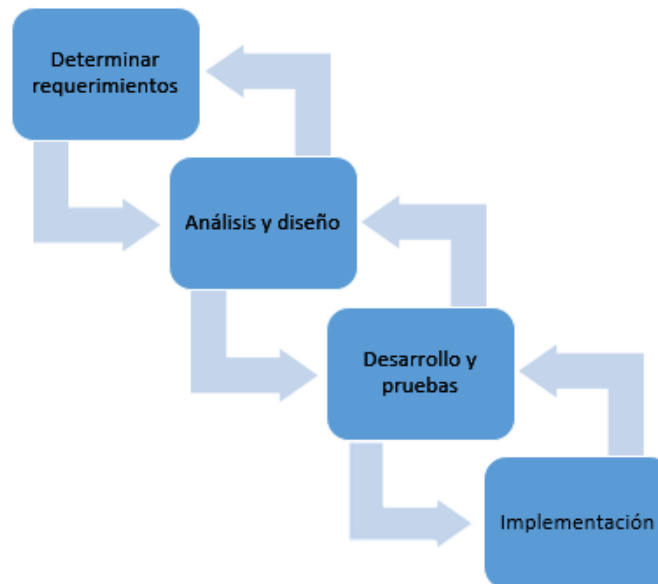


Figura 4.4 Diagrama de clases del sistema.

El modelo de trabajo implementado a lo largo del desarrollo del software es el conocido como cascada con retroceso y se caracteriza por contar con cuatro etapas básicas que consisten en determinar los requerimientos del software que se desarrollará, realizar un análisis para determinar cuáles son las metas principales y un diseño de cómo alcanzar mediante programación dichas metas. Posteriormente se realiza la codificación que se diseñó previamente y se implementan pruebas para evaluar el desempeño del software. Una vez terminado todo lo anterior se procede a implementar el software; sin embargo en todo

momento es posible regresar a pasos anteriores para definir ajustes derivados de imprevistos que surjan a lo largo del proceso. En la siguiente imagen se muestra el diagrama del proceso mencionado.



*Figura 4.5 Diagrama de cascada con retroceso.*

## 4.4 Interfaz gráfica

### 4.4.1 Determinación de requerimientos.

Pensando en que la interfaz gráfica en realidad sirve como elemento de prueba y validación de la herramienta, así como un ejemplo base para una implementación de interfaz gráfica más compleja, se ha optado por un diseño sencillo que permita la manipulación de los datos de entrada.

#### 4.4.2 Análisis y diseño.

Los elementos que componen la interfaz gráfica se muestran a continuación:

- *Área de dibujo:* Esta área permite al usuario agregar nodos que forman parte de los trazos, todo ello mediante el uso del mouse.
- *Nodos:* Lista de nodos agregados por el usuario, la cual muestra un identificador del nodo, las coordenadas en  $x,y$ , además de cuatro rangos de evaluación para la red neuronal(ver en el apartado 1.5.1 Objetivo General).
- *Segmentos:* Lista de segmentos en la cual cada fila representa una línea que conecta dos puntos, esto se logra indicando el punto de inicio y punto destino.
- *Polígonos:* Un listado de polígonos, al cual se le asocian los nodos y los segmentos que lo conforman, además existe la opción de indicar si es que otro elemento polígono es un orificio (agujero) de otro.
- *Botones:* Elementos botones que permitan actuar como detonadores de eventos de despliegue de ventanas para la selección de archivos, los cuales tienen el fin de importar o exportar los datos del dibujo a mallar, así como del mallado final.
- *Ventanas:* Una serie de ventanas estilo cajas de texto donde el usuario indica detalles para el mallado tales como el tamaño máximo y mínimo para los elementos, así como cajas de chequeo donde es posible manipular los parámetros evaluados por la red neuronal (esto solo durante la etapa de desarrollo y pruebas).

La siguiente imagen (ver Figura 4.6) muestra la interfaz gráfica propuesta, hay que recordar que el diseño para esta tesis es meramente para realizar pruebas y como muestra de una implementación.



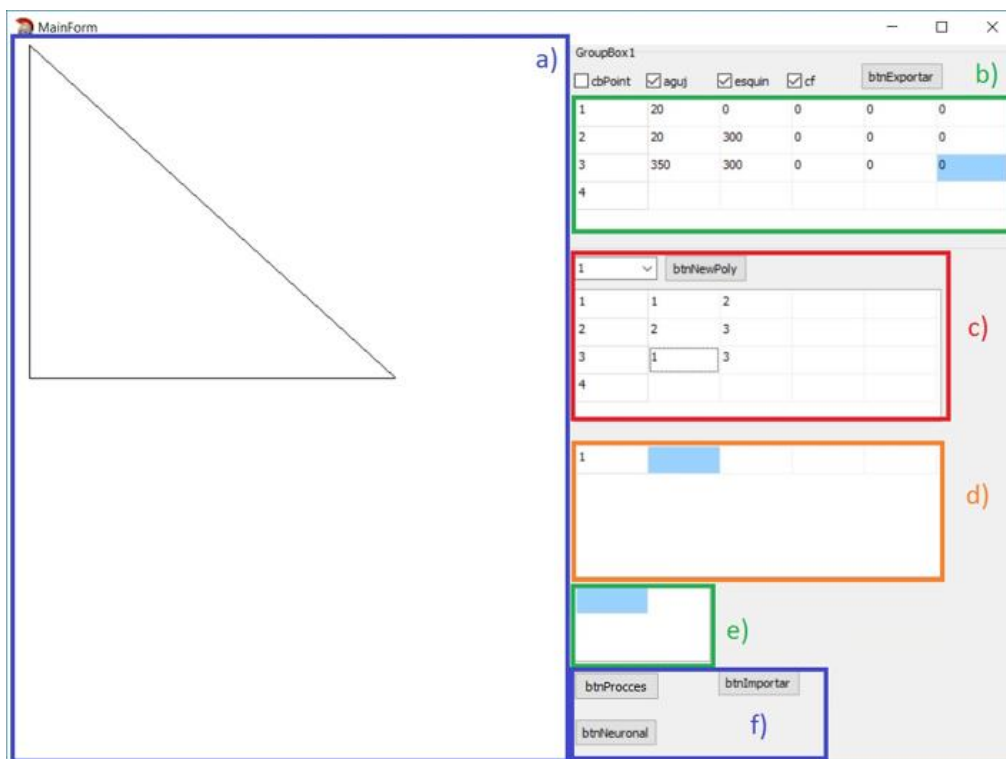


Figura 4.6 Interfaz gráfica. a) área de dibujo, b) lista de puntos del dibujo y botón exportar, c) lista de polígonos y tabla de segmentos por polígono, d) lista de marcas del polígono, e) lista de agujeros para el polígono, f) botones para mallar e importar.

#### 4.4.3 Codificación.

En la interfaz gráfica es posible marcar los vértices de los polígonos mediante el uso de mouse dando clic sobre el área de dibujo, o en su defecto editando las coordenadas directamente en la tabla situada en la parte superior de la interfaz. En dicha tabla el primer campo representa el identificador del punto (generado de manera automática). Los siguientes dos datos corresponden a las coordenadas  $x$  y  $y$  respectivamente. Cabe reiterar que en esta interfaz gráfica no se emplea una librería de gráficos como tal, en lugar de ello, se hace uso de las herramientas de dibujo más básicas y primitivas del sistema, por lo que el origen del plano se encuentra en la esquina superior izquierda, generando valores positivos en el eje  $x$  mientras se recorra hacia la derecha y para el caso del eje  $y$ , genera valores positivos al avanzar a la parte inferior de la ventana, como se puede apreciar en la Figura 4.7.

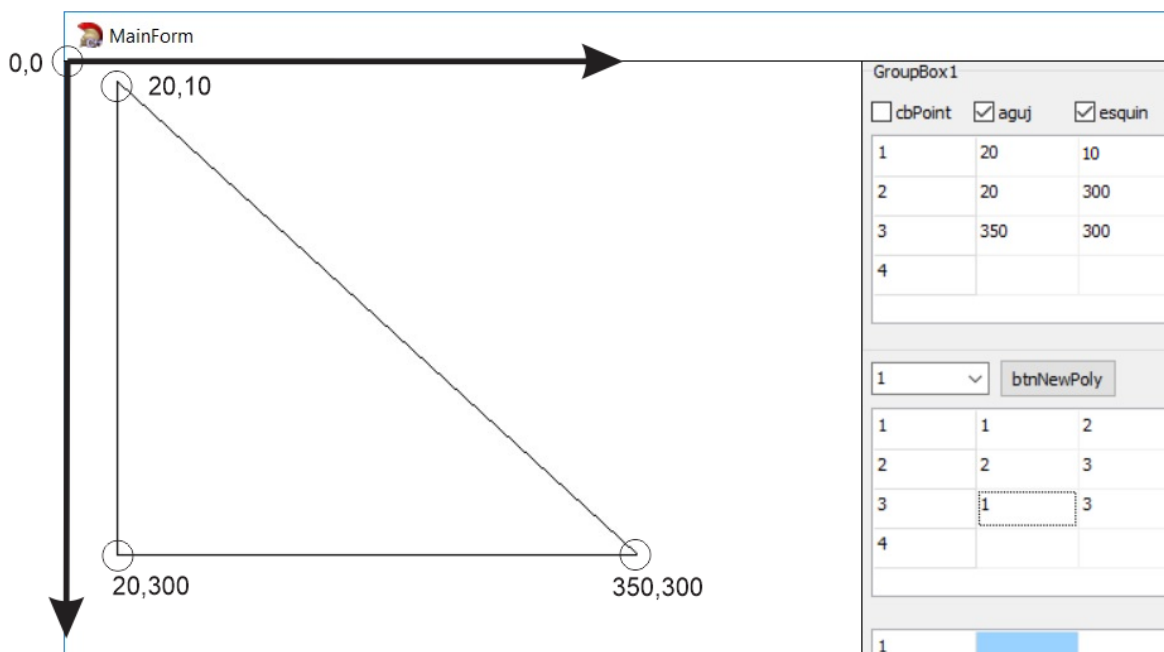


Figura 4.7 Demostración del sistema de coordenadas en el área de dibujo.

Adicionalmente, en caso de que el usuario quiera importar los vértices de un archivo de textos, se agregó un botón que despliega el componente estilo `TFileDialog` que muestra una ventana en donde se le solicita al usuario ubicar el archivo que contenga las coordenadas de los puntos, los cuales automáticamente se almacenan en la tabla de datos correspondiente. La estructura del archivo de texto es la siguiente: número flotante, seguido de espacio o tabulador, número flotante, seguido por último por un salto de línea; esto representado en una expresión regular propia del lenguaje C/C++: `%f\t%f\n`. El código que muestra este proceso aparece en el Anexo 8.1 .

Los campos adicionales en la primera tabla de la interfaz gráfica asisten el proceso de prueba, cada uno representa un criterio de interés para determinar la densidad de la malla en un punto dado. Estos valores se imponen de manera que el sistema no buscará evaluar el criterio si el campo al que corresponde ya está definido en la tabla.

Todos los nodos son almacenados en un listado global, de modo que son accesibles en todo momento, el listado está definido como una lista nombrada *p\_draw* de tipo `std::vector<Point2D*>` (ver Figura 4.4 Clase Utilities).

En la parte central de la interfaz aparece un elemento estilo TComboBox que despliega un enumerado de todos los polígonos creados hasta el momento. Los polígonos son almacenados en una colección con estructura de datos de tipo cola: `std::vector<Polygon*>` llamada *polyList* (ver Figura 4.4 Clase Utilities). El código empleado en la generación de los elementos de *p\_draw* se muestra en el Anexo 8.2.

La siguiente tabla dentro de la interfaz, representa los segmentos que conforman el polígono seleccionado (el contorno). El primer campo de la tabla representa el identificador del segmento, el segundo campo corresponde al identificador del punto inicial, el tercero corresponde al punto final, los siguientes campos en fila son empleados para controlar y realizar pruebas de funcionalidad. Cada segmento es almacenado dentro del polígono mediante la lista llamada *contour*, la cual es un miembro de la clase Polygon (ver Figura 4.4. Clase Polygon). Adicionalmente, cada uno de los segmentos también es guardado en una lista que es utilizada para realizar el dibujo de los trazos, dicha lista tiene por nombre *s\_draw* (ver Figura 4.4. Clase Utilities). Las subrutinas empleadas para dibujar los segmentos así como los puntos con sus respectivos identificadores se muestran en el Anexo 8.3 y Anexo 8.4. El código empleado en la generación de los elementos de *s\_draw* se muestra en el Anexo 8.5.

La tercera tabla de la interfaz se emplea para crear segmentos al igual que la anterior. Sin embargo, éstos no son almacenados en el listado que corresponde al contorno. Estos segmentos son guardados en el listado de las marcas, es decir, son segmentos que están dentro del dominio pero que no delimitan el contorno del dominio. Cabe señalar que existe la posibilidad que alguno de sus nodos sí forme parte del contorno; la lista empleada para estos datos fue nombrada *marks* y es miembro de la clase Polygon (ver Figura 4.4 Clase Polygon).

El ultimo listado que aparece en la parte inferior de la interfaz gráfica es un elemento tabla que almacena el identificador de aquellos polígonos que son agujeros o huecos del polígono que se está editando. Para el manejo de estos datos la clase polígono cuenta con un listado llamado *holes* (ver Figura 4.4 Clase Polygon). Cada vez que un polígono es considerado hueco de otro se emplea la instrucción *setHole(true)* para indicar que su variable *hole* debe cambiar a verdadero; este dato será leído más adelante para evitar que la malla se extienda al interior de estos polígonos.

