UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA MECÁNICA

DESARROLLO DE PLATAFORMA WEB PARA LA SIMULACIÓN DE PROBLEMAS DE MECÁNICA DE SÓLIDOS LINEALES ESTÁTICOS EN DOS DIMENSIONES UTILIZANDO EL MÉTODO DE ELEMENTOS FINITOS Y EL MÉTODO DE ELEMENTOS VIRTUALES

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS DE LA INGENIERÍA, MENCIÓN MECÁNICA

NICOLÁS JESÚS MUÑOZ GUAMÁN

PROFESOR GUÍA:
ALEJANDRO ORTIZ BERNARDIN

MIEMBROS DE LA COMISIÓN:
NANCY HITSCHFELD KAHLER
WILLIAMS CALDERÓN MUÑOZ

Este trabajo ha sido parcialmente financiado por:
PROYECTO ANID-FONDECYT Nº 1221325
ANID BECA MAGISTER/NACIONAL FOLIO 22220197

SANTIAGO DE CHILE
2025

DESARROLLO DE PLATAFORMA WEB PARA LA SIMULACIÓN DE PROBLEMAS DE MECÁNICA DE SÓLIDOS LINEALES ESTÁTICOS EN DOS DIMENSIONES UTILIZANDO EL MÉTODO DE ELEMENTOS FINITOS Y EL MÉTODO DE ELEMENTOS VIRTUALES

El presente proyecto de tesis detalla el desarrollo de una plataforma web denominada MECHAIDE, destinada a simular problemas de elasticidad lineal bidimensionales en mecánica de sólidos estáticos, utilizando el método de elementos finitos (MEF) y el método de elementos virtuales (MEV).

Se utilizaron tecnologías estándar como HTML, PHP, SQL y JavaScript, incorporando algoritmos para la lectura de geometrías CAD, generación y refinamiento de mallas de tipo Delaunay, Voronoi y mallas poligonales de tipo D-Polylla, algoritmo desarrollado en la presente tesis como variante del algoritmo Polylla [1]. La plataforma permite operar desde dispositivos móviles o de escritorio, guardar proyectos en la nube, cargar geometrías y mallas, configurar condiciones de contorno, calcular y visualizar resultados directamente en la página web.

Los métodos MEF y MEV fueron adaptados para JavaScript desde la metodología de cálculo presente en el software VEMLAB [2] originalmente desarrollado en Matlab.

En el marco del proyecto de FONDECYT Nº 1221325 se desarrolló el algoritmo Poly Remesh, el cual permite remallar modificando únicamente la conectividad de sectores problemáticos sin modificar la matriz nodal. Su desempeño es correcto para problemas con baja cantidad de nodos, sin embargo, la ausencia de algoritmos de detección de contacto compromete la estabilidad de la solución, por lo que su aplicación en escenarios de mayor complejidad requiere su adaptación a un entorno de cálculo más robusto.

La validación de la aplicación se realizó comparando resultados numéricos con soluciones analíticas y con simulaciones en software comercial y libre, evidenciando buena convergencia. Adicionalmente, se evidenció que el error de deformación nodal global evaluado mediante norma L^2 y seminorma H^1 es menor en mallas D-Polylla que con mallas de tipo Voronoi. En términos de eficiencia de cómputo, la principal limitación radica en el algoritmo de inversión matricial para JavaScript, lo que restringe el uso de la aplicación a mallas de tamaño pequeño o mediano, donde la descomposición de Cholesky ofrece rendimientos aceptables.

El desarrollo permite experimentar con nuevos algoritmos de simulación, con potencial para su uso en investigación, con posibilidades de extensión hacia modelos tridimensionales, formulaciones no lineales u otras áreas del análisis numérico, siempre que se resuelva la limitante asociada a la resolución de sistemas de ecuaciones de grandes dimensiones.

MECHAIDE está disponible en línea *www.tesis.mechaide.com*

Dedicatoria

Esta tesis se la dedico a mi hijo Nicolás, sé lo mucho que te esfuerzas por tus metas y veo el talento y capacidad con que las realizas. Te amo y quiero que sepas que cuentas conmigo para apoyarte en todo lo que necesites. Tú me has enseñado muchas cosas de la vida, tu alegría, tu perseverancia, decisión y templanza me emociona y me hace sentir orgulloso del hijo que tengo. Te amo Nico.

También dedico esta tesis a mis hijas gemelas Antonia y Sofía, su alegría, energía y cariño me llenan el corazón. Espero que cuando puedan aprendan a leer y vean esta dedicatoria, sepan que mi felicidad está a su lado, cuidándolas y dándoles amor. Deseo que crezcan felices y se sientan amadas en cada paso que den en esta vida.

En especial, dedico esta tesis a mi esposa, Lissette Lavín. Eres el alma y la fortaleza de nuestra familia. Gracias por tu amor, tu paciencia y por entregarte por completo al cuidado y formación de nuestros hijos. Siendo una profesional, elegiste acompañarnos y quedarte a su lado, dándoles tu tiempo, tu amor y tu sabiduría. Lissette, sabes que sin ti no sería el hombre que soy hoy. Todo lo logrado en esta tesis y en mi vida es fruto de tu dedicación, esfuerzo y amor. Le doy gracias a Dios que me ha bendecido con tu compañía, porque sin duda es el mejor regalo que me ha dado. Te amo.

Agradezco a Dios por darme la posibilidad de amarlos y tenerlos a mi lado, ustedes son la razón de mi vida y las personas que me llenan el alma y le dan paz a mí espíritu, para poder disfrutar, estudiar y comprender de la creación de Dios.

Agradecimientos

Quisiera expresar mi sincero agradecimiento a los miembros de la comisión evaluadora, profesores Nancy Hitschfeld y Williams Calderón, por sus valiosos comentarios y sugerencias, que contribuyeron significativamente a la mejora de esta tesis.

Agradezco también al profesor Rubén Fernández Urrutia, coordinador actual del programa de Magíster y a la Asistente Ejecutiva Talía Vilches, por sus gestiones para el reingreso y la información proporcionada durante todo el proceso de graduación. Asimismo, al profesor Ramón Frederick, coordinador anterior, quien me permitió ingresar al programa en su momento y me apoyó con los documentos necesarios para la postulación a la beca.

Extiendo mi gratitud al profesor guía, Alejandro Ortiz-Bernardin, por sus lineamientos, acotaciones y acompañamiento constante en cada etapa de esta tesis. Su paciencia, disposición para resolver dudas incluso en la madrugada y la generosidad al compartir sus conocimientos y código fueron fundamentales para concretar este trabajo, especialmente considerando los años que me tomó concluirlo.

Finalmente, agradezco a Dios por su amor y por permitirme estudiar y comprender, al menos a través de la simulación, una pequeña parte de su creación y el modo en que fue diseñada.

Tabla de Contenido

Índice de Figuras

Índice de Algoritmos

1 Introducción

1.1 Motivación

El desarrollo de las nuevas tecnologías de informática es un proceso constante y acelerado que ha tenido un gran impacto en la sociedad y en la forma en que se vive, en especial, el uso de las tecnologías de desarrollo web [3], que han revolucionado la forma en que nos comunicamos, divertimos, trabajamos y aprendemos.

La adaptación de software de cálculo y diseño de ingeniería, a estas nuevas tecnologías de desarrollo web, es un paso complejo pero necesario para la industria, debido a la diversificación de dispositivos electrónicos disponibles en el mercado [4], cuyo desarrollado clásico basado en dispositivos de escritorio, no es adecuado para la flexibilidad y alternativas de desarrollo que requieren estos dispositivos.

La estrategia comercial de proveedores de software en los últimos años, al considerar el software como un servicio bajo suscripción y no como un producto, con el fin de integrar el software de escritorio con los servicios en la nube [5], es un ejemplo claro de la dependencia de la conectividad y de la independencia del código del software instalado en el dispositivo.

Este proyecto de tesis, pretende dar un paso y demostrar la factibilidad de desarrollo de una herramienta de cálculo con tecnología web, considerando una tarea especifica: la simulación de problemas de mecánica de sólidos en dos dimensiones, mediante el método de elementos finitos MEF y el método de elementos virtuales MEV, permitiendo de esta forma dar accesibilidad a herramientas de cálculo complejas y personalizadas a todo usuario con una conexión a internet, sin tener que instalar ningún software, ni configurar ningún parámetro para su uso y cuyas actualizaciones y mejoras son aplicadas al instante para todos los dispositivos.

Por otro lado, el aspecto más importante para desarrollar esta aplicación tiene relación con la libertad en la experimentación. Las herramientas de cálculo comerciales son limitadas por su propia programación, utilizando metodologías estándar y seguras para el cálculo, o para mallado y remallado, actuando estos algoritmos como una caja negra, limitando al usuario únicamente a la modificación de ciertos parámetros. Al desarrollar por completo un software, es posible experimentar ya sea con algoritmo modernos de cálculo como el MEV o permitiendo la adaptación de código experimental para el uso de algoritmos de mallado o remallado. Esto conlleva una ventaja enorme para los investigadores, permitiendo mediante ensayo y error, el desarrollo de nuevos algoritmos.

1.2 Objetivos

1.2.1 Objetivo general

Desarrollar una plataforma web para simular problemas estáticos de mecánica de sólidos de elasticidad lineal en dos dimensiones, utilizando el método de elementos finitos y el método de elementos virtuales.

1.2.2 Objetivos específicos

1. Desarrollar algoritmo para la lectura de geometrías CAD en JavaScript.
2. Desarrollar algoritmo para la generación de mallas triangulares y poligonales con opción de refinamiento en JavaScript.
3. Desarrollar algoritmo de remallado para elementos poligonales sin modificar la posición espacial de los nodos de los elementos, de acuerdo con los lineamientos del proyecto Fondecyt 1221325.
4. Adaptar algoritmo de cálculo MEF y MEV a JavaScript en base a código Matlab de software VEMLAB 2.4.
5. Desarrollar interfaz de usuario, estructura de base de datos y formularios para que usuarios puedan generar, manipular y guardar proyectos, geometrías, mallas, remallados, condiciones de contorno, cargas y resultados.
6. Verificar el desarrollo, mediante la comparación de resultados numéricos, respecto a la solución analítica de dos problemas de mecánica de sólidos lineales estáticos, disponibles en la literatura técnica.
7. Verificar el desarrollo, mediante la comparación de resultados numéricos obtenidos mediante el uso de software comercial y software libre.
8. Evaluar alcances cómputo y cantidad de elementos límite a utilizar en la aplicación.

1.3 Alcances

El alcance del proyecto comprende el desarrollo y validación de la plataforma web para simular problemas estáticos de mecánica de sólidos en dos dimensiones, utilizando el MEF y el MEV.

Es importante señalar que los alcances propios de la capacidad de cómputo, así como el límite de elementos a utilizar en la aplicación son evaluados como un objetivo del proyecto de tesis luego del desarrollo mismo de la aplicación, permitiendo de esta manera establecer configuraciones ideales y limitaciones programáticas con el fin de no afectar la experiencia del usuario. Adicionalmente la evaluación de alcances en base a la programación permitirá definir el punto de partida, para establecer objetivos en futuros desarrollos o investigaciones.

A continuación, indican las limitaciones de la implementación:

1. La visualización se desarrolla en un ambiente tridimensional, pero la plataforma se limita al análisis de problemas bidimensionales.

2. La plataforma se limita al análisis de problemas de elasticidad lineal sin considerar otra teoría como el análisis de problemas tipo Poisson.

3. El algoritmo de remallado Poly Remesh (P-Remesh) fue programado de tal manera que permite la simulación iterativa de un mismo problema, generando diferentes instancias de deformación nodal, permitiendo la generación de animaciones. Estas soluciones en ningún caso corresponden a una solución de tipo dinámica, ni soluciones de tipo no lineal, siendo única y exclusivamente generada para demostrar la capacidad de remallado del algoritmo.

4. El desarrollado es evaluado en navegadores Google Chrome versión 123.0.6312.123 y Mozilla Firefox versión 132.0.1 ejecutados en laptop modelo ASUS FX504GE con Windows 11 Professional. No está garantizado el correcto funcionamiento del desarrollo al utilizar otro tipo de navegadores, sistema operativo, o equipo.

5. Los algoritmos presentes en este documento desarrollados son evaluados exclusivamente con la tecnología y herramientas de desarrollo presentes en el lenguaje de programación de JavaScript. La transcripción de estos algoritmos a otros lenguajes de programación podría requerir modificaciones.

6. Los alcances de capacidad de cómputo y tiempos definidos en los resultados de este documento se limitan a la capacidad del equipo utilizado, definido en el punto cinco de esta sección, por lo que los resultados podrían variar dependiendo de la capacidad del equipo en que se ejecute la aplicación.

7. Ningún algoritmo considera optimización mediante paralelismo mediante el uso de la GPU. Los algoritmos pueden tener esta optimización en futuros desarrollos.

1.4 Estructura de algoritmos presentes en el documento

Este documento presenta los principales algoritmos para desarrollar aplicaciones computacionales para el cálculo numérico de problemas de mecánica de sólidos lineales en dos dimensiones, pretendiendo ser un apoyo a los lectores en sus propios desarrollos. En este contexto los operadores y variables definidas en los algoritmos se plasman con diferentes formatos de fuente para mejorar la comprensión y navegación del código desarrollado.

Estos algoritmos se generan con una formulación genérica, no siendo código nativo de JavaScript, sino más bien un pseudo código que pretende simplificar el entendimiento de la metodología, para ser replicado en cualquier lenguaje de programación [6].

Los algoritmos por lo general son funciones estáticas que pueden ser utilizadas y reutilizadas en la medida que se le apliquen sus *argumentos* o *inputs* y retornan variables, objetos o arreglos llamados *outputs* a través de la lógica calculada.

Respecto al formato de fuente, se considera los operadores nativos de programación con fuente negrita y de color azul, por ejemplo: **for**, **if**, **else**, **foreach**, **null**, **false**, **true**. Para nombrar variables de texto o variables numéricas se considera fuente de color negro y cursiva, por ejemplo i, *value*, *diameter*, para nombrar arreglos u objetos se considera fuente negrita, cursiva y de color negro, por ejemplo ***coods***, ***connect***, ***point***. Particularmente en este tipo de variables se puede acceder al contenido del objeto o arreglo a través de su referencia de índice, por ejemplo $\boldsymbol{coords}[i]$, donde se referencia al elemento de índice i del arreglo ***coords***, por otro lado, se puede referenciar las características de los objetos de jerarquía inferior con un punto $\boldsymbol{coords}[i].x$, que identifica a la dimensión x del elemento i del arreglo ***coords***. Para comentarios dentro del algoritmo se considera fuente en negrita y de color verde, presidido de una doble barra lateral, por ejemplo **// Iteración principal**, para textos planos se considera fuente de color rojo oscuro y con doble comilla, por ejemplo "CIRCLE", "ARC". Finalmente, los nombres de funciones desarrolladas en el documento tienen un formato de fuente de color verde-agua en negrita, por ejemplo, **calc_APR**, **point_segment_distance**.

2 Marco teórico

2.1 Mecánica de sólidos lineales estáticos

En el contexto de los problemas de elasticidad lineal estática en dos dimensiones, las metodologías MEF y MEV resuelven de forma numérica ecuaciones diferenciales parciales (EDP) con sus respectivas condiciones de borde, mediante aproximaciones locales que conjuntamente forman el dominio global [2].

Mediante el método de residuos ponderados (MRP) se puede obtener la llamada forma débil, una ecuación integral equivalente a la EDP, la cual trata de encontrar la mejor solución aproximada que satisfaga las condiciones de contorno, minimizando el residuo en todas las partes del dominio [7].

La discretización de la forma débil puede tomar la siguiente forma [8]:

$$\boldsymbol{u}(x,y) = \sum_{i=1}^{N} \boldsymbol{N}_i(x,y)\, \boldsymbol{u}_i \tag{1}$$

donde $\boldsymbol{u}_i$ corresponde a los valores nodales del campo de desplazamiento de cada elemento y $\boldsymbol{N}_i$ corresponde a las funciones de forma del mismo elemento.

Es importante mencionar que la principal diferencia entre ambos métodos es la capacidad de MEV de afrontar problemas en mallas poliédricas arbitrarias, mediante una representación algebraica de la matriz de rigidez elemental $\boldsymbol{K}_E$, sin la evaluación explícita de las funciones de base [2].

En términos generales, para resolver mediante MEF y MEV, se pueden considerar los siguientes pasos:

1. Generar la malla en el dominio, mediante la subdivisión en subintervalos, denominados elementos, por lo general para MEF corresponden a polígonos triangulares o cuadriláteros y para MEV corresponden a cualquier polígono no complejo, es decir, un polígono que no tiene aristas no consecutivas, que compartan un vértice [9].

2. Definir las funciones aplicadas a los grados de libertad GL de los nodos de frontera $\boldsymbol{\Gamma}$, ya sea aplicando tracción en el contorno natural, denominado también como contorno de Neumann $\boldsymbol{\Gamma}_f$ o restringiendo los desplazamientos en el contorno esencial o también llamado contorno de Dirichlet $\boldsymbol{\Gamma}_g$, donde $\boldsymbol{\Gamma} = \boldsymbol{\Gamma}_g \cup \boldsymbol{\Gamma}_f$ [2].

3. Determinar la matriz de rigidez de cada elemento $\boldsymbol{K}_E$ y el vector de fuerzas corporales $\boldsymbol{f}_{b,E}$, ya sea para MEF o para MEV.

4. Ensamblar las matrices elementales para obtener las matrices globales $\boldsymbol{K}_{global}$ y $\boldsymbol{f}_{global}$. Para esto se utilizan las ecuaciones de conectividad, las cuales relacionan los nodos de los elementos con las ecuaciones globales.

5. Imponer las condiciones de contorno directamente sobre el sistema global. Para imposición de la condición de borde natural:

$$\boldsymbol{f}_{global} = \boldsymbol{f}_{global} + \boldsymbol{f}_{b,f} \tag{2}$$

donde $\boldsymbol{f}_{b,f}$, corresponde a el vector de tracción definido al aplicar las funciones en los GL en $\boldsymbol{\Gamma}_f$.

Para imposición de la condición de borde esencial:

$$\boldsymbol{f}_{global} = \boldsymbol{f}_{global} - \boldsymbol{K}_{global} \cdot \boldsymbol{u}_g \tag{3}$$

donde $\boldsymbol{u}_g = \{\boldsymbol{u}_i | i \in GL\ en\ \boldsymbol{\Gamma}_g\}$.

Invertir el sistema global para obtener la solución al sistema de ecuaciones de la forma en los GL libres:

$$\boldsymbol{u}_f = \boldsymbol{K}_f^{-1} \boldsymbol{f}_f \tag{4}$$

donde $\boldsymbol{f}_f$ es el vector de tracción en los nodos con GL libres, $\boldsymbol{K}_f$ es la matriz de rigidez en los nodos con sus GL libres y $\boldsymbol{u}_f$ es el vector de desplazamientos nodales con su GL libres, es decir, la solución del problema de elasticidad lineal.

Finalmente se une la matriz de desplazamientos con GL libres, determinada en el punto 6 con la matriz GL restringidos, para obtener el campo de desplazamientos total $\boldsymbol{u}$:

$$\boldsymbol{u} = \boldsymbol{u}_g + \boldsymbol{u}_f \tag{5}$$

2.1.1 Ecuaciones elementales para MEF

La matriz de rigidez elemental para MEF está dada por:

$$\boldsymbol{K}_E = \int_{\Omega_E} \boldsymbol{B}_E^\top \boldsymbol{D} \boldsymbol{B}_E \, d\Omega \tag{6}$$

donde $\boldsymbol{D}$ es la matriz constitutiva para un material lineal elástico e isotrópico [2]. Para la deformación plana está dada por:

$$\boldsymbol{D} = \frac{E_Y}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ v & 1-\nu & \nu \\ 0 & 0 & 2(1-2\nu) \end{bmatrix} \tag{7}$$

y para el esfuerzo plano:

$$\boldsymbol{D} = \frac{E_Y}{(1-\nu^2)} \begin{bmatrix} 1 & v & 0 \\ v & 1 & 0 \\ 0 & 0 & 2(1-\nu) \end{bmatrix} \tag{8}$$

siendo E_Y es el módulo de Young y ν es el coeficiente de Poisson.

$\boldsymbol{B}_E$ corresponde a la matriz derivada de las funciones de forma en elemento triangular de tres nodos:

$$\boldsymbol{B}_E = [\boldsymbol{B}_1 \quad \boldsymbol{B}_2 \quad \boldsymbol{B}_3] \tag{9}$$

donde $\boldsymbol{B}_i$ esta dado por:

$$\boldsymbol{B}_i = \begin{bmatrix} N_{i,x} & \mathbf{0} \\ \mathbf{0} & N_{i,y} \\ N_{i,y} & N_{i,x} \end{bmatrix} \tag{10}$$

$N_{i,x}$ y $N_{i,y}$ son las derivadas parciales de las funciones de forma respecto de x e y.

El vector de fuerzas corporales $\boldsymbol{f}_{b,f}$ se determina de la siguiente manera:

$$\boldsymbol{f}_{b,f} = \int_{\Omega_E} \boldsymbol{N}_E^\top \, b \, d\Omega \tag{11}$$

donde $\boldsymbol{N}_E$ corresponde a la matriz de funciones de forma del elemento y b representa las fuerzas corporales distribuidas en el dominio.

2.1.2 Ecuaciones elementales para MEV

El vector de fuerza elemental corporal en MEV se expresa como:

$$\boldsymbol{f}_{f,e} = \frac{|E|}{n_e} \begin{bmatrix} \hat{b}_x \\ \hat{b}_y \end{bmatrix} \tag{12}$$

donde $\hat{b}_i$ es el valor promedio de fuerza corporal del elemento para la CB i y n_e es el número de vértices del elemento.

La matriz de rigidez elemental en MEV se descompone en dos partes: la matriz de consistencia y la matriz de estabilidad [10]. Para el problema de elasticidad lineal estática, la matriz de rigidez del elemento para el MEV se define como:

$$\boldsymbol{K}_E = \boldsymbol{K}_E^c + \boldsymbol{K}_E^s = |E|\boldsymbol{W}_c\boldsymbol{D}\boldsymbol{W}_c^\top + (\boldsymbol{I}_{2N} - \boldsymbol{P}_P)^\top \boldsymbol{S}_E(\boldsymbol{I}_{2N} - \boldsymbol{P}_P), \tag{13}$$

donde la matriz de consistencia $\boldsymbol{K}_E^c$ está dada por el término $|E|\boldsymbol{W}_c\boldsymbol{D}\boldsymbol{W}_c^\top$ y la matriz de estabilidad $\boldsymbol{K}_E^s$ corresponde al término $(\boldsymbol{I}_{2N} - \boldsymbol{P}_P)^\top \boldsymbol{S}_E(\boldsymbol{I}_{2N} - \boldsymbol{P}_P)$, donde:

$$\boldsymbol{P}_P = \boldsymbol{H}_R\boldsymbol{W}_R^\top + \boldsymbol{H}_C\boldsymbol{W}_C^\top \tag{14}$$

$$\boldsymbol{H}_C = [(\boldsymbol{H}_C)_1 \dots (\boldsymbol{H}_C)_a \dots (\boldsymbol{H}_C)_N]^\top,\ (\boldsymbol{H}_C)_a = \begin{bmatrix} (x_{1a} - \bar{x}_1) & 0 \\ 0 & (x_{2a} - \bar{x}_2) \\ (x_{2a} - \bar{x}_2) & (x_{1a} - \bar{x}_1) \end{bmatrix}^\top \tag{15}$$

$$\boldsymbol{W}_C = [(\boldsymbol{W}_C)_1 \dots (\boldsymbol{W}_C)_a \dots (\boldsymbol{W}_C)_N]^\top,\ (\boldsymbol{W}_C)_a = \begin{bmatrix} 2q_{1a} & 0 \\ 0 & 2q_{2a} \\ q_{2a} & q_{1a} \end{bmatrix}^\top \tag{16}$$

$$\boldsymbol{H}_R = [(\boldsymbol{H}_R)_1 \dots (\boldsymbol{H}_R)_a \dots (\boldsymbol{H}_R)_N]^\top,\ (\boldsymbol{H}_R)_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ (x_{2a} - \bar{x}_2 & -(x_{1a} - \bar{x}_1) \end{bmatrix} \tag{17}$$

$$\boldsymbol{W}_R = [(\boldsymbol{W}_R)_1 \dots (\boldsymbol{W}_R)_a \dots (\boldsymbol{W}_R)_N]^\top,\ (\boldsymbol{W}_R)_a = \begin{bmatrix} \bar{\phi}_a & 0 \\ 0 & \bar{\phi}_a \\ q_{2a} & -q_{1a} \end{bmatrix}^\top \tag{18}$$

en donde $\boldsymbol{I}_{2n}$ corresponde a la matriz de identidad de tamaño $2N$, $\boldsymbol{x}$ corresponde a la matriz de coordenadas del elemento a, $\bar{\phi}_a = \frac{1}{N}$, con N es el número de lados del elemento. q_{ia} se puede calcular de la siguiente manera:

$$q_{ia} = \frac{1}{4|E|}(|e_{a-1}|\{n_i\}_{a-1} + |e_a|\{n_i\}_a) \tag{19}$$

$|E|$ es el área del polígono y se obtiene como:

$$|E| = \sum_{a=1}^{N} \frac{x_{1a-1}x_{2a} - x_{2a-1}x_{1a}}{2}, \tag{20}$$

$|e_i|$ corresponde a la longitud del i-ésimo lado del polígono y se calcula como:

$$|e_i| = [(x_{1a} - x_{1a-1})^2 + ((x_{2a} - x_{2a-1})^2]^{0,5}, \tag{21}$$

$\boldsymbol{n}_a$ corresponde al vector de normal unitario apuntando hacia afuera del polígono, que se obtiene mediante:

$$\boldsymbol{n}_a = \begin{bmatrix} \frac{x_{2a} - x_{2a-1}}{|e_i|} & \frac{x_{1a-1} - x_{1a}}{|e_i|} \end{bmatrix} \tag{22}$$

y $\boldsymbol{S}_E$ es una aproximación de la matriz de rigidez elemental, la cual es más conveniente para su implementación. Se elije de modo tal, que no se pierda la coercitividad de la forma

bilineal. Dentro de las variadas posibilidades para esta aproximación, una de ellas está dada por:

$$\boldsymbol{S}_E = \alpha_E \boldsymbol{I}_{2N}, \alpha_E = \frac{\gamma |E| \text{trace}(D)}{\text{trace}(\boldsymbol{H}_C^{\boldsymbol{T}} \boldsymbol{H}_C)} \tag{23}$$

2.2 Tecnologías de desarrollo

El código se desarrolla aplicando una programación orientada a objetos, con el fin estructurar el software en piezas simples y reutilizables [11]. Se limitará en gran medida la cantidad de clases, sin considerar paralelismo, para facilitar la navegación y compresión del código, con el fin de que el programa pueda ser modificado por profesionales cuya especialidad no sea exclusivamente las ciencias de la computación.

Se considerará para alojar el servicio, un servidor compartido (shared hosting) de pago, basado en Linux, con cPanel, PHP, MariaDB y un servidor LiteSpeed. Se utilizará PHP para ejecutar las consultas en la base de datos. No se utilizarán Frameworks de JavaScript, como React o Angular, esto con el fin de evitar posibles limitaciones de desarrollo producto de la estructuración de estas librerías [12].

El código necesario para resolver operaciones matemáticas, cálculos matriciales y renderización se desarrollará utilizando JavaScript, esto se define para que la capacidad de cómputo dependa del dispositivo del usuario. Utilizando PHP, es posible realizar los cálculos desde el servidor, pero no es recomendable, debido a que las operaciones a realizar son complejas y duraderas por lo que consumirían de sobre manera la capacidad de transferencia de datos del servicio de pago.

2.3 Geometrías en dos dimensiones CAD

Para la importación de geometrías CAD, se utiliza el tipo de archivo .DXF. Este formato de visualización y modelamiento paramétrico en dos dimensiones se puede guardar directamente desde Autodesk AutoCAD, formato que permite visualizar sin encriptación de las operaciones geométricas de las figuras dibujadas, permitiendo su manipulación y extracción directa [13].

2.4 Generación de mallas

El desarrollo informático para el mallado se realizará mediante la adaptación de librerías ya desarrolladas que contienen algoritmos requeridos de forma parcial, para su uso en arreglos de geometrías en JavaScript.

Las principales librerías con operadores geométricos y de mallado en dos dimensiones se detallan a continuación:

1. *fast-poisson-disk-sampling.js:* Pequeña librería para JavaScript que permite la generación arreglos de puntos en dos dimensiones de acuerdo con el algoritmo de muestreo de disco de Poisson. Este algoritmo a su vez permite generar de forma aleatoria una malla de puntos muy compacta de manera tal, que mantienen una distancia relativamente igual entre sí [14].

2. *turf.js:* Turf es una biblioteca JavaScript para análisis espacial. Incluye operadores geométricos booleanos, además de herramientas de estadística y clasificación.

3. *d3.js:* Completa biblioteca JavaScript de código abierto para visualizar datos, la cual contiene herramientas para la generación de elementos triangulares utilizando el algoritmo de Delaunay y para la generación de elementos poligonales a través del algoritmo de teselaciones de Voronoi centroidales (CVT) [15] .

2.4.1 Calidad de malla poligonal

El método de elementos virtuales permite el uso de cualquier tipo de malla poligonal para el desarrollo del cálculo. Las mallas poligonales al igual que las mallas triangulares o cuadradas utilizadas para MEF deben tener elementos consistentes geométricamente para la convergencia de la solución.

El autor [16] desarrolla un algoritmo acumulativo para determinar el denominado *indicador de calidad de malla* $\varrho(\mathcal{T}^h)$. Este algoritmo capaz de entregar un valor escalar sobre el comportamiento de la malla, el cual depende exclusivamente de la geometría de los elementos, entregando valores entre 0 y 1, donde valores cercanos a 1 equivalen a una malla con excelente convergencia y baja probabilidad de error, mientras que un valor cercano a 0 equivale a una baja convergencia y alta probabilidad de error [16].

Las ecuaciones (24) a (28) presentan el algoritmo para determinar el *indicador de calidad de malla* (ϱ), donde $\mathcal{T}^h$ es la malla que contiene los elementos E, $k(E)$ corresponde al área del kernel, $\mathcal{I}_E^i$ corresponden a las aristas de E, h_E es el diámetro del elemento circunscrito, es decir, la mayor distancia entre dos puntos del elemento E. Finalmente e, corresponde a una arista del polígono:

$$\varrho(\mathcal{T}^h) = \left[\frac{1}{\#\{E \in \mathcal{T}^h\}} \sum_{E \in \mathcal{T}^h} \frac{\varrho_1(E)\varrho_2(E) + \varrho_1(E)\varrho_3(E) + \varrho_1(E)\varrho_4(E)}{3} \right]^{0,5} \tag{24}$$

$$\varrho_1(E) = \frac{k(E)}{|E|} \tag{25}$$

$$\varrho_2(E) = \frac{\min\left(|E|^{0,5}, \min\limits_{e \in \partial E} |e|\right)}{\max(|E|^{0,5}, h_E)} \tag{26}$$

$$\varrho_3(E) = \frac{3}{\#\{e \in \partial E\}} \tag{27}$$

$$\varrho_4(E) = \min_i \frac{\min\limits_{e \in \mathcal{J}_E^i} |e|}{\max\limits_{e \in \mathcal{J}_E^i} |e|} \tag{28}$$

En resumen, el factor ϱ_1 favorece a elementos de ángulos y aristas similares la ϱ_2 penaliza elementos con lados muy pequeños respecto al tamaño del elemento, ϱ_3 penaliza elementos con muchos lados y el factor ϱ_4 penaliza elementos con aristas dispares. Finalmente $\varrho(\mathcal{T}^h)$ entrega una métrica global de todos estos factores.