Los últimos 3 botones restantes en la parte inferior tienen las siguientes funciones:

- *btnProcces*: Crea una malla de manera automática sin pasar por la evaluación de la red neuronal (proceso y anexo explicados en párrafos siguientes).
- *btnNeuronal*: Crea una malla analizando la información de los dominios mediante una red neuronal (proceso y anexo explicados en párrafos siguientes).
- *btnImportar*: Importa los datos del mallado obtenido a dos archivos de texto, uno con los vértices del mallado y otro con las relaciones entre los vértices para generar los elementos (ver Anexo 8.6).

## 4.5 Módulo de mallado.

### 4.5.1 Determinación de requerimientos.

El objetivo central del software consiste en la generación de la malla y según como se mostró en el marco teórico, los métodos de triangulación son un método eficaz de lograr este fin. Adicionalmente se mostró que los métodos de triangulación derivados del método Delaunay son confiables en cuanto a generar elementos de buena calidad. Es entonces necesario el desarrollo de un módulo para generar una triangulación de este tipo o la implementación de una librería ya existente que emplee el mismo lenguaje de programación que el proyecto para cubrir esta necesidad.

#### 4.5.2 Análisis y diseño.

Por cuestiones de tiempo de desarrollo, se confió esta tarea a la librería *Triangle* (versión 1.6) que fue desarrollada por Jonathan Richard Shewchuk con fines de generar mallados para análisis de elementos finitos, por lo cual es una solución viable a implementar. Sin embargo aún que la estructura de datos empleada para representar las figuras son muy similares a las empleadas por la librería de triangulación, existen diferencias entre los tipos de datos y paradigmas de programación empleados, por lo cual se adecuaron los datos obtenidos en la interfaz gráfica para que sean compatibles con los de la librería de triangulación. Así mismo, los resultados obtenidos por la librería de triangulación deberán de ser adecuados a la estructura de datos que se emplea en el proyecto general.

#### 4.5.3 Codificación.

La librería *Triangle* maneja una estructura de datos tanto para la entrada como para la salida de información que se denomina como *triangulateio*; si se desea emplear como entrada de datos o como salida, hay que especificar una serie de parámetros para cada opción.

Cuando se trata de implementar una entrada de datos se define la variable *triangulateio in* y se definen las variables propias de una salida como *cero* y *null* para evitar conflictos de datos; esto se muestra en el Anexo 8.7. A continuación es necesario preparar los datos de los polígonos para después pasarlos a la librería *Triangle*, el primer paso recomendado por diversas fuentes consiste en ordenar los segmentos de modo en que sigan el sentido horario, esto facilita de gran manera el procesamiento de la información, así como la velocidad en la búsqueda de datos y mejora el tiempo de respuesta de la herramienta significativamente. Para llevar a cabo el ordenamiento de los segmentos existen diversos métodos pero la mayoría solo sirven para polígonos que son convexos o para el caso contrario (no convexos), por lo que se ideó una manera de generalizar este problema a ambos casos. El primer paso consiste en generar un par de puntos a la mitad del segmento, este par de puntos se alejarán del

segmento de manera perpendicular a una distancia  $\Delta = \min\_len * 0.01$  como se ve en la Figura 4.8

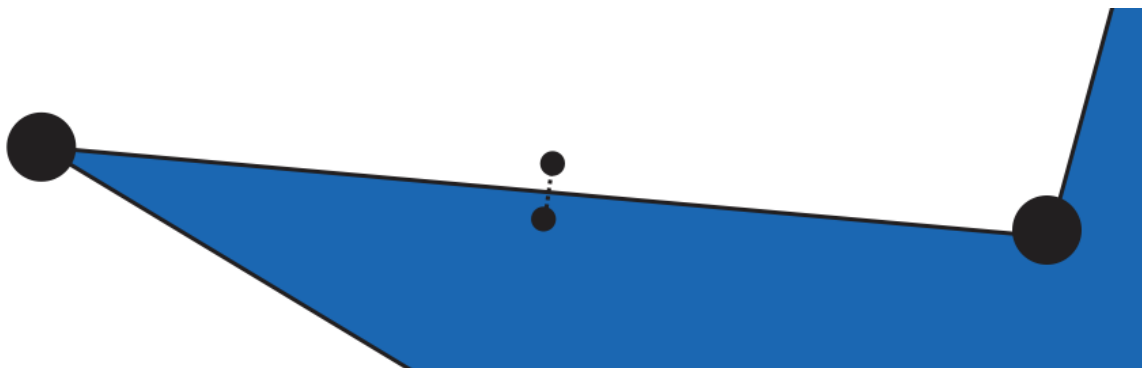


Figura 4.8 Muestreo de puntos para determinar sentido de los segmentos.

Donde  $\min\_len$  representa la longitud del segmento más corto presente en el polígono. Posteriormente se evalúa si alguno de los dos puntos está dentro del polígono, si ninguno de ellos lo está, se reduce la distancia hasta conseguir esto. Con los datos del punto auxiliar que está adentro del polígono, es fácil determinar el sentido que tiene el segmento según la posición de los puntos del segmento y los ángulos que tiene con base en el punto auxiliar.

Para evaluar si un punto está o no adentro del polígono se implementa el método llamado *Ray Casting* el cual consiste en evaluar el punto de interés en aquellos segmentos cuyos puntos tengan coordenadas en  $y$  que estén por encima y debajo del punto de interés y que además tenga algún punto con sus coordenadas en  $x$  antes de la posición en  $x$  del punto a evaluar. Se considera que el punto recorre de manera horizontal la figura partiendo desde el exterior de ella. El punto está dentro de la figura si “*atraviesa*” o “*brinca*” un número impar de segmentos que conforman el contorno de la figura hasta alcanzar su posición final (la coordenada en  $x$  que tiene el punto a evaluar); la siguiente imagen ayuda a entender el concepto de este método.

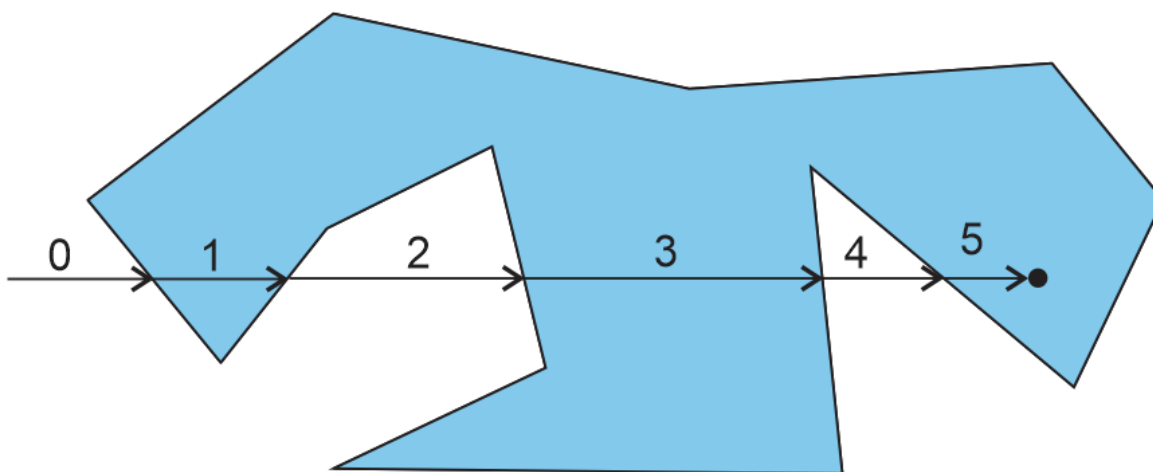


Figura 4.9 Ejemplificación del método Ray Casting.

El código empleado para ordenar el sentido de los segmentos se llama *onClockWise* y el método para determinar si un punto está dentro de un polígono *pointInside* que recibe como único parámetro un objeto *Point2D* a evaluar y retorna un booleano como respuesta. El código de los métodos mencionados puede verse en el Anexo 8.8 para *onClockWise* y Anexo 8.9 para *pointInside*.

Ya que los segmentos están ordenados, el sistema evalúa si el mallado es genérico o inteligente según el botón que se empleó para ejecutar las rutinas. El siguiente pasos consiste en generar un pre-mallado de manera automática. El pre-mallado consiste en agregar nodos sobre el contorno de la geometría. La cantidad de nodos puede ser modificada, sin embargo de manera predeterminada se agregan catorce nodos. Esta pre-malla será utilizada por la librería de triangulación como una primera nube de puntos, lo cual mejora el tiempo de respuesta de la librería según su documentación, ya que estos facilitan las primeras etapas de procesamiento de la librería.

Para generar los nodos adicionales se adopta la estrategia de considerar a cada segmento como si fuera un vector; se representa la dirección del vector mediante el vector unitario y variando la magnitud del mismo, se determina la distancia a la que se colocará cada nodo

adicional. Para obtener el vector unitario que representa el sentido de un segmento es necesario considerar al mismo como un vector de la siguiente manera:

- Todo segmento parte de un punto origen (A) con dirección a un punto destino (B).
- Todo segmento tiene una cierta longitud comparable a la magnitud de un vector.

Con estos dos puntos podemos describir entonces que el segmento como se ve en la siguiente Figura 4.10.

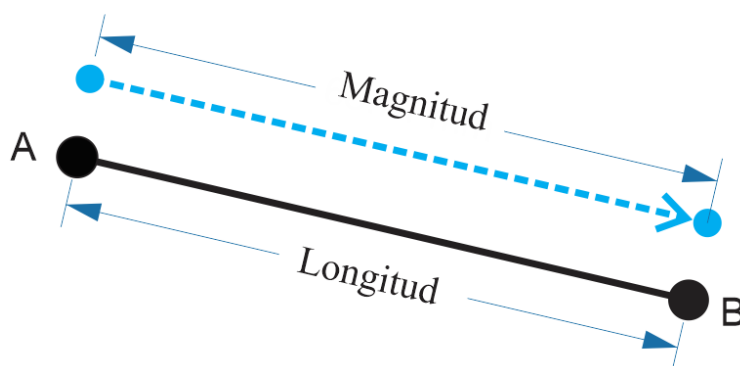


Figura 4.10 Representación de segmento como vector, línea punteada ejemplifica el vector que representa al segmento mostrado con la línea continua.

El vector  $\vec{u} = \overrightarrow{AB}$  descrito en la imagen anterior sirve para representar el segmento del punto A al punto B. El vector unitario está definido como

$$\vec{v} = \frac{\vec{u}}{|\vec{u}|} \quad (4.1)$$

para determinar el modulo o magnitud del vector tenemos el teorema de Pitágoras que al ser aplicado al segmento, obtenemos

$$|u| = \sqrt{(B_x - A_x)^2 + (B_y - A_y)^2} \quad (4.2)$$



Con esto es posible entonces descomponer el sentido del vector  $\vec{u}$  en un par de ecuaciones que describen la dirección que sigue el vector en el eje  $x$  y  $y$

$$Dx = \frac{B_x - A_x}{|u|} \quad (4.3)$$

$$Dy = \frac{B_y - A_y}{|u|} \quad (4.4)$$

Posteriormente para generar los puntos adicionales (para el pre-mallado) es necesario determinar sus coordenadas mediante las siguientes formulas

$$x = A_x + Dx * \Delta_x \quad (4.5)$$

$$y = A_y + Dy * \Delta_y \quad (4.6)$$

donde  $\Delta_x$  y  $\Delta_y$  representan la distancia a la que estarán separados los nodos adicionales entre ellos. De este modo se agregan nodos adicionales utilizados en el proceso de triangulación para facilitar el proceso de mallado. Cabe recalcar que todo esto es necesario para garantizar que los nodos adicionales estén sobre los segmentos, ya que si un nodo está fuera del área del domino, se cambiarían las dimensiones del mismo y esto ocasionaría graves problemas al intentar resolver el modelo mediante el MEF. Todo este proceso es desencadenado en cada uno de los segmentos de un polígono por medio de la instrucción `segmentMesh` (ver Figura 4.4 Clase Polygon) que a su vez hace llamado al método `linearMesh` presente en cada objeto segmento (ver Figura 4.4 y Anexo 8.10).

Ya con los nodos internos del segmento almacenados en cada uno mediante la lista `meshPoints` (ver Figura 4.4) se procede a calcular el área a triangular. Para ello se aplica el método conocido como determinantes de Gauss. Para aplicar este método, primero deben colocarse los segmentos en sentido anti-horario; ésta es una tarea fácil ya que solo se debe invertir el sentido en que se recorren las listas de los segmentos que fueron ordenados en sentido horario anteriormente. Después se aplica la siguiente fórmula

$$A = \frac{1}{2} \left| \sum_{i=1}^n x_i y_{i+1} - x_{i+1} y_i \right| \quad (4.7)$$

en la cual  $A$  es el área del polígono,  $n$  es el número de lados del mismo (segmentos de la lista *contour*) y las coordenadas  $x$  y  $y$  corresponden a las coordenadas de los vértices. Este método fue elegido debido a que permite determinar el área de manera efectiva para polígonos cóncavos y convexos por igual. Con este último dato se procede a preparar las variables que se implementan en la librería de triangulación.

Así como fue definida una estructura de datos del tipo *triangulateio* para convertir los datos de entrada, es necesario especificar una variable de salida por cuestiones de simplicidad, a esta variable se le llamara *out*. Implementando diversos ciclos y listas auxiliares, los datos de cada polígono son iterados y almacenados en la estructura *in*. Los datos de todos los puntos o nodos que conforman el dominio son almacenados en *in.pointlist* mediante la invocación del método *getPolyPoints*. Los datos de los segmentos son almacenados mediante *in.segmentlist* utilizando la rutina *getPolySegments*. Hay que reiterar que los segmentos y puntos de agujeros dentro del polígono a mallar también son contados en estos listados, incluso, debido a la manera en que opera la librería *Triangle*, se requiere que se genere un nodo adicional que debe estar ubicado en el interior del área que se considera como hueco. Este nodo extra es generado con la instrucción *makePH* que pertenece a la clase *Polygon* (ver Figura 4.4 Clase *Polygon*), esta instrucción sigue la misma mecánica empleada para generar un punto en el interior de un polígono planteado anteriormente, los datos de este nodo son almacenados en el miembro de clase *p\_h* de *Polygon*.

Nótese que estos datos no son descritos con mayor detalle ya que esta información pertenece al diseño propio de la librería *Triangle*, para conocer su codificación es preciso adentrarse en los manuales de usuario presentes en la documentación de la librería.

Una vez terminada la carga de la información se ejecuta la orden *triangulate*, mediante la cual se genera en *out* el listado de los nodos resultantes y su respectiva relación para generar

los elementos de la triangulación. En la instrucción anterior es posible especificar diferentes comportamientos y configuraciones propias que se desea que tenga la triangulación tales como tamaño máximo y mínimo de elementos, repetición de nodos o generar un refinado si los datos de entrada eran de una triangulación anterior.

El procedimiento típico de la librería consiste en generar un grafo plano con los datos recabados en etapas anteriores, después de esto se procede a realizar una triangulación compresada (descrita en el apartado 3.3 Triangulación Delaunay) y se procede entonces a refinar la triangulación hasta alcanzar los valores requeridos. El código encargado de realizar la preparación de las variables *in* y *out* así como de la ejecución de *triangulate* se expone en el Anexo 8.11. El código para exportar la información obtenida se muestra en el Anexo 8.12.

El resultado obtenido por la herramienta hasta este momento consiste en una malla estructural, por lo cual no presenta ninguna mejora en comparación con otras herramientas de mallado. Para cubrir los objetivos planteados en esta tesis, se desarrolló una red neuronal capaz de modificar los parámetros del mallado a modo de que los elementos se ubiquen en zonas de interés, sin modificar en gran medida el código expuesto en este apartado.

## **4.6 Módulo de red neuronal.**

### **4.6.1 Determinación de requerimientos.**

Para que el mallado sea adaptativo se debe implementar un algoritmo capaz de determinar qué zonas requieren una mayor o menor cantidad de elementos. Siguiendo los estudios mostrados en el marco teórico y estado del arte, una solución adecuada es el desarrollar e implementar una red neuronal. La red neuronal evaluará los criterios planteados en los objetivos y dará como resultado una tasa de crecimiento que se aplicará en el proceso de triangulación.

El resultado de la red neuronal debe seguir un comportamiento particular. Aunque existen diversos criterios para evaluar, existe una jerarquía entre ellos. Los cambios de condición de frontera en un punto son de mayor relevancia que la presencia de ángulos rectos o menores, a su vez, este último es más importante que la proximidad con orificios; de modo en que las combinaciones entre los criterios deben producir a su vez diferentes resultados.

#### 4.6.2 Análisis y diseño.

Según los requerimientos se crearon un conjunto de clases que son mostradas a continuación.

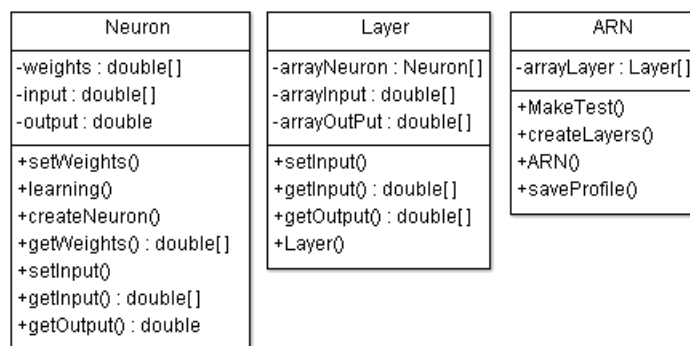


Figura 4.11 Diagrama de clases de red neuronal.

Mediante estas clases es posible crear una red neuronal que permita cumplir con los objetivos planteados. Como se mencionó en el marco teórico, existen diferentes consideraciones para llevar a cabo una buena implementación de la red neuronal, las cuales se pueden de limitar en tres apartados, primeramente, cómo se conectan las neuronas entre sí, segundo, qué función de activación se empleará y finalmente, cómo se entrena la red para definir el valor de los pesos sinápticos.

Debido a que no es posible generar una colección de datos para el entrenamiento de la red por la complejidad del problema, es conveniente implementar un diseño de red que aplique estrategias de aprendizaje no supervisado; para este caso *mapas de auto organizado multicapa*, los cuales contienen características deseables para este fin. Por un lado los mapas

de auto organizado pueden ser entrenados de manera no supervisada y las diversas capas de la red permiten la evaluación individual de cada criterio de interés. Así mismo, se busca que cada criterio de interés repercuta en el resultado de tal manera que el cambio de condiciones de frontera tenga una mayor relevancia que los otros dos criterios, seguido de la presencia de ángulos agudos y por último la proximidad con agujeros. Podría suponerse entonces, que de tener que representar la importancia de los tres criterios mediante un valor numérico la siguiente podría ser una sencilla solución al problema:

- Cambio en condición de frontera : importancia = 3
- Ángulo interno igual o inferior a noventa grados : importancia = 2
- Proximidad con orificios : importancia = 1

Sin embargo, este tipo de ponderación no contempla los casos donde se presenten combinaciones de puntos con dos o más características, en cuyo caso se podrían dar empates en importancia, lo cual no debería de ser así. La Tabla 4.1 muestra de manera descendente el orden de las combinaciones posibles que se pueden presentar en relación con su nivel de importancia.

*Tabla 4.1. Diferencias de importancia entre combinaciones de criterios de interés.*

| Cambio en condiciones de frontera | Ángulo interno igual o inferior a noventa grados | Proximidad con orificios | Importancia              |
|-----------------------------------|--|--------------------------|--------------------------|
|                                   |  |                          | Nada importante          |
|                                   |  | X                        |                          |
|                                   | X  |                          |                          |
|                                   | X  | X                        |                          |
| X                                 |  |                          |                          |
| X                                 |  | X                        |                          |
| X                                 | X  |                          |                          |
| X                                 | X  | X                        | Completamente importante |

Como se puede apreciar en la tabla anterior, existen ocho clasificaciones diferentes que la red neuronal deberá contemplar y según la combinación encontrada, determinar la población de elementos en esa zona según dicha combinación.

La red neuronal retorna un valor que se almacena en el punto de la clase Point2D y que se evalúa en su componente llamado growthrate (ver Figura 4.4 Clase Point2D) o tasa de crecimiento. Este valor será aplicado en el proceso de mallado para alterar el proceso de triangulación a modo de que se genere una malla con cualidades diferentes.

Para evaluar la presencia de los criterios de interés es necesario el desarrollo de rutinas que permitan analizar esto, el resultado obtenido debe ser almacenado en el objeto Point2D que se evaluó.

#### 4.6.3 Codificación.

Para determinar la presencia de los criterios de interés antes de realizar la ejecución de la instrucción segmentMesh, se itera cada segmento y se evalúa la sobre sus dos vértices la variable de init\_cond (ver Figura 4.4 Clase Point2D):

- si dicha variable es falsa, se procede a imponer el id\_bc (ver Figura 4.4 Clase Segment) del segmento como condición de frontera sobre el vértice evaluado;
- si init\_cond es verdadero entonces se compara la condición de frontera del vértice con el id\_bc del segmento, si ambos coinciden, la variable bound\_cond del vértice se mantiene sin cambios (por defecto definida en falso), en caso contrario cambia a verdadero, lo que significa que existen dos segmentos que comparten un vértice y que cada segmento tiene una condición de frontera diferente.

Como se pudo haber intuido, id\_bc representa una condición de frontera específica de un segmento, este valor se impone al segmento mediante la interfaz gráfica haciendo uso de la tercera columna de la lista de segmentos (ya sea de contorno o de marcas). Según el fenómeno físico que se trate, la cantidad de condiciones de frontera puede variar, es por esta razón que se ha optado por permitir al usuario que determine los identificadores con los que quiera representar las condiciones de frontera ya que finalmente la herramienta solo compara que

estos valores no sean iguales. El código empleado para evaluar la continuidad sobre las condiciones de frontera se ve en el Anexo 8.13.

Prosiguiendo con el proceso para determinar el ángulo interno de cada vértice, se recorren los segmentos en pares y se aplica el teorema del coseno mostrado a continuación

$$\cos \alpha = \frac{a^2 - b^2 - c^2}{-2bc} \quad (4.8)$$

En donde  $c$  y  $b$  son las longitudes de los segmentos al ángulo  $\alpha$  que se quiere conocer y  $a$  representa la longitud del lado opuesto al ángulo.

Aplicando la función recíproca de coseno al resultado de esta fórmula (arco coseno), se consigue el ángulo interior de cada vértice, después de esto se evalúa si está por debajo del rango definido como crítico (es decir menor a noventa grados) y de estarlo, entonces se indica como verdadera la bandera `angle` dentro de la clase `Point2D`. Anexo 8.14.

El proceso que se sigue para evaluar si un vértice está próximo a un orificio, se lleva a cabo con el método `getIfIsNearH` que recibe un parámetro que representa la distancia a la que debe estar un vértice para considerarse cerca o no de un agujero. La mecánica para esto consiste en generar un círculo cuyo centro es el punto que se quiere analizar y el radio del mismo está dado por el valor que recibió `getIfIsNearH`. Como se cuenta con los datos de los agujeros y ahora se tiene un círculo que representa la zona de proximidad del punto a evaluar, se busca mediante la operación booleana intersección si existe un área que ambas figuras (círculo y agujero) compartan; de ser el caso, entonces existe una proximidad con un orificio. El código que se encarga de este paso está en el Anexo 8.15. Después de cada punto, el resultado obtenido es guardado mediante la instrucción `setNearOfHole` propia de cada elemento `Point2D`.

Los datos recabados hasta este punto pasan a ser procesados por la red neuronal, la cual está conformada por un arreglo de elementos Neurona y Layers. Cada red neuronal debe entrenarse antes de proceder a la aplicación de la misma, para esto la red neuronal se configuró cómo se ilustra en la Figura 4.12.

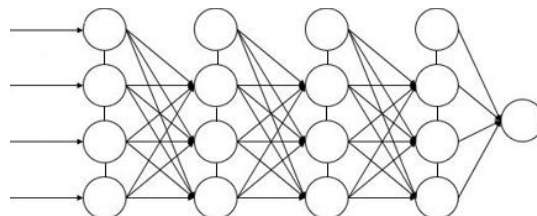


Figura 4.12 Esquema de red neuronal para el proceso de entrenamiento.

Como se puede apreciar en la imagen anterior, la red neuronal empleada puede verse también como un plano de un mapa de auto organizado (un arreglo cuadrado de neuronas), esto debido a la mecánica de red neuronal que se pretende manejar. En un mapa de auto organizado clásico se dispone únicamente de dos capas de neuronas, una para procesar datos de entrada y el plano en donde se genera el ordenamiento de los datos y el resultado (ver Figura 3.12). Sin embargo, esto no es obligatorio y puede ser revertido, teniendo una última capa con una única neurona que genera una solución global promediando los resultados obtenidos con anterioridad en capas pasadas. Esta mecánica de alterar la estructura de un mapa de auto organizado permite desarrollar dos variantes de mapas: las redes de auto organizado multicapa, conocidos también como mapa en multicapa (los empleados para esta tesis) y los mapas de crecimiento jerárquico (*Growing Hierarchical Self-Organizing Map*).

Las bondades del esquema que aquí se implementa (redes de auto organizados multicapa) es que permite el aplicar un entrenamiento competitivo clásico de un entrenamiento no supervisado y a su vez la comprobación mediante heurísticas del resultado obtenido.

Una neurona típicamente se representa con el diseño de la Figura 4.13



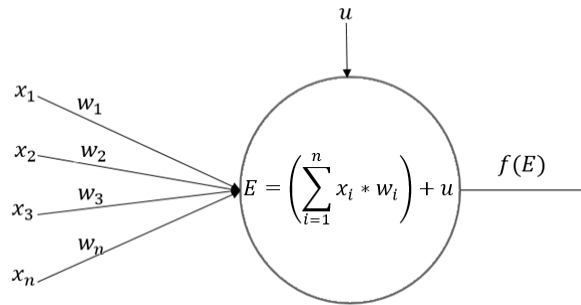


Figura 4.13 Diagrama general de una red neuronal.