El autor [17] a través del desarrollo y análisis de su propio algoritmo de remallado, al comparar los resultados de las métricas de mallado con los resultados del error obtenido, deduce que el indicador $\varrho(\mathcal{T}^h)$ es impreciso debido a que sobrestima el efecto de la reducción del número de lados en la malla durante el proceso de remallado, además, considera la relación área perímetro de un determinado elemento, denominado APR por sus siglas en inglés, sí se condice con los errores de desplazamiento obtenidos.

El Algoritmo 1 presenta la estructura para calcular el APR a través de un arreglo de puntos que definen al polígono, utilizando la fórmula del área de Gauss [18] para calcular el área del polígono.

Algoritmo 1. Calcular APR (**calc_APR**)

Inputs ***polygonCoords*** arreglo de n puntos en dos dimensiones que conforman el polígono.

```
area ← 0
perimeter ← 0|
for i ← 0 to i < n do
    j ← (i + 1)% n
    xi ← polygonCoords[i].x
    yi ← polygonCoords[i].y
    xj ← polygonCoords[j].x
    yj ← polygonCoords[j].y
    area ← area + xi · yj − xj · yi
```

```
    perimeter ← perimeter + hypot(xj − xi, yj − yi)
  end
  area ← abs(area) / 2
  return perimeter = 0 ? Infinity : area / perimeter
end
```

2.4.2 Algoritmo de mallado Polylla

El algoritmo Polylla, permite la generación de mallas poligonales a partir de una triangulación inicial de Delaunay, permitiendo crear polígonos convexos y no convexos funcionales para su uso en la aplicación de MEV [1].

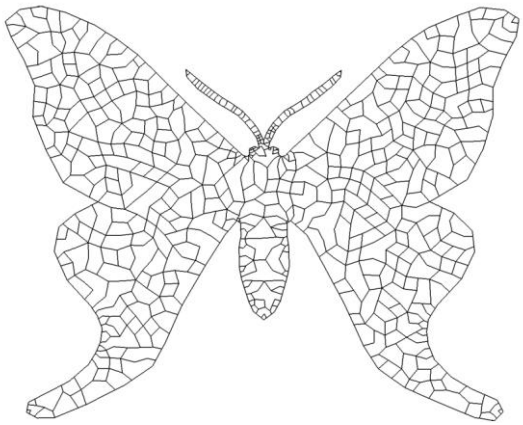


Figura 1. Ejemplo de mallado tipo Polylla. Tomado de [1].

El algoritmo se divide en tres fases principales:

1. *Etiquetado:* Se etiquetan todas las aristas de cada elemento triangular, identificando la *arista terminal* la cual corresponde a la arista más larga que comparten dos triángulos adyacentes, la *arista de frontera* la cual se puede encontrar en el borde del dominio y no puede ser una *arista terminal* y la *arista interna* la cual corresponde la más larga de sólo uno de los dos triángulos.

   Se selecciona un *triángulo semilla*, como punto de partida para generar el polígono.

2. *Recorrido:* Se generan los polígonos, desde una arista del *triángulo* semilla recorriendo los triángulos de la vecindad identificando las *aristas frontera* en orden antihorario hasta encontrar la misma arista para generar un polígono cerrado. Si el polígono creado es *polígono complejo*, es decir, el polígono contiene *aristas frontera* que comparten ambos vértices, entonces estos polígonos se toman para la fase de reparación.

3. *Reparación:* Se identifica el vértice de las *aristas frontera* que comparten ambos vértices en el *polígono complejo* (arista 29-30 de la Figura 2), posteriormente, se

determina el *grado de la arista* el cual corresponde al número de aristas adyacentes. Dependiendo del grado de la arista se divide al polígono uniendo el vértice de la arista compleja con algún vértice del polígono, dividiendo el *polígono complejo* en dos *polígonos simples*.

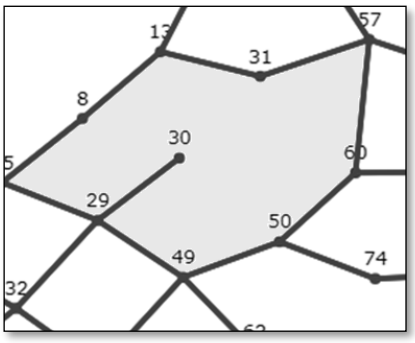


Figura 2. Geometría de polígono complejo.

El autor establece que la calidad de la malla triangular inicial afecta tanto al número de reparaciones realizadas, como también en los resultados obtenidos en simulaciones realizadas aplicando MEV, por lo que recomienda el uso de herramientas computacionales especializadas para la generación de la malla base.

2.4.3 Estructuras de almacenamiento de mallas

Existen diferentes algoritmos que permiten almacenar de manera eficiente la geometría triangular o poligonal de una malla. Estos algoritmos comúnmente son utilizados en gráficos computacionales y pretenden almacenar de manera eficiente la malla para su navegación y modificación. La estructura Indexed Face Set (IFS) [19] utilizada en VEMLAB consta un arreglo de coordenadas, en conjunto con un arreglo de dos dimensiones, el cual se almacena los índices de las coordenadas para representar la cara del polígono. Este arreglo se denomina conectividad de la malla.

Existen otros algoritmos de almacenamiento de mallas tales como Half-Edge (HE) [20], cuya principal característica, es el almacenamiento de las aristas de un polígono en conjunto la arista espejo polígono contiguo. Esta propiedad del algoritmo facilita enormemente la navegación, identificando de forma inmediata las aristas compartidas entre polígonos. El autor en [1] no menciona el uso de la estructura de almacenamiento de mallas para definir la metodología de remallado Polylla, pero si está disponible en la última versión del código fuente disponible en [21].

2.5 Algoritmos de inversión de matriz

En la resolución de problemas lineales estáticos de MEF o MEV la matriz de rigidez $\boldsymbol{K}$ es simétrica y definida positiva [22], esto permite ejecutar algoritmos de inversión de matriz acotados y optimizados para este tipo de problemas. En la Figura 3 se presenta el diagrama de matriz cuadrada simétrica de un problema específico, el cual permite visualizar la estructura de las matrices a resolver.

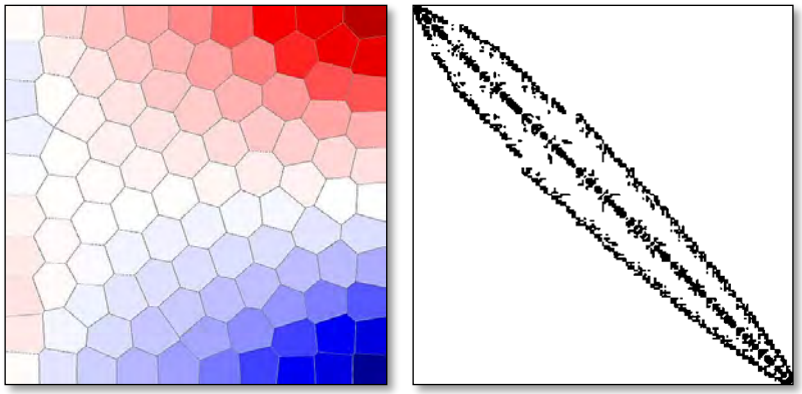


Figura 3. Diagrama de matriz simétrica 100 elementos poligonales.

A continuación, se presentan algunos algoritmos para resolver sistemas lineales de matrices cuadradas simétricas y definidas positivas [23]:

1. *Factorización L-U*: Descompone la matriz objetivo $\boldsymbol{A}$ en una matriz triangular inferior $\boldsymbol{L}$ y en una matriz triangular superior $\boldsymbol{U}$ tal que $\boldsymbol{A} = \boldsymbol{LU}$. Posteriormente se aplica la sustitución hacia adelante donde se resuelve el sistema $\boldsymbol{Ly} = \boldsymbol{b}$ y luego se realiza la sustitución hacia atrás, donde se resuelve el sistema $\boldsymbol{Ux} = \boldsymbol{y}$, donde $\boldsymbol{x}$ es la solución del sistema. El Algoritmo 2 al Algoritmo 5 se detalla el procedimiento de factorización.

Algoritmo 2. Factorización L-U (lu)

Inputs: $\boldsymbol{A}$ matriz cuadrada, $\boldsymbol{b}$ vector

$L \leftarrow$ Matriz de ceros de tamaño $n \times n$

$U \leftarrow$Matriz de ceros de tamaño $n \times n$

$\boldsymbol{y} \leftarrow$Vector de ceros de tamaño n

$\boldsymbol{x} \leftarrow$Vector de ceros de tamaño n

$\{\boldsymbol{L}, \boldsymbol{U}\} \leftarrow$ lu_desc$(\boldsymbol{A})$

$\boldsymbol{y} \leftarrow$ lu_fs$(\boldsymbol{L}, \boldsymbol{b})$

$\boldsymbol{x} \leftarrow$ lu_bs$(\boldsymbol{U}, \boldsymbol{y})$

return $\boldsymbol{x}$

end

Algoritmo 3. Descomposición de matriz L-U (lu_desc)

Inputs: $\boldsymbol{A}$ matriz cuadrada

$\boldsymbol{L} \leftarrow$ Matriz de ceros de tamaño $n \times n$
for $i \leftarrow 0$ **to** $i < n$ **do**
  for $k \leftarrow i$ **to** $k < n$ **do**
    $a \leftarrow 0$
    for $j \leftarrow 0$ **to** $j < i$ **do**
      $a \leftarrow a + L_{i,j} \cdot U_{j,k}$
    end
    $U_{i,k} \leftarrow A_{i,k} - a$
  end
  for $k \leftarrow i$ **to** $k < n$ **do**
    if $i = k$ **then**
      $L_{i,i} \leftarrow 1$
    else
      $a \leftarrow 0$
      for $j \leftarrow 0$ **to** $j < i$ **do**
        $a \leftarrow a + L_{k,j} \cdot U_{j,i}$
      end
      $L_{k,i} \leftarrow \dfrac{A_{k,i} - a}{U_{i,i}}$
    end
  end
end
return $\{\boldsymbol{L}, \boldsymbol{U}\}$
end

Algoritmo 4. Sustitución hacia adelante L-U (lu_fs)

Input: $\boldsymbol{L}$ matriz triangular inferior, $\boldsymbol{b}$ vector

$\boldsymbol{y} \leftarrow$ Vector de ceros de tamaño n
for $i \leftarrow 0$ **to** $i < n$ ***do***
  $y_i \leftarrow b_i$
  for $j \leftarrow 0$ ***to*** $j < i$ **do**
    $y_i \leftarrow y_i - y_j \cdot L_{i,j}$
  end
  $y_i \leftarrow \dfrac{y_i}{L_{i,i}}$
end

return $\boldsymbol{y}$ // Solución del sistema $\boldsymbol{Ly} = \boldsymbol{b}$
end

Algoritmo 5. Sustitución hacia atrás L-U (lu_bs)

Input: $\boldsymbol{U}$ *matriz triangular superior*, $\boldsymbol{b}$ *vector*

$\boldsymbol{x} \leftarrow$ *Vector de ceros de tamaño* n
for $i \leftarrow n-1$ **to** $i \geq 0$ **do**
  $x_i \leftarrow b_i$
  for $j \leftarrow i+1$ **to** $j < n$ **do**
    $x_i \leftarrow x_i - x_j \cdot U_{i,j}$
  end
  $x_i \leftarrow \dfrac{x_i}{U_{i,i}}$
end
return $\boldsymbol{x}$ // **Solución del sistema** $\boldsymbol{Ux = y}$
end

2. *Factorización de Cholesky:* Descompone la matriz objetivo $\boldsymbol{A}$ en una matriz triangular inferior $\boldsymbol{L}$, tal que $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^\top$. Posteriormente se aplica la sustitución hacia adelante donde se resuelve el sistema $\boldsymbol{Ly} = \boldsymbol{b}$ y luego se realiza la sustitución hacia atrás, donde resuelve el sistema $\boldsymbol{L}^\top\boldsymbol{x} = \boldsymbol{y}$, donde $\boldsymbol{x}$ es la solución del sistema. El Algoritmo 6 detalla el procedimiento de factorización.

Algoritmo 6. Factorización de Cholesky (cholesky)

Inputs: $\boldsymbol{A}$ matriz cuadrada, $\boldsymbol{b}$ vector

$L \leftarrow$ Matriz de ceros de tamaño $n \times n$
$\boldsymbol{y} \leftarrow$Vector de c|eros de tamaño n
$\boldsymbol{x} \leftarrow$Vector de ceros de tamaño n
$\boldsymbol{L} \leftarrow$ **cholesky_desc**$(\boldsymbol{A})$
$\boldsymbol{y} \leftarrow$ **cholesky_fs**$(\boldsymbol{L}, \boldsymbol{b})$
$\boldsymbol{x} \leftarrow$ **cholesky_bs**$(\boldsymbol{L}^\top, \boldsymbol{y})$
return $\boldsymbol{x}$
end

Algoritmo 7. Descomposición de matriz Cholesky (cholesky_desc)

Inputs: $\boldsymbol{A}$ matriz cuadrada

$\boldsymbol{L} \leftarrow$ Matriz de ceros de tamaño $n \times n$
for $i \leftarrow 0$ **to** $i < n$ **do**
  for $j \leftarrow 0$ **to** $j \leq i$ **do**
    $a \leftarrow 0$
    if $j = i$ **then**
      for $k \leftarrow 0$ **to** $k < j$ **do**
        $a \leftarrow a + L_{j,k} \cdot L_{j,k}$

```
            end
            L_{j,j} ← (A_{j,j} − a)^{−0.5}
        else
            for k ← 0 to k < j do
                a ← a + L_{i,k} · L_{j,k}
            end
            L_{i,j} ← (A_{i,j} − a) / L_{j,j}
        end
    end
end
return L
end
```

Algoritmo 8. Descomposición de matriz Cholesky (**cholesky$_{\text{fs}}$**)

```
Inputs: L matriz, b vector
y ← Vector de ceros de tamaño n
for i ← 0 to i < n do
    a ← 0
    for j ← 0 to j < i do
        a ← a + L_{i,j} · y_j
    end
    y_i ← (b_i − a) / L_{i,i}
end
return y
end
```

Algoritmo 9. Sustitución hacia atrás Cholesky (**cholesky$_{\text{bs}}$**)

```
Inputs: LT matriz transpuesta de L, y vector
x ←Vector de ceros de tamaño n
for i ← n − 1 to i ≥ 0 do
    a ← 0
    for j ← i + 1 to j < n do
        a ← a + LT_{i,j} · x_j
    end
    x_i ← (y_i − a) / LT_{i,i}
end
return x
end
```

3. *Método del gradiente conjugado:* Método iterativo, el cual estima un valor inicial para el vector $\boldsymbol{x}$, donde el residuo inicial esta dado por $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$, luego se determina el producto punto del residuo con su transpuesta $rs_{old} = \boldsymbol{r}^\top \boldsymbol{r}$. Dentro del bucle, se determina el tamaño del paso $\alpha_i = \frac{rs_{old}}{\boldsymbol{p}_i^\top \boldsymbol{A}\boldsymbol{p_i}}$ en donde $\boldsymbol{p}_i = \boldsymbol{r}_i + \beta_i \boldsymbol{p}_{i-1}$, además, se determina un nuevo valor solución para $\boldsymbol{x}_i = \boldsymbol{x}_i + \alpha_i \boldsymbol{p}_i$ y su respectivo residuo $\boldsymbol{r}_i = \boldsymbol{r}_i - \alpha_i(\boldsymbol{A}\boldsymbol{p_i})$, finalmente se determina un nuevo valor para $rs_{new} = \boldsymbol{r}_i^\top \boldsymbol{r}_i$ y finalmente se determina el valor de $\beta_i = \frac{rs_{new}}{rs_{old}}$. El bucle finaliza cuando $\left|\left|\boldsymbol{r}_i\right|\right|$ es menor a la tolerancia definida. El Algoritmo 10 detalla el método.

Algoritmo 10. Método del Gradiente conjugado (cg)

Inputs: $\boldsymbol{A}$ matriz cuadrada de dimensiones $n \times n$, $\boldsymbol{b}$ vector, *tol* tolerancia, *maxIter* iteraciones máximas

$\boldsymbol{x} \leftarrow$ Vector de ceros de tamaño n
$\boldsymbol{r} \leftarrow \boldsymbol{b} - \boldsymbol{A} \cdot \boldsymbol{x}$
$\boldsymbol{p} \leftarrow \boldsymbol{r}$
$rs_{old} \leftarrow \boldsymbol{r}^\top \boldsymbol{r}$
for $iter \leftarrow 0$ **to** $iter < maxIter$ **do**
  $\alpha \leftarrow \frac{rs_{old}}{\boldsymbol{p}^\top \boldsymbol{A}\boldsymbol{p}}$
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \boldsymbol{p}$
  $\boldsymbol{r} \leftarrow \boldsymbol{r} - \alpha \boldsymbol{A}\boldsymbol{p}$
  $rs_{new} \leftarrow \boldsymbol{r}^\top \boldsymbol{r}$
  if $(rs_{new})^{-1/2} < tol$ **then break for**
  $\beta \leftarrow \frac{rs_{new}}{rs_{old}}$
  $\boldsymbol{p} \leftarrow \boldsymbol{r} + \beta \boldsymbol{p}$
  $rs_{old} \leftarrow rs_{new}$
end
return $\boldsymbol{x}$
end

2.6 Normas de error global

La norma L^2 permite medir el error global de desplazamientos entre una solución exacta y la solución aproximada generada a partir de una simulación mediante MEF o MEV. Permite verificar de forma general la precisión del campo de desplazamientos.

$$\frac{\|\boldsymbol{u} - \boldsymbol{u}^h\|_{L^2}}{\|\boldsymbol{u}\|_{L^2}} = \left(\frac{\sum_{\boldsymbol{E}} \int_{\boldsymbol{E}} (\boldsymbol{u} - \boldsymbol{u}^h)(\boldsymbol{u} - \boldsymbol{u}^h) d\boldsymbol{x}}{\sum_{\boldsymbol{E}} \int_{\boldsymbol{E}} \boldsymbol{u} \cdot \boldsymbol{u} d\boldsymbol{x}} \right)^{1/2} \tag{29}$$

La seminorma H^1, evalúa el error energía interna del sistema. Es más exigente que L^2 dado que se ve afectada por el gradiente de desplazamientos.

$$\frac{\|\boldsymbol{u}-\boldsymbol{u}^h\|_{H^1}}{\|\boldsymbol{u}\|_{H^1}} = \left(\frac{\sum_{\boldsymbol{E}} \int_{\boldsymbol{E}} (\boldsymbol{\varepsilon}-\boldsymbol{\varepsilon}^h)(\boldsymbol{\sigma}-\boldsymbol{\sigma}^h) d\boldsymbol{x}}{\sum_{\boldsymbol{E}} \int_{\boldsymbol{E}} \boldsymbol{\varepsilon}\cdot\boldsymbol{\sigma} d\boldsymbol{x}} \right)^{1/2} \tag{30}$$

En ambos casos la integración sobre el elemento $\boldsymbol{E}$ se realiza mediante integración numérica [17].

3 Características del desarrollo

3.1 Generalidades

La arquitectura de sitio está estructurada en base a los lineamientos de desarrollo definidos en los objetivos del presente documento. Estos lineamientos exigen una estructura flexible, para abarcar características gráficas, ejecución de cálculos iterativos complejos, exportación e importación de archivos, registro y modificación de base de datos.

Tanto en *backend* como en el *frontend*, se prioriza el uso de tecnologías compatibles con servidores Linux disponibles de manera nativa en los entornos de hosting con cPanel, permitiendo de esta forma que la aplicación pueda ejecutarse sin requerir configuraciones o recursos adicionales.

Se define la utilización de JavaScript para la realización de cálculos iterativos, renderización y visualización en tres dimensiones, de esta forma se aprovecha la potencia del dispositivo del cliente sin afectar al servidor. Por otro lado, al ser JavaScript un código que se ejecuta en el dispositivo del cliente, su contenido se ve expuesto al usuario, por esta razón, para limitar la ingeniería inversa y proteger la propiedad intelectual se utiliza la librería *JavaScriptObfuscator.js*¸ librería que ofusca parcialmente el código de JavaScript.

3.1.1 Librerías externas

La aplicación requiere una variedad de librerías externas para su correcta funcionalidad. A continuación, se detalla el propósito de cada librería y utilizada.

1. *jQuery:* Manipulación del *modelo de objetos del documento* DOM y ejecución eventos ligados al servidor mediante JavaScript asincrónico y XML AJAX.
2. *jQuery UI:* Herramientas para modificar el DOM para arrastrar modales.
3. *Bootstrap 5.3:* Framework CSS para el diseño de formularios con estilos predefinidos y con capacidad responsiva.
4. *Three.js:* Librería para visualizar y renderizar geometrías y mallas en 3D, con opciones de definición de materiales, luces y físicas ópticas [24].
5. *Monaco Editor:* Editor de código para sitio web.
6. *Bootstrap Table:* Plugin para Bootstrap para la creación de tablas con funcionalidades avanzadas.

3.1.2 Estructura del software

Se desarrolla la aplicación considerando ingreso y registro de usuario basado en PHP, MariaDB y HTML. Se configura respuesta automática mediante correo electrónico para el restablecimiento de credenciales.

La aplicación web, cuenta con tres aplicaciones “Estático 2D – Análisis”, “Estático 2D. Cálculo” y “Estático 2D. Proyecto” accesibles desde la barra lateral responsiva de acuerdo con la Figura 4.

Figura 4. Menú lateral de aplicaciones.

Adicionalmente el software entrega opciones al usuario para cambiar el tema de visualización modo oscuro, junto con los colores del entorno. La configuración de visualización queda registrada en la base de datos asociada al usuario (ver Figura 5).

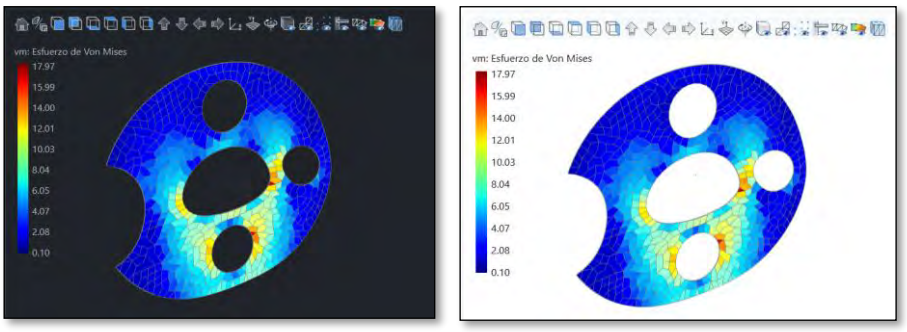


Figura 5. Modos de visualización de entorno

3.1.2.1 Aplicación de administrar proyectos

Permite visualizar y administrar los proyectos guardados en la nube. Utilizando *BootstrapTable.js* se presenta una tabla con nombres, descripción y última captura de pantalla del proyecto generada al guardar el proyecto (ver Figura 6). Desde esta aplicación se pueden cargar los proyectos previamente guardados al entorno de cálculo, permitiendo cambiar el

nombre del proyecto y su descripción. Adicionalmente, se habilita la opción de descargar al dispositivo del usuario toda la información de los proyectos guardados en formato JSON.

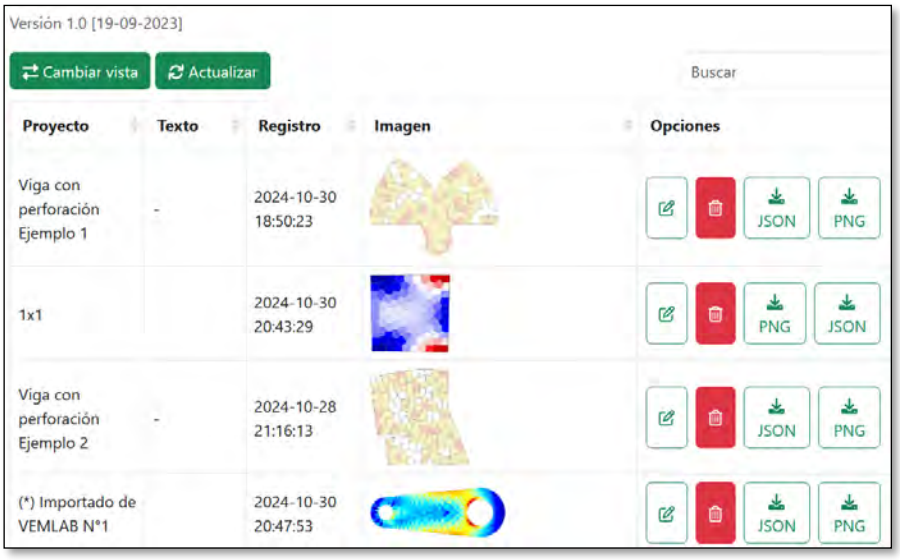


Figura 6. Aplicación para la administración de proyectos.

3.1.2.2 Aplicación cálculo y visualización

La aplicación de cálculo se diseña para ofrecer una interfaz intuitiva para el usuario, imitando la estructura y comportamientos de herramientas y pestañas de software de escritorio basados en Microsoft Office o Autodesk Inventor. En la Figura 7 se presenta a modo ejemplo la pestaña *Vista*, la cual entrega herramientas de manipulación espacial y estética para la representación del elemento de cálculo.

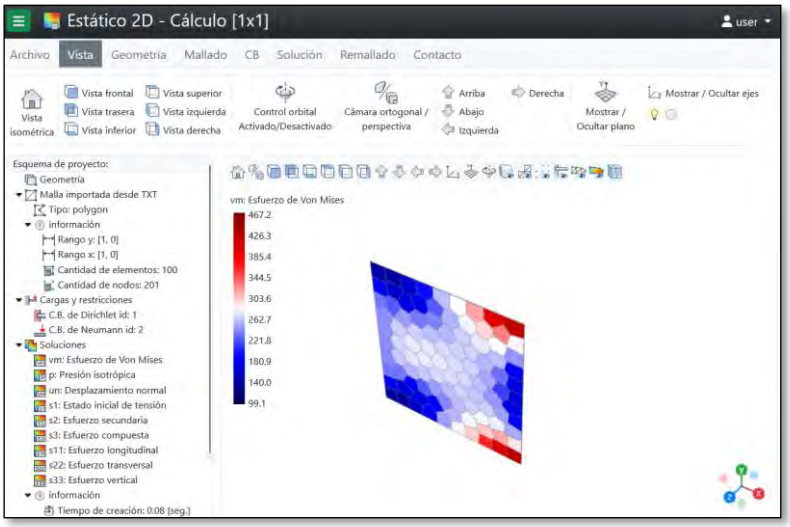


Figura 7. Entorno de cálculo. Pestaña de Vista

3.1.2.3 Estructura de datos de proyecto

Se define una estructura de datos basada en objetos de tipo JSON [25] el cual corresponde a un formato de almacenamiento moderno, fácil de leer y escribir, por su configuración de pares clave-valor.

En la Figura 7 en la sección izquierda, se pueden ver en el menú lateral, como se presenta la información del contendida en el proyecto en formato de etiquetas con iconos. La confección del archivo que maneja la información del proyecto es fundamental para plasmar tanto parámetros que mapeen iconos, como nombres y datos al momento de interactuar con las diferentes opciones y datos del proyecto.

Se establece entonces que cada sección del archivo JSON del proyecto debe tener al menos cuatro claves principales: "*name_es*, *type*, *data* y *content*" sólo en caso de que se quiera presentar como etiqueta en la barra lateral. Esta estructura estándar, permite la visualización dinámica del menú lateral al realizar de forma recursiva un análisis del archivo JSON en cada modificación realizada, de esta forma se puede detectar las claves principales, lo que permite agregar o borrar la información contenida en el archivo en el menú lateral.

La clave *name_es* define el nombre de la etiqueta, *type* establece el tipo de etiqueta con el cual se referencia el icono, junto con las opciones e interacciones con en el menú contextual, *data* almacena hijos delegados de etiqueta y *content* maneja los datos que se cargan o se almacenan dependiendo de la interacción definida por el usuario.

A modo de ejemplo se presenta estructura básica de un proyecto sin geometría, con malla de elementos cargada desde archivo .TXT externo:

```
{
    "name": "Viga empotrada",
    "type": "proyect",
    "data": {
        "geometry": {
            "type": "geometry_all",
            "estado": false,
            "properties": [ ],
            "name_es": "Geometría",
            "content": [ ],
            "data": [ ]
        },
    "mesh": {
        "type": "mesh_2D",
        "name_es": "Malla importada desde TXT",
        "estado": true,
        "refinamiento": 0,
```

```
"content": {
    "he_data": {
        "vértices":
            {
                "x": -7.7895e-12,
                "y": -8.0545e-12,
                "index": 0,
                "incidentEdge": 250,
                "isBorder": true
            }, ...
```


En la estructura se identifican las claves principales definidas anteriormente y se suman otras claves específicas de cada operación de la aplicación tales como *estado, refinamiento,* entre otras.

En la Figura 8 y en la Figura 9, se presenta la vista de la pestaña *Archivo* donde tanto en la importación, como en la exportación, la estructura de datos JSON es transversal tanto para la carga o descargar de proyectos completos como la carga o descargar de datos parciales como lo es la geometría y la malla.

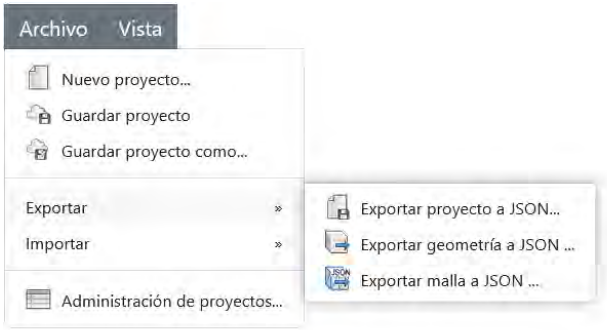


Figura 8. Entorno de cálculo. Exportación de datos en formato JSON

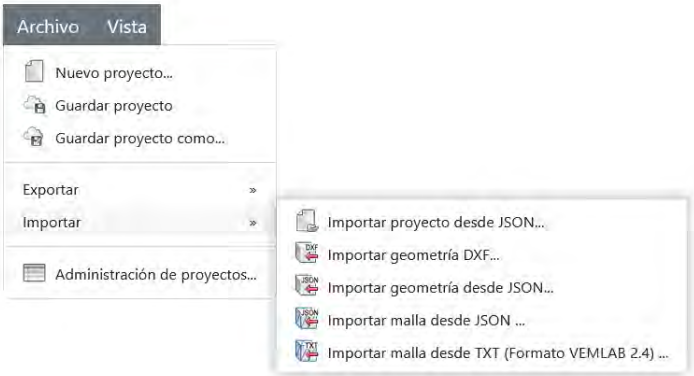


Figura 9. Entorno de cálculo. Importación de datos en JSON

3.1.2.4 Almacenamiento en la nube

La estructura de datos de proyecto definida en la sección anterior, donde se puede organizar la información geométrica, de malla, condiciones de borde y resultados en solo archivo JSON, permite de manera simple guardar los proyectos en una carpeta no publica, asociando el registro del ID del proyecto de la base de datos, a la dirección del servidor donde se guarda el archivo del proyecto, la imagen de vista previa y al nombre del usuario.

Tanto para guardar el proyecto como cargarlo, se utiliza JQuery con AJAX para llamar a través de PHP los datos del usuario y del proyecto mediante consulta a la base de datos de MariaDB del Shared Host.

El método utilizado permite escalabilidad a través de la relación comercial con el proveedor de servicios alojamiento, en la medida que la capacidad de almacenamiento y ancho de banda no se adecuen a la demanda requerida.

3.1.2.5 Diseño de iconos

Se desarrolla un total de 95 iconos de 32x32 píxeles y 43 de 16x16 píxeles, para facilitar la interacción del usuario con la plataforma desarrollada. Los iconos son desarrollados mediante el uso del software de diseño gráfico Paint.NET basadas principalmente herramientas presentes en Autodesk Inventor Professional 2016.