Esto se representa computacionalmente como un vector de datos de entrada ( $x_n$ ), un vector de pesos sinápticos  $w_n$ , un ajuste de entrenamiento  $u$  y una función de activación  $f(E)$ , la cual se encarga de generar la salida de la neurona y que será la entrada de otra neurona. Pensando en la naturaleza del problema que aquí se maneja (determinar la tasa de crecimiento de los elementos y con ello la cantidad de los mismos cerca de un punto) el resultado obtenido puede ser un número real, por lo que se requiere que la función de activación permita el manejo de los mismos sin restricción alguna y por esto se emplea la función de activación tangente hiperbólica (mostrada en la siguiente ecuación) que permite generar números en este rango

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (4.9)$$

Para facilitar el proceso de entrenamiento así como validar los resultados adquiridos hasta el momento, se realizó una corrida de la misma red neuronal en un ambiente de desarrollo llamado NeuroPH de distribución gratuita y desarrollado en el lenguaje Java; sin embargo, los resultados de recrear la red neuronal en este entorno pueden ser fácilmente comparados e importados al modelo que se desarrolló en C++.

La reconstrucción de la red neuronal a red neuronal en la herramienta mencionada anteriormente se puede apreciar en la Figura 4.14.

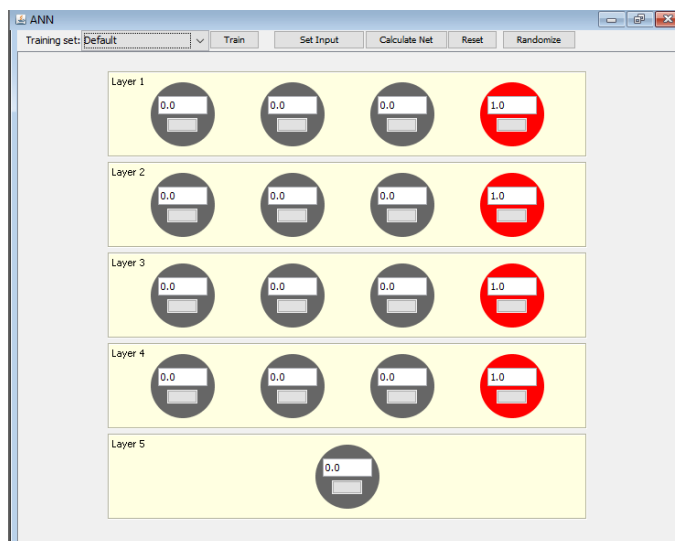


Figura 4.14 Recreación de la red neuronal en Neuroph.

El resultado del proceso de entrenamiento ha dado como resultado un margen del 0.0099 por ciento de error en clasificación al paso de 10431 iteraciones. Este error representa la dispersión que existe entre los posibles resultados (tasas de crecimiento) que se pueden generar de presentarse cada una de las combinaciones mostradas en la Tabla 4.1. La siguiente imagen muestra las estadísticas en del entrenamiento en la herramienta Neuroph.

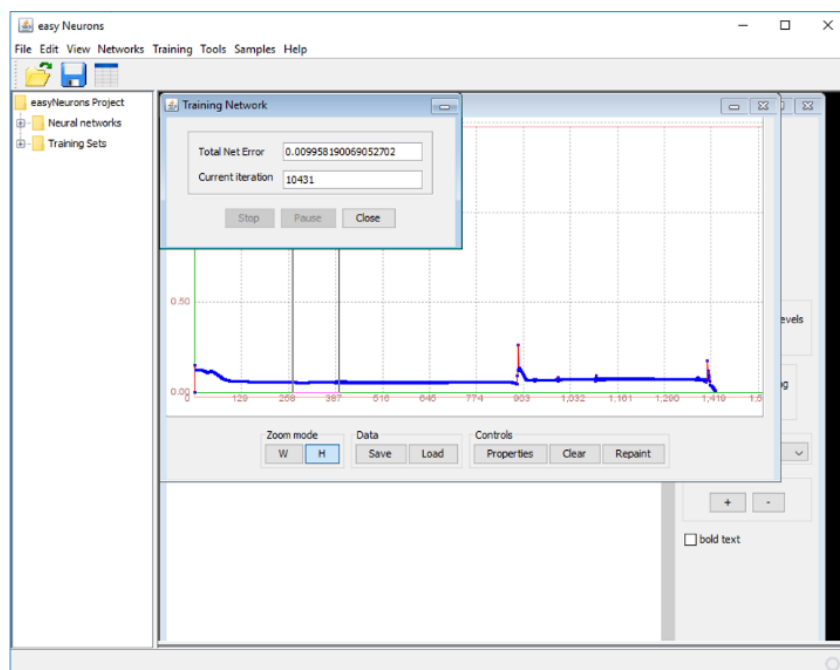
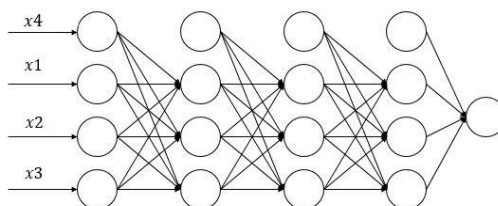


Figura 4.15 Ventana de entrenamiento de la red neuronal.

Para realizar el entrenamiento y determinar el porcentaje de error en el resultado de la red neuronal, en cada iteración del entrenamiento se evalúa si con la configuración de pesos sinápticos de dicha iteración se logra obtener un resultado que cumpla con el nivel de importancia, esto de acuerdo a cada una de las ocho clasificaciones mostradas en párrafos anteriores (Tabla 4.1. Diferencias de importancia entre combinaciones de criterios de interés.). Si con los valores de una iteración los resultados obtenidos por la red alcanzan a generar una tasa de crecimiento que satisface lo anterior, se considera entonces que los parámetros (los pesos sinápticos) de esa iteración son exitosos; los pesos sinápticos son guardados en un archivo de registro y son implementados en la siguiente ronda de entrenamiento como datos de partida a manera de buscar el “patrón” en los datos de entrada con base en un resultado positivo previo. Esto provoca que los pesos sinápticos se “afinen” de modo que en próximas rondas de entrenamiento se ajusten dichos pesos al valor que deben tener para generar una tasa de crecimiento correcta. El porcentaje de error es determinado por la dispersión que existe entre los resultados de cada caso de clasificación, es decir, se busca que la dispersión entre las tasas de crecimiento (dato de salida de la red) sea pareja entre los posibles casos (datos de entrada de la red) que la red neuronal evalúa, y mantenga una coherencia; por ejemplo, si en caso de tener el menor nivel de importancia se obtiene una tasa de crecimiento de 1.5 y para el caso de mayor relevancia una tasa de .1, los niveles de importancia intermedios deben tener valores entre estos límites manteniendo un crecimiento parejo y gradual entre cada nivel de importancia.

Posterior al proceso de entrenamiento la estructura de la red neuronal resultante es la que se muestra a continuación (ver Figura 4.16):



*Figura 4.16 Red neuronal resultante. x1 corresponde a la existencia de un cambio en las condiciones de frontera, x2 corresponde a los picos sobre el contorno, x3 representa la proximidad con algún agujero u orificio, x4 ajuste de datos(rangos de tasa de crecimiento) .*

Los resultados que corresponden a los pesos sinápticos para las capas intermedias (o mejor conocidas como ocultas) y la capa de salida se muestra en la Tabla 4.2.

*Tabla 4.2 Pesos sinápticos resultantes*

| Capa oculta | Neuronas | x 1    | x 2     | x3     | x4     |
|-------------|----------|--------|---------|--------|--------|
| 1           | 1        | 5.438  | -1.912  | -2.168 | 1.737  |
|             | 2        | 2.261  | -1.236  | -0.931 | -1.805 |
|             | 3        | 0.5736 | -0.280  | 0.944  | 0.141  |
| 2           | 4        | 0.827  | 1.136   | 0.815  | 0.110  |
|             | 5        | 4.606  | 2.569   | -0.325 | 0.303  |
|             | 6        | -0.461 | -1.038  | 0.247  | 0.238  |
| 3           | 7        | -0.841 | -3.349  | 0.405  | -0.458 |
|             | 8        | 0.286  | -2.2503 | -0.311 | -1.220 |
|             | 9        | 1.403  | 0.034   | -1.095 | 1.139  |
| 4           | 10       | 0.633  | 0.447   | 1.536  | 0.392  |

Con estos resultados es posible entonces reconstruir la misma red neuronal en el entorno de desarrollo Embarcadero y mandar a llamar la red neuronal mediante la función `analysePoints` presente en Polygon (ver Figura 4.4 Clase Polygon).

---

#### **4.7 Implementación.**

Debido a que el mallado es por sí solo una parte independiente de proceso del MEF cabe mencionar que la implementación directa de la misma se llevará a cabo en un software de simulación que está siendo desarrollado por el CIMAV y se espera que en un futuro dicho software de simulación sea lanzado al público.

Con los tres módulos listos e implementados se pasa a realizar una serie de pruebas mediante las cuales se verifica la calidad del resultado obtenido, sin embargo, esto será mencionado a mayor detalle en el próximo capítulo.

## CAPÍTULO V. RESULTADOS Y DISCUSIÓN

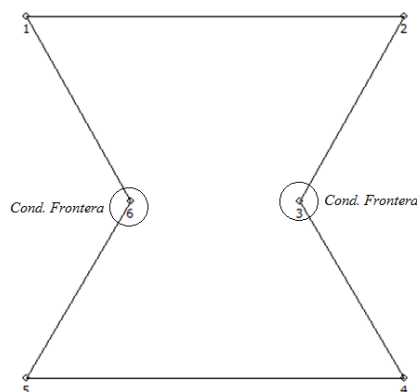
En esta sección se presentan unas series de pruebas realizadas a la librería para probar su desempeño. La primera sección de pruebas tiene como meta determinar el rendimiento de la herramienta en cuanto a tiempo de procesamiento y cantidad de memoria requerida para su función. La segunda sección hace referencia a la implementación de la librería para solucionar un problema del tipo MEF. Todas las pruebas fueron realizadas en un equipo de cómputo con las siguientes especificaciones.

*Tabla 5.1. Especificaciones del equipo empleado para las pruebas*

|                          |  |
|--------------------------|--|
| <b>Disco duro</b>        | 1TB a 5400 rpm                             |
| <b>Procesador</b>        | Intel® Core™ i7-6700HQ (6M Cache, 2.6 GHz) |
| <b>Tarjeta de video</b>  | NVIDIA® GeForce® GTX™ 960M 4GB             |
| <b>Sistema operativo</b> | Windows 10 Home 64                         |
| <b>Memoria</b>           | 8.0GB PC4-17000 DDR4 2133 MHz              |

### 5.1 Pruebas de rendimiento

La prueba comparativa de rendimiento (procesamiento y memoria) fue probada sobre la siguiente geometría (ver Figura 5.1), aplicando sobre el dominio diferentes tipos de malla que a continuación se describen:



*Figura 5.1 Geometría para prueba de tiempo.*

- Malla creada por la librería Triangle sin refinar (ver Figura 5.2 inciso a).

- Malla aplicando los criterios de la red neuronal (ver Figura 5.2 inciso b.1).
- Malla después de un refinamiento utilizando como base el punto anterior (ver Figura 5.2 inciso b.2), y aplicando criterios de optimización de la librería de triangulación (Triangle).
- Malla genérica y homogénea creada en el software COMSOL Multiphysics (ver Figura 5.2 inciso c).

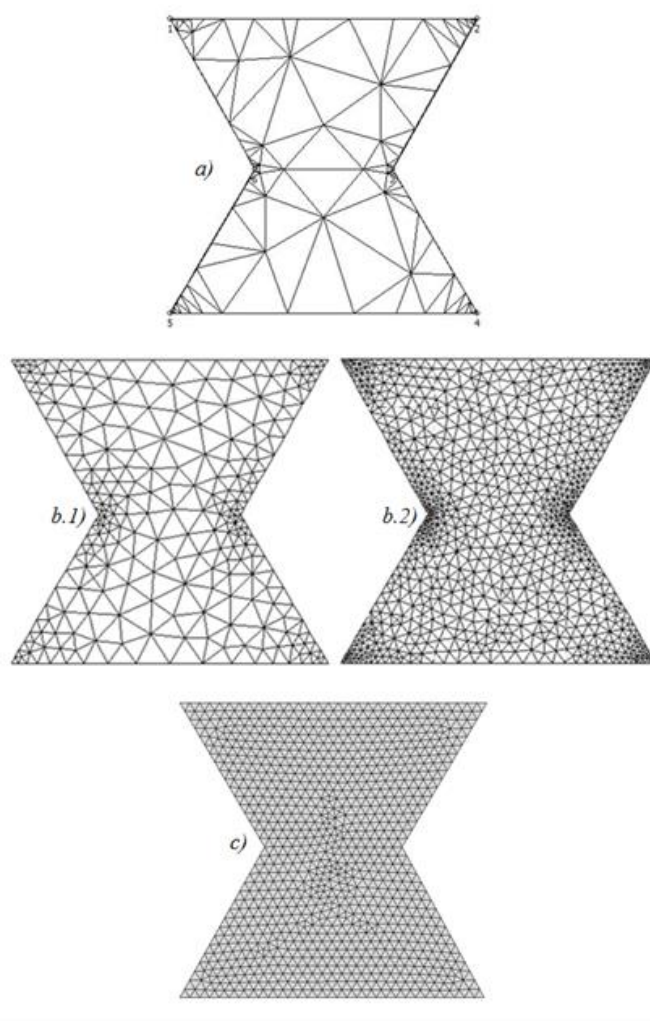


Figura 5.2 Distintos tipos de mallado aplicados al mismo dominio.