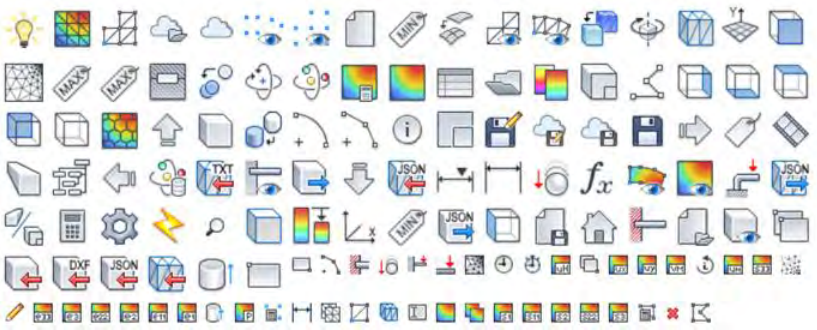


Figura 10. Diseño de iconos

3.2 Importación de geometría CAD

La importación de archivos desde software de diseño asistido por computadora se desarrolla con el fin de que el usuario pueda generar el diseño fuera del entorno de cálculo. La Figura 11 presenta las herramientas para manipulación, importación y exportación de la geometría.

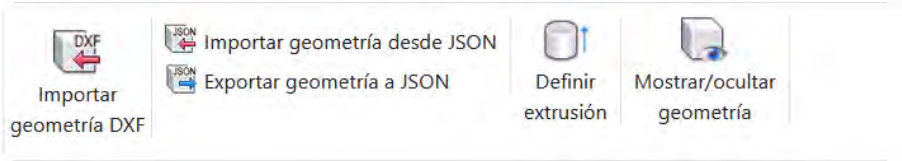


Figura 11. Herramientas para manipulación de geometría.

No se utilizaron librerías externas, debido a que las librerías disponibles no consideran la importación de regiones, propiedad fundamental del diseño para asegurar que la geometría es cerrada.

La metodología de importación fue desarrollada para extraer únicamente la última región creada en el diseño, permitiendo importar la región con diferencias (regiones compuestas de otras regiones, en otros términos, una geometría cerrada con perforaciones).

3.2.1 Importación de geometrías formato DXF

El Algoritmo 29 (Anexo A) permite leer el diseño asistido por computadora en formato .DXF versión *AutoCAD R12/LT2 DXF*. Este archivo se puede generar en cualquier versión de AutoCAD, modificando la opción en el cuadro de selección en la ventana para guardar.

Una vez extraída la información de la última geometría generada, se debe ordenar los vértices de cada operación ya sea *LINE*, *ARC* o *CIRCLE* en sentido antihorario para la región de contorno exterior y en sentido horario para las regiones internas o perforaciones. El Algoritmo 30 (Anexo A) presenta la metodología de orden de vértices considerando que la última región extraída del archivo importado corresponde al contorno exterior de la figura.

Ya con las operaciones geométricas ordenadas, el resultado permite almacenar la información de la geometría de manera compacta, pero no permite su renderización en el entorno de visualización, debido a que, las operaciones geométricas radiales *ARC* y *CIRCLE* extraídas del archivo .DXF deben ser expresadas en trazos rectos. Dependiendo del nivel de detalle de la visualización que se requiera estos trazos rectos pueden ser de menor o mayor dimensión. El Algoritmo 11 permite la transformación de la geometría radial a trazos rectos para su renderización, a través de la división del arco en varios segmentos. Este algoritmo también es utilizado en la generación de malla necesaria para el cálculo MEF o MEV, definiendo el tamaño del elemento al definir la separación entre vértices que dividen el contorno.

Algoritmo 11. Renderización de geometría DXF (**rend_DXF**)

Inputs: ***DXF*** Arreglo de arreglo de n operaciones geométricas ordenadas, *paso_arco* valor numérico flotante que define la distancia entre vértices que dividen un arco o valor entero que define en cuantas partes se divide un arco, *dividir_arco* valor booleano que establece si *paso_arco* se utiliza para definir el paso o como valor entero que divide el arco.

$\boldsymbol{DXF_{Aux}} \leftarrow$ Arreglo de arreglo de n operaciones geométricas
$idu \leftarrow -1$
for $i \leftarrow 0$ **to** $i < n$ **do**
  // **Extracción de valores actuales**
  $tipo \leftarrow \boldsymbol{DXF}[i].tipo$
  $x0 \leftarrow \boldsymbol{DXF}[i].x0$
  $y0 \leftarrow \boldsymbol{DXF}[i].y0$
  $x1 \leftarrow \boldsymbol{DXF}[i].x1$
  $y1 \leftarrow \boldsymbol{DXF}[i].y1$
  $rx \leftarrow \boldsymbol{DXF}[i].rx$
  $ry \leftarrow \boldsymbol{DXF}[i].ry$
  $r \leftarrow \boldsymbol{DXF}[i].r$
  $ars \leftarrow \boldsymbol{DXF}[i].ars$
  $are \leftarrow \boldsymbol{DXF}[i].are$
  $id \leftarrow \boldsymbol{DXF}[i].id$
  $region\_id \leftarrow \boldsymbol{DXF}[i].region\_id$
  $giro \leftarrow \boldsymbol{DXF}[i].giro$
  // **Identificación de región futuro**
  if $!i \leftarrow n-1$ **then**
    $region\_id\_fut \leftarrow \boldsymbol{DXF}[i+1].region\_id$
  end
  // **Identificación de operación**
  if $tipo =$ "ARC" **or** $tipo =$ "CIRCLE" **then**
    $dife\_angle \leftarrow are - ars$
    $arco \leftarrow \textbf{abs}(r \cdot dife\_angle)$
    //Parámetros de ajuste empíricos
    if $dividir\_arco =$ **true then**
      $n_{paso} \leftarrow \textbf{floor}(dife\_angle \cdot 13)$
    else
      $n_{paso} \leftarrow \dfrac{arco}{paso\_arco}$
    end
    if $abs(dife\_angle) > 1.25 \cdot \text{PI}$ **and** $n_{paso} < 5$ **then**
      $n_{paso} \leftarrow 6$
    else if $abs(dife\_angle) > 0.55 \cdot \text{PI}$ **and** $n_{paso} < 2$ **then**
      $n_{paso} \leftarrow 2$
    end
    if $n_{paso} = 0$ **then**
      $paso\_ang \leftarrow dife\_angle$
      $n_{paso} \leftarrow 1$
    else

```
            paso_ang ← difε_angle / n_paso
        end
        if giro = 1 then
            for j ← 0 to j ≤ n_paso do
                x0 ← rx + r cos(ars + j · paso_ang)
                y0 ← ry + r sin(ars + j · paso_ang)
                idu ← idu + 1
                DXF_Aux.add(id, idu, tipo, region_id, x: x0, y: y0)
            end
        end
            for j ← n_paso to j ≥ 0 do
                x1 ← rx + r cos(ars + j · paso_ang)
                y1 ← ry + r sin(ars + j · paso_ang)
                idu ← idu + 1
                DXF_Aux.add(id, idu, tipo, region_id, x: x1, y: y1)
            end
        end
    else if tipo = "LINE" then
        idu ← idu + 1
        DXF_Aux.add(id, idu, tipo, region_id, x: x0, y: y0)
        if i = n − 1 or region_{id_fut} not region_id then
            idu ← idu + 1
            DXF_Aux.add(id, idu, tipo, region_id, x: x1, y: y1)
        end
    end
    region_id_old ← region_id
    id_old ← id
end
return DXF_Aux
end
```

La Figura 12 presenta la representación gráfica de una geometría diseñada en dos dimensiones con AutoCAD, la cual fue importada al software utilizando la metodología presente en esta sección. El software desarrollado permite realizar una extrusión simple de la figura, solo para fines estéticos, no aplicable a la metodología de cálculo.

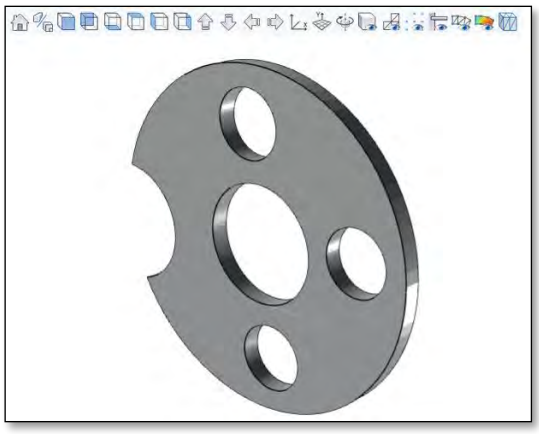


Figura 12. Representación gráfica de geometría 2D con extrusión en 3D.

3.3 Generación de mallas

Los tres algoritmos de generación de malla desarrollados en este documento basan su metodología global de mallado de la librería de Matlab PolyMesher [26] aplicando modificaciones para su compatibilidad con código a JavaScript y el uso de librerías de terceros disponibles.

Esta metodología definida para su aplicación en JavaScript a grandes rasgos permite adaptar cualquier tipo de algoritmo de mallado al contorno o perforaciones de la geometría en dos dimensiones.

En términos generales la metodología se puede expresar en siete pasos:

1. Utilizando Algoritmo 11, se procede realizar la aproximación poligonal de arcos y circunferencias.

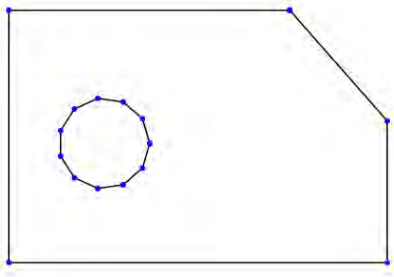


Figura 13. Aproximación poligonal de arcos y circunferencias.

2. Se subdivide el contorno aplicando el Algoritmo 12, considerando como variables de entrada la distancia de separación entre vértices. Los vértices creados en este paso quedan fijos y no se ven afectados por la relajación en los pasos siguientes. El

fundamento de vértices de contorno fijos y separados de manera uniforme, evitando que en la relajación tenga sesgos por falta o exceso de limitación en el movimiento de los vértices interiores.

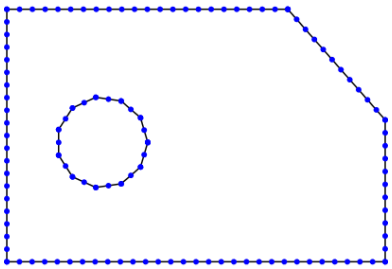


Figura 14. Subdivisión de contorno

Algoritmo 12. Generador de puntos de contorno (**generate_new_points**)

Inputs: ***points*** Arreglo de n puntos en dos dimensiones, d Distancia nominal entre puntos.

newPoints ←Arreglo de puntos en dos dimensiones
//Iteración sobre los puntos de la región
for $i \leftarrow 0$ **to** $i < n$ **do**
  currentPoint $\leftarrow$ ***points***$[i]$
  nextPoint $\leftarrow$ ***points***$[(i+1)\ \%\ n]$
  $edgeLength \leftarrow$ **distance**(***currentPoint***, ***nextPoint***)
  if $edgeLength \geq d$ **then**
    $numNewPoints \leftarrow$ **floor** $\left(\frac{edgeLength}{d}\right)$
    for $j \leftarrow 0$ **to** $j \leq numNewPoints$ **do**
      $t \leftarrow \frac{j}{numNewPoints}$
      $x \leftarrow$ ***currentPoint***$.x + t \cdot$ (***nextPoint***$.x -$ ***currentPoint***$.x$)
      $y \leftarrow$ ***currentPoint***$.y + t \cdot$ (***nextPoint***$.y -$ ***currentPoint***$.y$)
      newPoints$.$**add**(x, y);
    end
  end
end
return ***newPoints***
end

3. Se aplica el muestreo el algoritmo de discos rápidos de Poisson, presente en la librería externa [14]. Este algoritmo permite la generación de puntos de forma eficiente en un área rectangular de manera relativamente uniforme. El área rectangular, se define a

través de la detección de valores máximos y mínimos de los vértices de contorno generados en el paso anterior, determinados mediante el Algoritmo 32 (Anexo A).

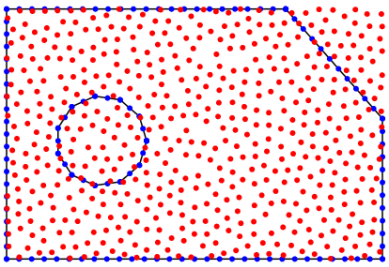


Figura 15. Aplicación de muestreo de disco rápidos de Poisson

4. A través del algoritmo de verificación de puntos en polígono presente en la librería externa *Turf.js* se aplica la función $turf.booleanPointInPolygon$, la cual permite eliminan los vértices que estan fuera del contorno y los vértices que estan dentro de las perforaciones.

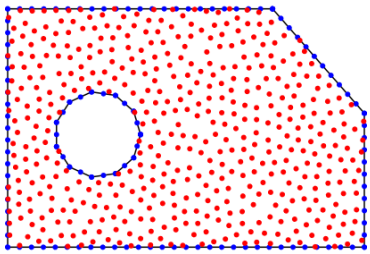


Figura 16. Eliminación de vértices fuera del contorno.

5. A través del Algoritmo 13 y Algoritmo 14 estructurados de acuerdo con [27] se identifican y eliminan los vértices en al vecindad del contorno y de las perforaciones. Esto permite que al momento de aplicar relajamiento a la malla, los vértices interiores se acomoden sin generar atascos con los vértices de contorno, esto considerando que los vértices de contorno son fijos.

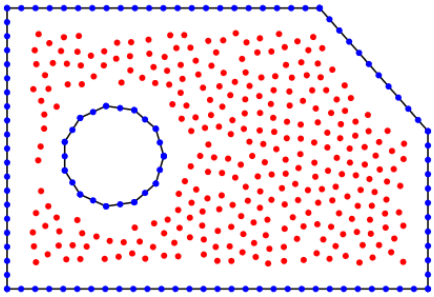


Figura 17. Eliminación de vértices en la vecindad del contorno.

Algoritmo 13. Distancia de un punto a un polígono (distance_to_polygon)

Inputs: ***point*** punto cartesiano en dos dimensiones, ***polygon*** arreglo de n puntos que conforman el polígono.

```
minDist ← Infinity
for i ← 0 to i < n do
    a ← polygon[i]
    b ← polygon[(i + 1)%n]
    dist ← point_segment_distance(point, a, b)
    if dist < minDist then
        minDist ← dist
    end
end
return minDist
end
```

Algoritmo 14. Distancia de un punto a un segmento (point_segment_distance)

Inputs: $\boldsymbol{a}, \boldsymbol{b}$ puntos cartesianos que conforman un segmento, $\boldsymbol{p}$ punto cartesiano en dos dimensiones

$[p_x, p_y] \leftarrow \boldsymbol{p}$
$[a_x, a_y] \leftarrow \boldsymbol{a}$
$[b_x, b_y] \leftarrow \boldsymbol{b}$
$d_x \leftarrow b_x - a_x$
$d_y \leftarrow b_y - a_y$
if $d_x = 0$ **and** $d_y = 0$ **then**
  return $[(p_x - a_x)^2 + (p_y - a_y)^2]^{0.5}$
end
$$t \leftarrow \frac{(p_x - a_x) \cdot d_x + (p_y - a_y) \cdot d_y}{d_x \cdot d_x + d_y \cdot d_y}$$
if $t < 0$ **then**
  $\boldsymbol{closest} \leftarrow \boldsymbol{a}$
else if $t > 1$ **then**
  $\boldsymbol{closest} \leftarrow \boldsymbol{b}$
else
  $\boldsymbol{closest} \leftarrow [a_x + t \cdot d_x, a_y + t \cdot d_y]$
end
return $[(p_x - \boldsymbol{closest}[0])^2 + (p_y - \boldsymbol{closest}[1])^2]^{\frac{1}{2}}$
end

6. Se aplica relajación de Lloyd [28] de acuerdo con el Algoritmo 33 (Anexo B). Este algoritmo solo se aplica a los vértices interiores, sin considerar los nodos de contorno

los cuales son fijos, permitiendo que se acomoden al contorno manteniendo una distancia similar entre ellos. En la medida que se consideren más iteraciones, la distancia promedio entre vértices de la malla será más uniforme. Cabe destacar que el número de iteraciones para la relajación de Lloyd es un parámetro de entrada disponible para el usuario en el formulario para la generación de mallas en el software.

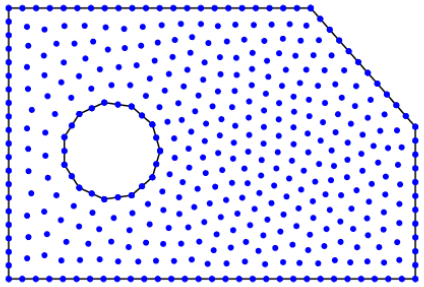


Figura 18. Relajación de Lloyd en vértices interiores.

7. Ya con los vértices generados y con relativa uniformidad en su distribución, se pueden aplicar el tipo de algoritmo de mallado deseado. Posterior a la aplicación del algoritmo triangulación de Delaunay utilizando la librería externa d3.js, se debe filtrar los elementos triangulares que están en las perforaciones, mientras que para las mallas de tipo Voronoi se deben aplicar operaciones booleanas que permitan recortar los polígonos que se generan dentro de las perforaciones. Si bien la generación de polígonos mediante la teselación de Voronoi requiere el uso de la una triangulación inicial, la eliminación de triángulos no es factible dado que los vértices de los triángulos son el centro del elemento poligonal, lo que obliga a aplicar recortes a los polígonos generados. Estas operaciones geométricas booleanas están optimizadas para elementos en dos dimensiones disponibles en la librería Turf.js.

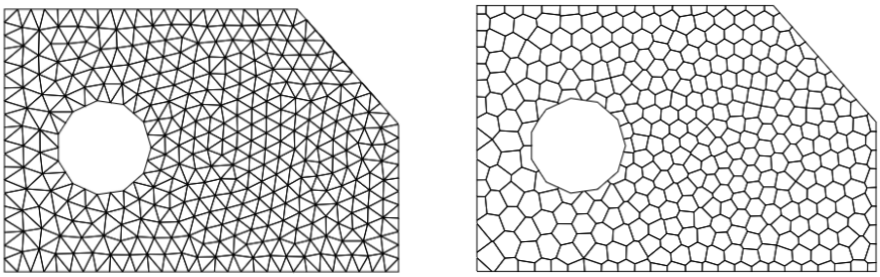


Figura 19. Aplicación de algoritmo de mallado de tipo Delaunay y Voronoi.

El algoritmo para la generación de polígonos D-Polylla requiere como entrada el arreglo de puntos de una malla triangular de tipo Delaunay no requiriendo eliminación de triángulos dado que está ya se realiza al crear la malla triangular inicial.

Con la metodología definida anteriormente, el usuario puede crear mallas luego de cargar la geometría, o cargar las mallas generadas mediante VEMLAB 2.4. La Figura 20 presenta las diferentes herramientas para crear, refinar, exportar o importar las mallas.

Figura 20. Herramientas de manipulación de malla

La ventana de creación de malla presente en la Figura 21, permite seleccionar entre los tres algoritmos desarrollados: "Triangular, poligonal Voronoi y poligonal D-Polylla", adicionalmente permite la selección del tamaño del elemento y la cantidad de iteraciones de relajación.

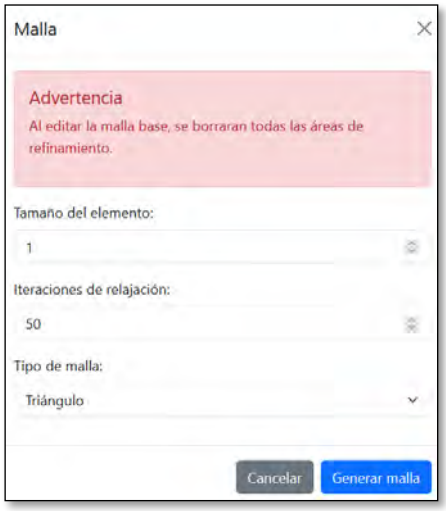


Figura 21. Ventana de creación de malla

3.3.1 Algoritmo para malla poligonal tipo Direct Polylla

El algoritmo Direct Polylla, es una modificación del algoritmo original Polylla para la generación de mallas poligonales definido en [1]. Esta modificación pretende simplificar la forma en que se utiliza la recursión, para identificar las aristas problemáticas en la fase etiquetado y posteriormente reparar la geometría de forma directa en la construcción de los polígonos, eliminando el paso de *reparación de polígonos*, definido de forma nativa en el algoritmo. La Figura 22 presenta un esquema comparativo general entre ambas metodologías.

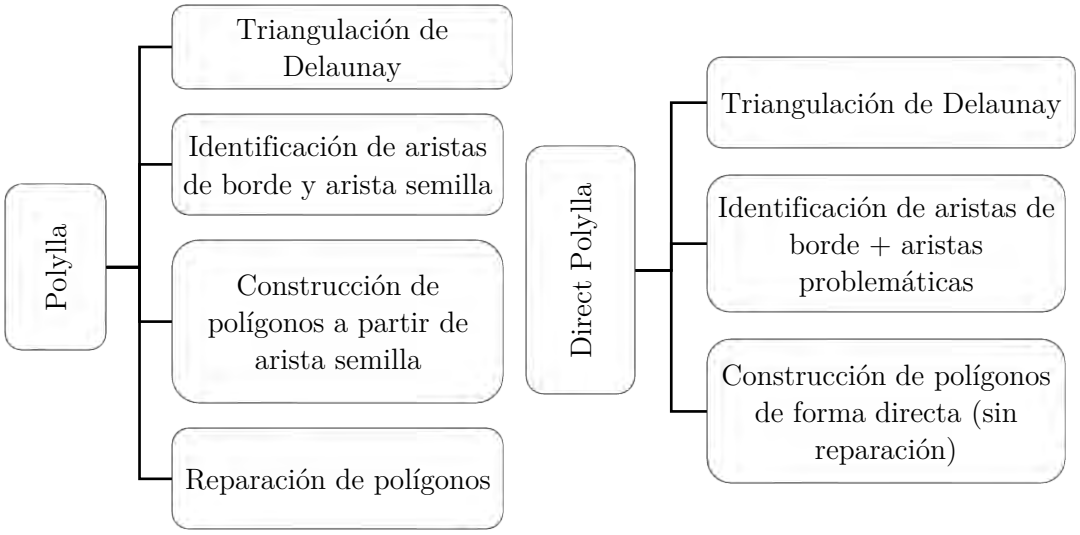


Figura 22. Comparación de algoritmo Polylla versus D-Polylla

La estructura de almacenamiento de malla es fundamental para navegar de manera eficiente en la geometría de los elementos de la malla, por lo que es necesario migrar la estructura a IFS generada a partir de los algoritmos de generación de malla, a una estructura HE donde se pueden detectar las aristas de contacto o aristas gemelas entre polígonos adyacentes, sin recurrir a una sobre recursión sobre los vértices o su conectividad.

El Algoritmo 15 construye la estructura HE en base a el objeto IFS de la malla triangular. Tras inicializar las variables, se crean los objetos para almacenar vértices, caras y el arreglo de claves únicas para las aristas. Luego, para cada triángulo se identifican sus aristas relacionado sus vértices y asignando sus gemelos de los triángulos adyacentes utilizando el arreglo de *claves únicas*. Posteriormente, se relacionan los aristas previas y siguientes del mismo triángulo, para mejorar aún más las opciones de navegación. Finalmente, las aristas sin gemelos se les asigna una etiqueta de *aristas de borde*.

Algoritmo 15. Construcción de HE a partir de arreglo IFS (build_half_edge)

```
Inputs: polygons objeto IFS con coordenadas y conectividad.
    vertices ← [ ]
    faces ← [ ]
    edgeMap ← new Map
    coords ← polygons.coords
    connect ← polygons.connect
    halfEdgeIndex ← 0
    // Crear vértices con índices
    foreach (point, index) of coords do
        vertices[index] ← {
```

```
        x: point.x,
        y: point.y,
        index: index,
        incidentEdge: null,
        isBorder: false,
    }
end
// Función para generar una clave única para una arista
function get_edge_key(a, b) {
    return a + "-" + b
end
// Procesar cada polígono
foreach (polygon, faceIndex) of connect do
    face ← { edge: null, index: faceIndex}
    faceHalfEdges ← [ ]
    // Crear HE para el polígono actual
    foreach i of polygon do
        originIndex ← polygon[i]
        targetIndex ← polygon[(i + 1) % polygon.length]
        origin ← vertices[originIndex]
        target ← vertices[targetIndex]
        key ← get_edge_key(originIndex, targetIndex)
        twinKey ← get_edge_key(targetIndex, originIndex)
        he ← {
            target: target,
            origin: origin,
            twin: null,
            next: null,
            prev: null,
            face: face,
            key: key,
            isBorder: false,
            marked: false,
            index: halfEdgeIndex++,
        }
        halfEdges.push(he)
        faceHalfEdges.push(he)
        // Asignar incidentEdge si no está asignado
        if origin.incidentEdge = null then
            origin.incidentEdge = he.index
        end
```

```
        // Asignar aristas gemelas utilizando edgeMap
        if edgeMap.has(twinKey) then
            twinHeIndex ← edgeMap.get(twinKey)
            he.twin = twinHeIndex
            halfEdges[twinHeIndex].twin = he.index
            edgeMap.delete(twinKey)
        else
            edgeMap.set(key, he.index)
        end
    end
    // Establecer arista previa y el siguiente para cada HE del polígono
    for i = 0 to i < faceHalfEdges.length do
        currentHe ← faceHalfEdges[i]
        nextHe ← faceHalfEdges[(i + 1) % faceHalfEdges.length]
        prevHe = faceHalfEdges[(i − 1
                    + faceHalfEdges.length) % faceHalfEdges.length]
        currentHe.next = nextHe.index
        currentHe.prev = prevHe.index
    end
    // Asignar un índice como punto de partida
    face.edge = faceHalfEdges[0].index
    faces.push(face)
end
// Marcar los HE restantes en edgeMap como bordes
foreach (heIndex, key) of edgeMap do
    he ← halfEdges[heIndex]
    he.isBorder = true
    he.twin = null
    he.origin.isBorder = true
end
return {
    vertices,
    halfEdges,
    faces
}
end
```

Ya con la estructura HE de los elementos de la malla se puede proceder con el etiquetado de cada arista calculando la arista de mayor dimensión con Algoritmo 16.

Algoritmo 16. Etiquetado de arista máxima de triángulo (label_max_edges)

```
Inputs: he_data Estructura HE de la malla triangular.
    maxEdges ← new Map
    foreach face of he_data.faces do
        // Se hace un listado con las aristas del triángulo
        heIndices ← [face.edge]
        heIndices.push(he_data.halfEdges[heIndices[0]].next)
        heIndices.push(he_data.halfEdges[heIndices[0]].next)
        heList ← heIndices.map((i) →  he_data.halfEdges[i])
        // Se calculan los largos de cada arista y luego se comparan
        lengths ← heList.map((he) → edge_length_squared(he))
        maxIndex ← 0
        if lengths[1] > lengths[maxIndex] then maxIndex = 1
        if lengths[2] > lengths[maxIndex] then maxIndex = 2
        maxHe ← heList[maxIndex]
        maxHe.isMax = true
        maxEdges.add(maxHe.index)
    end
    return maxEdges
end
```

Algoritmo 17. Largo de arista al cuadrado (edge_length_squared)

```
Inputs: he Estructura HE de cada arista.
    origin ← he.origin
    target ← he_data.halfEdges[he.next].origin
    dx ← origin.x − target.x
    dy ← origin.y − target.y
    return dx² + dy²
end
```

El Algoritmo 18 se aplica para la identificación de las aristas de frontera. El algoritmo recorre las aristas HE de la malla triangular, etiquetando las aristas del contorno y aristas interiores que no debe ser la más larga, ni ella misma, ni su gemela.

Algoritmo 18. Etiquetado de aristas de frontera (label_frontier_edges)

```
Inputs: he_data Estructura HE de la malla triangular.
    frontierEdges ← new Set()
    foreach he of he_data.halfEdges do
        if he.isBorder then
            // Arista de borde, etiquetarla como frontier-edge
```

```
        he.isFrontier = true
        frontierEdges.add(he.index)
    else
        twinHe ← he_data.halfEdges[he.twin]
        isNotMaxEdgeInBothTriangles ← !he.isMax and !twinHe.isMax
        if isNotMaxEdgeInBothTriangles then
            he.isFrontier = true
            frontierEdges.add(he.index)
        end
    end
end
return frontierEdges
end
```

La Figura 23 presenta un ejemplo del etiquetado de *aristas frontera* mediante el Algoritmo 18 a partir de una malla triangular.

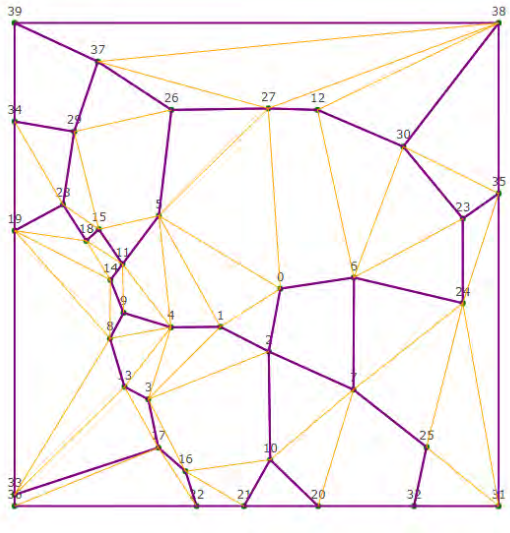


Figura 23. Etiquetado de aristas frontera en color purpura.

Posterior al etiquetado de *aristas frontera* el Algoritmo 19 procede a la identificación de aristas problemáticas que generar polígonos complejos. Estas aristas se identifican al contar la concurrencia de los vértices de destino de las aristas y luego identificando las aristas cuyo vértice de destino solo se repite una sola vez en la malla. Esta metodología también identifica aristas de contorno no problemáticas, las cuales se filtran en un paso posterior.

Algoritmo 19. Identificación de aristas a reparar (**label_to_repair**)

Inputs: ***he_data*** Estructura HE de la malla triangular, ***frontierEdges*** arreglo de aristas etiquetadas como frontera de polígonos

```
vertices ← [ ]
edges_to_repair ← [ ]
// Extraer los índices de los vértices de destino
foreach index of frontierEdges do
    edge ← he_data.halfEdges[index]
    vertices.push(edge.target.index)
    edges_to_repair.push(edge); // Guardar todas las aristas inicialmente
end
// Contar las ocurrencias de cada vértice
vertexCounts ← vertices.reduce((countMap, vertex) → {
    countMap[vertex] = (countMap[vertex] or 0) + 1
    return countMap
}
// Encontrar el valor mínimo de ocurrencias
minCount ← min(...Object.values(vertexCounts))
// Encontrar todos los vértices con ocurrencia igual a minCount
leastFrequentVertices ← Object.keys(vertexCounts) .filter((vertex)
                    → vertexCounts[vertex] = minCount)
// Filtrar las aristas que tienen cualquiera de los vértices menos frecuentes
edgesLeastFrequentVertices ← edges_to_repair.filter(
    (edge) → return leastFrequentVertices.includes(edge.target.index)
}
// Retornar todas las aristas que deben repararse
return edgesLeastFrequentVertices
end
```

La reparación de aristas se realiza utilizando Algoritmo 20. Este algoritmo agrega nuevas aristas al conjunto de aristas de frontera, de forma tal que la nueva arista une la arista problemática en el vértice de destino con la arista más larga que esté conectado a este vértice.