A continuación se muestra en la siguiente tabla una comparativa entre el número de elementos, el tiempo de procesamiento y la memoria requerida para generar cada uno de los mallados anteriormente enunciados.

*Tabla 5.2. Comparativa entre número de elementos, tiempo de procesamiento y uso de memoria.*

| <b>Caso</b> | <b>Número de elementos</b> | <b>Memoria aplicada (MB)</b> | <b>Tiempo de procesamiento (segundos)</b> |
|-------------|----------------------------|------------------------------|---|
| a           | 243                        | 1.8                          | 0.125                                     |
| b.1         | 453                        | 2.2                          | 0.282                                     |
| b.2         | 1749                       | 3.1                          | 0.301                                     |
| c           | 1736                       | 7                            | *   |

*\*No se pudo determinar la lectura de tiempo con precisión, por lo cual se deja fuera del estudio.*

Claramente se puede apreciar en estos resultados una tendencia natural presente en todo software, en donde, a más trabajo (a mayor número de elementos) mayor es el esfuerzo realizado (mayor es también la cantidad de memoria y tiempo de procesamiento empleado para generar la malla). También es importante mencionar que la malla genérica creada con COMSOL requirió de una mayor cantidad de memoria en comparación con aquel mallado generado en la librería, bajo las mismas condiciones en similitud de elementos (b.2).

Adicionalmente se recrearon pruebas en donde se desarrollaron diferentes geometrías para comprobar la correcta lectura de los datos de entrada implementado las rutinas explicadas en los párrafos anteriores. En la Figura 5.3 se puede apreciar un conjunto de figuras a las cuales se les han impuesto condiciones que la red neuronal debe detectar para realizar un mallado adecuado según el caso que corresponda. En esta evaluación se pretende mostrar cómo la librería produce un resultado acorde a cada geometría que se le presente.



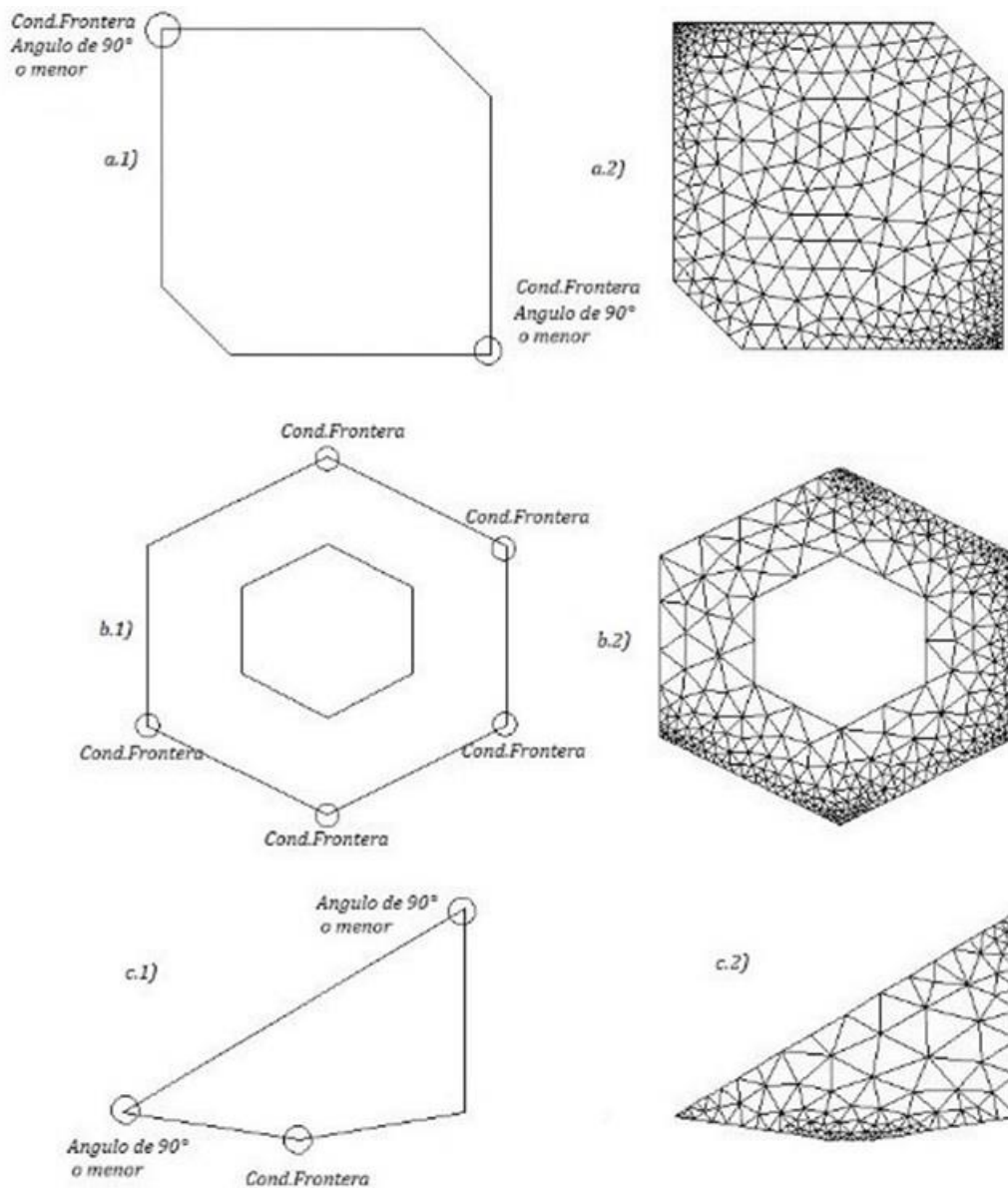
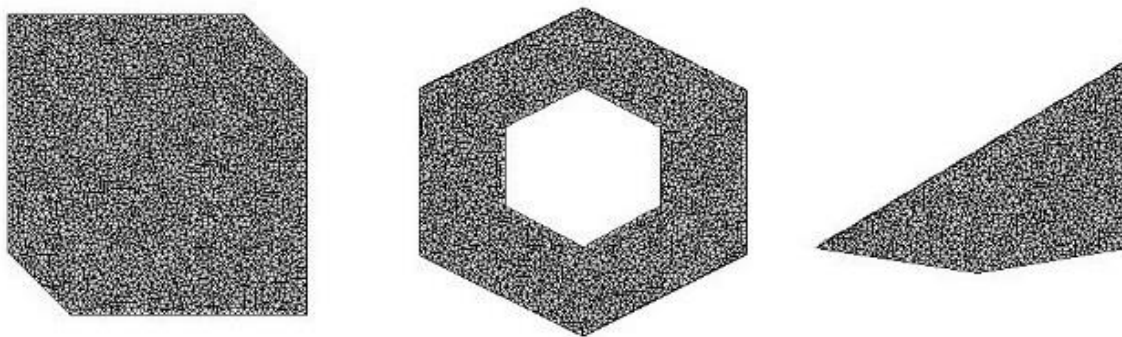


Figura 5.3 Juego de geometrias con condiciones.

En la imagen anterior se puede observar en la columna izquierda de figuras los dominios originales y el planteamiento propuesto para cada uno, en la columna derecha el resultado la librería. Se observa que en las zonas cercanas a los puntos donde se presentó un criterio de interés existe una mayor concentración de elementos, por otro lado, los elementos se acomodaron según los criterios presentados en cada dominio.

Si se deseara crear una malla homogénea que concentrara la misma cantidad de elementos que los resultados presentados en la Figura 5.3 en las zonas de interés, sería necesario refinar todos los elementos hasta tal punto que tuvieran la misma área promedio que presentan los elementos aledaños a los puntos de interés. La siguiente imagen ejemplifica situación en los mismos tres dominios presentados en la Figura 5.4.



*Figura 5.4 Malla homogénea refinada.*

## 5.2 Prueba de experimentación

Para evaluar la calidad del mallado desde el punto de vista del cálculo por elementos finitos se propuso una probeta plana en un experimento de mecánica de fractura, en donde se calculó la energía de fractura, contemplando como única variable el mallado, de acuerdo al siguiente escenario:

Se considera que el dominio tiene un espesor de  $0.01\text{ m}$ ,  $0.1\text{ m}$  de alto y de largo, además, la geometría está hecha de acero estructural, el cual presenta un módulo de Young de  $200\text{ GPa}$  y un coeficiente de Poisson de  $0.33$ .

Se desea evaluar la energía de fractura, considerando que en el estado inicial el dominio tiene una grieta de  $0.04$  metros de largo, la cual crece a partir del borde izquierdo, justo a la mitad de la probeta, existiendo un esfuerzo de tensión de  $150e^6\text{ N/m}^2$  que jala al dominio por la parte superior e inferior del mismo (ver Figura 5.5).

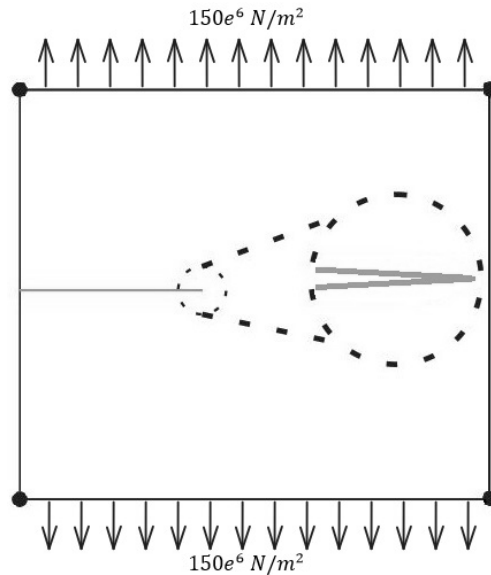


Figura 5.5 Modelo de probeta original.

Para calcular la energía de fractura se prevé un crecimiento infinitesimal de la grieta ( $\Delta l = 10\mu m$ ), para con ello poder calcular la energía de fractura  $E_f$  a partir de las energías de deformación ( $E_1$ ) antes y después del crecimiento de grieta ( $E_2$ ). La energía de deformación está dada por la integral del producto del tensor de tensión ( $\sigma$ ) y el tensor de deformación ( $\varepsilon$ ), típicamente se representa con la letra  $w$ , definido entonces como

$$w = \int_{\Omega} (\bar{\sigma} : \bar{\varepsilon}) d\Omega \quad (5.1)$$

Con la energía de deformación calculada es entonces es posible conocer el valor de la energía de fractura mediante la siguiente ecuación:

$$E_f = \frac{E_1 - E_2}{\Delta l * 1cm} \quad (5.2)$$

Haciendo un análisis sobre la geometría y las cargas se puede considerar la existencia de simetría en el eje y dentro del dominio, lo que hace posible entonces realizar la siguiente simplificación (ver Figura 5.6)

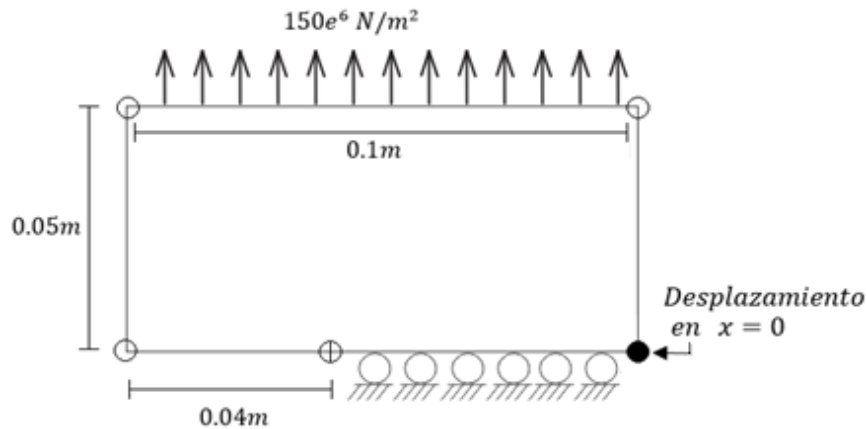


Figura 5.6 Modelo de probeta para experimento.

Con el modelo planteado anteriormente se procede a realizar el experimento que permite evaluar el nivel de eficiencia de una malla generada por la librería. Para ello la mecánica empleada consintió en importar al software COMSOL una serie de mallas realizadas por la librería a fin de evaluar en dicho software el resultado del modelo planteado. A continuación se enumeran las condiciones de mallado con las cuales se realizaron las pruebas:

- Un caso en donde se aplica una malla genérica realizada por las configuraciones automáticas de COMSOL.
- Un caso donde se evalúa por parte de la red neuronal el punto que representa la grieta exclusivamente.
- Por último se aplica una malla donde la librería evalúa en cada punto del dominio todos los criterios de interés que la red neuronal puede analizar.

Cabe señalar que para comparar el resultado y determinar la veracidad de los resultados, es primeramente necesario conocer el resultado (“valor real” o aproximación más cercana) de la energía de fractura en el experimento. Para ello, empleando el mismo software COMSOL se genera un mallado extremadamente fino (de 137846 elementos), con un polinomio de interpolación de cuarto grado y se procede a realizar el estudio, el resultado obtenido para la

energía de fractura fue de  $33865 \text{ J/m}^2$ . La malla empleada para llegar a este resultado garantiza que el resultado del MEF sea lo más preciso posible (tener una malla con una gran cantidad de elementos y un polinomio de interpolación elevado favorece a la convergencia del MEF). Por el número de elementos y lo elevado del polinomio este experimento requirió de un equipo de cómputo con características de Workstation, la Tabla 5 expone las especificaciones de dicho equipo de cómputo.