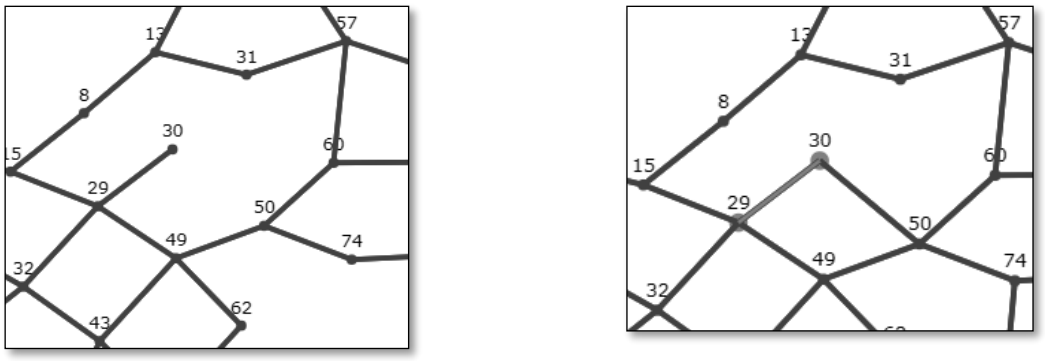
Algoritmo 20. Reparación de aristas problemáticas (**repair_edges**)

Inputs: ***he_data*** Estructura HE de la malla triangular, ***to_repair*** arreglo de aristas identificadas para su reparación, ***frontierEdges*** arreglo de aristas etiquetadas como frontera de polígonos

```
result ← frontierEdges
foreach problematicEdge of to_repair do
    // Identificar el vértice problemático
    problematicVertex ← problematicEdge.target
```

```
    // Obtener todas las aristas no de frontera conectadas a este vértice
    connectedFrontierEdges ← [ ]
        foreach he of he_data.halfEdges do
            if !he.isFrontier and (he.origin.index = problematicVertex.index or
            he.target.index = problematicVertex.index) then
                connectedFrontierEdges.add(he)
            end
    end
    // Seleccionar la arista más larga conectada al vértice problemático
    longestEdge ← connectedFrontierEdges[0]
    maxLength ← edge_length_squared(longestEdge)
    foreach he of connectedFrontierEdges do
        length ← edge_length_squared(he)
        if length > maxLength then
            longestEdge ← he
            maxLength ← length
        end
    end
    // Agregar la arista seleccionada y su gemela al conjunto de aristas de frontera
    result.add(longestEdge.index)
    if longestEdge.twin ≠ null then
        result.add(longestEdge.twin)
    end
    // Modificar la propiedad isFrontier para indicar que estas aristas ahora son fronteras
    he_data.halfEdges[longestEdge.index].isFrontier = true
    if longestEdge.twin ≠ null then
        he_data.halfEdges[longestEdge.twin].isFrontier = true
    end
end
return result // Retornar el estado de reparación
end
```

A modo de ejemplo la Figura 24 presenta en (a) la identificación de la arista problemática 29-30, mediante la identificación de la concurrencia unitaria del vértice terminal 30. En (b) se presenta la reparación del problema detectado en la malla, agregando una arista 30-50, lo que permite integrar correctamente dicha región a la malla y eliminar el problema asociado a la arista de frontera detectada.



(a) Identificación de aristas problemática 29-30.

(b) Reparación de arista problemática con nueva arista de frontera 30-50.

Figura 24. Identificación y reparación de aristas problemáticas.

El Algoritmo 21 construye polígonos cerrados a partir de las aristas fronteres. Se toma como argumento la malla con estructura HE y las aristas etiquetadas como frontera. Cada iteración recorre las aristas de cada triángulo identificando las aristas frontera y marcando como "visitada" las aristas utilizadas en la construcción de un polígono. Si un triángulo considera tres aristas frontera, se cierra el polígono y se inicia la construcción de uno nuevo. Si no se cumple esta condición, se continua con el análisis del triángulo adyacente. Una manera simple de garantizar que un polígono está cerrado es contar la ocurrencia de vértices de las aristas frontera seleccionadas en cada iteración, si todos los vértices se repiten dos veces, se considera un ciclo cerrado y por lo tanto un polígono completo.

Algoritmo 21. Creación de polígonos (**build_polygons**)

Inputs: ***he_data*** Estructura HE de la malla triangular, ***frontierEdges*** arreglo de aristas etiquetadas como frontera de polígonos

```
polygons ← [ ]

visitedEdges ← new Set()
foreach edgeIndex of frontierEdges do
    // Si la arista ya fue visitada no se revisa nuevamente
    if visitedEdges.has(edgeIndex)) return
    seedEdge ← he_data.halfEdges[edgeIndex]
    polygon ← [ ]
    vertices ← [ ]
    actual ← seedEdge
    while
        // Obtener las aristas del triángulo actual
        next ← he_data.halfEdges[actual.next]
```

```
        prev ← he_data.halfEdges[actual.prev]
        triangle ← [actual, next, prev]
        triangle_id = actual.origin.index + "-" + next.origin.index + "-"
                    + prev.origin.index
        seedTriangleEdges ← triangle.filter((edge) → edge.isFrontier)
        empty_edges ← triangle.filter((edge) → !edge.isFrontier)
        seed_id ← ""
        foreach he of seedTriangleEdges do
            seed_id += "[" + he.origin.index + "-" + he.target.index + "]"
            if !visitedEdges.has(he.index) then
                visitedEdges.add(he.index)
                polygon.push(he)
                vertices.push(he.origin.index, he.target.index)
            end
        end
        // Si ya se tiene tres aristas de frontera en el triángulo, el polígono está completo
        if seedTriangleEdges.length = 3 then
            break
        end
        // Contamos las ocurrencias de cada vértice
        vertexCounts ← vertices.reduce((countMap, vertex) → {
            countMap[vertex] = (countMap[vertex] or 0) + 1
            return countMap
        }
        allVerticesPaired ← Object.values(vertexCounts).every((count) → count = 2)
        if allVerticesPaired then
            break // El polígono está cerrado, se sale del bucle
        end
        // Pasar a un triángulo adyacente solo si la arista tiene una gemela y no es de frontera
        twinEdge = he_data.halfEdges[empty_edges[empty_edges.length − 1].twin]
        if twinEdge and !twinEdge.isFrontier and !visitedEdges.has(twinEdge.index) then
            actual ← twinEdge // Continuar por la gemela
        else
            // Si no hay más gemelas no frontera, se termina el polígono
            break
        end
    end
    polygons.push(polygon); // Añadir el polígono una vez construido
end
return polygons
end
```

Finalmente, el Algoritmo 22 aplica toda la metodología presente en esta sección para dar como resultado la malla poligonal utilizando los mismos vértices de la malla triangular original.

Algoritmo 22. Direct Polylla en malla de tipo IFS (**dpolylla**)

Inputs: ***polygons*** objeto IFS con coordenadas y conectividad.

```
he_data ← build_half_edge(polygons)
maxEdges ← label_max_edges(he_data)
frontierEdges ← label_frontier_eges(he_data, maxEdges)
to_repair ← label_to_repair(he_data, frontierEdges)
frontierEdges ← repair_edges(he_data, to_repair, frontierEdges)
polygons ← build_polygons(he_data, frontierEdges)
return polygons
end
```

3.3.2 Algoritmo de refinamiento de malla

El algoritmo de refinamiento tiene un enfoque nodal donde el usuario selecciona los nodos de la malla mediante un cuadro de selección generado al arrastrar el mouse (ver Figura 25 y Figura 26). Los nodos seleccionados se tintan de color verde al ser seleccionados. Se permite selección multiple presionando *Ctrl* y arrastrando el mouse.

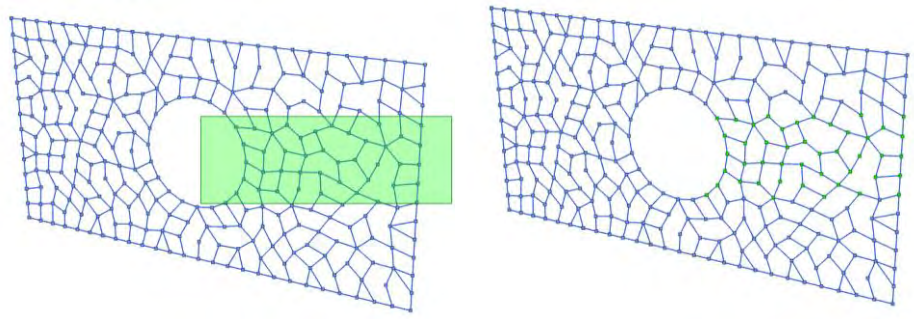


Figura 25. Selección para refinamiento de malla

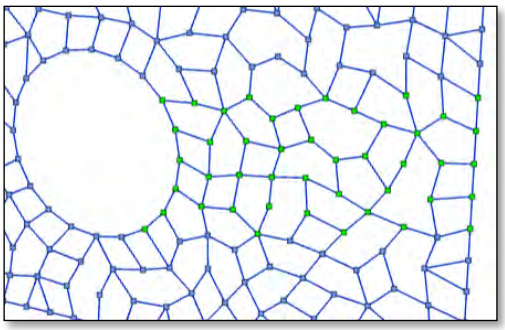


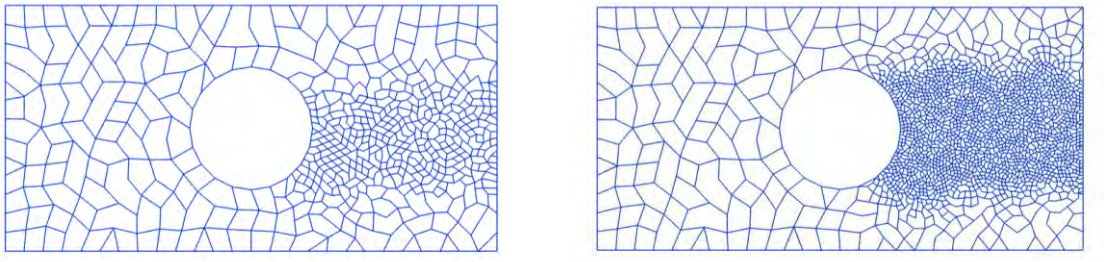
Figura 26. Detalle de nodos seleccionados para el refinamiento de la malla

El formulario de refinamiento presente en la Figura 27, permite definir el tamaño del elemento refinado, las iteraciones de relajación de las regiones de refinamiento y las iteraciones de relajación global. Éste último parámetro permite generar cierta transición entre elementos refinados y los elementos originales de la malla al relajar las posiciones de los nodos en toda la malla.

Figura 27. Formulario de refinamiento de malla y resultado del refinamiento

El algoritmo de refinamiento se puede aplicar a los tres tipos de malla ya se triangular, poligonal Voronoi y poligonal D-Polylla debido a que el refinamiento actúa sobre los nodos, no sobre los elementos. El comportamiento es estable para refinamientos de más de 2000 elementos.

En la Figura 28 se puede comparar dos mallas generadas sobre una misma malla base de tipo D-Polylla en ambos casos tanto para las mallas de 500 elementos, como para la malla de 2000 elementos, se aprecia que sólo los elementos con nodos seleccionados se ven afectados por el refinamiento, junto con los elementos colindantes a las regiones seleccionadas principalmente afectadas por la relajación global.



(a) Malla refinada de 500 elementos. (b) Malla refinada de 2000 elementos.

Figura 28. Ejemplo de malla refinada

El Algoritmo 23 simplificado, agrupa los polígonos que contienen los vértices seleccionados en clústeres, definidos como conjuntos de polígonos conectados por vértices compartidos basado en la conectividad de la malla. A partir de la estructura HE, se extrae el contorno exterior de cada cluster identificando sus aristas de borde. Posteriormente, se eliminan los vértices de los polígonos dentro de los clusters, conservando los vértices del resto de la malla original. Se generan nuevos vértices en estas regiones aplicando la técnica de mallado descrita en la sección 3.3, que toma como entrada el contorno del cluster, la dimensión del elemento, y el número de iteraciones de relajación.

Para lograr una transición suave entre la malla original y las regiones refinadas, se aplica una relajación global a la malla completa con un parámetro de relajación más bajo que el usado en el refinamiento con el fin de evitar una dispersión excesiva de los nodos en las zonas refinadas.

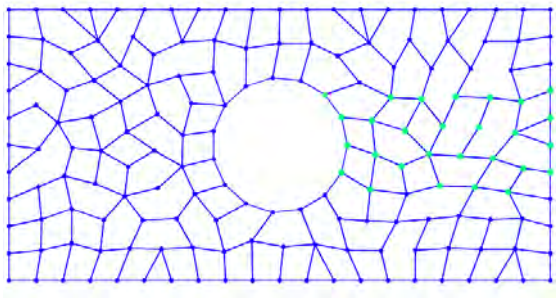
Algoritmo 23. Refinamiento de malla (**refinamiento**)

Inputs: ***original_mesh*** arreglo de datos e información de malla original, ***p_select_vertex*** vértices de la malla seleccionados para el refinamiento, *dr* distancia nominal de refinamiento, *iter_r* número de iteraciones de relajación del refinamiento, *iter_g* número de iteración de relajación global

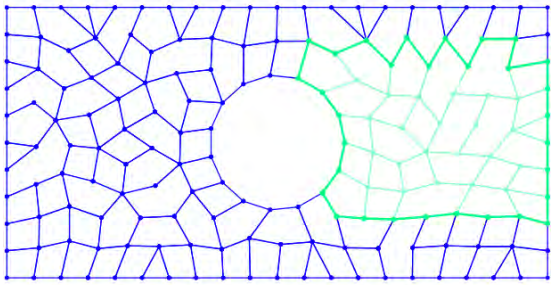
```
original_coords ← original_mesh.coords
original_connect ← original_mesh.connect
original_he ← original_mesh.he_data
original_contorno ← original_mesh.contorno
original_hoyos ← original_mesh.hoyos
// Encontrar los índices y coordenadas de vértices y polígonos que incluyen los vértices seleccionados.
p_id ← get_polygons_from_vertices(original_connect, p_select_vertex)
p_connect ← original_connect.filter((index) → p_id.includes(index))
points_to_remove ← new Set(p_connect.flat())
filtered_original_coords ← original_coords.filter((index)
                → !points_to_remove.has(index))
// Conformar los grupos y obtener los contornos
groups ← group_polygons(p_connect, p_id)
contours ← countours_from_HE(original_he, groups)
new_points ← [ ]
// Generación de puntos de cada clúster
for contour of contours do
    new_points.push(// ...aplica procedimiento sección 3.3 hasta punto 6 para generar puntos
    dentro de cluster. Utiliza como argumento contour , iter_r y dr)
end
// Eliminación de puntos antiguos y generación de nuevos.
puntos_new ← new_points.flatMap((p) → p.centros)
puntos_new ← [...filtered_original_coords, ...puntos_new]
```

```
    refined_mesh ← lloyd_relaxation(puntos_new, filtered_original_coords, iter_g, ...)
    return refined_mesh
end
```

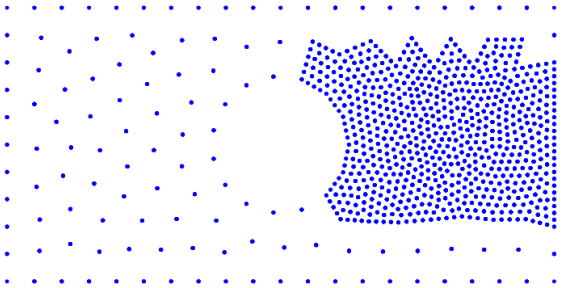
La Figura 29 presenta un ejemplo que detalla el paso a paso del Algoritmo 23.



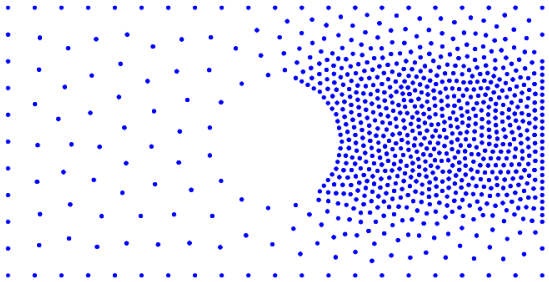
(a) Identificación de nodos selección de nodos



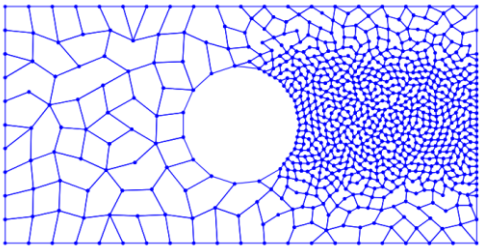
(b) Agrupación de elementos involucrados en la selección e identificación de contornos de clúster



(c) Aplica sección 3.3 hasta punto 6 para generar vértices dentro de clúster.



(d) Relajación de Lloyd aplicada a todo el dominio.



(e) Aplicación de algoritmo Direct Polylla en malla de puntos relajadas.

Figura 29. Representación gráfica del refinamiento

El Algoritmo 34 (Anexo B) presenta la metodología para obtener los polígonos de los vértices seleccionados, el Algoritmo 25 permite la agrupación de estos polígonos y finalmente

con el Algoritmo 35 (Anexo B) se puede extraer el contorno de los grupos para su posterior refinación.

3.4 Remallado Fondecyt N° 1221325

Se desarrolla algoritmo de remallado de acuerdo con los lineamientos definidos en el proyecto Fondecyt N° 1221325 "Development and Assessment a Combined Nodal Integration and Virtual Element Method for Small and Large Deformation Solid Mechanics Simulations".

El proyecto exige un remallado localizado que mantenga la posición espacial de los nodos en todo momento, lo que a su vez obliga, a que el remallado únicamente modifique la reorganización de conectividad de los elementos que conforman la malla.

Para ejecutar de manera local el remallado, es necesario identificar los elementos de baja calidad que limitan el rendimiento en la resolución de problemas en términos de error de aproximación y tasa de convergencia mediante el uso de MEV.

3.4.1 Algoritmo de remallado Poly Remesh (P-Remesh)

El algoritmo fue diseñado con el fin de reemplazar la conectividad de los polígonos mal formados utilizando los vértices propios del elemento, junto con los vértices de la vecindad, de esta manera la formación de nuevos polígonos se realizará considerando nuevas posibles configuraciones de polígonos con una amplia cantidad de vértices.

El algoritmo sólo es viable sobre una malla base de tipo D-Polylla, en ningún caso se podrá utilizar otro algoritmo de mallado poligonal de tipo Voronoi.

Al igual que en [17] se considera el parámetro APR como métrica de calidad de malla para la identificación de polígonos mal formados, considerando un umbral a definir por el usuario. Los polígonos que no cumplen el umbral se marcan como problemáticos, guardando sus índices, nodos involucrados y coordenadas. El Algoritmo 24 aplica el cálculo del APR y compara su resultado con el umbral en cada polígono de la malla.

Algoritmo 24. Filtros polígonos problemáticos (**filter_problematic_polygons**)

Inputs: ***connect*** matriz de conectividad de la malla de tipo D-Polylla, ***coords*** arreglo de coordenadas de la malla, *quality_umbral* umbral para determinar con APR si polígono tiene mal formación.

```
problematicPolygons ← {
    polygonIndices: [ ],
    nodeIndices: new Set( ), // Índices globales de nodos en los polígonos problemáticos
    mesh: [ ] // Coordenadas de los polígonos problemáticos
```

```
    }
    foreach polygonNodeIndices, polygonIndex of connect do
        polygonCoords ← polygonNodeIndices.map(nodeIndex → coords[nodeIndex])
        APR ← calc_APR(polygonCoords)
        if APR < quality_umbral then
            // Agregar el índice del polígono a la lista
            problematicPolygons.polygonIndices.push(polygonIndex)
            // Agregar los índices de nodos al conjunto
            for nodeIndex in polygonNodeIndices do
                problematicPolygons.nodeIndices.add(nodeIndex)
            end
            // Agregar las coordenadas del polígono a 'mesh'
            problematicPolygons.mesh.push(polygonCoords)
        end
    end
    return problematicPolygons
end
```

Luego de la detección de polígonos en conjunto con su información geométrica se procede a obtener los polígonos vecinos que comparten al menos dos vértices. Definir que los polígonos vecinos deben compartir dos vértices puede ser un criterio discutible, dado que en ciertas circunstancias dentro de la simulación se requiere mayor flexibilidad en el remallado donde se requerirá una mayor cantidad de vértices para la formación de nuevos polígonos. Si se considera dentro del remallado la vecindad de elementos que comparten solo un vértice con el polígono problemático, la cantidad de vértices disponibles para el remallado aumentaría considerablemente. Una mayor cantidad de vértices implica una mayor posibilidad de reconfigurar polígonos no problemáticos, pero aumenta el costo computacional y tiempo de procesamiento.

Luego de la identificación de los elementos problemáticos y su vecindad, se procede a la formación de grupos (ver Algoritmo 25). y detección de contornos de estos (ver Algoritmo 35 Anexo B). En esta parte de la metodología el algoritmo de remallado P-Remesh comparte algunas similitudes con el algoritmo de refinamiento desarrollado en 3.3.2, donde se obtiene el contorno a partir de los grupos de polígonos.

Es importante mencionar que, para la búsqueda de las conexiones entre polígonos, se utiliza el algoritmo de búsqueda de amplitud, también conocida por sus siglas en inglés como BFS [29] *breadth first search.* Esta metodología asegura que se recorran todos los elementos que están conectados directa o indirectamente al polígono inicial.

Algoritmo 25. Agrupación de polígonos (**group_polygons**)

Inputs: ***connect*** matriz de conectividad de la malla de tipo D-Polylla, ***polygonIDs*** identidad de polígonos en la matriz de conectividad.

```
n_polygons ← connect.length
polygon_vertices ← { }
// Mapear cada polígono a su conjunto de vértices
for idx ← 0 to idx < n_polygons do
    polygon ← connect[idx]
    polygonID ← polygonIDs[idx]
    polygon_vertices[polygonID] = new Set(polygon)
end
// Construir la lista de conexiones por dos o más vértices compartidos
conexiones ← { }
foreach polygonID of polygonIDs do
    conexiones[polygonID] = new Set( )
end
// Buscar conexiones entre polígonos comparando cada par de polígonos, verificando que comparten 2 o más vértices.
for i = 0 to i < n_polygons do // Recorre todos los polígonos uno por uno
    polygonID_i ← polygonIDs[i]
    vertices_i ← polygon_vertices[polygonID_i]
    // Compara el polígono i con todos los siguientes sin repetir. Compara i/j pero no j/i.
    for j = i + 1 to j < n_polygons do
        polygonID_j ← polygonIDs[j]
        vertices_j ← polygon_vertices[polygonID_j]
        // Contar los vértices compartidos
        sharedVerticesCount ← 0
        foreach v of vertices_i do
            if vertices_j.has(v) then
                sharedVerticesCount ← sharedVerticesCount + 1
            end
        end
        if sharedVerticesCount ≥ 2 then
            // Añadir una conexión bidireccional
            conexiones[polygonID_i].add(polygonID_j)
            conexiones[polygonID_j].add(polygonID_i)
        end
    end
end
// Encontrar los componentes conectados, búsqueda de amplitud BFS
visited ← new Set()
groups ← [ ]
polygonIDToGroup ← { }
foreach polygonID of polygonIDs do
```

```
        if !visited.has(polygonID) then
            group ← [ ]
            cola ← [polygonID]
            visited.add(polygonID)
            while cola.length > 0 do // Mientras la cola no esté vacía, sigue la recursión
                current ← cola.shift() // Primer polígono
                group.push(current)
                polygonIDToGroup[current] = group // Mapear polígono al grupo
                foreach vecino of conexiones[current] do
                    if !visited.has(vecino) then
                        visited.add(vecino)
                        cola.push(vecino) // Se agrega a la cola para explorarlo después.
                    end
                end
            end
            groups.push(group)
        end
    end
    // Remover duplicados en los grupos
    foreach group of groups do
        uniquePolygons ← [...new Set(group)]
        group.length = 0
        group.push(...uniquePolygons)
    end
    return groups
end
```

La función de la obtención de los grupos antes de la eliminación de la conectividad permite limitar el remallado únicamente a los sectores problemáticos, aplicando el algoritmo D-Polylla de forma unitaria en cada grupo, de esta forma el costo computacional de remalladlo es mucho menor que si se tuviese que regenerar la malla completa.

A partir de la conectividad original, sus coordenadas, la información de los grupos de polígonos a remallar y la conectividad de los elementos no conflictivos se procede mediante el Algoritmo 26 a reconstruir las zonas problemáticas generado una nueva conectividad y manteniendo intactas las demás regiones.

En una primera instancia el Algoritmo 26, ya con la determinación de contorno de cada grupo de polígonos, procede al cálculo del área, con el fin de identificar al contorno exterior y las perforaciones, es decir, el contorno de mayor área identificado corresponde al contorno de exterior del conjunto de polígonos, las demás regiones identificadas corresponderían a perforaciones internas del mismo conjunto.

Ya con las perforaciones y contorno exterior identificado se procede a aplicar el algoritmo de mallado D-Polylla sobre cada conjunto problemático. Finalmente se une la conectividad de los nuevos polígonos generados en las zonas problemáticas, con la conectividad de los polígonos en las regiones estables de la malla.

Es importante entender que la triangulación de Delaunay utilizada en el algoritmo D-Polylla, maximiza el ángulo mínimo de los triángulos, garantizando una distribución uniforma y estable de triángulos, evitando la formación de triángulos alargados o con ángulos agudos [9], lo cual es deseable para su posterior uso en la conformación de polígonos de la malla cuyo APR permita su procesamiento en el cálculo numérico.

Por otro lado, la definición del contorno exterior y perforaciones de los grupos permite filtrar triángulos, conservando únicamente aquellos cuyo centroide está dentro del contorno circunscrito de la geometría. Este último punto es muy relevante, dado que la triangulación tiene como entrada únicamente los vértices, no discriminando entre triángulos dentro o fuera del dominio.

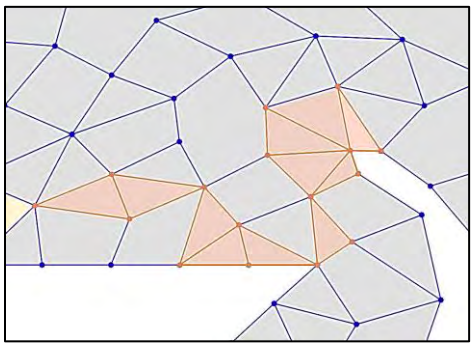
Algoritmo 26. Remallado para malla de tipo D-Polylla (**polyremesh**)

Inputs: ***connect*** matriz de conectividad de la malla original de tipo D-Polylla, ***coords*** arreglo de coordenadas de la malla, ***groups_info*** grupos de polígonos problemáticos, ***aux_connect***

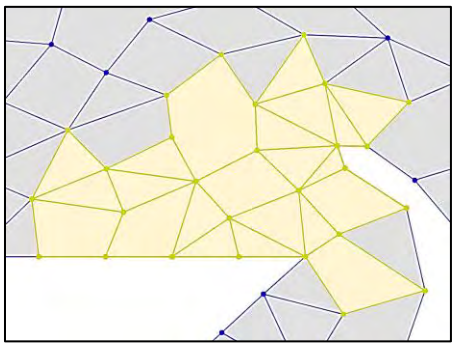
```
newConnect ← aux_connect
indicesToRemove ← new Set()
Triangles ← [ ]
plotData ← [ ]
// Calcular el área de las regiones identificadas de cada grupo
foreach (grupo, i) of groups_info do
    grupos_connect ← grupo.group.map((idPoligono) → connect[idPoligono])
    contoursWithAreas ← grupo.contoursCoordinates.map(contourCoords → {
      area ← polyarea(contourCoords)
      return { contourCoords, area}
    })
    // Ordenar los contornos por área absoluta en orden descendente identificar contorno exterior de perforaciones
    contoursWithAreas.sort((a, b) → abs(b.area) − abs(a.area))
    contorno_grupo_coords ← contoursWithAreas[0].contourCoords
    holes ← contoursWithAreas.slice(1).map((item) → item.contourCoords)
    // Filtrar triángulos fuera del contorno exterior y dentro de las perforaciones
    polygonNodeIndices ← Array.from(new Set(grupos_connect.flat()))
    polygonCoords ← polygonNodeIndices.map((nodeIndex) → coords[nodeIndex] )
    allTriangles ← triangulation(polygonCoords)
    filteredTriangles ← allTriangles.filter((triangle) → {
        centroid ← compute_centroid(triangle)
        return is_centroid_valid(centroid, contorno_grupo_coords, holes)
```

```
        })
        meshDataTriangles = extract_mesh(filteredTriangles)
        meshData
        ← dpolylla(meshDataTriangles.coords, meshDataTriangles.connect)
        polygons_connect
        ← to_global_connect(coords, meshData.coords, meshData.connect)
        newConnect.push(...polygons_connect)
    end
    return newConnect
end
```

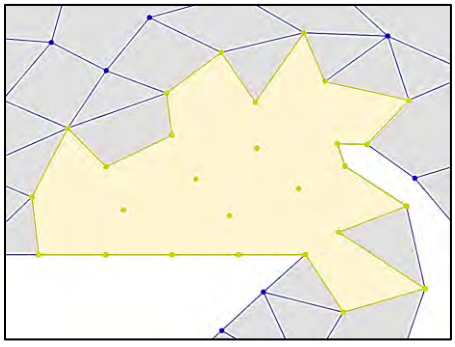
La Figura 30 presenta un ejemplo que detalla el paso a paso del algoritmo de remallado Poly Remesh.



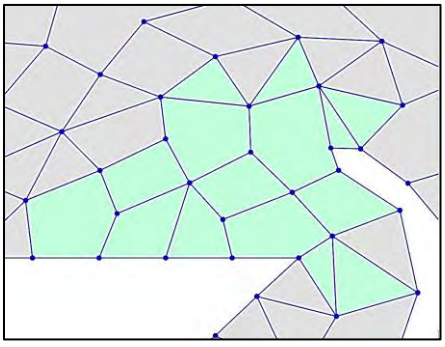
(a) Identificación de polígonos problemáticos según su APR



(b) Identificación y agrupación de vecindad de polígonos que comparten aristas con polígonos problemáticos.



(c) Identificación de contorno y eliminación de conectividad de grupos de polígonos.



(d) Aplicación de algoritmo Direct Polylla limitado al grupo de polígonos.

Figura 30. Representación gráfica del remallado Poly Remesh

La Figura 31 presenta las herramientas desarrolladas para la configuración y cálculo del remallado.

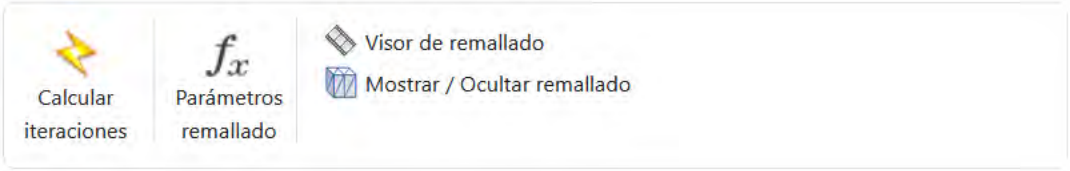


Figura 31. Herramientas para el remallado

Se establece la posibilidad de seleccionar el factor de calidad donde se considerarán elementos a remallar los polígonos que tengan un APR menor a lo indicado en el formulario (ver Figura 32). Adicionalmente, se puede establecer la cantidad de iteración a utilizar para la animación.

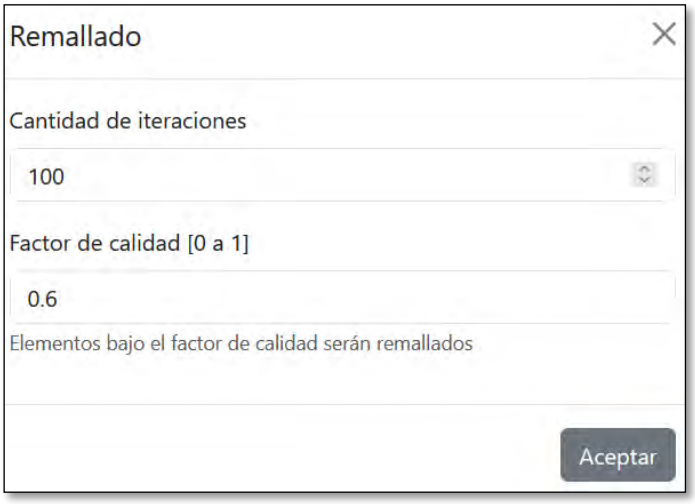


Figura 32. Modal de opciones de remallado.