*Tabla 5.3 Especificaciones de Workstation*

|                         |                                |
|-------------------------|--------------------------------|
| <b>Disco duro</b>       | 1TB a 7200 rpm                 |
| <b>Procesador</b>       | Intel® Core™ Xeon (3.5 GHz)    |
| <b>Tarjeta de video</b> | NVIDIA® GeForce® GTX™ 770M 4GB |
| <b>Memoria</b>          | 64 GB DDR3 1866 MHz            |

Una vez que se tiene el resultado contra el cual comparar los valores de la energía de fractura para cada mallado planteado, se recreó el dominio en la librería y se realizaron los tres casos mencionados anteriormente. Después de esto, las mallas fueron aplicadas mediante herramientas de importación a COMSOL y se resolvió el mismo modelo para cada caso, siendo la única variante para cada caso el mallado empleado para resolver el modelo. La Figura 5.7 muestra los tres tipos de malla que se emplearon, *a)* el mallado genérico creado por COMSOL, *b)* el mallado por parte de la librería considerando sólo el punto de la fractura y *c)* el mallado de la librería pero considerando todos los criterios sobre todos los puntos del dominio.

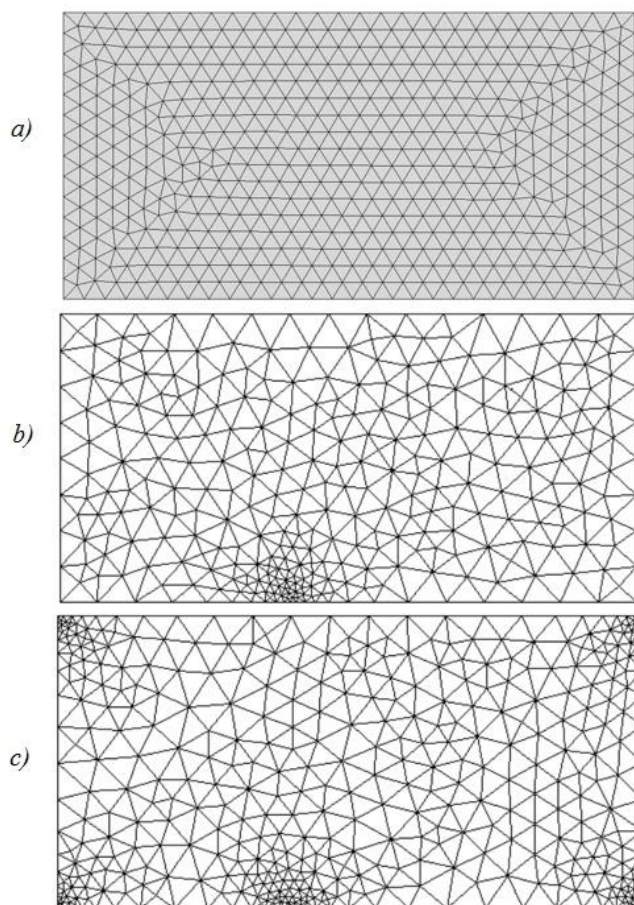


Figura 5.7 Tipos e mallado para la prueba

Los resultados obtenidos se pueden ver en la siguiente tabla, en la cual se expone el porcentaje de error obtenido contra el valor más preciso ( $33865 \text{ J/m}^2$ ).

Tabla 5.4 Porcentajes de error obtenidos.

| Número de elementos | Porcentaje de error |                             |                                  |
|---------------------|---------------------|-----------------------------|----------------------------------|
|                     | Mallado genérico    | Considerando solo la grieta | Considerando todos los criterios |
| 172                 | 31.2062602          | 4.919533442                 | 5.5366898                        |
| 346                 | 4.54746789          | 3.006053448                 | 1.6920124                        |
| 1060                | 0.69393179          | 1.768787834                 | 1.34652296                       |
| 4068                | 0.89177617          | 0.085634136                 | 0.10039864                       |



Estos valores se muestran en la siguiente figura para facilitar su interpretación:

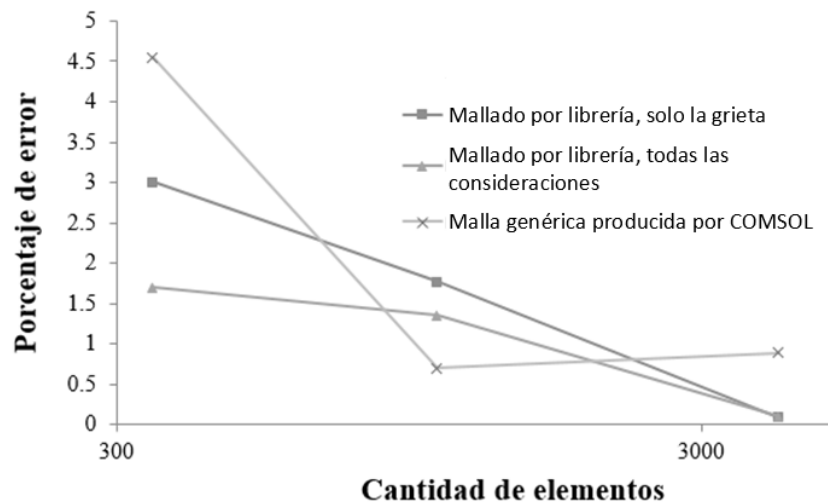


Figura 5.8 Gráfica de resultados en prueba de mecánica de fractura.

En la gráfica anterior se puede apreciar que las mallas provenientes de la librería tienen un porcentaje de error menor, aún y cuando emplean casi la misma cantidad de elementos, siendo la diferencia radical la ubicación de los elementos. Esto demuestra que el acomodo de los elementos repercute directamente en el resultado obtenido y que es posible disminuir la cantidad de elementos necesarios para lograr una buena aproximación. Evaluando los resultados obtenidos por la malla homogénea en el experimento, se aprecia que al elevar la cantidad de elementos el porcentaje de error disminuye, sin embargo, es claro que debido a su funcionamiento se necesitan más elementos que en las mallas generadas por la librería para alcanzar el mismo resultado que el obtenido por estas últimas con menos elementos.

Después de verificar en el experimento anterior la convergencia de cada uno de los casos, se presenta un nuevo experimento en el cual se emplea el mismo dominio y modelo presentado anteriormente. La intención de este segundo experimento es determinar la relevancia y el comportamiento que presenta la librería con respecto a los elementos internos del dominio que están alejados de las zonas críticas. A continuación se enlistan los casos de estudio para el experimento:



- Un mallado creado por la librería donde se evalúa cada nodo del dominio, el crecimiento entre los elementos en el interior del dominio (lejos del contorno) está configurado para ser de 1 a 10 veces más pequeño o grande (1/10).
- Un mallado con el mismo planteamiento que en el punto anterior, sólo que en esta ocasión el crecimiento es de 1/5.

De manera paulatina se fue refinando cada uno de los dos casos de experimentación y se realizó una lectura de la energía de fractura calculada en el modelo. Esto se hizo con la intención de tener un conjunto de datos para cada caso de experimentación y poder determinar el comportamiento que cada uno lleva. En la siguiente tabla se muestra la cantidad de elementos y la energía de fractura calculada en cada caso.

*Tabla 5.5. Lectura de la energía de fractura para segundo experimento.*

| Caso de crecimiento 1/5 |                                | Caso de crecimiento 1/10 |                                |
|-------------------------|--------------------------------|--------------------------|--------------------------------|
| Número de elementos     | Energía de fractura( $J/m^2$ ) | Número de elementos      | Energía de fractura( $J/m^2$ ) |
| 224                     | 34383                          | 286                      | 32097                          |
| 644                     | 33800                          | 706                      | 32812                          |
| 2222                    | 33520                          | 2268                     | 31240                          |
| 8366                    | 33929                          | 8418                     | 33340                          |
| 33153                   | 33560                          | 33508                    | 33896                          |

En la Figura 5.9 muestra la gráfica de los resultados obtenidos según su porcentaje de error en contraste con el valor de referencia ( $33865 J/m^2$ ).

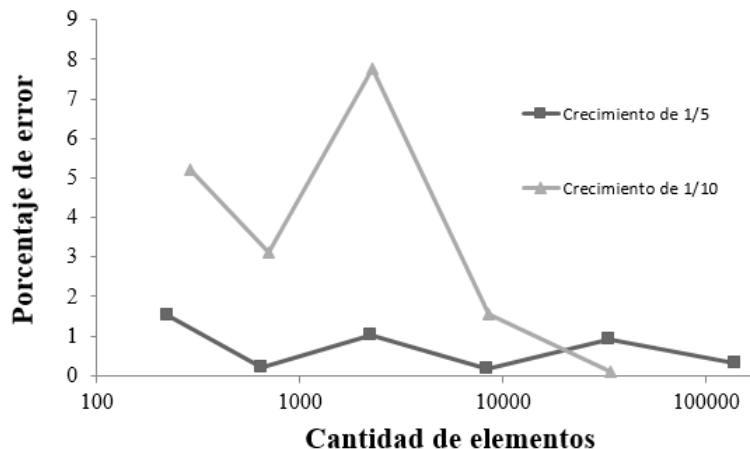


Figura 5.10 Velocidad de convergencia en resultados.

En la imagen anterior se alcanza a ver que un crecimiento de elementos en proporciones elevadas retarda la convergencia del resultado, requiriendo entonces más elementos para alcanzar un resultado correcto, en contraste, el uso de elementos que tiene un tamaño que concuerda con la teoría presentada en el marco teórico (de 1.5 a 2 veces crecimiento o decremento en el área de los elementos) produce una convergencia en el método con una menor cantidad de elementos.

## CAPÍTULO VI. CONCLUSIONES

En este trabajo se ha realizado un análisis sobre la elaboración de un proceso de mallado óptimo. Dicho mallado se estudió con el fin de que fuera el adecuado para ser implementado en el MEF. Como se mencionó en los capítulos primero y tercero, existen diferentes métodos para realizar dicha tarea, sin embargo, hay consideraciones que pueden mejorar la aportación o desempeño que tiene la malla en el MEF; de las cuales se pueden recalcar: el generar elementos con determinadas propiedades y el acomodo de los elementos. Para crear una herramienta de software capaz de crear una malla adecuada para el MEF, lo cual es el objetivo de este trabajo, se optó por implementar una mecánica de triangulación para generar elementos con propiedades adecuadas para el MEF y se creó una red neuronal que determina la manera en que se concentran los elementos sobre el dominio estudiado.

El análisis de la funcionalidad y del desarrollo de esta librería de mallado condujo a buscar satisfacer cada objetivo específico planteado, y en conjunto, el objetivo general: crear una herramienta de software libre capaz de realizar un mallado optimizado sobre figuras 2D, valiéndose de una red neuronal para lograr dicha optimización. Con base en las pruebas realizadas y las capturas de la herramienta mostradas se concluyeron los siguientes resultados para cada objetivo específico:

- *Desarrollar la herramienta de dibujo:* La interface de dibujo es capaz de realizar diversos trazos, de importar y exportar figuras geométricas además de permitir al usuario la operación de las variables sobre las cuales desea se realice el mallado. Por lo anterior, se considera que este objetivo se cumplió satisfactoriamente a excepción del lado estético, ya que es una interface burda.
- *Implementar la triangulación Delaunay:* Coincidiendo que el implementar software que está probado conlleva el ahorro de tiempo, y viendo la complejidad que es el programar la triangulación Delaunay sin base alguna, es un acierto el implementar una librería ya existente que realizara esta labor. Sin embargo fue preciso realizar las adecuaciones necesarias como nuevos tipos de dato, rutinas de código, modelado de

clases, entre otros, para lograr una correcta implementación y fácil manejo de la misma. Las pruebas de rendimiento demuestran que la librería crea elementos según lo propuesto en el marco teórico, por lo que el objetivo se cumplió correctamente.

- *Crear un módulo de análisis a figuras 2D por medio de redes neuronales:* Se logró encontrar un modelo de red neuronal que satisface la serie de consideraciones que se plantearon para la misma, tales como que fuera de entrenamiento no supervisado, y que permitiera la implementación de heurísticas. Sumado a esto, se realizaron las rutinas de código necesarias para poder extraer la información de la geometría a mallar. Con base en los resultados obtenidos en el proceso de prueba, se considera que estas rutinas como la red neuronal funciona correctamente.
- *Definir entrenamiento de la red neuronal:* Se logró definir de manera clara los criterios que la red neuronal debe evaluar, así como la estrategia que debe emplear para lograr su objetivo. Para esto último, se creó un listado de posibles casos a los cuales la red neuronal se podría enfrentar, logrando crear un resultado coherente después de un tiempo de entrenamiento.
- *Validar la eficiencia de la malla:* Las pruebas presentadas a la librería fueron fundamentales para garantizar que la misma fuera confiable. Particularmente las pruebas experimentales de la mecánica de fractura ayudaron a determinar en su momento fallos en la herramienta que fueron corregidos. Después de esto, los resultados alcanzados coinciden claramente con lo presentado en el marco teórico, en donde se comentó que es posible mejorar la calidad de la aproximación del MEF si existe una mayor concentración de elementos en zonas donde exista un alto gradientes de los valores de la solución numérica (desplazamientos y esfuerzos en un problema de mecánica).