La ventana de animación presente en la Figura 33 permite visualizar de manera dinámica los cálculos realizados en la iteración. Esta animación permite identificar los elementos que fueron detectados en la iteración anterior como problemáticos en color naranjo y su vecindad en amarillo.

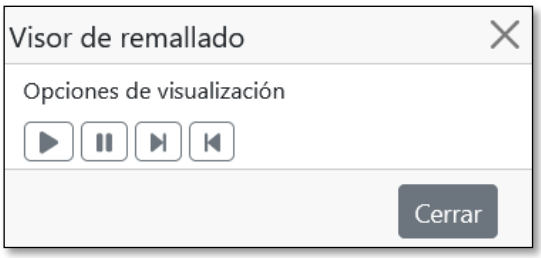


Figura 33. Visor de animación de remallado.

A modo de ejemplo la Figura 34 presenta la deformación de un cuerpo durante el proceso de iteración, donde los elementos problemáticos y su vecindad rompen su conectividad para luego reorganizarse y formar nuevos polígonos.

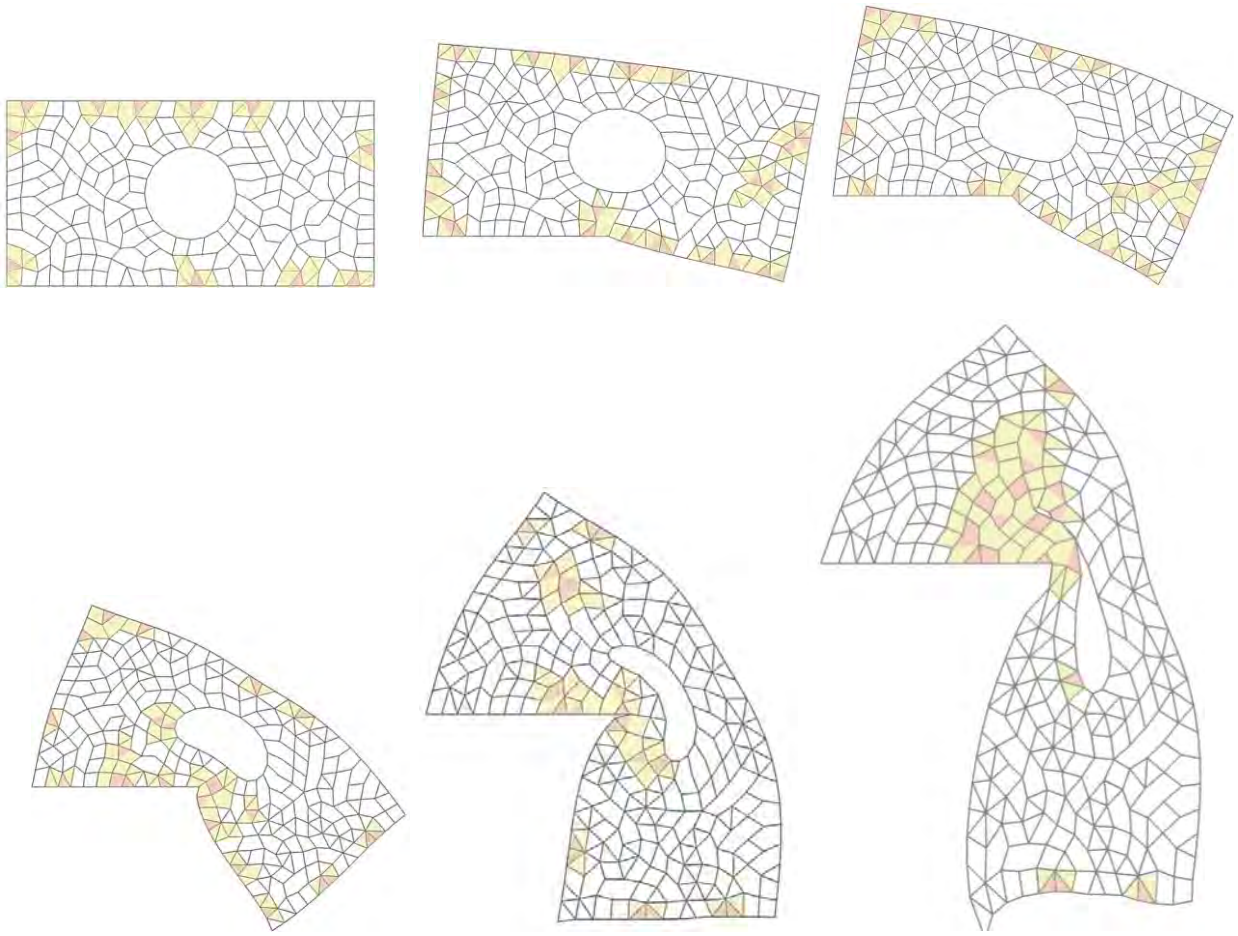


Figura 34. Animación bajo remallado algoritmo P-Remesh

3.5 Condiciones de contorno

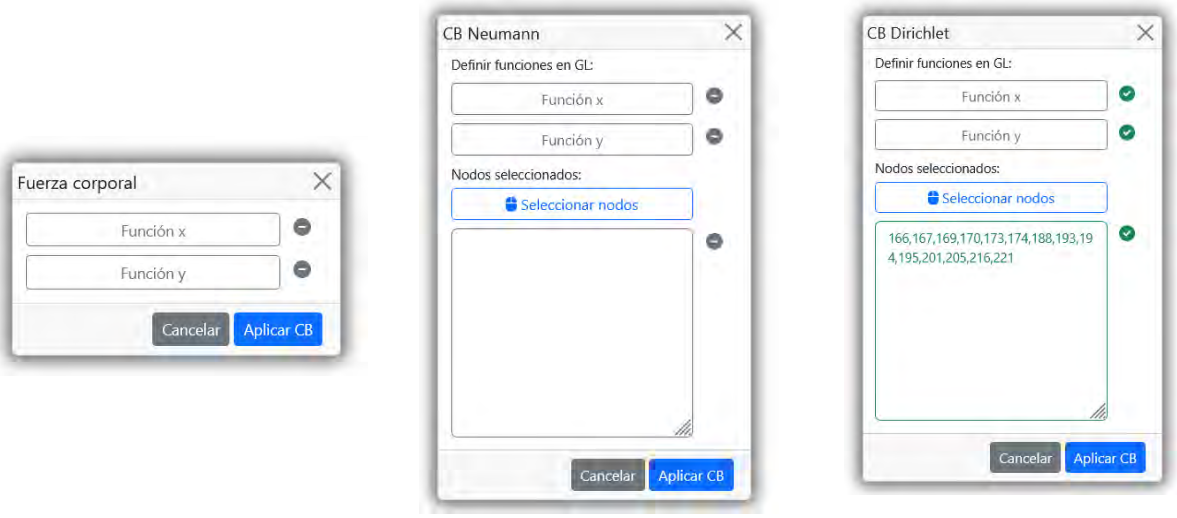
El modelo de asignación de condiciones de contorno en VEMLAB 2.4 se realiza a partir de la identificación de los nodos de contorno, a los cuales se les asigna una función que limita o modifica el comportamiento de los grados de libertada de dichos nodos durante el cálculo del problema.

La asignación de condiciones de borde en software MECHAIDE se realiza de la misma manera, pero con la posibilidad de seleccionar los nodos, a los cuales se les aplicará la condición de contorno seleccionada ya sea de Neumann, Dirichlet o de fuerza corporal.

La Figura 35 presenta las herramientas que permiten crear una nueva condición de contorno.

Figura 35. Herramientas condiciones de contorno

Cada icono al ser presionado abre los diferentes formularios presentes en la Figura 36 los cuales permiten asignar la función para cada grado de libertad y la selección de nodos utilizados en el entorno de visualización. Particularmente la Figura 36(a) no tiene opción para selección de nodos, dado que la condición de fuerza corporal se aplica a todos los nodos de la malla.



(a) Fuerza corporal (b) CB Neumann (c) CB Dirichlet

Figura 36. Formularios condiciones de contorno.

En la configuración de las funciones presentada en la Figura 37 se pueden utilizar las variables x e y correspondientes a la posición unitaria de los nodos seleccionados, también se puede acceder al arreglo de posiciones de los nodos seleccionados denominados como ***arr_x*** y ***arr_y***. Adicionalmente, es posible acceder a las propiedades del material mediante la llamada al objeto ***matProps*** , el cual tiene la información respecto al tipo de comportamiento elástico *plane_state* ya sea esfuerzo plano y deformación plana, el módulo de elasticidad E_y, el coeficeinte de Poisson nu, la matriz constitutiva para un material lineal elástico $\boldsymbol{D}$, entre otros parámetros constitutivos del material seleccionado.

Figura 37. Configuración de funciones de condiciones de contorno

Una vez asignadas las condiciones de contorno, el entorno de visualización asigna un número y un color aleatorio a los vértices, con el fin de identificar la condición utilizada en los nodos seleccionados. (ver Figura 38).

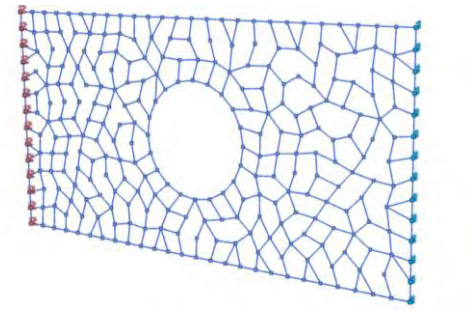


Figura 38. CB en el entorno de visualización.

3.6 Solucionador

El código presente en VEMLAB 2.4 contiene toda la metodología de cálculo para dar solución a problemas estáticos de mecánica de sólidos de elasticidad lineal en dos dimensiones, utilizando MEF y MEF.

Cabe destacar que la estructura de código de VEMLAB comprende varias características no implementadas en MECHAIDE, tales como la factibilidad de cálculo mediante matrices

dispersas o la opción de dar solución mediante la ecuación de Poisson. Las limitaciones del desarrollo actual se especifican de forma detallada en los objetivos y alcances definidos en la sección 1.3 del presente documento.

3.6.1 Dificultades de la adaptación

La principal complejidad que implica la adaptación del código de Matlab a JavaScript es el desarrollo o búsqueda de funciones de operación matricial presentes de forma nativa en Matlab, las cuales no están disponibles para JavaScript, ni existen como librerías externas para dicho lenguaje. Si bien Matlab es un lenguaje con enfoque para cálculos numéricos y matrices para científicos e ingenieros [30], JavaScript esta netamente optimizado para la manipulación del DOM [31], por esta razón en muchas ocasiones se desarrollaron estrategias y algoritmos iterativos no optimizados, implementados para solventar en parte estas operaciones matriciales.

Otro aspecto importante que mencionar en la adaptación del código de Matlab a JavaScript corresponde al número de inicio en índice en arreglos. En Matlab el índice de matrices o arreglos empieza desde uno, mientras que JavaScript el índice empieza desde cero. Aunque a simple vista es diferencia menor, en el contexto de una adaptación de código complejo, donde los cálculos recursivos se encuentran en prácticamente todas las funciones, una conversión incorrecta puede generar errores difíciles de detectar.

3.6.2 Validación del algoritmo de VEMLAB

Se establece el Algoritmo 27 el cual permitió enfocar los esfuerzos de desarrollo sólo en las funciones estrictamente necesarias, reutilizando código ya existente disponible en internet y validando los algoritmos generados en el proceso.

Algoritmo 27. Validación de adaptación desde Matlab a JavaScript

Input: *CodeMatlab* código de la función en Matlab, ***Example*** arreglo con datos de entrada de la función
Copiar *CodeMatlab* al entorno de desarrollo de JavaScript
CodeJS ← Adaptar variables y recursiones de *CodeMatlab*
A:
if ¿*CodeJS* tiene operación no nativa? **then**
Buscar operación en librerías externas disponibles
if ¿se encontró operación? **then**
Operación ← Operación de librería disponible
goto ***B***
else

```
        Buscar operación en librerías disponibles en la web
        if ¿se encontró operación? then
            Operación ← Operación de librería disponible en la web
            goto B
        else
            Operación ← Desarrollar operación
            goto B
        end
    end
  end
end A
B:
  CodeJS ← CodeJS + Operación
  resultMatlab ← Ejecutar CodeMatlab con datos de Example
  resultJS ← Ejecutar CodeJS con datos de Example
  if resultMatlab ! = resultJS then
    goto A
  end
end B
return
end
```

3.6.3 Adaptación de algoritmos de inversión de matrices

Los algoritmos de inversión de matriz presentes en MECHAIDE se detallan en la sección 2.5 del presente documento. Adicionalmente se agregan alternativas de inversión de matriz nativas en librerías matemáticas externas, como la presente en la librería *Numeric.js* [32].

3.6.4 Estructura de la solución

Se desarrolla una estructura global de solución basada en VEMLAB donde se puede especificar las características del cálculo requeridos, tales como el tipo de elasticidad, algoritmo de inversión de matriz, método de cálculo ya sea MEV o MEF, estabilidad y escala de deformación. Adicionalmente se le permite al usuario seleccionar las soluciones disponibles para su visualización, dentro de las cuales está disponible el desplazamiento, deformación, esfuerzo de von Mises, entre otras. La Figura 40 presenta el formulario con las opciones mencionadas anteriormente, el cual es disponible en el menú de herramientas del solucionador presentado en la Figura 39.

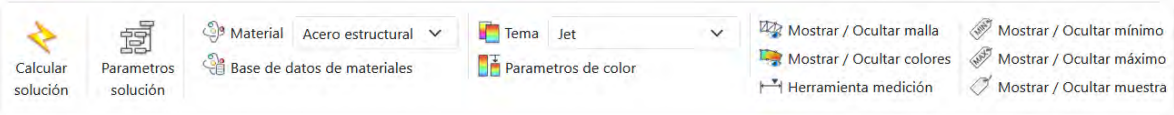


Figura 39. Herramientas del solucionador

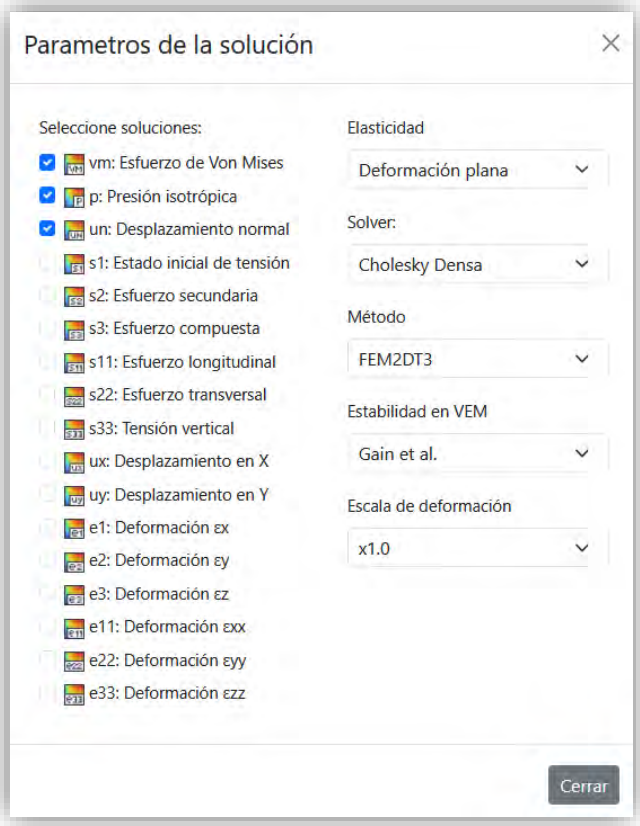


Figura 40. Parámetros de configuración del solucionador

La función desarrollada que toma la parametrización del cálculo definida por el usuario es capaz de capturar los argumentos y adaptarse a las diferentes características, permitiendo sin importar el método, tipo de elasticidad o método de inversión de matriz, entregar el resultado acorde a las condiciones establecidas.

En términos generales el Algoritmo 28 basado en la metodología de VEMLAB permite la simulación numérica del problema de elasticidad lineal en dos dimensiones ya sea mediante MEF o MEV, resolviendo la matriz de rigidez con sus condiciones de borde y entregando como resultado los desplazamientos, tensiones y deformaciones en la interfaz gráfica.

Algoritmo 28. Calcular problema (**calc**)

Inputs: ***material*** Tipo de material seleccionado, ***mesh*** Información de la malla poligonal o malla triangular, *plane_state* comportamiento elástico, *solver* tipo de algoritmo de inversión de matriz, *stability_type* tipo de estabilidad para MEV, *method* algoritmo de solución ya sea MEV o MEF, *scale* Escala de deformación de la solución, ***node_info*** Información de nodos seleccionados respecto a sus funciones y tipo de condición de borde.

```
// Propiedades del material
Ey ← parseFloat(material.E_GPa)
nu ← parseFloat(material.v)
matProps ← material_parameters_linelast(Ey, nu, plane_state, method)
lam ← matProps.lam
// Condiciones de borde
config ← {module, method, stability_type}
dirichlet_nodes ← node_info.all_dir_nodes
neumann_nodes ← node_info.all_neu_nodes
dirichlet_fun_values ← create_dirichlet_functions(node_info)
neumann_fun_values ← create_neumann_functions(node_info)
body_force_fun_values ← create_body_force_functions(node_info)
domainMesh ← format_mesh(mesh, neumann_nodes, dirichlet_nodes)
// Ensamble matriz de rigidez y vector de fuerzas nodales
        {K_global, f_global}
        = assembly(domainMesh, config, matProps, body_force_fun_values))
// Aplicar sobre cada nodo seleccionado su condición de borde de tipo Neumann
Neumann_BCs
← compute_Neumann_BCs(domainMesh, config, domainMesh.neumann_nodes, do
f_global ← sumar_vector_vectord(f_global, Neumann_BCs)
// Aplicar condición de borde de Dirichlet en cada nodo seleccionado
dirichlet_dofs ← domainMesh.dirichlet_dofs
u_nodal_sol ← new Array(2 · domainMesh.coords.length).fill(0)
DB_dofs
← compute_Dirichlet_BCs(domainMesh, config, dirichlet_nodes, dirichlet_dofs, diri
foreach (index, i) of DB_dofs.indexes do
    u_nodal_sol[index] = DB_dofs.values[i]
end
f_global ← subtract_sparse_vector(f_global, mult_sparse_vector(K_global, u_nodal_sol))
// Filtrar grados de libertad libres
num_nodes ← domainMesh.coords.length
free_dofs ← Array.from(0 : 2 · num_nodes − 1).filter(x
                → !DB_dofs.indexes.includes(x))
// Calcular solución
A ← filter_sparse(K_global, free_dofs)
b ← filter_vector(f_global, free_dofs)
solution_aux ← solve_options(solver, A, b)
```

```
    solution ← rellenar_soluciones(solution_aux, free_dofs, 2 · num_nodes)
    // Asigna los valores de Dirichlet a la solución completa
    foreach (index, i) of DB_dofs.indexes do
        solution[index] = DB_dofs.values[i]
    end
    // Reconstruir malla con la solución
    nodes ← domainMesh.coords
    nodesNumber ← nodes.length
    polygons ← domainMesh.connect
    u_x ← solution.filter((_, i) →  i % 2 = 0)
    u_y ← solution.filter((_, i) →  i % 2 ≠ 0)
    u_n ← u_x.map((x, i) →  sqrt(x^2  +  u_y[i]^2))
    nodes_dispx ← nodes.map(([x, y], i) → {
        return [x + u_x[i] · scale , y]
    })
    nodes_dispy ← nodes.map(([x, y], i) → {
        return [x, y +  u_y[i] · scale]
    })
    nodes_deformed ← nodes.map(([x, y], i) → {
        return [x +  u_x[i] · scale, y + u_y[i] · scale]
    })
    deformedPolygons ← polygons.map(polygon → {
        return polygon.map(nodeIndex → nodes_deformed[nodeIndex])
    })
    // Calcular tensiones y deformaciones
    if method = "FEM" then
        {stress, strain} ← fem_stress_and_strain (solution, domainMesh, matProps)
    else if vemlab_method = "VEM" then
        {stress, strain} ← vem_stress_and_strain (solution, domainMesh, matProps)
    end
end
```

La implementación completa de los algoritmos utilizados en el Algoritmo 28 del solucionador basada en VEMLAB, está disponible en la sección Anexo C desde el Algoritmo 44 al Algoritmo 77.

La Figura 41 y Figura 42 presentan algunos ejemplos de los problemas con resolución, tanto para esfuerzo de von Mises como para el desplazamiento normal de forma respectiva.

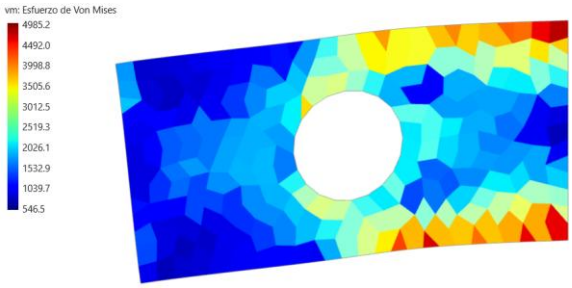


Figura 41. Visualización de resultados, esfuerzo de von Mises

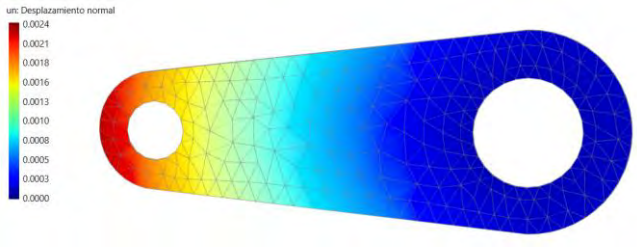


Figura 42. Visualización de resultados, desplazamiento normal.

3.7 Base de datos de materiales

Se establece una tabla en la base de datos, que asociada al usuario que permite guardar diferentes materiales asignado el módulo de elasticidad y el coeficiente de Poisson. El la Figura 43 se presenta el formulario que muestra los diferentes materiales, los cuales se pueden eliminar o seleccionar para ser utilizados en la solución del problema.

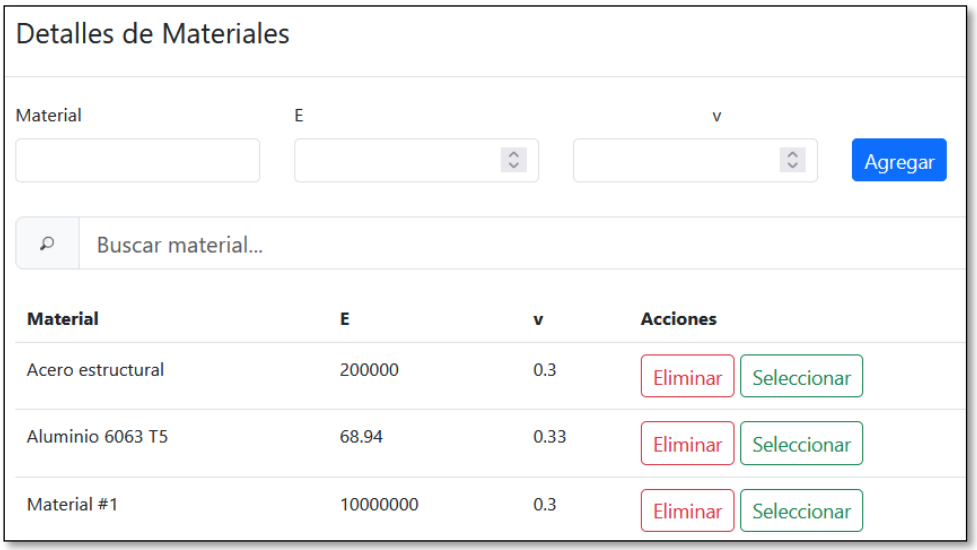


Figura 43. Formulario de materiales

3.8 Mapas multicolores

Basado en la estructura de colores RGB de los mapas la librería Matplotlib para Python [33], se generaron dieciocho alternativas de esquemas de color para la aplicación. La Figura 44 muestra tres de las posibles alternativas disponibles a seleccionar.

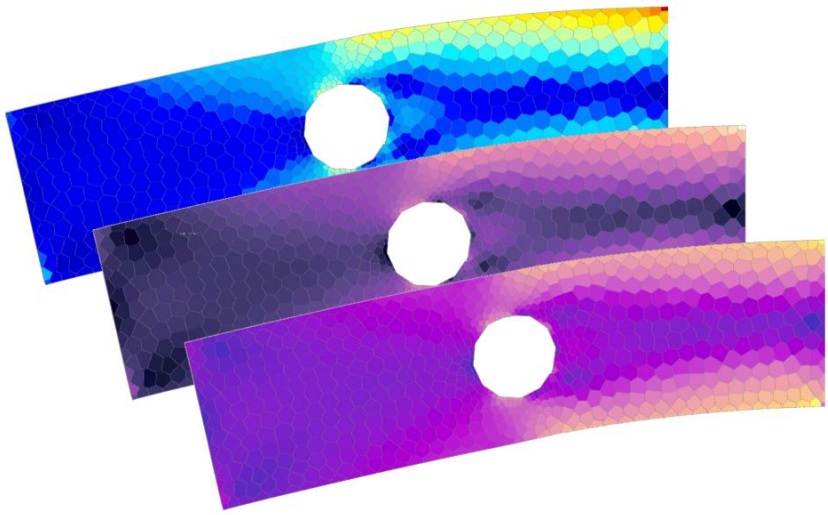


Figura 44. Mapas multicolores

A través del formulario de configuración presente en la Figura 45, el usuario puede seleccionar el rango aplicable en relación con los límites de color del mapa. Adicionalmente, es posible definir la cantidad de divisiones o marcas de referencia en la escala de colores.

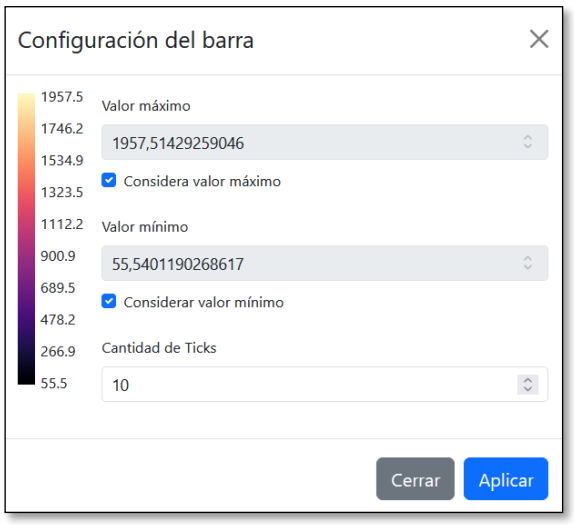


Figura 45. Formulario de configuración de color

3.9 Herramienta de medición de geometría

Se desarrolla una herramienta que permite medir la distancia entre dos puntos dentro de un plano bidimensional. El usuario puede seleccionar un primer punto y luego un segundo punto en el mismo plano para medir deformaciones o dimensiones de la geometría.

Adicionalmente, se desarrolla el "magnetismo" hacia los nodos. Esto significa que el usuario al seleccionar un punto cercano a un nodo, se toma la posición del nodo en lugar de la posición seleccionada. Esta funcionalidad se puede hacer mapeando la posición de los nodos y comparando con la posición seleccionada.

El usuario puede seleccionar el magnetismo ya sea para los nodos de la malla original, los nodos de la malla calculada o los nodos de la malla generada en la iteración del remallado (ver Figura 46).

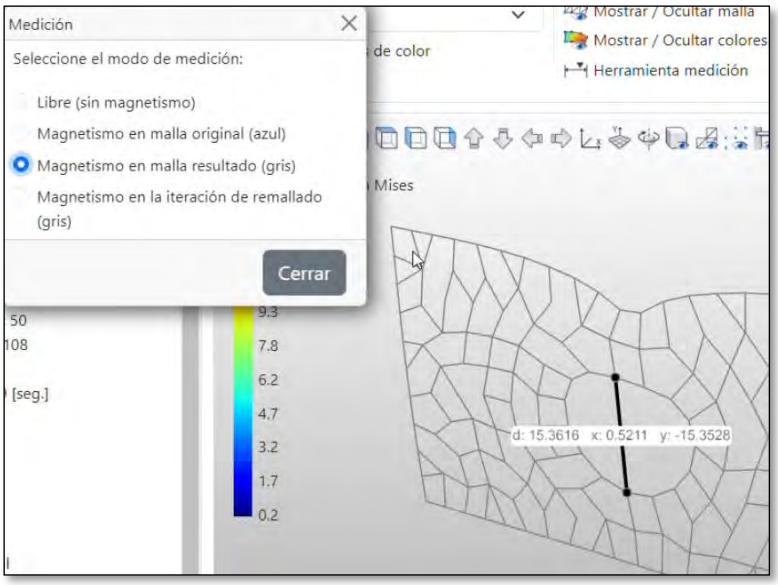


Figura 46. Herramienta de medición

3.10 Herramienta etiqueta de valores

Se desarrolla herramienta para etiquetar los valores calculados en base a las posiciones seleccionadas por el usuario (ver Figura 47). Estas etiquetas muestras el valor específico del elemento o nodo seleccionado, según el tipo de gráfico visualizado.

Las etiquetas se acumulan, permitiendo presentar tantas como sean necesarias y se eliminan cuando el usuario desactiva la herramienta.

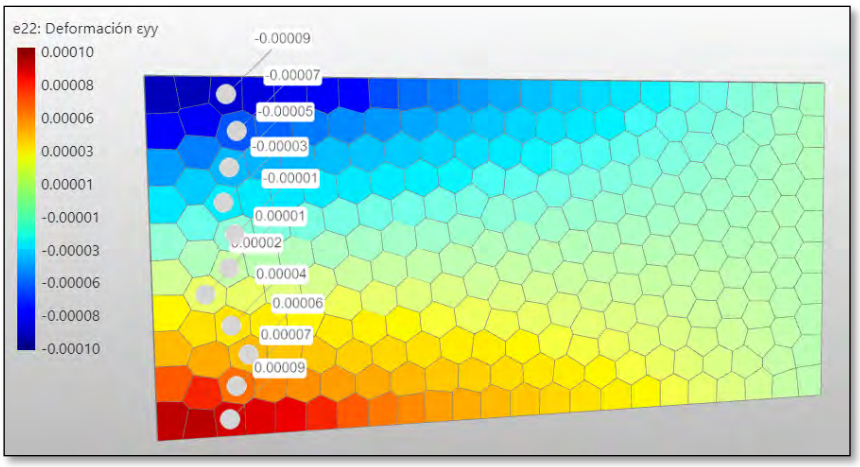


Figura 47. Herramienta etiquetas de valores

Adicionalmente, se desarrollan el etiquetado de los máximas y mínimos los cuales se generarán de forma automática al activar las herramientas. Al igual que la herramienta de selección manual, estas etiquetas dependen del gráfico visualizado.

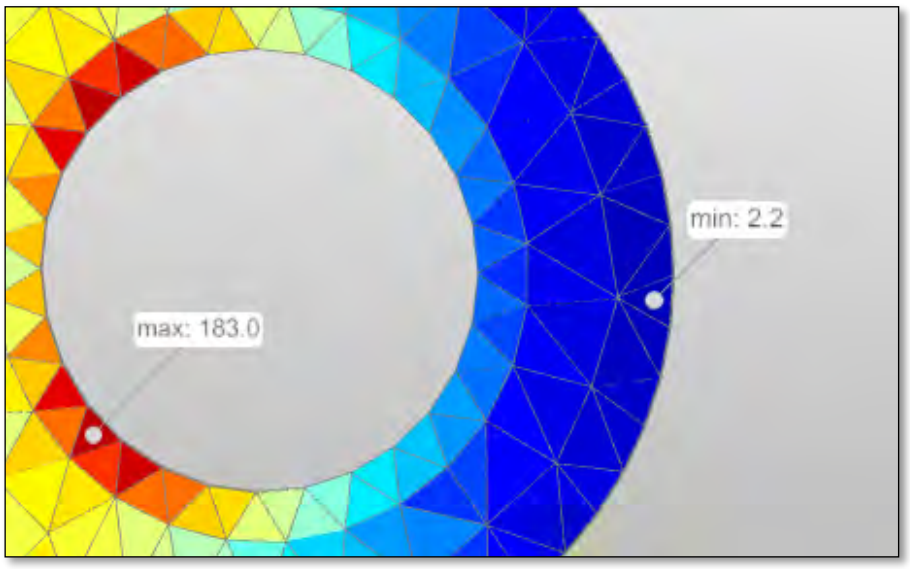


Figura 48. Herramienta máximos y mínimos

3.11 Exportación e importación de mallas a VEMLAB

Debido a que el software MECHAIDE replica el algoritmo de solución de VEMLAB 2.4.1, la importación y exportación de mallas permite realizar análisis comparativo efectivo entre ambos softwares, permitiendo utilizar la potencia de cálculo de Matlab a través de VEMLAB, pero utilizando la simplicidad de configuración de mallas y refinamiento generadas por MECHAIDE.

La Figura 49 se presentan dos ejemplos de exportaciones de malla triangular y malla tipo D-Polylla refinadas en dominios de tipo rectangular desde MECHAIDE (mallas en azul) a VEMLAB 2.4 (mallas en negro). En caso de que la configuración de condiciones de borde de Dirichlet y de Neumann ya se hayan definido en MECHAIDE, la exportación de malla asocia por defecto una geometría de malla de tipo rectangular *RectangularDomain* a la cual se le asignan nodos bajo condición de borde de Neumann a la etiqueta del lado derecho *rightNodes*, mientras que los nodos con condiciones de borde de Dirichlet se asocian a la etiqueta de nodos del lado izquierdo *leftNodes.*

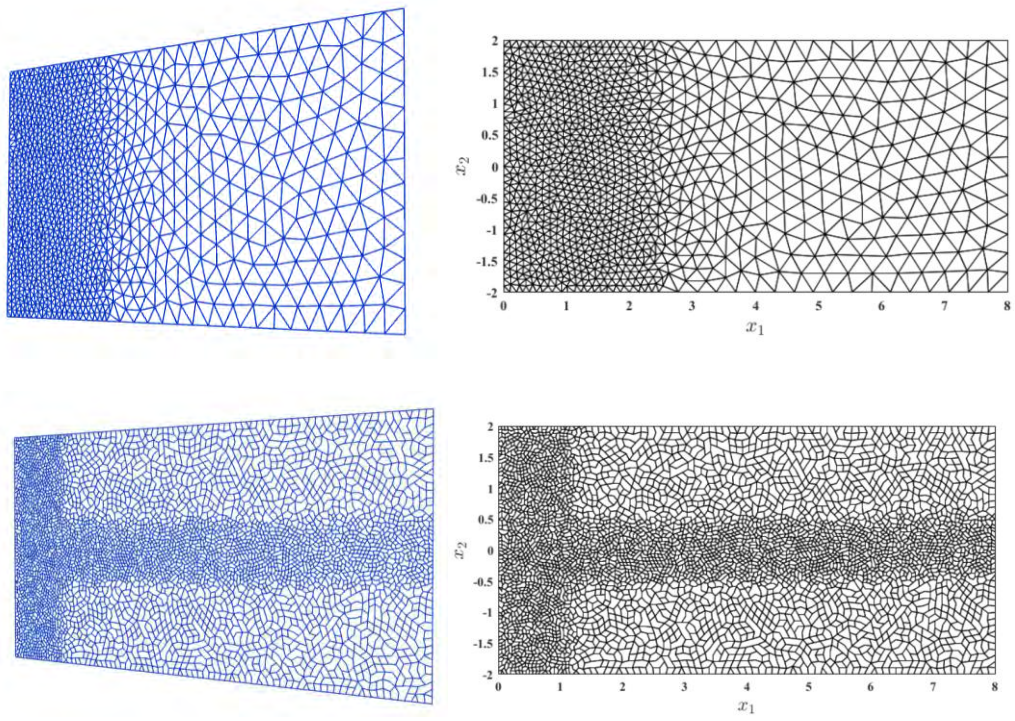


Figura 49. Mallas de MECHAIDE (azul) a VEMLAB (negro)

4 Validación de resultados

4.1 Análisis comparativo

Se realiza análisis comparativo entre diferentes programas computacionales tomando como criterios el módulo de la deformación máximo y esfuerzo de von Mises máximo.

Se considera la comparación de diferentes problemas en el análisis de valores obtenidos mediante software desarrollado MECHAIDE en comparación con software de licencia libre VEMLAB v2.4.1 desarrollado en Matlab R2023A y software comercial Autodesk Inventor Profesional 2016.

Para la comparación con VEMLAB se utiliza las mallas ejemplo para la verificación de MECHAIDE, permitiendo de esta forma reutilizar las mallas y la configuración de las condiciones de borde en los mismos nodos en ambos programas.

4.1.1 Comparación de resultados con VEMLAB

Esta comprobación permite validar los algoritmos utilizados para el cálculo MEF y MEV permitiendo de esta forma validar la correcta implementación y adaptación del código fuente desde Matlab a JavaScript.

En todos los problemas presentes en esta sección VEMLAB está configurado para resolver los problemas con algoritmos de inversión de matriz nativo de Matlab para matrices dispersas, mientras que MECHAIDE utiliza la lógica de inversión Cholesky para matrices densas. En casos que se utilicen MEV se considera el parámetro de estabilidad tipo 1 definido en ambas configuraciones.

4.1.1.1 Biela en flexión

En VEMLAB y MECHAIDE el problema se calcula mediante MEV considerando una malla poligonal Voronoi de 400 elementos y 803 nodos disponible en la documentación de VEMLAB 2.4.1 con el nombre de *wrench_400poly_elems.txt*, el cual considera un módulo de elasticidad adimensional de $E_Y = 10^7$ y una relación de Poisson de $\nu = 0{,}3$, para una presión de carga de $P = -35$ aplicada en la superficie inferior de la perforación de menor dimensión de la biela, considerando restricción de los grados de libertad de los nodos circunscritos en la superficie de la perforación de mayor dimensión.

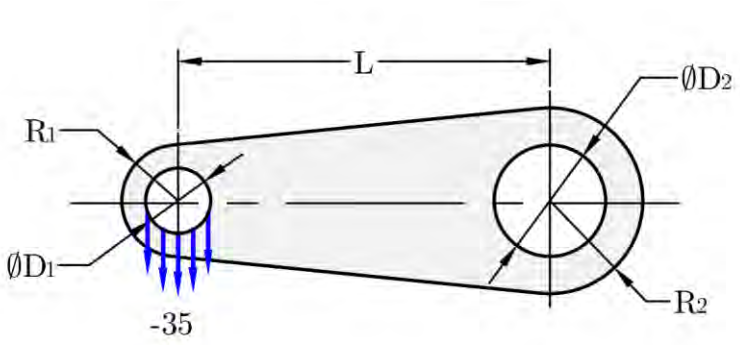


Figura 50. Geometría de biela

La Figura 51 y la Figura 52 presentan los resultados obtenidos mediante MECHAIDE y VEMLAB de manera respectiva. En ambos softwares se obtuvieron valores idénticos tanto para esfuerzo de von Mises $\sigma_{VM,\max}$ con un valor de 214,6, como para la deformación normal $u_{n,\max}$ equivalente a 0,00014. El tiempo de procesamiento de VEMLAB, con renderización de gráficos equivale a 8.4 segundos, mientras que MECHAIDE realizó la misma operación en 3,8 segundos.

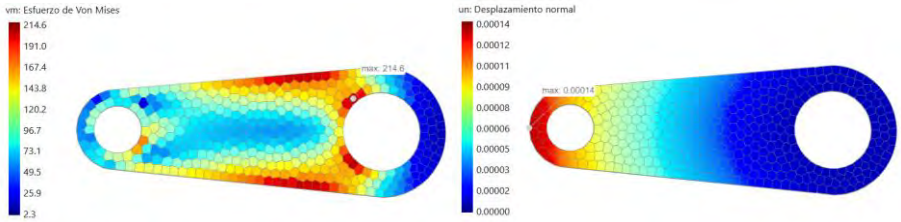


Figura 51. Esfuerzo von Mises y deformación biela, malla Voronoi 400 elementos, 803 nodos con MECHAIDE

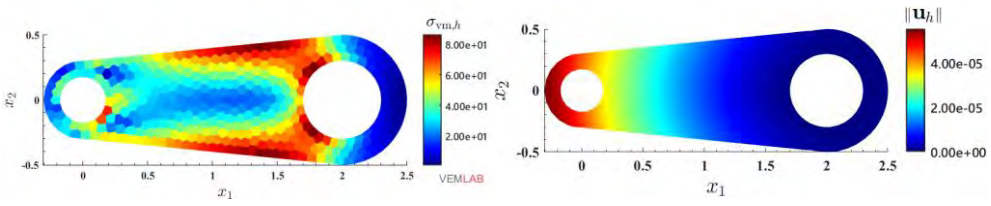


Figura 52. Esfuerzo von Mises y deformación normal biela, malla Voronoi 400 elementos, 803 nodos con VEMLAB

Se considera la misma configuración de problema para una malla más refinada, presente en el archivo *wrench_3000poly_elems.txt* de 3000 elementos y 5993 nodos, obteniendo, al igual que en el caso anterior, valores idénticos para esfuerzo de von Mises $\sigma_{VM,\max} = 222{,}7$, y para la deformación normal $u_{n,\max} = 0{,}00012$. El tiempo de procesamiento de VEMLAB para el problema es de 13.3 segundos mientras que MECHAIDE lo realiza en 18 minutos y 5 segundos.

Se considera la misma configuración de problema para una malla aún más refinada, presente en el archivo *big_wrench_4000poly_elems.txt* de 4000 elementos y 7926 nodos (ver Figura 53), obteniendo al igual que en el caso anterior valores idénticos para esfuerzo de von Mises $\sigma_{VM,\max} = 196{,}2$ (ver Figura 54) y para la deformación normal $u_{n,\max} = 0.0025$. VEMLAB logra calcular en 15.8 segundos mientras que MECHAIDE realiza el cálculo en 1 hora con 12 minutos. Es importante aclarar que la geometría de la biela presente en la Figura 50, cambia respecto de los dos primeros problemas, por esta razón los resultados entre ambas soluciones variar considerablemente.

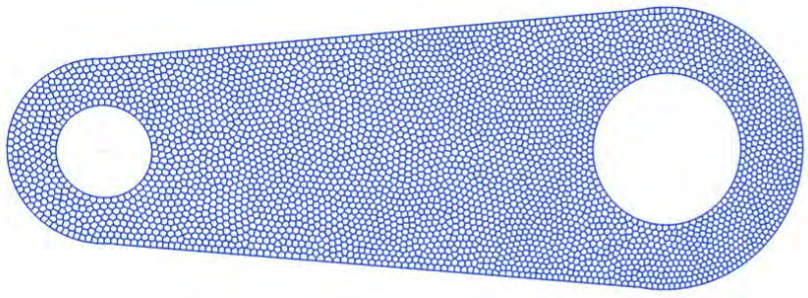


Figura 53. Malla Voronoi 4000 elementos poligonales, 7926 nodos.

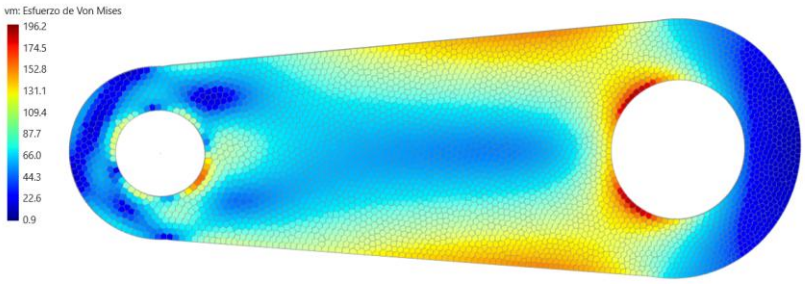


Figura 54. Esfuerzo de von Mises de biela, malla Voronoi 4000 elementos, 7926 nodos con MECHAIDE

Se procede a validar bajo la misma configuración de problema el algoritmo MEF considerado la malla disponible en el archivo *big_wrench_444_t3_elems.txt* de 444 elementos triangulares y 286 nodos. En ambos softwares los resultados coinciden exactamente $\sigma_{VM,\max} = 189{,}8$, y $u_{n,\max} = 0{,}0025$. VEMLAV calcula en 10,8 segundos y MECHAIDE en 0,21 segundos. Utilizando la metodología MEV para la misma malla, VEMLAB procesa el problema en 8,3 segundos, mientras que MECHAIDE lo realiza en 0.17 segundos.

4.1.1.2 Viga en voladizo bajo carga vertical parabólica

Se ejecuta el problema de viga en voladizo bajo carga parabólica $P = -1000$ lb_f aplicada en el extremo libre bajo estado de deformación plana. La Figura 55 presenta la geometría y

sus condiciones de contorno. Se considera un módulo de elasticidad $E = 10^7$y una relación de Poisson $\nu = 0{,}3$, para un largo $L = 8$ in y un espesor y profundidad de $t = h = 4$ in.

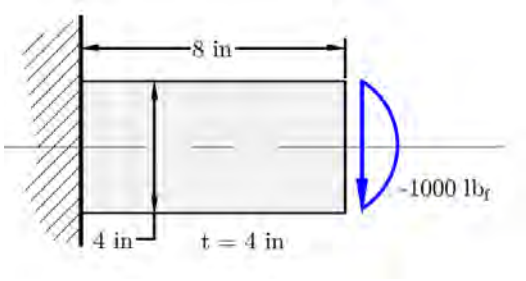


Figura 55. Diagrama de carga viga en voladizo carga parabólica.

Se procede a validar bajo la misma configuración de problema el algoritmo MEV considerado la malla disponible en el archivo *cantilever_beam_250poly_elems.txt* de 250 elementos poligonales y 286 nodos. En ambos softwares los resultados son idénticos $\sigma_{VM,\max} = 2300$ psi, y $u_{n,\max} = 0.00301$ in. VEMLAV calcula en 1.9 segundos y MECHAIDE en 1.1 segundos, siempre utilizando la MEV para la misma malla.

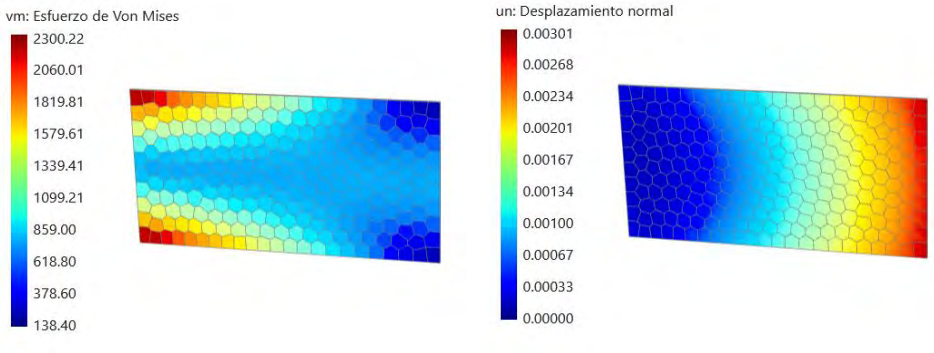


Figura 56. Viga bajo carga parabólica, malla Voronoi 250 elementos, 286 nodos con MECHAIDE.

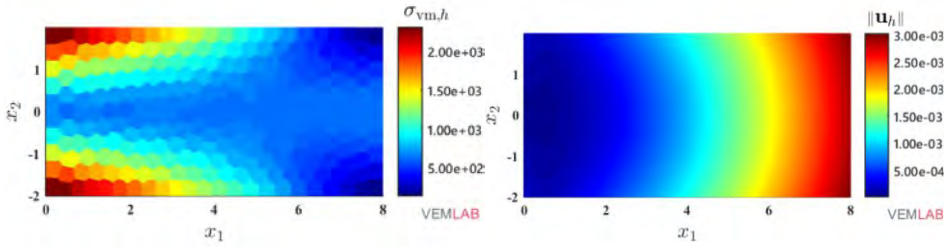


Figura 57. Viga bajo carga parabólica, malla Voronoi 250 elementos, 286 nodos con VEMLAB

Se procede con la exportación de mallas tipo D-Polylla desde MECHAIDE a VEMLAB para comprobación de lógica de solución inversa. Esta exportación permite realizar análisis de convergencia numérica para el algoritmo de mallado D-Polylla en la sección 4.3.

Se genera malla de tipo D-Polylla de 719 nodos y 492 elementos, bajo las mismas condiciones de borde determinadas por el problema utilizando el algoritmo MEV. Como se presenta en la Figura 58 y

Figura 59, en ambos softwares los resultados son idénticos $\sigma_{VM,\max} = 2463.18$ psi, y $u_{n,\max} = 0{,}003$ in. VEMLAV calcula en 1.9 segundos y MECHAIDE en 2.1 segundos, siempre utilizando la MEV para la misma malla.

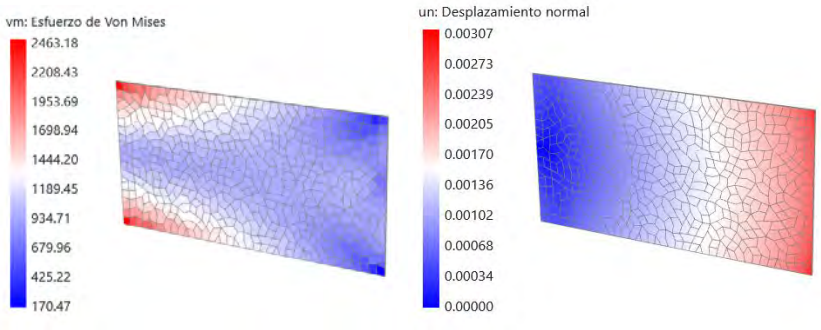


Figura 58. Viga bajo carga parabólica, malla Direct Polylla, 492 elementos, 719 nodos con MECHAIDE

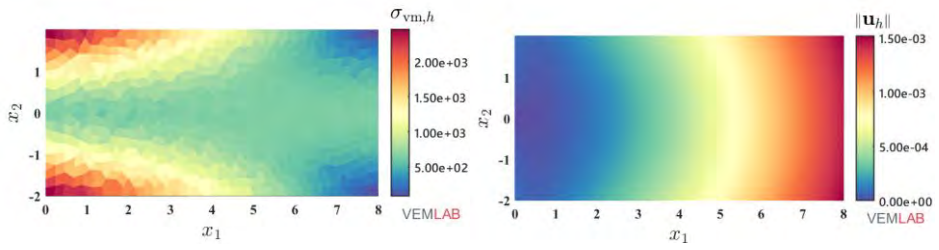


Figura 59. Viga bajo carga parabólica, malla Direct Polylla, 492 elementos, 719 nodos con VEMLAB

4.1.2 Comparación con literatura técnica y software comercial

Se modelan dos problemas del comportamiento lineal elástico de cuerpos sólidos mediante el MEF presentes en la literatura técnica, uno bajo cargas de tracción y otro bajo carga de flexión pura.

4.1.2.1 Viga empotrada bajo carga de tracción

El primer problema extraído de [34] considera un cuerpo rectangular de sección uniforme, módulo de elasticidad de $E_Y = 210$ GPa y relación de Poisson de $\nu = 0{,}3$, cuyas dimensiones son 200 mm de altura, 400 mm de largo y 20 mm de espesor empotrado en un extremo y sometido un esfuerzo por tracción de 7 MPa, como se presenta en la Figura 60. Se solicita determinar los desplazamientos nodales en base a dos elementos triangulares.

Como la carga está definida como unidad de presión y bajo la suposición de comportamiento elástico en condiciones de esfuerzo plano, el espesor es un dato prescindible, del punto de vista numérico, dado que los esfuerzos de corte bajo esta condición se asumen nulos. En casos donde se especifique cargas distribuidas o puntuales, el espesor si adquiere relevancia, principalmente para transformar dichas fuerzas en cargas por presión aplicada a los nodos.

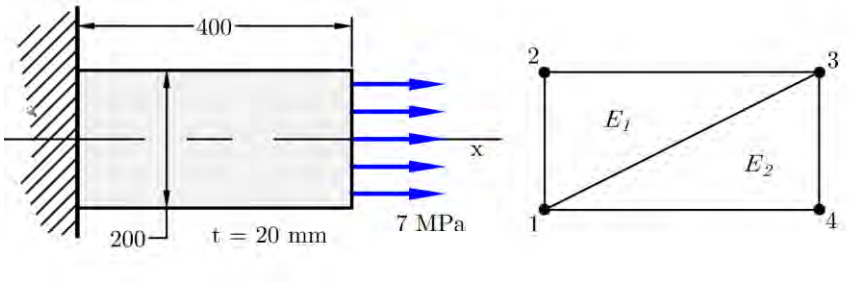


Figura 60. Problema cuerpo bajo presión de tracción.

De acuerdo con las condiciones de borde solicitadas, se considera una restricción de los grados de libertad mediante la definición de una condición de borde Dirichlet en los nodos 1 y 2 $u_{\{1,2\}} = 0$ mm y $v_{\{1,2\}} = 0$ mm, y se aplica la carga por presión de tracción mediante la una condición de Neumann, en los nodos 3 y 4 $f_{x,\{3,4\}} = 7$ MPa, $f_{y,\{3,4\}} = 0$ MPa. El algoritmo de solución de MECHAIDE entrega resultados idénticos a los valores de desplazamiento nodal calculados de manera analítica en la literatura tanto para todos sus grados de libertad.

$$\begin{Bmatrix} u_3 \\ v_3 \\ u_4 \\ v_4 \end{Bmatrix} = \begin{Bmatrix} 0{,}012192 \\ 0{,}000833 \\ 0{,}013274 \\ 0{,}0020817 \end{Bmatrix}$$

La Figura 61 presenta el esquema de desplazamientos con las etiquetas en los respectivos nodos.

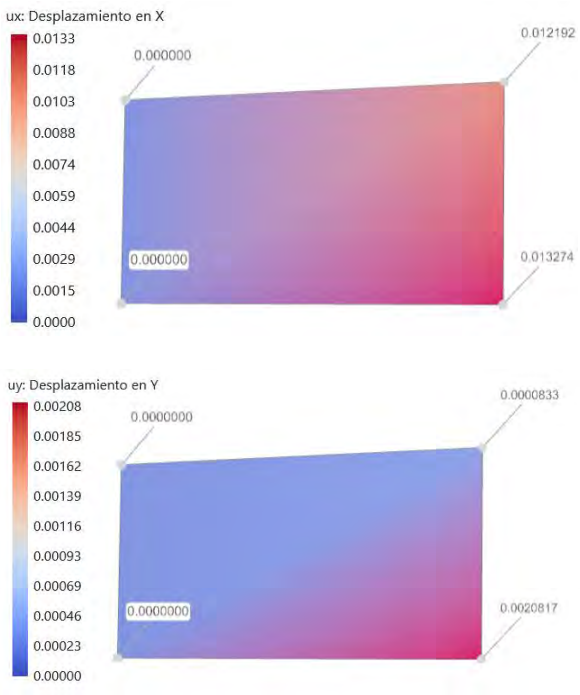


Figura 61. Desplazamientos nodales cuerpo bajo tracción, malla triangular, 2 elementos, 4 nodos con MECHAIDE.

El autor en [35] al comparar la solución MEF o MEV a la solución analítica, la cual esta dada por $\delta = \frac{PL}{AE} \approx 0{,}0133$ mm concluye que los resultados son razonables, debido que a pesar de que un sólo nodo se aproxima al valor teórico, el comportamiento es esperable por la tosquedad del modelo simulado.

Debido a que el esfuerzo axial de 7 MPa actúa únicamente en la dirección x sobre él la sección transversal del cuerpo, se esperaría que el esfuerzo calculado en cada elemento sea cercano a 7 MPa, tal como se evidencia en los resultados del esfuerzo de von Mises calculado con el software $\sigma_{VM,\{\max\}} = 6{,}71$ MPa ≈ 7 MPa.

Utilizando Autodesk Inventor Professional 2016, se realiza la simulación mediante MEF del cuerpo tridimensional definido en el problema, restringiendo el desplazamiento de las caras cuya normal es perpendicular a la dirección de la carga, en conjunto con las condiciones de borde y cargas que impone el problema y considerando la menor cantidad de elementos, dentro de las opciones de configuración admitidas por el software.

La Figura 62 presenta los resultados tanto de desplazamiento máximo $u_{n,\max} = 0{,}012$ mm como de esfuerzo de von Mises $\sigma_{VM,\max} = 6{,}683$ MPa que evidencia un comportamiento similar a la resolución analítica y la solución numérica obtenida por MECHAIDE.

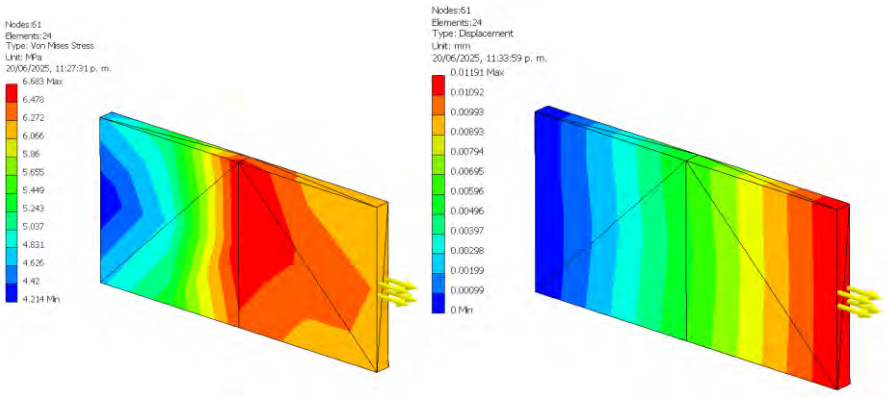


Figura 62. Viga bajo carga axial, malla tetraédrica, 24 elementos, 61 nodos, con Autodesk Inventor.

4.1.2.2 Viga empotrado bajo carga puntual en flexión

Se considera una viga empotrada de largo $L = 1000$ mm, cuya sección uniforme tiene un alto de $h = 100$ mm y una profundidad de $t = 120$ mm, con un extremo empotrado y en el otro extremo una carga vertical puntual $P = -4000$ N de acuerdo con el diagrama de cuerpo presente en la Figura 63. Se considera un módulo de elasticidad de $E = 200$ GPa y un módulo de rigidez de $G = 77{,}5$ GPa, se solicita determinar desplazamiento nodal máximo y esfuerzo de von Mises.

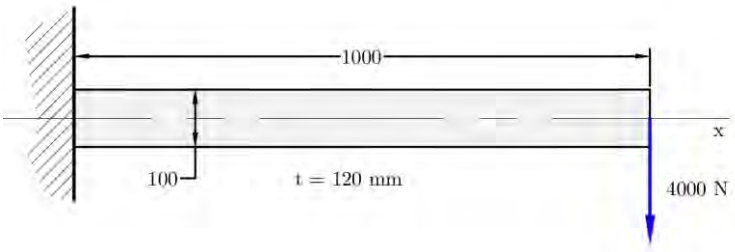


Figura 63. Diagrama de cuerpo viga bajo flexión.

Dado que la carga está definida en unidades de fuerza y que algoritmo de MECHAIDE no permite la imposición de condiciones de contorno de Neumann sobre un solo nodo, se debe determinar la carga por presión equivalente, la cual se calcula al determinar el área de presión entre los dos nodos más cercanos al punto de aplicación de la carga puntual (ver Figura 64.), multiplicando por el espesor t teórico del cuerpo en la superficie. Al distribuir la carga en la superficie, se determina presión teórica para la simulación $f_y = F/td$, donde d corresponde a la separación entre los nodos. El procedimiento se debe repetir en cada iteración de malla, dado que la separación entre nodos varía según tamaño del elemento.

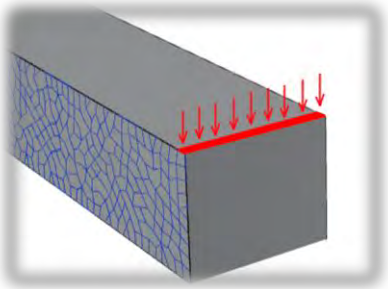


Figura 64. Superficie de aplicación de carga.

La Figura 65 presenta los resultados obtenidos de MECHAIDE tanto para el desplazamiento $u_n = 0{,}662\ mm$ como para el esfuerzo de von Mises $\sigma_{VM,\max} = 20{,}45$ MPa.

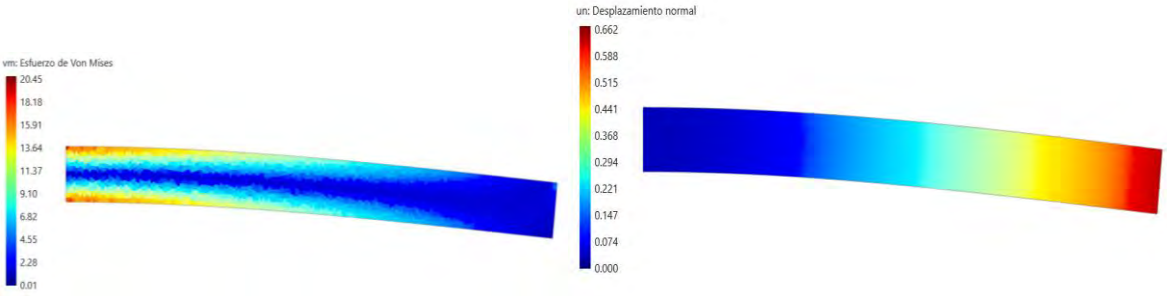


Figura 65. Viga bajo flexión, malla Direct Polylla 1233 elementos, 1841 nodos, con MECHAIDE.

Cabe destacar que el desplazamiento máximo teórico esta dado por la ecuación $v_{\max} = \frac{PL^3}{3E_YI} + \frac{6PL}{5AG} = 0{,}66$ mm y el esfuerzo máximo esta dado por $\sigma_{VM,\max} \approx \sigma_x = \frac{PLh}{2I} = 20$ MPa.

Para este problema se realiza comprobación con software comercial Autodesk Inventor 2016 y se verifican resultados obtenidos por MECHAIDE, donde el esfuerzo máximo de von Mises $\sigma_{VM,\max} = 20{,}3$ MPa y el desplazamiento máximo normal se evalúa en $u_{n,\max} = 6.62$ mm.

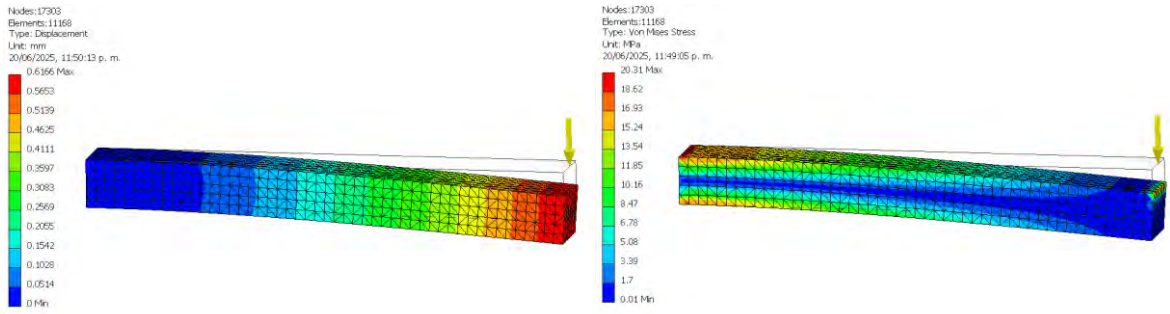


Figura 66. Viga bajo flexión, malla tetraédrica, 11168 elementos, 17303 nodos con Autodesk Inventor.