Considerando los resultados de cada objetivo específico y el material presentado en el capítulo anterior, se determina que la librería cumple con su objetivo general de manera adecuada. Cabe destacar, que en este aspecto la librería no se presenta como una solución específica para un caso o fenómeno en particular, por lo cual, es claro que los criterios que evalúa la red en algunos casos no satisfaga la necesidad en su totalidad para un caso en

especial, sin embargo, puede utilizarse el planteamiento presentado en este documento para desarrollar una herramienta que evalúe los criterios que se deseen. Existe una amplia posibilidad, de que un software que aplique la misma mecánica para realizar el mallado pero que evalúe datos específicos de un fenómeno en particular tenga un mejor desempeño que la librería aquí presentada; sin embargo, el resultado alcanzado en este trabajo atacan de manera general el MEF por lo cual puede ser utilizado en toda aplicación del mismo, lo que le da mayor alcance en cuanto a la aplicación, lo que es una ventaja en contraste a atacar un problema particular.

En cuanto a mejoras a la librería presentada actualmente, se ha considerado el implementar ajustes que le permita a la red dejar de evaluar determinados criterios de interés o que alteren la importancia que tienen los criterios ya existentes, esto le daría a la librería una mayor capacidad de adaptación a problemas específicos, sin embargo, es necesario evaluar qué características de la red neuronal (o que modelos de red) permiten crear un entrenamiento que acepte este tipo de variantes sin recaer en un problema de entrenamiento.

Adicionalmente se desea que la librería forme parte de un software mayor que realice el proceso completo de simulación mediante el MEF. Dicho software maneja la misma filosofía de software libre que la librería, adicionalmente para este caso será necesario crear una interfaz con una mejor presentación estética para facilitar al usuario el manejo del módulo de mallado (la librería), siendo esta implementación su aplicación o implementación final en el ciclo de vida del desarrollo del software.

## VII. BIBLIOGRAFÍA

- A. Bahreininejad, B. H. (1997). *Finite element mesh partitioning using neural networks*. Edimburgo: Elsevier.
- Ahn, C.-H. (1991). *GENERATION, A SELF-ORGANIZING NEURAL NETWORK APPROACH FOR AUTOMATIC MESH*. Seoul: IEEE TRANSACTIONS ON MAGNETICS, VOL. 27.
- Alfonzetti, S. (1998 ). *A Finite Element Mesh Generator based on an Adaptive Neural Network* . Catania.
- Anna Paszynska, M. P. (2008). *Graph Transformations for Modeling hp-Adaptive Finite Element Method with Triangular Elements*. Berlin: Springer.
- Chew, L. P. (1989). *Constrained Delaunay Triangulations*. New York: Algorithmica.
- D. N. Dyck, D. A. (1992). *DETERMINING AN APPROXIMATE FINITE ELEMENT MESH* . Montreal: IEEE TRANSACTIONS ON MAGNETICS, VOL. 28.
- Dolska, B. (2002). *Finite element mesh desing expert system*. Marivor: Elsevier.
- E. Kang, K. H. (1995). *Intelligent Finite Element Mesh Generation* . West Lafayette: Engineering with Computers.
- Fauseet, L. (1994). *Fundamentals of neural networks. Architectures, algorithms, and aplicaciones*. Upper Saddle River, New Jersey: Prentice Hall.
- Gouri Dhatt, G. T. (2012). *Finite Element Method*. Hoboken: ISTE Ltd and Jhon Willey & Sons.
- H. Jilani, A. B. (2009). *Adaptive finite element mesh triangulation using self-organizing neural networks*. Tehran: Elsevier.
- K. Srasuay, A. C. (2009). *Mesh generation of FEM by ANN on iron*. Phitsanulok: Naresuan University.
- K.H., L. (1988). *Finite element mesh generation methods: a review and classification* . Elsevier.
- Kohonen, T. (1982). Self-Organized Formation of Topologically Correct Feature Maps. *Department of Technical Physics, Helsinki University of Technology*.
- Kriesel, D. (2005). *A Brief Introduction to Neuronal Networks*. Bonn.
- Labridis, D. G. (2002). *A Finite-Element Mesh Generator Based on Growing*. Greece: IEEE TRANSACTIONS ON NEURAL NETWORKS,VOL. 13.

- 
- Larry Manevitz, M. Y. (1997). *Finite-Element Mesh Generation Using Self-Organizing Neural Networks*. Malden: Blackwell Publishers.
- Moaveni, S. (1990). *Finite element analysis*. Prentice hall.
- Pascal Jean Frey, P.-L. G. (2000). *Mesh generation: Application to finite elements*. Hoboken: Wiley.
- R. Chedid, N. N. (1996). *Automatic Finite-Element Mesh Generation Using Artificial Neural Networks*. New York: IEEE TRANSACTIONS ON MAGNETICS.
- Ruppert, J. (1995). A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. *Journal of Algorithms*, 548-585.
- S.Alfonzetti, S. (1988). *Elfin:an n-dimensional finite-element codefor the computation of electromagnetic fields*. Catania: Universita di Catania.
- S.Alfonzetti, S. S. (1996). *Automatic Mesh Generation by the Let-It-Grow Neural Network*. Catania: Universita di Catania.
- S.H. Lo, D. (2015). *Finite Element Mesh Generation*. New York: CRC Press.
- Shewchuk, J. R. (1997). Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. *Applied Computational Geometry: Towards Geometric Engineering*.
- Siu-Wing Cheng, T. K. (2012). *Delaunay Mesh Generation*. New York: CRC Press.
- Stuart J. Russell, P. N. (1995). *Artificial Intelligence A Modern Approach*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.
- T. Nishizeki, N. (1991). *Planar Graphs:Theory and Algorithms*. New York: Elsevier.

## VIII. ANEXOS

### 8.1 Exportar datos de dibujo.

```
void __fastcall TMainForm::btnExportarClick(TObject *Sender)
{
    if(FileOpenDialog1->Execute()){
        sgPoints->RowCount=1;
        std::ifstream file;
        file.open(FileOpenDialog1->FileName.c_str());
        std::string line;
        int c=1;
        while(std::getline(file, line)){

            if (*line.rbegin() == '\r') {
                line.erase(line.length() - 1);
            }
            if(line.empty() || line.size() < 2) {
                continue;
            }
            String param="";
            for(int i=0,j=0;i<line.size() && j<2; i++){
                if(line[i]!=' ' && line[i]!='\t' )
                    param += line[i];
                if ((i==line.size()-1) || line[i]==' ' || line[i]=='\t' || line[i]=='\n') {
                    sgPoints->Cells[j+1][c-1]=param;
                    j++;
                    param="";
                    if (j==2){
                        sgPoints->Cells[0][c-1]=IntToStr(c);
                        c++;
                        sgPoints->RowCount=c;
                    }
                }
            }
            MakeDPoints();
            DrawPoints();
        }
    }
}
```



## 8.2 Crear elementos Point2D para área de dibujo.

```
void TMainForm::MakeDPoints() {
p_draw.clear();
    for (int i = 0; i < sgPoints->RowCount; i++) {
        if (sgPoints->Cells[1][i] != "" && sgPoints->Cells[2][i] != "") {
            int id = StrToInt(sgPoints->Cells[0][i]);
            float X = StrToFloat(sgPoints->Cells[1][i]);
            float Y = StrToFloat(sgPoints->Cells[2][i]);
            bool hole=false;
            bool cf_p=false;
            int cf_id=0;
            float a=0.0;
            if(sgPoints->Cells[3][i]!="1"){
                hole=true;
            }

            if(sgPoints->Cells[4][i]!="1"){
                a=3.14159/2.0;
            }else{a=3.14159;}
            if(sgPoints->Cells[5][i]!="1"){
                cf_p=true;
                cf_id= StrToInt( sgPoints->Cells[4][i]);
            }
            p_draw.push_back(new Point2D(id, X, Y));
            p_draw.at(p_draw.size()-1)->setHoleNear(hole);
            p_draw.at(p_draw.size()-1)->setCfId(cf_id);
            p_draw.at(p_draw.size()-1)->setCfP(cf_p);
            p_draw.at(p_draw.size()-1)->setAng(a);
        }
    }
}
```

## 8.3 Realizar trazos de segmentos en área de dibujo

```
void TMainForm::DrawSegments(std::vector<Segment*>segmentos) {
    for (size_t i = 0; i < segmentos.size(); i++) {
        DrawingBoard->Canvas->MoveTo(segmentos[i]->getP1()->getX(),
            segmentos[i]->getP1()->getY());
        DrawingBoard->Canvas->LineTo(segmentos[i]->getP2()->getX(),
            segmentos[i]->getP2()->getY());
        PaintBox1->Canvas->Draw(0, 0, DrawingBoard);
    }
}
```

## 8.4 Generar puntos en área de dibujo.

```
void TMainForm::DrawPoints() {  
    initializePaintBox();  
    for (size_t i = 0; i < p_draw.size(); i++) {  
        int X = p_draw[i]->getX();  
        int Y = p_draw[i]->getY();  
        if(cbPoint->Checked){  
            DrawingBoard->Canvas->Ellipse(X - 3, Y - 3, X + 3, Y + 3);  
            DrawingBoard->Canvas->TextOutA(X - 3, Y + 4, IntToStr(p_draw[i]->getId()));  
        }  
    }  
    PaintBox1->Canvas->Draw(0, 0, DrawingBoard);  
}
```

## 8.5 Crear elementos Segment para área de dibujo.

```
void TMainForm::MakeDSegments() {
    std::vector<Segment*>aux;
    for (int id = 1, i = 0; i < sgSegments->RowCount; i++) {
        if (sgSegments->Cells[1][i] != "" && sgSegments->Cells[2][i] != "") {

            bool p1f = false;
            bool p2f = false;

            Point2D* p1;
            Point2D* p2;

            bool c_s=false;
            int id_cf=0;

            if(sgSegments->Cells[3][i]!=""){
                c_s=true;
            }
            if(sgSegments->Cells[4][i]!=""){
                id_cf=StrToInt(sgSegments->Cells[4][i]);
            }
            for (size_t j = 0; j < p_draw.size(); j++) {
                if (p_draw[j]->getId() == StrToInt(sgSegments->Cells[1][i])) {
                    p1 = p_draw[j];
                    p1f = true;
                }
                if (p_draw[j]->getId() == StrToInt(sgSegments->Cells[2][i])) {
                    p2 = p_draw[j];
                    p2f = true;
                }
                if (p1f && p2f) {
                    j = p_draw.size();
                }
            }
            if (p1f && p2f) {
                p1->s_list.insert(std::pair<int,int> (id,id));
                p2->s_list.insert(std::pair<int,int> (id,id));
                aux.push_back(new Segment(id, p1, p2));
                aux.at(aux.size()-1)->setCs(c_s);
                aux.at(aux.size()-1)->setCf(id_cf);
                id++;
            }
        }
    }
    polyList.at(cbPolyList->ItemIndex)->setContour(aux);
}
```

## 8.6 Importar datos del mallado.

```
void __fastcall TMainForm::btnImportarClick(TObject *Sender)
{
    if(FileSaveDialog1->Execute()){
        FILE *fp= fopen(((AnsiString)FileSaveDialog1->FileName).c_str(), "w+");
        for (int i = 0; i < out.numberofpoints; i++) {
            fprintf(fp, "d%\t%f\t%f\n", i+ 1, out.pointlist[i * 2], out.pointlist[i * 2 + 1]);
        }
        fclose(fp);
    }

    if(FileSaveDialog1->Execute()){
        FILE *fp= fopen(((AnsiString)FileSaveDialog1->FileName).c_str(), "w+");
        for (int i = 0; i < out.numberoftriangles; i++) {
            fprintf(fp, "d%\t%d\t%d\n",out.trianglelist[i*3], out.trianglelist[(i*3)+1], out.trianglelist[(i*3)+2]);
        }
        fclose(fp);
    }
}
```

## 8.7 Inicialización de entrada de datos para librería de triangulación.

```
triangulateio in;
in.numberofpointattributes=0;
in.pointattributelist=(double*)NULL;
in.numberofedges = 0;
in.numberofregions = 0;
in.numberoftriangleattributes = 0;
in.triangleattributelist==(double*)NULL;
in.pointmarkerlist = (int*)NULL;
in.segmentmarkerlist = (int*)NULL;
```