Para este problema en particular se realizaron 82 simulaciones con diferentes tamaños de elementos en MECHAIDE, considerando diferentes tipos de malla con el fin de evaluación eficiencia y precisión de los algoritmos implementados. La Figura 67 y Figura 68 presentan la distribución del error relativo para el desplazamiento normal u_n y esfuerzo de von Mises, de manera respectiva, considerando MEF y MEV para mallas de tipo triangular, D-Polylla y Voronoi.

Las mallas de tipo Voronoi en términos del desplazamiento se evidencia una pequeña desviación de un 1% respecto al valor teórico. Las mallas de tipo triangular y de tipo D-Polyllla tanto para la metodología MEF y MEV tienen un comportamiento más disperso respecto a las mallas de Voronoi, con un error promedio de 2 a 3% respecto al valor teórico máximo real.

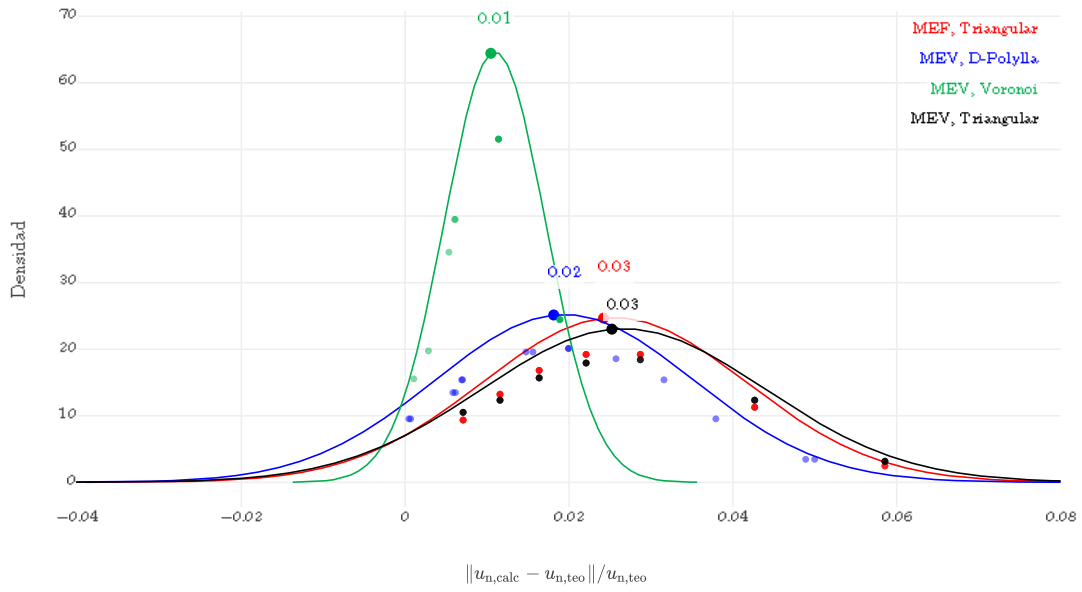


Figura 67. Distribución del error relativo para el desplazamiento máximo u_n.

El comportamiento de las curvas en el análisis de la distribución del error relativo para esfuerzo máximo de von Mises, difiere con el comportamiento de las curvas de desplazamiento. Se puede evidenciar una marcada diferencia entre la resolución en base a mallas triangulares y las mallas poligonales, donde las mallas triangulares son hasta un cincuenta por ciento más precisas, respecto al error relativo medio máximo. De igual manera el error promedio de las mallas poligonales es de un 8 a 9% respecto al valor teórico mientras que las mallas triangulares mantienen un error relativo medio de un 4%.

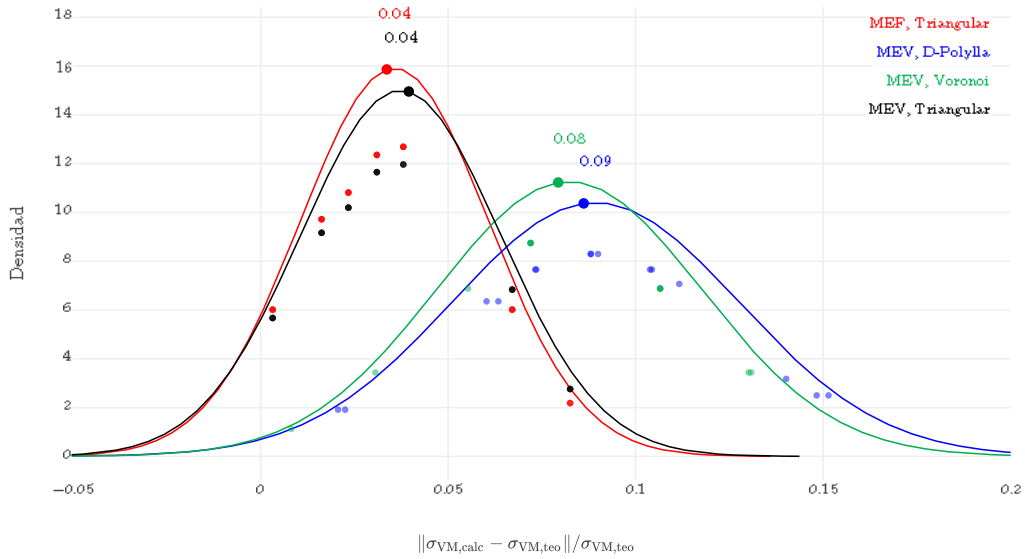


Figura 68. Distribución del error relativo esfuerzo de von Mises máx. σ_{VM}.

4.2 Análisis de eficiencia

Para establecer rangos óptimos de uso de la aplicación se efectuaron ochenta y dos simulaciones del problema de la sección 4.1.2.2 considerando diferentes tamaños de elementos y diferentes tipos de malla. Todos los gráficos se encuentran disponibles en la aplicación “Estático 2D – Analisis”.

4.2.1 Eficiencia de algoritmos de generación de malla

Para analizar la malla, se genera un registro en la base de datos que permite guardar tiempos de generación de malla asociando a su tipo, dimensiones del elemento, cantidad de nodos y cantidad de elementos generados.

La Figura 69 y Figura 70 presentan el tiempo de procesamiento para la generación de la malla versus la cantidad de nodos y elementos de manera respectiva.

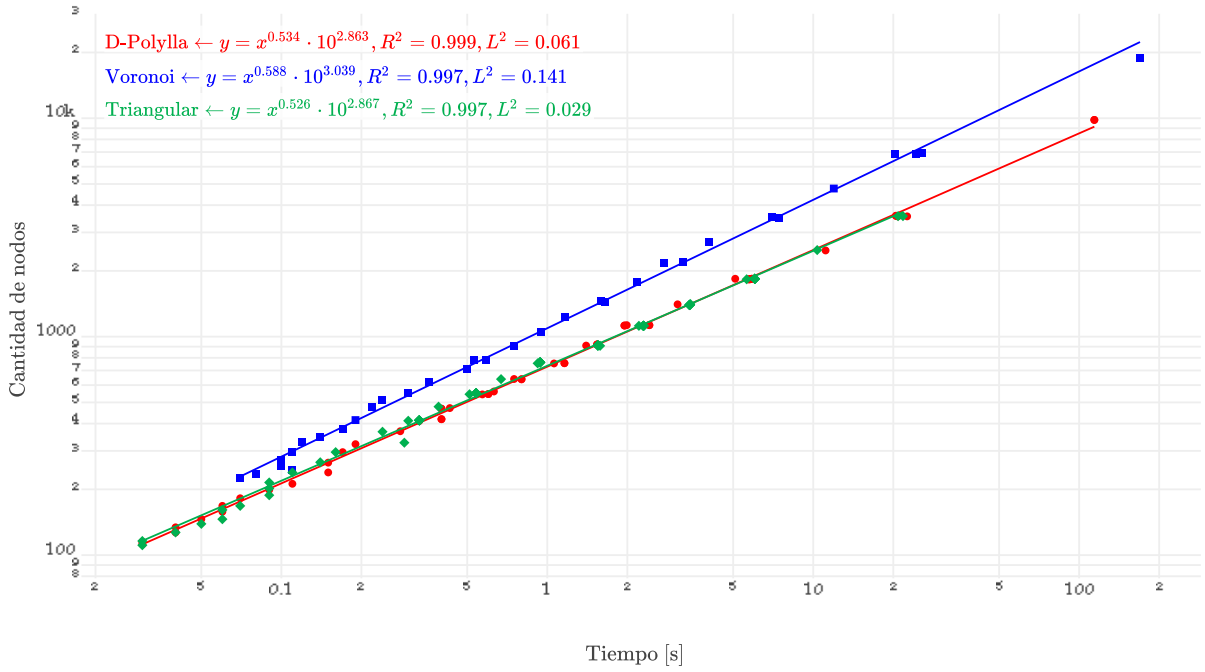


Figura 69. Nodos versus tiempo de cálculo de malla.

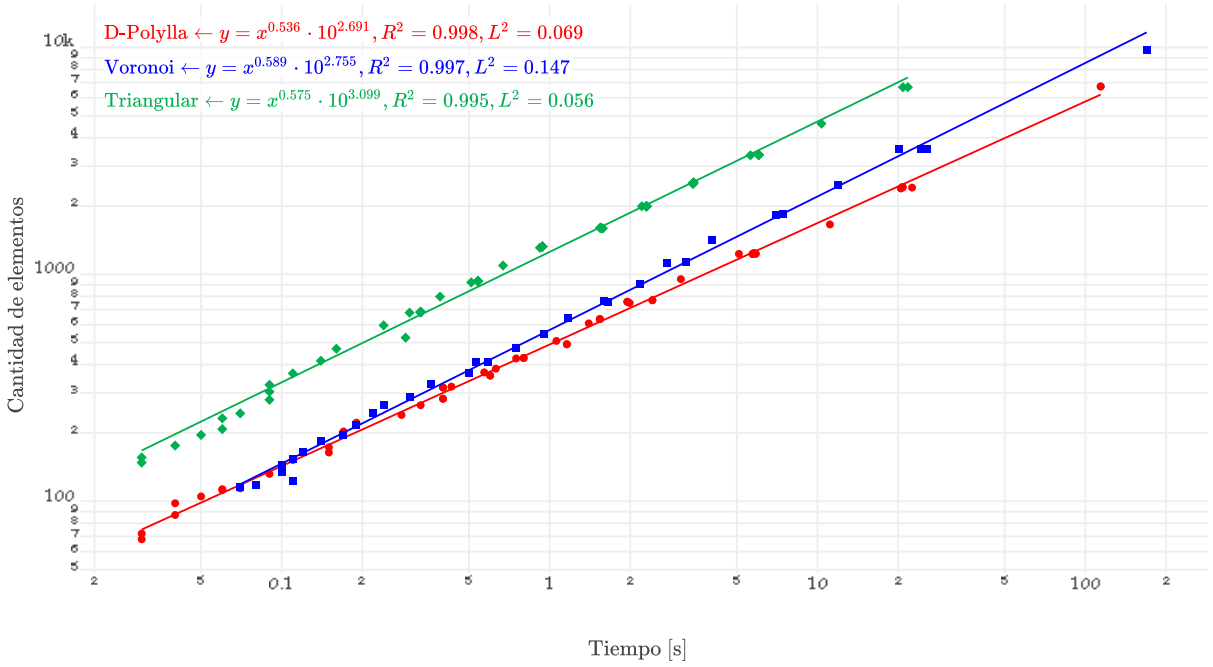


Figura 70. Elementos versus tiempo de cálculo de malla

En ambas gráficas se observa un comportamiento de dependencia potencial, evidenciado un ajuste de potencia con determinación R^2 muy cercano a la unidad, permitiendo de esta manera estimar con precisión el tiempo de generación de malla de cada tipo de manera efectiva, aplicando las regresiones de potencia presentes en la leyenda de cada gráfica. El

análisis porcentual presente en esta sección considera el promedio de la diferencia de valores calculados en base estas regresiones de potencia, tomando en consideración un rango de cero a cien segundos.

En términos de generación de nodos en un determinado tiempo, el algoritmo de generación de malla D-Polylla es un 2% más eficiente que el algoritmo para la generación de malla triangular Delaunay, mientras que el algoritmo de Voronoi es un 87% más eficiente que el algoritmo D-Polylla.

En relación con la generación de elementos, el algoritmo de generación de malla triangular D-Polylla es un 41% menos eficiente que el algoritmo de Voronoi. Comparando con algoritmo de malla triangular D-Polylla es un 196% menos eficiente.

4.2.2 Eficiencia de algoritmos de inversión de matriz

Para evidenciar la eficiencia de los algoritmos de inversión de matriz basados en JavaScript, la Figura 71 presenta la representación gráfica de cincuenta y seis simulaciones realizadas al problema de la sección 4.1.2.2 utilizando diferentes densidades malla triangulares, considerando diferentes metodologías de cálculo.

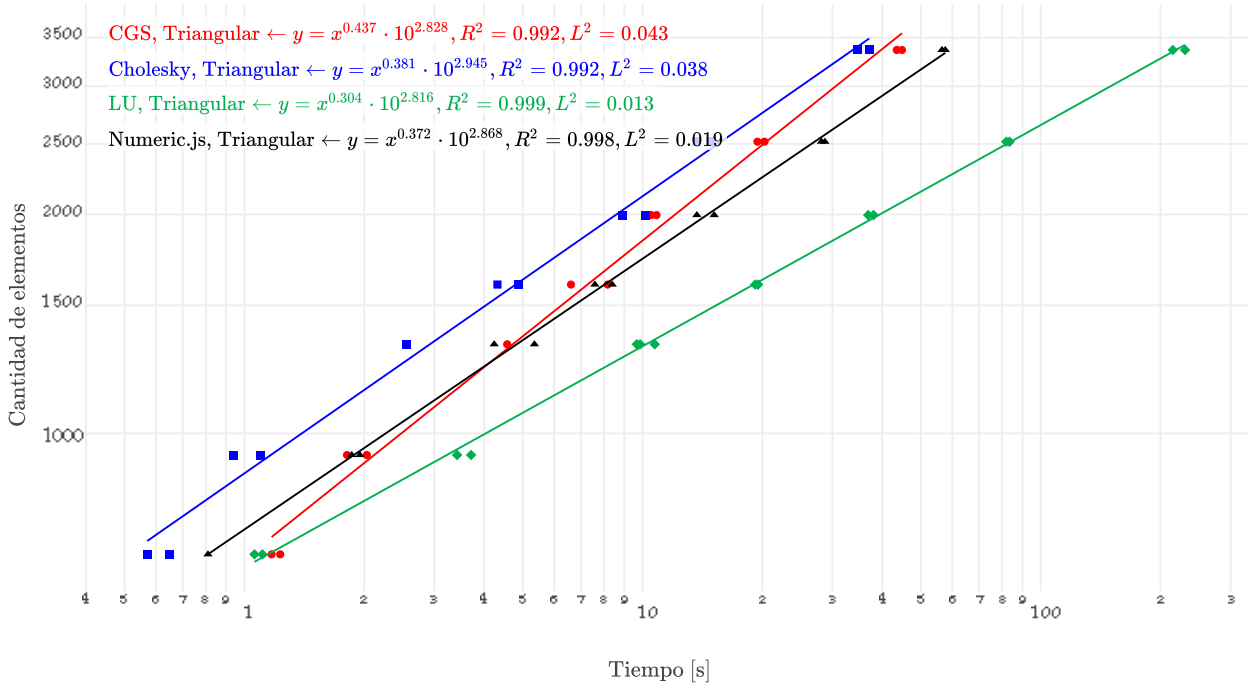


Figura 71. Elementos versus Tiempo cálculo según solver para malla triangular

Basado en el mismo enfoque de comparación porcentual respecto al valor de diferencia entre regresiones medias, se puede definir que al algoritmo de Cholesky es un 6% más eficiente que el algoritmo CGS, siendo un 19% más eficiente que algoritmo de Numeric.js y un

44% más eficiente que el algoritmo L-U. Cabe destacar que el algoritmo CGS podría ser más eficiente que el algoritmo de Cholesky en mallas que superen los 5500 elementos, tomando en consideración la extrapolación de las regresiones de potencia de cada modelo de solución.

4.2.3 Eficiencia de algoritmos de solución según tipo de malla

La Figura 72 establece la gráfica comparativa entre diferentes mallas utilizadas para el problema de viga bajo carga por flexión, respecto a la cantidad de elementos utilizados en la simulación. En términos de tiempo de resolución considerando la misma malla triangular de tipo Delaunay, se puede deducir que los métodos de resolución MEF y MEV tienen tiempos de procesamiento similares siendo más eficiente MEF por sólo un 0,15% respecto a la resolución mediante MEV.

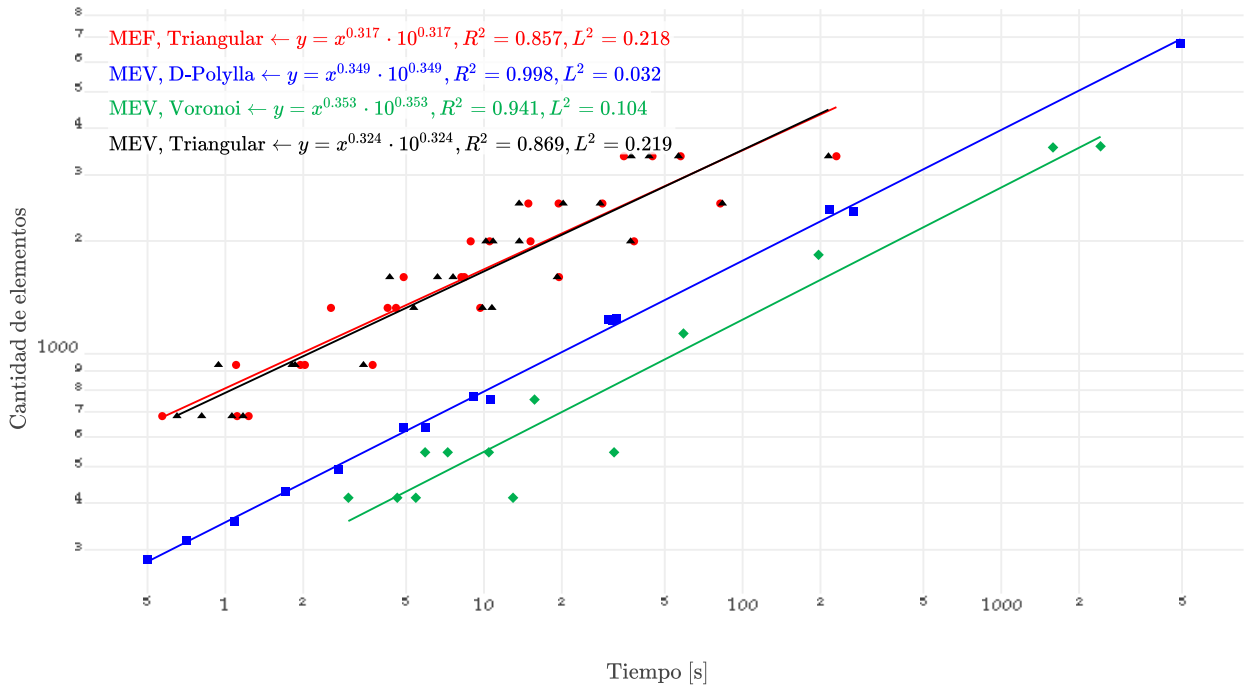


Figura 72. Método-Malla versus Tiempo de cálculo para solver de tipo Cholesky

Respecto al algoritmo de generación de malla D-Polylla, se puede deducir que al utilizarlo puede ser 31% más eficiente en tiempo de resolución, en comparación con una malla de elementos poligonales generados con el algoritmo de tipo Voronoi. Cabe destacar que el uso de mallas de tipo D-Polylla es al menos un 102% menos eficiente, en comparación con mallas de tipo triangular Delaunay tanto para MEV como para MEF.

4.2.4 Comportamiento remallado P-Remesh

Se realizan diferentes simulaciones con geometrías y mallado aleatoria, para verificar comportamiento general del algoritmo de remallado P-Remesh definido en la sección 3.4 del presente documento.

MECHAIDE puede animar el comportamiento de deformación al guardar la geometría de deformación en cada iteración, de tal manera que las cargas se apliquen sobre la malla deformada de la iteración anterior. De esta manera se puede analizar el comportamiento del algoritmo de remallado bajo situaciones límite. Adicionalmente, MECHAIDE colorea los polígonos para identificar problemas geométricos: los polígonos naranjas indican elementos con baja calidad según el criterio APR, mientras que los polígonos amarillos representan su vecindad, es decir, elementos adyacentes conectados por aristas compartidas.

Es importante resaltar que el objetivo es visualizar el comportamiento del remallado y en ningún caso se busca simular el comportamiento real plástico del material utilizado. El comportamiento no lineal del material bajo condición de deformación permanente esta fuera de los alcances del desarrollo expuesto en el presente documento.

El primero problema consiste en un disco perforado con restricción en sus grados de libertad en todo el perímetro de la perforación inferior y con una carga en los nodos superiores. La Figura 73 presenta la geometría del cuerpo considerando varios estados de deformación para una restricción de APR de 0,1.

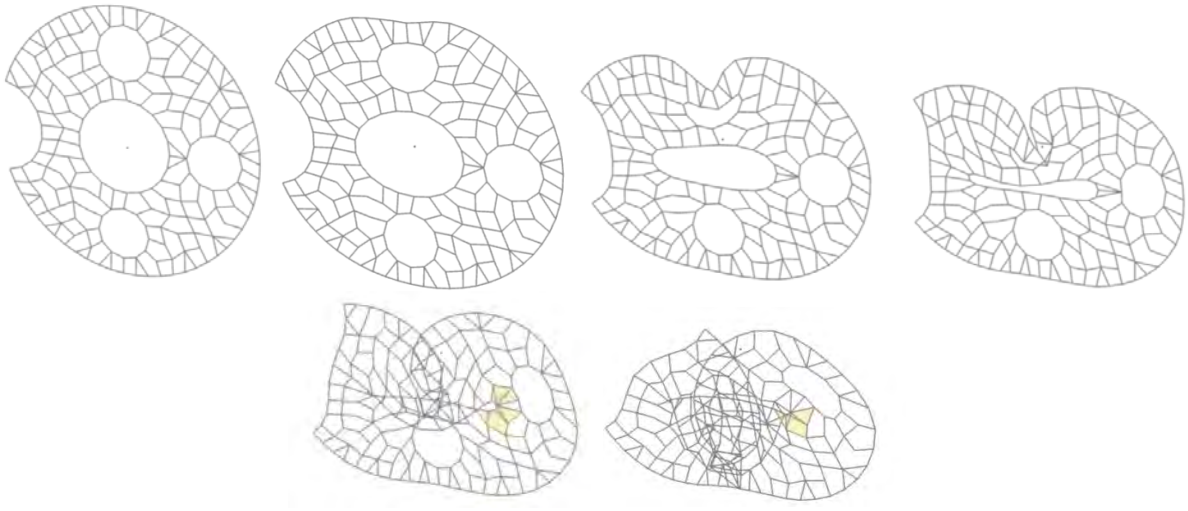


Figura 73. P-Remesh circulo perforado 205 nodos, APR límite 0,1

En términos de rendimiento para un total de 205 nodos se considera, cada iteración de generación de malla y cálculo de solución se realizó como promedio en 77 milisegundos por iteración.

Analizando la Figura 73 en los últimos tres estados de deformación, se puede evidenciar claramente el traslape de elementos poligonales no siendo esto un comportamiento deseado en el contexto de simulación de sólidos. Este comportamiento ocurre con un valor de APR bajo, donde sólo algunos elementos son remallados. Cabe destacar que el método MEV logra procesar de igual manera el cálculo.

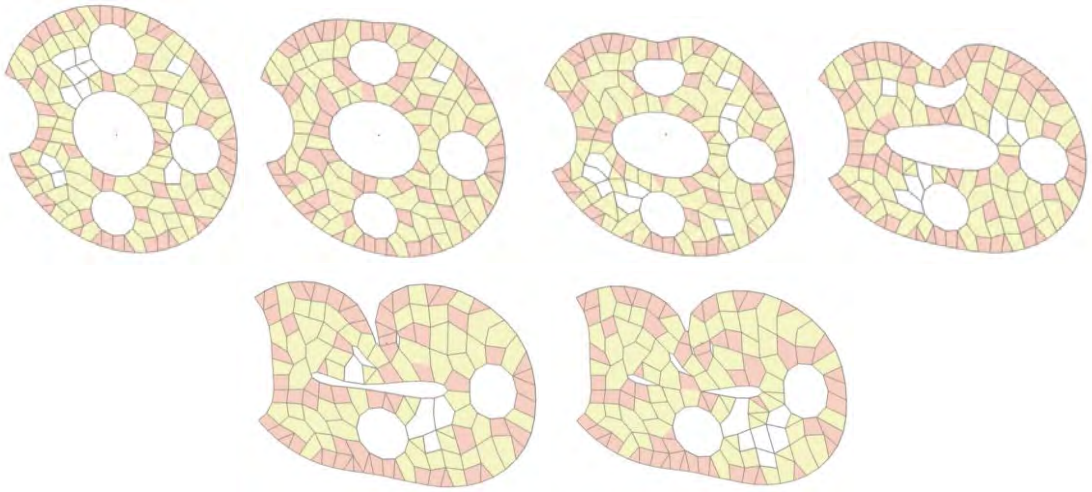


Figura 74. P-Remesh circulo perforado 205 nodos, APR límite 0,3

La Figura 74 presenta el mismo problema mencionado en el párrafo anterior considerando un límite de APR de 0,3. El tiempo de procesamiento promedio es de 85 milisegundos. Se evidencia claramente la acción del remallado en prácticamente todos los elementos, donde los polígonos color naranjo corresponden a los elementos detectados como problemáticos mientras que los elementos en amarillo son los elementos de la vecindad. Si bien en términos de tiempo de procesamiento de la solución ambas situaciones son similares, el comportamiento difiere completamente debido a que se evidencia la acción del remallado al romper la conectividad en los zonas donde existe contacto, permitiendo que se generen nuevos elementos que conformen las zonas de contacto y con esto se genere la inercia para resistir con esta nueva geometría las cargas, y por consiguiente evitando que se generen deformaciones donde se traslapen los elementos.

Se procede con la geometría de un perfil tubular bajo carga de fuerza corporal vertical decendente. Se consideran 605 nodos con un APR límite inicial de 0,3. La Figura 75 presenta las iteraciones de deformación cuyo tiempo de procesamiento promedio se evalúa en 1 segundo. Al analizar la deformación ocurre un comportamiento similar al observado en de la Figura 73, donde el traslape es evidente a pesar de mantener un APR más alto.

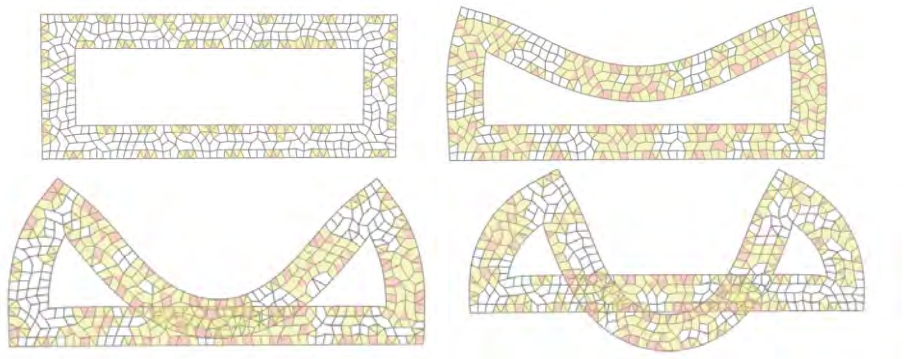


Figura 75. P-Remesh tubular 605 nodos, APR límite 0,3

La Figura 76 presenta la misma situación considerando un APR de 0,5, donde las deformaciones en las iteraciones muestran un comportamiento resiliente, con fusión de elementos en contacto. El procesamiento de las iteraciones en la Figura 76 es un poco más costosa del punto de vista computacional, donde cada iteración toma 1,2 segundos.

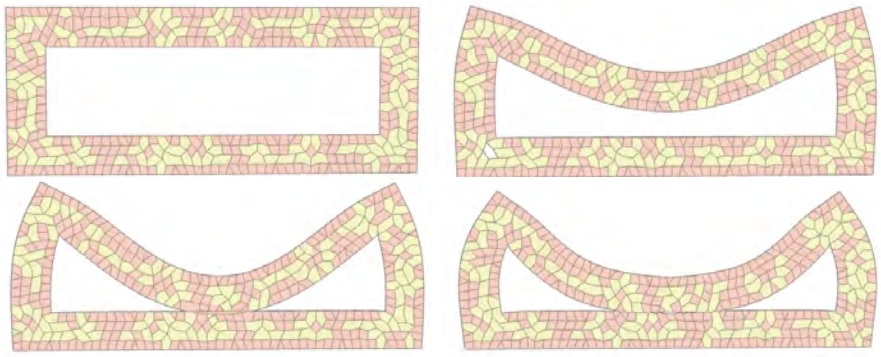


Figura 76. P-Remesh tubular 605 nodos, APR límite 0,5

Se procede con la simulación de una placa perforada, bajo carga puntual en la zona central dirección vertical descendente, donde se fijan los GL de la zona inferior a excepción de los nodos centrales. En la Figura 77 se presentan las deformaciones representativas de la iteración, donde cada una demoró 37 milisegundos en promedio.

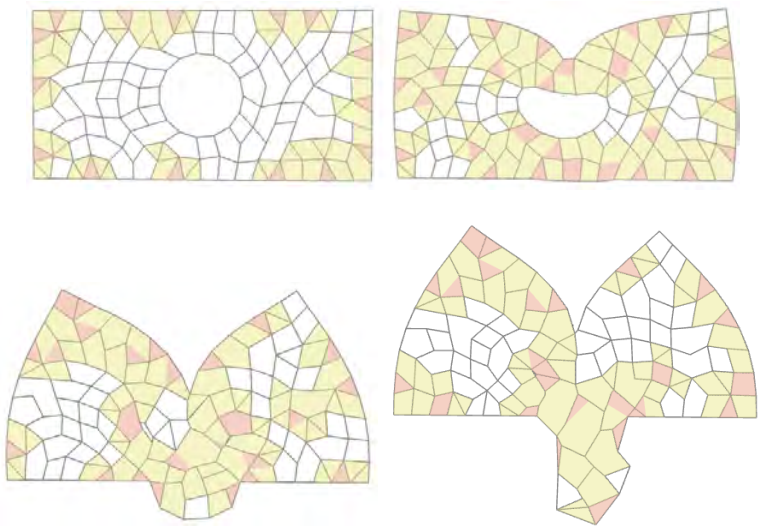


Figura 77. Placa perforada 169 nodos, APR límite 0,8

Se procede a realizar dos simulaciones adicionales considerando el mismo problema, pero con mallas con mayor refinación en sus elementos. La Figura 78 y la Figura 79 presentan dos simulaciones para 656 nodos y de 2568 nodos, de manera respectiva. Para la simulación de 656 nodos en promedio cada iteración tiene un tiempo de procesamiento de 1.7 segundos mientras que para la malla de 2568 nodos el tiempo de procesamiento es de 2 minutos.