## 8.8 Rutina onClockWise

```
void onClockWise() {
    for (int j = 0; j < contour.size(); j++) {
        float x1 = contour[j]->getP1()->getX();
        float y1 = contour[j]->getP1()->getY();
        float x2 = contour[j]->getP2()->getX();
        float y2 = contour[j]->getP2()->getY();

        double h = std::sqrt(std::pow(x2 - x1, 2) + std::pow(y2 - y1, 2));
        float dx = (x2 - x1) / h;
        float dy = (y2 - y1) / h;

        double min_len = getDistance
            (contour[j]->getP1()->getX(), contour[j]->getP1()->getY(),
            contour[j]->getP2()->getX(), contour[j]->getP2()->getY());

        for (int i = 0; i < contour.size(); i++) {
            double aux = getDistance
                (contour[i]->getP1()->getX(), contour[i]->getP1()->getY(),
                contour[i]->getP2()->getX(), contour[i]->getP2()->getY());
            min_len = min_len < aux ? min_len : aux;
        }
        float ang = getAngle(x1, y1, x2, y2);
        double pi = 3.14159265358979323846;
        Point2D p_;
        double d = min_len * 0.01;
        double x_i, y_i;
        bool control = false;
        do {
            if (control == false) {
                x_i = std::cos(ang + (pi / 2)) * d + (dx * (h / 2) + x1);
                y_i = std::sin(ang + (pi / 2)) * d + (dy * (h / 2) + y1);
                p_ = Point2D(-2, x_i, y_i);
                control = true;
            }
            else {
                x_i = std::cos(ang + (pi * 1.5)) * d + (dx * (h / 2) + x1);
                y_i = std::sin(ang + (pi * 1.5)) * d + (dy * (h / 2) + y1);
                p_ = Point2D(-2, x_i, y_i);
                control = false;
                d = d * 0.1;
            }
        }
        while (!pointInside(p_)); // punto de referencia
        if (control == true) {
            x_i = std::cos(ang + (pi / 2)) * h + (dx * (h / 2) + x1);
            y_i = std::sin(ang + (pi / 2)) * h + (dy * (h / 2) + y1);
        }
        else {
            x_i = std::cos(ang + (pi * 1.5)) * h + (dx * (h / 2) + x1);
            y_i = std::sin(ang + (pi * 1.5)) * h + (dy * (h / 2) + y1);
        }

        double ang1, ang2;

        ang1 = getAngle(x_i, y_i, x1, y1);
        ang2 = getAngle(x_i, y_i, x2, y2);

        if (x_i < x1 && x_i < x2) {
            float max = y1 > y2 ? y1 : y2;
            if ((y2 < y_i && y_i < y1) || (y1 < y_i && y_i < y2) ||
                (y_i == max)) {
                ang1 = y1 > y2 ? (ang1 + 2 * pi) : ang1;
                ang2 = y2 > y1 ? (ang2 + 2 * pi) : ang2;
            }
        }
        if (ang2 > ang1) {
            Point2D* p_new = new Point2D(p_.id, p_.getX(), p_.getY());

            p_new = contour[j]->getP1();
            contour[j]->setP1(contour[j]->getP2());
            contour[j]->setP2(p_new);
        }
    }
}
```

## 8.9 Rutina pointInside

```
bool pointInside(Point2D p) {
    bool is_inside = false;
    bool point_in_hole = false;
    for (int i = 0; i < holes.size(); i++) {
        if (holes[i]->pointInside(p)) {
            point_in_hole = true;
            i = holes.size();
        }
    }
    if (point_in_hole == false) {
        for (int i = 0; i < contour.size(); i++) {
            if ((contour[i]->getP1()->getY() < p.getY() && contour[i]->getP2()
                ->getY() >= p.getY()) || (contour[i]->getP2()->getY()
                < p.getY() && contour[i]->getP1()->getY() >= p.getY()))
            {
                if (contour[i]->getP1()->getX() +
                    (p.getY() - contour[i]->getP1()->getY()) /
                    (contour[i]->getP2()->getY() - contour[i]->getP1()->getY()) *
                    (contour[i]->getP2()->getX() - contour[i]->getP1()->getX()) < p.getX()) {
                    is_inside = !is_inside;
                }
            }
        }
    }
    return is_inside;
}
```

## 8.10 Rutina linearMesh

```
void linearMesh() {
    meshPoints.clear();
    double l = getDistance(p1->getX(), p1->getY(), p2->getX(), p2->getY());
    double dirX;
    double dirY;
    float x, xn, yn, y, d_r, d;
    double g_r;
    Point2D * aux;
    meshPoints.push_back(p1);
    x = p1->getX();
    y = p1->getY();
    dirX = (p2->getX() - p1->getX()) / l;
    dirY = (p2->getY() - p1->getY()) / l;
    g_r = p1->getGrowthRate();
    g_r = g_r == 0.0 ? 2 : g_r;
    d = (g_r / 100) * p1->small_Length;

    do {
        xn = x + d * dirX;
        yn = y + d * dirY;
        aux = new Point2D(-1, xn, yn);
        meshPoints.push_back(aux);
        x = xn;
        y = yn;
        d_r += d;
        d = d * g_r;
    }
    while ((d + d_r) < (l/2));
    int contador=meshPoints.size();
    x = p2->getX();
    y = p2->getY();
    dirX = (p1->getX() - p2->getX()) / l;
    dirY = (p1->getY() - p2->getY()) / l;
    g_r = p2->getGrowthRate();
    g_r = g_r == 0.0 ? 2 : g_r;
    d = (g_r / 100) * p2->small_Length;
    d_r=0.0;
    do {
        xn = x + d * dirX;
        yn = y + d * dirY;
        aux = new Point2D(-1, xn, yn);
        meshPoints.push_back(aux);
        x = xn;
        y = yn;
        d_r += d;
        d = d * g_r;
    }
    while ((d + d_r) < (l/2));
    std::reverse(meshPoints.begin() + contador, meshPoints.end());
    meshPoints.push_back(p2);
}
};
```

## 8.11 Conversión de datos de polígono a grafo para librería de triangulación.

```
std::vector<Segment*>_s = polyList[i]->getPolySegments();
std::map<int, Point2D*>_p = reDefineId(_s);
std::vector<Point2D>ph;
for (int p = 0; p < polyList[i]->getHoles().size(); p++) {
    ph.push_back(polyList[i]->getHoles().at(p)->getPH());
}
in.numberofsegments = _s.size();
in.segmentlist = (int*)malloc(in.numberofsegments * 2 * sizeof(int));
in.numberofholes = ph.size();
in.holelist = (REAL*)malloc(in.numberofholes * 2 * sizeof(REAL));

out.pointlist = (REAL*)NULL;
out.pointmarkerlist = (int*)NULL;
out.trianglelist = (int*)NULL;
out.segmentlist = (int*)NULL;
out.segmentmarkerlist = (int*)NULL;

for (int j = 0; j < in.numberofsegments; j++) {
    in.segmentlist[j * 2] = _s[j]->getP1()->getId();
    in.segmentlist[j * 2 + 1] = _s[j]->getP2()->getId();
}
for (int p = 0; p < in.numberofholes; p++) {
    in.holelist[p * 2] = ph[p].getX();
    in.holelist[p * 2 + 1] = ph[p].getY();
}
if (polyList[i]->getElements().size() <= 0) {
    in.numberoftriangles = 0;
    in.trianglearealist = (double*)NULL;
    in.numberofpoints = _p.size();
    in.pointlist = (REAL*)malloc(in.numberofpoints * 2 * sizeof(REAL));
    int c = 0;
    for (std::map<int, Point2D*>::iterator it = _p.begin();
         it != _p.end(); ++it, c++) {
        in.pointlist[c * 2] = it->second->getX();
        in.pointlist[c * 2 + 1] = it->second->getY();
    }
    triangulate("pqa15", &in, &out, (triangulateio*)NULL);
}
else {
    in.numberoftriangles = polyList[i]->getElements().size();
    in.trianglelist = (int*)malloc(in.numberoftriangles * 3 * sizeof(int));
    in.numberofcorners = 3;
    in.numberofpoints = polyList[i]->getMeshPoints().size();
    in.pointlist = (REAL*)malloc(in.numberofpoints * 2 * sizeof(REAL));
    for (int x = 0; x < in.numberoftriangles; x++) {
        in.trianglelist[x * 3] = polyList[i]->getElements().at(x)->getS1()->getId();
        in.trianglelist[x * 3 + 1] = polyList[i]->getElements().at(x)->getS2()->getId();
        in.trianglelist[x * 3 + 2] = polyList[i]->getElements().at(x)->getS3()->getId();
    }
    for (int l = 0; l < polyList[i]->getMeshPoints().size(); l++) {
        in.pointlist[l * 2] = polyList[i]->getMeshPoints().at(l)->getX();
        in.pointlist[l * 2 + 1] = polyList[i]->getMeshPoints().at(l)->getY();
    }
    triangulate("Drpa70", &in, &out, (triangulateio*)NULL);
}
std::vector<Point2D*>out_p;
for (int j = 0; j < out.numberofpoints; j++) {
    out_p.push_back(new Point2D(j + 1, out.pointlist[j * 2], out.pointlist[j * 2 + 1]));
}
std::vector<Segment*>out_seg;
for (int j = 0; j < out.numberoftriangles; j++) {
    out_seg.push_back(new Segment(j * 3 + 1, out_p[out.trianglelist[j * 3] - 1],
                                   out_p[out.trianglelist[j * 3 + 1] - 1]));
    out_seg.push_back(new Segment(j * 3 + 2, out_p[out.trianglelist[j * 3 + 1] - 1],
                                   out_p[out.trianglelist[j * 3 + 2] - 1]));
    out_seg.push_back(new Segment(j * 3 + 3, out_p[out.trianglelist[j * 3 + 2] - 1],
                                   out_p[out.trianglelist[j * 3] - 1]));
}
DrawSegments(out_seg);
std::vector<Element*>_e;
for (int j = 0; j < out.numberoftriangles; j++) {
    _e.push_back(new Element(j + 1, out_p[out.trianglelist[j * 3] - 1],
                              out_p[out.trianglelist[j * 3 + 1] - 1], out_p[out.trianglelist[j * 3 + 2] - 1]));
}
polyList[i]->setElements(_e);
polyList[i]->setMeshPoints(out_p);
}
```



## 8.12 Exportar malla

```
if(FileSaveDialog1->Execute()){
    FILE *fp= fopen(((AnsiString)FileSaveDialog1->FileName).c_str(), "w+");
    for (int i = 0; i < out.numberofpoints; i++) {
        fprintf(fp, "d%\t%f\t%f\n", i+1, out.pointlist[i * 2], out.pointlist[i * 2 + 1]);
    }
    fclose(fp);
}

if(FileSaveDialog1->Execute()){
    FILE *fp= fopen(((AnsiString)FileSaveDialog1->FileName).c_str(), "w+");
    for (int i = 0; i < out.numberoftriangles; i++) {
        fprintf(fp, "d%\t%d\t%d\n",out.trianglelist[i*3], out.trianglelist[(i*3)+1], out.trianglelist[(i*3)+2]);
    }
    fclose(fp);
}
```

## 8.13 Determinar continuidad en condiciones de frontera

```
void findIdBoundSeq(){
    for (int i = 0; i < contour.size(); i++) {
        if(contour.at(i)->getP1()->hasCfP()==true){
            if (contour.at(i)->getP1()->getCfId()==-1) {
                contour.at(i)->getP1()->setCfId(contour.at(i)->getCf());
            }else{
                bool seq=contour.at(i)->getP1()->getCfId()==contour.at(i)->getCf() ? true:false;
                contour.at(i)->getP1()->setCfP(seq);
            }
        }

        if(contour.at(i)->getP1()->hasCfP()==true){
            if (contour.at(i)->getP2()->getCfId()==-1) {
                contour.at(i)->getP2()->setCfId(contour.at(i)->getCf());
            }else{
                bool seq=contour.at(i)->getP2()->getCfId()==contour.at(i)->getCf() ? true:false;
                contour.at(i)->getP2()->setCfP(seq);
            }
        }
    }
}
```

## 8.14 Determinar ángulo interno

```
void processInterAngle() {
    Segment* v1;
    Segment* v2;
    for (int i = 0; i < contour.size(); i++) {
        v1 = contour[i];
        if (i == contour.size() - 1) {
            v2 = contour[0];
        }
        else {
            v2 = contour[i + 1];
        }
        float hip = std::sqrt
            (std::pow(v2->getP2()->getX() - v1->getP1()->getX(), 2) +
             std::pow(v2->getP2()->getY() - v1->getP1()->getY(), 2));

        float cos_ang = ((hip * hip) - (v1->getLength() * v1->getLength())
            - (v2->getLength() * v2->getLength())) /
            (-2 * v1->getLength() * v2->getLength());

        float ang = std::acos(cos_ang);

        v1->getP2()->setAng(ang);
    }
}
```

## 8.15 Determinar proximidad con agujeros

```
void getIfIsNearH(float radio){
    std::map<int, Point2D*>aux;

    std::map<int, Point2D*>aux_s;

    for (size_t i = 0; i < holes.size(); i++) {
        std::map<int, Point2D*>aux2;
        aux2 = holes[i]->getPolyPoints();
        aux.insert(aux2.begin(), aux2.end());
    }
    for (size_t i = 0; i < contour.size(); i++) {
        aux_s.insert(std::pair<int, Point2D*>(contour[i]->getP1()->getId(),
            contour[i]->getP1()));
        aux_s.insert(std::pair<int, Point2D*>(contour[i]->getP2()->getId(),
            contour[i]->getP2()));
    }

    for (size_t i = 0; i < marks.size(); i++) {
        aux_s.insert(std::pair<int, Point2D*>(marks[i]->getP1()->getId(),
            marks[i]->getP1()));
        aux_s.insert(std::pair<int, Point2D*>(marks[i]->getP2()->getId(),
            marks[i]->getP2()));
    }

    for (size_t i = 0; i < inner_points.size(); i++) {
        aux_s.insert(std::pair<int, Point2D*>(inner_points[i]->getId(),
            inner_points[i]));
    }
    for(std::map<int,Point2D*>::iterator i=aux_s.begin(); i!=aux_s.end(); ++i){
        for(std::map<int,Point2D*>::iterator j=aux.begin(); j!=aux.end(); ++j){
            bool finish=false;
            float distance=getDistanceBetPoints(i->second,j->second);
            if (distance<=radio) {
                finish=true;
                i->second->setHoleNear(true);
            }
            if(finish){
                j=aux.end();
            }
        }
    }
}
```