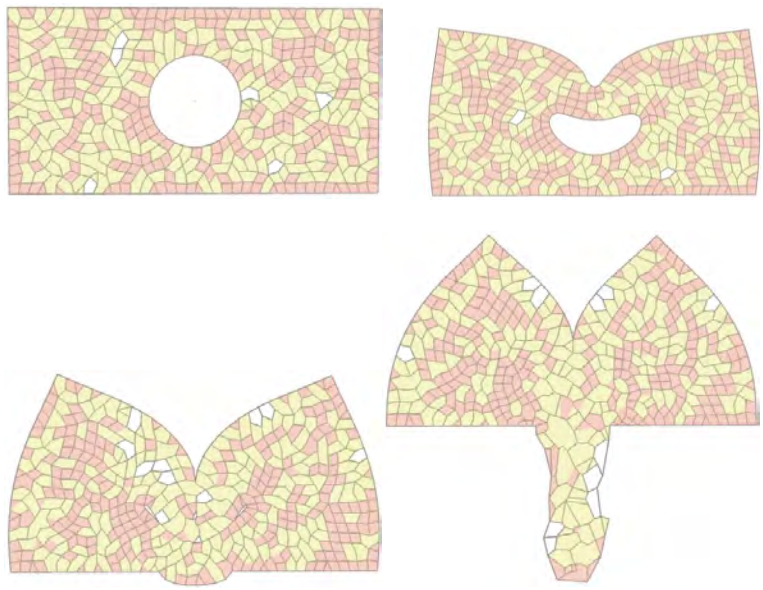


Figura 78. Placa perforada 656 nodos, APR límite 0,6

Es importante mensionar que las geometrías de deformación del problema de 656 nodos y 2568 nodos se realizaron con una carga de mayor magnitud, que la utilizada en el problema de 169. Esto fue necesario devido a que el navegador falla al procesar y almacenar el cálculo iterativo. Este error presumiblemente se puede ver asociado a la capacidad de memoria que puede manejar el navegador, pero el mismo no presenta mayor detalle del por qué se produce.

Al aumentar la carga, las deformaciones en cada iteración auméntan por lo que la sosbreposición de elementos o traslape es mucho más prominente. Particularmente el problema con 656 nodos no permite más de 34 iteraciones, mientras que el problema de 2568 nodos sólo permite 4 iteraciones antes que se produscan errores en la generación de la matriz de rigidez. En la última iteración de la Figura 79 se puede evidenciar comportamiento de la geometría con sobreposición.

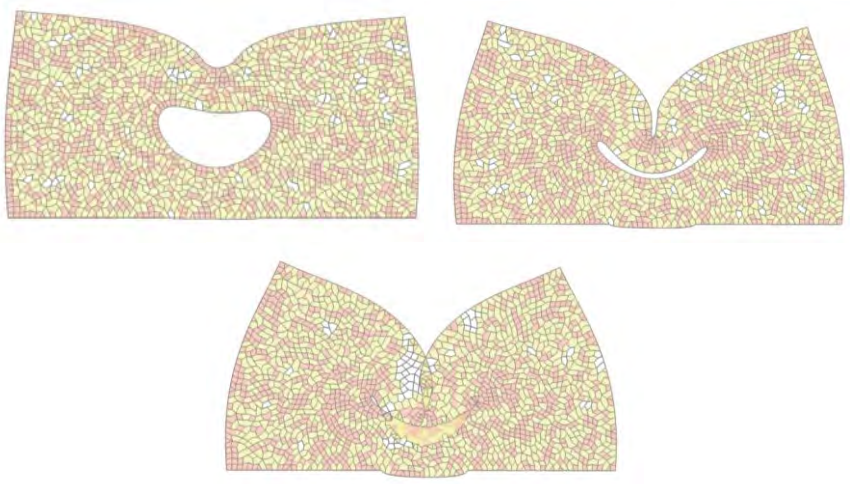


Figura 79. Placa perforada 2568 nodos, APR límite 0,6

4.3 Convergencia numérica mallado tipo D-Polylla

MECHAIDE no contempla un módulo de cálculo de convergencia de desplazamientos nodales respecto a la solución exacta, por lo que el cálculo de normales de convergencia se realizó en VEMLAB importando las mallas D-Polylla generadas desde MECHAIDE.

Se utiliza para la verificación el problema de la viga en voladizo bajo carga vertical parabólica estudiado en la sección 4.1.1.2 considerando 7 mallas D-Polylla adicionales en conjunto con las mallas disponibles en VEMLAB para el cálculo de este problema en particular.

Este problema tiene solución analítica exacta definida en [36] y utilizada en [2], la cual permite determinar el error global del desplazamiento a través de la norma L^2 y el error de global en términos de energía H^1.

La Figura 80 presenta la regresión potencial de la norma L^2 que especifica el error desplazamientos nodales respecto al total de grados de libertad considerando diferentes tipos de mallas. Si bien todas las metodologías entregan errores bajos para un problema de elasticidad lineal, existen diferencias entre los algoritmos.

Utilizando MEV, el algoritmo de mallado D-Polylla tiene un error de desplazamiento nodal un 36% menor que el algoritmo de mallado de tipo MEV Voronoi. Ahora bien, el modelo MEV D-Polylla es un 57% menos preciso que el método MEF T3 para elementos triangulares y un 96% menos preciso el método MEF Q4 para cuadriláteros.

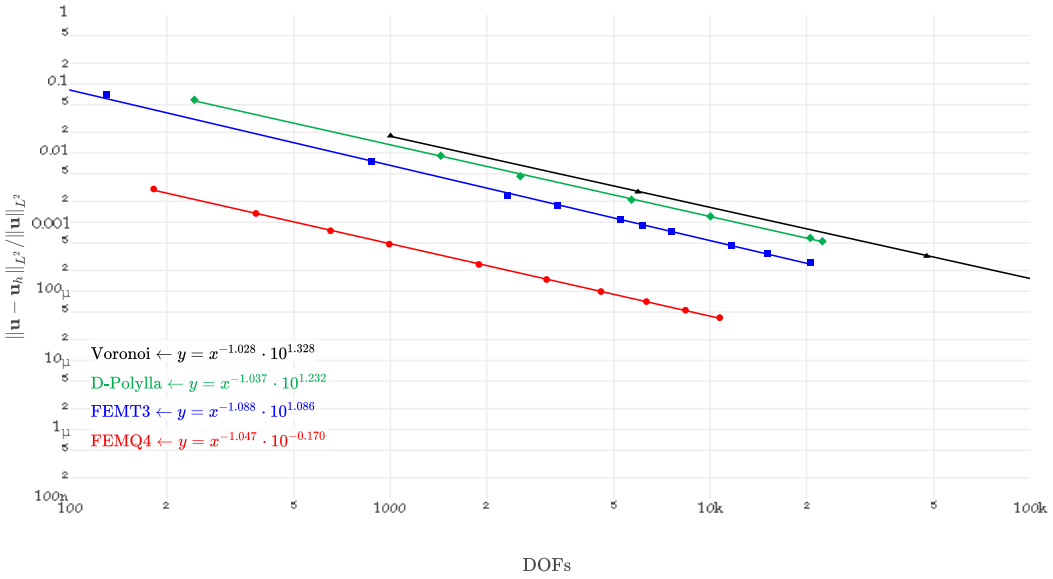


Figura 80. Norma L^2 desplazamientos nodales versus grados de libertad

La Figura 81 presenta la regresión potencial de la semi norma H^1 que especifica el error en términos de energía, respecto al total de grados de libertad considerando diferentes tipos de mallas. Al igual que el caso de la norma L^2 todas las metodologías entregan errores bajos para un problema de elasticidad lineal, pero es relevante mencionar las diferencias objetivas entre los diferentes algoritmos.

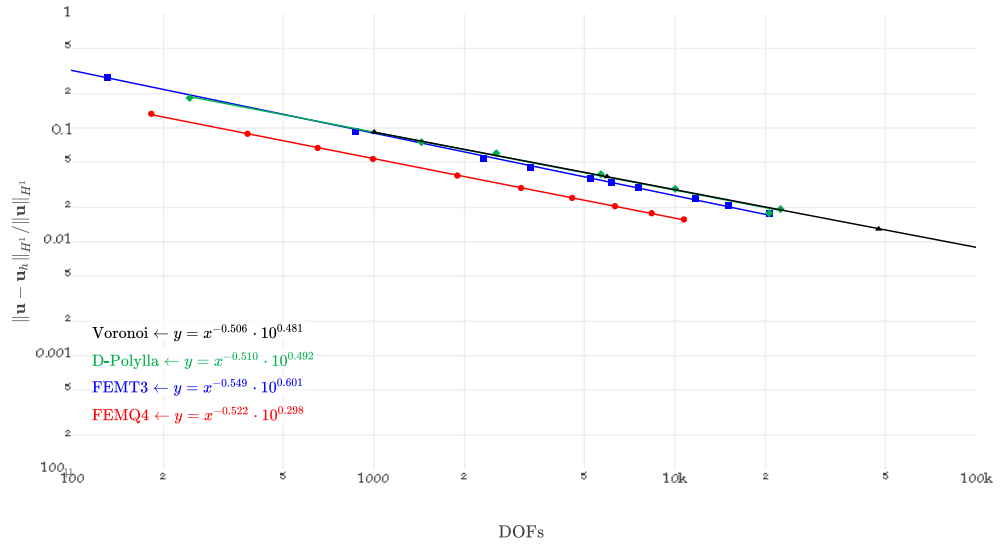


Figura 81. Semi norma H^1 desplazamientos nodales versus grados de libertad.

La metodología MEV D-Polylla tiene un error energético respecto a la solución exacta un 1,4% menor que el algoritmo de mallado de MEV Voronoi, por otro lado, el modelo MEV D-Polylla es un 12% menos preciso que el método MEF T3 para elementos triangulares y un 43% menos preciso el método MEF Q4 para cuadriláteros.

5 Conclusiones

En la presente tesis se desarrolló una aplicación de cálculo para la simulación de problemas de elasticidad lineal en mecánica de sólidos bidimensionales mediante el uso del método de elementos virtuales y el método de elementos finitos utilizando tecnologías de desarrollo web.

La aplicación desarrollada al operar en entorno web, no requiere instalación, ni depende de un hardware específico para funcionar, por lo que puede operar sin mayor configuración en cualquier dispositivo móvil o de escritorio, permitiendo de esta forma que los usuarios puedan ejecutar simulaciones medianamente complejas desde cualquier dispositivo con conexión a internet.

La aplicación a través de la interfaz de usuario en HTML con JavaScript y la estructura de base de datos basada en MariaDB que interactúa con la API desarrollada en PHP, permite a los usuarios generar, manipular, exportar, importar y guardar proyectos, geometrías, mallas, remallados, condiciones de contorno y resultados.

La plataforma permite la importación directa de geometrías CAD cerradas y con perforaciones mediante el algoritmo desarrollado para JavaScript para la importación de regiones desde archivos con formato DXF. Adicionalmente, se implementaron los algoritmos de generación de malla triangular Delaunay y poligonal Voronoi reutilizando código basado en librerías externas. Cabe destacar que en esta tesis se desarrolló un nuevo algoritmo de mallado Direct Polylla (D-Polylla), basado en el algoritmo original Polylla, con optimizaciones para su uso en algoritmos de remallado adaptativo. Es importante resaltar que, a través de la interfaz del programa, todos los tipos de mallas pueden ser refinadas, a través de la selección de nodos en las regiones que el usuario requiera, permitiendo controlar las condiciones para la simulación.

A través del análisis de más de ochenta iteraciones de mallado para un mismo problema se establecieron regresiones no lineales que permiten deducir de forma parcial la cantidad de nodos o elementos, en un determinado tiempo de cálculo. Basado en estas funciones se establecieron diferencias globales que permiten comparar de manera objetiva los diferentes algoritmos de generación de mallas.

De acuerdo con lo expuesto en el párrafo anterior, el algoritmo D-Polylla comparte similitudes de comportamiento con el algoritmo de triangulación de Delaunay respecto a la cantidad de nodos generados en una misma cantidad de tiempo, esto se explica porque los vértices en relajación de ambas mallas corresponden a coordenadas perimetrales de los polígonos y triángulos. Las mallas de tipo Voronoi no son comparables dado que este tipo de malla considera los vértices en relajación como centros geométricos del polígono y la

intersección entre ellos forman los nodos perimetrales del polígono. Esto explica la diferencia de más del 80% en términos de generación de nodos para las mallas de Voronoi por sobre las mallas triangular Delaunay y D-Polylla para un mismo tiempo de generación.

Se estima que en 60 segundos el algoritmo D-Polylla puede generar una malla de 6.500 nodos con 4.400 elementos, el algoritmo generación de malla de Voronoi genera para el mismo tiempo 12.100 nodos para 6.300 elementos, mientras que el algoritmo de generación de malla triangular Delaunay genera 6.300 nodos para 13.200 elementos.

La aplicación cuenta con el algoritmo de cálculo de solución para JavaScript adaptado desde VEMLAB, software desarrollado en Matlab, el cual permite la simulación de problemas estáticos de elasticidad lineal en dos dimensiones para MEV y para MEF. La adaptación del código de VEMLAB fue satisfactoria, obteniendo a través de la simulación de diferentes problemas, resultados exactos en ambos softwares para una misma malla y configuración de condiciones de contorno.

JavaScript no cuenta con soporte nativo para operaciones matriciales optimizadas por lo que fue necesario implementar diferentes algoritmos basados en literatura técnica y librerías de terceros. En particular el algoritmo de inversión de matriz para dar solución al sistema de ecuaciones de la matriz de rigidez se abordó analizando cuatro métodos: "descomposición de Cholesky, L-U, CSG y el algoritmo de la librería externa Numeric.js". Considerando el problema de viga empotrada bajo carga puntual en el extremo, para una malla de 3000 elementos triangulares, se estima que el algoritmo de Cholesky podría resolverlo en 25 segundos, el algoritmo CSG lo resolvería en 33 segundos, Numeric.js lo resolvería en 45 segundos y el algoritmo L-U en 2 minutos con 30 segundos.

Si bien Cholesky presentó mejor rendimiento que los demás métodos, el algoritmo CGS al ser un método iterativo, podría ser más efectivo que la descomposición de Cholesky para problemas sobre los 5500 elementos. Este análisis basado en extrapolación de las regresiones de potencia generadas a partir de los resultados obtenidos podría ser demostrado de manera experimental. Es importante mencionar que esta comprobación es irrelevante considerando los tiempos que maneja MECHAIDE para la resolución de problemas de mayor complejidad por sobre 3000 elementos poligonales, donde se evidencian limitaciones de rendimiento significativos.

Considerando el problema de la biela bajo carga vertical, MECHAIDE para una malla de 3000 elementos, utilizando el modelo de inversión de matriz más eficiente, tarda 18 minutos en completar el cálculo, respecto a los 13,3 segundos que demora VEMLAB. Para mallas de 4.000 elementos los tiempos de cómputo son aún más elevados donde MECHAIDE demora 1 hora con 12 minutos mientras que VEMLAB con el algoritmo de inversión de matriz de Matlab demora 15,8 segundos.

Adicional a la comprobación de resultados del desarrollo frente a VEMLAB, se realizó el análisis comparativo utilizando dos problemas con solución analítica, evaluando los valores de desplazamiento normal máximo y esfuerzo de von Mises máximo. Estos mismos problemas fueron simulados en software de licencia comercial en tres dimensiones (Autodesk Inventor Professional 2016), donde se evidenció el correcto comportamiento de los resultados obtenidos mediante MECHAIDE.

MECHAIDE a través de sus herramientas desarrolladas, permite exportar las mallas generadas al formato de malla requerido por VEMLAB 2.4 con el fin de utilizar el algoritmo de cálculo de normas de errores de desplazamientos nodales L^2 y errores de energía H^1, respecto a la solución real. Considerando el problema de viga empotrada bajo carga parabólica, basado en los resultados obtenidos en [2], los resultados obtenidos permiten evidenciar una buena convergencia, para todas las mallas incluyendo la nueva malla de tipo D-Polylla, la cual es un 36% más preciso según la norma L^2 y un 1,4% más preciso según la seminorma H^1 en comparación con la malla poligonal de tipo Voronoi. El análisis de error global basado en normas es indicador que integra el error de todo el dominio, siendo una medida de desempeño más objetiva que el análisis del error relativo para el desplazamiento máximo utilizado en la sección 4.1.1.2.

En términos de eficiencia de cómputo las mallas triangulares de tipo Delaunay son las más eficientes en todos los casos. Se estima que, para resolver un problema de 3000 elementos poligonales, las mallas de tipo D-Polylla demoraría 7 minutos con 30 segundos, con mallas de tipo Voronoi demoraría 20 minutos con 15 segundos y para una malla triangular Delaunay el tiempo de resoluciones se estima en 1 minuto.

De acuerdo con los lineamientos del proyecto Fondecyt 1221325, se desarrolla algoritmo Poly Remesh (P-Remesh) el cual permite regenerar la conectividad de los nodos de elementos poligonales problemáticos. Este algoritmo utiliza el APR como indicador de calidad del polígono, identificando en base a un umbral a elección, los elementos problemáticos y sus vecindades, para luego romper sus conectividades, sin modificar la posición espacial de los nodos. Al romper la conectividad, el algoritmo procede a generar una nueva malla local con el algoritmo D-Polylla, generando polígonos más estables en la nube de puntos libres del sector.

La aplicación desarrollada permite generar animaciones del comportamiento de la deformación en cada iteración, aplicando las cargas sobre los mismos nodos, pero considerando la malla deformada de la iteración anterior. Este desarrollo permite visualizar el comportamiento del remallado en situaciones límite, pero no intenta simular el comportamiento plástico del sólido, modelamiento que se podría incluir en futuros desarrollos, donde la solución podría considerar la simulación de cuerpos sólidos considerando deformación permanente.

El algoritmo Poly Remesh en términos generales tiene un comportamiento adecuado, se observa en los diferentes problemas donde se prueba el algoritmo, una correcta eliminación de conectividad de nodos y por consiguiente una regeneración de elementos poligonales en las zonas problemáticas. Los nuevos polígonos generados permiten que MEV pueda dar solución al problema evidenciando la validez del método. El modelo es bastante estable y flexible a tal punto que se observa un comportamiento interesante, donde el sólido se adecua a las restricciones del contorno teniendo un comportamiento similar al flujo viscoso.

Al probar el funcionamiento del algoritmo P-Remesh se detectaron problemas en la identificación de los elementos problemáticos mediante el cálculo del APR, donde el umbral fijo establecido para todas las iteraciones puede causar comportamientos no deseados, como el traslape de elementos de la malla. Si bien se identifica que para umbrales de APR de 0,5 a 0,8 el comportamiento es adecuado, el valor del APR idóneo depende de las características del problema y no solo la malla utilizada. Si las cargas aplicadas generan deformaciones en el sólido de tal manera que los elementos se sobreponen, en algunos casos, se producen errores que impiden el cálculo debido a incongruencias en la construcción de la matriz de rigidez. Este comportamiento no deseado, se podría solventar con algoritmos más complejos que permitan la detección de contactos y que a su vez modifiquen de forma dinámica las cargas en los nodos de contorno. Otro enfoque que se podría utilizar es el uso de un indicador de calidad de malla diferente al APR que permita adaptarse a las condiciones de la malla en las diferentes iteraciones.

El estudio detallado del comportamiento de Poly Remesh en condiciones de alta carga computacional queda fuera del alcance de este documento, debido a errores detectados en modelos sobre 2500 nodos, donde la pestaña del navegador falla. Estos errores pueden ser asociados a limitaciones de memoria del navegador, por lo que para establecer alcances de uso y capacidades de la metodología es necesario adaptar el algoritmo a sistemas más robustos de cómputo, no experimentales, fuera del entorno de desarrollo web basado en JavaScript.

El párrafo anterior evidencia la limitación estructural del desarrollo web para su uso en aplicaciones de cálculo numérico. Si bien MECHAIDE puede resolver problemas de mediana complejidad su escalabilidad se ve limitada por la capacidad para resolver sistemas de ecuaciones sobre 3000 nodos. Por esta razón si se proyecta el desarrollo de la aplicación, hacia modelos de cálculo tridimensionales o con algoritmos de carácter no lineal, es imperativo desarrollar algoritmos de inversión de matriz robustos, que permitan el uso de paralelismo y cómputo utilizando la GPU de la máquina, siempre a través del navegador.

Finalmente, si bien los resultados obtenidos por MECHAIDE presentan limitaciones importantes en términos de eficiencia computacional frente a herramientas comerciales o software libre desarrolladas para aplicaciones de cálculo especializado, su valor reside en el

potencial que ofrece para la investigación, al no depender de software de pago, ni de entornos de cálculo cerrados, permitiendo experimentar con toda libertad diferentes soluciones o modificaciones tales como los algoritmos diseñados en el presente documento tanto para el mallado de tipo *Direct Polylla* o el remallado *Poly Remesh*. Adicionalmente, al estar desarrollado en base a tecnologías web, su accesibilidad, escalabilidad y adaptabilidad sin la necesidad de instalar actualizaciones, podría permitir posicionar al software como una herramienta pedagógica ideal para dar solución no sólo a problemas de mecánica de sólidos, sino también a cualquier otro proceso físico que requiera análisis numérico.

Bibliografía

[1] S. Salinas, N. Hitschfeld-Kahler, A. Ortiz-Bernardin y S. Hang, «Polylla: polygonal meshing algorithm based on terminal-edge,» *arXiv,* vol. 38, nº 4545-4567, 2022.

[2] A. Ortiz-Bernardin, «Método del Elemento Virtual,» *Apuntes del curso ME7800, Departmento de Ingeniería Mecánica, Universidad de Chile.,* 2018.

[3] D. Leiva, Impacto de las tecnologías de la información y comunicación en la economía chilena, Santiago: Universidad de Chile, 2003.

[4] B. Arbeláez, «Convertir aplicaciones de escritorio en aplicaciones móviles con Java,» Gestión Básica de la Información, 2010.

[5] Autodesk, Interrupción de las licencias perpetuas para versiones autónomas de productos de Autodesk, 2016.

[6] T. Cormen, Introduction to Algorithms, Third Edition, Cambridge: MIT Press, 2009.

[7] A. Ortiz-Bernardin, «Método del Elemento Finito 1D,» *Apuntes del curso ME7800, Departmento de Ingeniería Mecánica, Universidad de Chile.,* 2020.

[8] Y. Kwon y H. Bang, The finite element method using MATLAB, New York: CRC Press, 1997.

[9] F. Preparata, Computational Geometry: An Introduction, New York: Springer-Verlag, 1985.

[10] A. Ortiz-Bernardin, R. Silva-Valenzuela, S. Salinas-Fernández, N. Hitschfeld-Kahler, S. Luza y B. Rebolledo, «A node-based uniform strain virtual element method for compressible and nearly incompressible elasticity,» *International Journal for Numerical Methods in Engineering,* vol. 128, nº 8, pp. 1818-1855, 2022.

[11] V. Illanes y A. Bergel, Búsqueda automatizada para generar código usando programación genética en un lenguaje de programación orientado a objetos, Santiago: Universidad de Chile, 2021.

[12] V. V. Alvarez, Tecnologías para el desarrollo de sistemas web, Mazatlán, Sinaloa: Universidad Politécnica De Sinaloa, 2017.

[13] Autodesk, «AutoCAD Release 12 DXF Reference,» Autodesk, Inc., 1993.

[14] R. Bridson, «Fast Poisson Disk Sampling in Arbitrary Dimensions,» de *SIGGRAPH07: Special Interest Group on Computer Graphics and Interactive Techniques Conference*, San Diego California , 2007.

[15] C. P. C. Álvarez, Virtual element method for linear elasticity problems in modifiable meshes, Santiago: FCFM, 2017.

[16] T. Sorgente, VEM and the Mesh, Genova: IMATI, 2021.

[17] S. Luza, Desarrollo de una plataforma interactiva de mallado y remallado de mallas poligonales para su uso en un esquema de integración nodal basado en el método del, Santiago: Universidad de Chile, 2022.

[18] B. B., «The Surveyor's Area Formula,» *The College Mathematics Journal,* vol. 17, nº 4, p. 326–337, 1986.

[19] A y J. Peng, «Technologies for 3D mesh compression,» *JVCI,* vol. 16, p. 688–733, 2005.

[20] N. Kim, «Half Edge Data Structure,» de *CSE 528 Stony Brook* , New York, 2016.

[21] S. Salinas, «POLYLLA: Polygonal meshing algorithm based on terminal-edge regions/Polylla-Mesh,» 22 11 2023. [En línea]. Available: https://github.com/ssalinasfe/Polylla-Mesh. [Último acceso: 26 03 2025].

[22] O. Zienkiewicz, The Finite Element Method: Its Basis and Fundamentals, Sixth edition, Oxford: Elsevier, 2005.

[23] J. Gilbert, Sparse matrices in Matlab: Design and implementation, SIAM, 1992.

[24] P. Chico, Elaboración de un motor de animación 3D para three.js, Madrid: Universidad Autonoma de Madrid, 2017.

[25] D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), The Internet Society, 2006.

[26] C. Talischi, «PolyMesher: a general-purpose mesh generator for polygonal elements written in Matlab,» *Struct Multidisc Optim,* nº 45, p. 309–328, 2012.

[27] C. Kimberling, «Triangle Centers and Central Triangles,» *Utilitas Mathematica Publishing,* vol. Congr. Numer. 129, nº 129, pp. 1-295, 1998.

[28] Q. Du, «Convergence of the Lloyd algorithm for computing centroidal Voronoi tessellations,» *SIAM Journal on Numerical Analysis,* nº 4, pp. 102-119, 2006.

[29] J. Holdsworth, «The Nature of Breadth-First Search,» James Cook University, 1999.

[30] S. Attaway, MATLAB: A Practical Introduction to Programming and Problem Solving, Fourth Edition, Cambridge: Elsevier, 2017.

[31] T. Rascia, Understanding the DOM — Document Object Model, New York: DigitalOcean, 2020.

[32] S. Loisel, «Numeric JS - Numerical analysis in Javascript,» 20 12 2012. [En línea]. Available: https://github.com/sloisel/numeric/. [Último acceso: 16 05 2025].

[33] Matplotlib, «Choosing Colormaps in Matplotlib,» [En línea]. Available: https://matplotlib.org/stable/users/explain/colors/colormaps.html. [Último acceso: 04 06 2025].

[34] D. L. Logan, A First Course in the Finite Element Method, Boston: Cengage, 2023.

[35] L. L. Daryl, A First Course in the Finite Element Method, Fifth Edition, Stamford: Cengage Learning, 2012.

[36] S. a. G. J. Timoshenko, Theory Of Elasticity, New York: McGraw-Hill, 1951.

[37] F. Bry y S. Drab, Methods for Polygonalization of a Constructive Solid Geometry Description in Web-based Rendering Environment, LMU, 2012.

[38] B. Neperud, «Visualizing Mesh Data Structures and Algorithms,» Michigan Technological University, Houghton, 2005.

Anexos

Anexo A. Algoritmos auxiliares para importación de geometría

A continuación, se presentan los algoritmos auxiliares para la importación de la geometría desde el archivo .DXF.

Algoritmo 29. Lectura y extracción información CAD (**extractDXF**)

```
input: rows arreglo de n filas del archivo .DXF
    Inicio ← false
    DXF ← Arreglo de arreglos de valores variables
    // Identificar último bloque
    for line_id ← 0 to line_id < n do
        if rows[line_id] = "BLOCK" then
            last_block_id ← line_id
        end
    end
    // Procedimiento de lectura de datos del último bloque
    for id_bloc ← 0 to id_bloc < n do
        if id_bloc > last_block_id then
            line ← rows[x]
            if line = "ARC" or line = "LINE" or line = "CIRCLE" then
                Inicio ← true
            else
                if Inicio ≠ true then
                    continue
                end
            end
            if line = "ENDBLK" then
                return DXF
            end
            // Publicación de variables de Línea
            if x0_OK = true then
                x0 ← line
                x0_OK ← false
            end
            if y0_OK = true then
                y0 ← line
                y0_OK ← false
            end
```

```
if x1_OK = true then
    x1 ← line
    x1_OK ← false
end
if y1_OK = true then
    y1 ← line
    y1_OK ← false
    line_OK ← false
    DXF.add("LINE", x0, y0, x1, y1)
end
// Publicación de variables de Arco
if rx_OK = true then
    rx ← line
    rx_OK ← false
end
if ry_OK = true then
    ry ← line
    ry_OK ← false
end
if r_OK = true then
    r ← line
    r_OK ← false
end
if ars_OK = true then
    ars ← line · PI/180
    ars_OK ← false
end
if are_OK = true then
    are ← line · PI/180
    are_OK ← false
    if are < ars then
        are ← are + 2PI
    end
    x0 ← rx + r cos(ars)
    y0 ← ry + r sin(ars)
    x1 ← rx + r cos(are)
    y1 ← ry + r sin(are)
    arc_OK ← false
    DXF.add("ARC", x0, y0, x1, y1, rx, ry, r, ars, are)
end
// Publicación de variables de circulo
if x_cOK = true then
    rx_c ← line
    rx_cOK ← false
```

```
        end
        if ry_cOK = true then
            ry_c ← line
            ry_cOK ← false
        end
        if r_cOK = true then
            r_c ← line
            r_cOK ← false
            x0 ← rx_c + r_c cos(0)
            y0 ← ry_c + r_c sin(0)
            circle_OK ← false
            DXF.add("CIRCLE", x0, y0, x0, y0, rx_c, ry_c, r_c)
        end
        // Detección de tipo de operación
        if line = "LINE" then
            line_OK ← true
        else if line = "ARC" then
            arc_OK ← true
        else if line = "CIRCLE" then
            circle_OK ← true
        end
        // Asignación de datos a la operación
        if line_OK = true then
            if IsNumeric(line) = true then
                if line = 10 then
                    x0_OK ← true
                else if line = 20 then
                    y0_OK ← true
                else if line = 11 then
                    x1_OK ← true
                else if line = 21 then
                    y1_OK ← true
                end
            end
        else if arc_OK = true then
            if isNumeric(line) = true then
                if line = 10 then
                    rx_OK ← true
                else if line = 20 then
                    ry_OK ← true
                else if line = 40 then
                    r_OK ← true
                else if line = 50 then
                    ars_OK ← true
                else if line = 51 then
```

```
            are_OK ← true
          end
        end
      else if circle_OK ← true then
        if IsNumeric(line) = true then
          if line = 10 then
            rx_cOK ← true
          else if line = 20 then
            ry_cOK ← true
          else if line = 40 then
            r_cOK ← true
          end
        end
      end
    end
  end
end
```

Algoritmo 30. Orden de operaciones geométricas (**orderDXF**)

Input: $\boldsymbol{DXF}$ Arreglo de arreglo de n operaciones geométricas

```
DXF_new ← Arreglo de arreglo de n valores variables
first_ok ← true
region_id ← −1
// Inicio iteración de operaciones geométricas
for i ← 0 to i < n do
  // Extracción de valores actuales
  tipo = DXF.tipo
  x0 = DXF.x0[i]
  y0 = DXF.y0[i]
  x1 = DXF.x1[i]
  y1 = DXF.y1[i]
  rx = DXF.rx[i]
  ry = DXF.ry[i]
  r = DXF.r[i]
  ars = DXF.ars[i]
  are = DXF.are[i]
  id = DXF.id[i]
  region_id = DXF.region_id[i]
  giro = DXF.giro[i]
  //Extracción de posiciones futuras
  if i < n − 1 then
    x0_fut ← DXF.x0[i + 1]
```

```
        y0_fut ← DXF.y0[i + 1]
        x1_fut ← DXF.x1[i + 1]
        y1_fut ← DXF.y1[i + 1]
    end
    //Procesamiento de círculos
    if tipo_old = "CIRCLE" then
        first_ok ← true
    end
    if tipo = "CIRCLE" then
        if i = n then
            giro ← 1
        else
            giro ← −1
        end
        ars ← 0
        are ← 2 · PI
        first_ok ← true
    end
    // Procesamiento de líneas
    if tipo = "LINE" then
        if first_ok then
            if (x0_fut = x1 and y0_fut = y1) or (x1_fut = x1 and y1_fut = y1) then
                giro ← 1
            else
                x0_temp ← x0
                y0_temp ← y0
                x0 ← x1
                y0 ← y1
                x1 ← x0_temp
                y1 ← y0_temp
                giro ← −1
            end
        else if x0 = x1_old and y0 = y1_old then nothing
        else if x1 = x1_old and y1 = y1_old then
            x0_temp ← x0
            y0_temp ← y0
            x0 ← x1
            y0 ← y1
            x1 ← x0_temp
            y1 ← y0_temp
        end
        giro ← 0
    end
    //Procesamiento de arcos
```

```
if tipo = "ARC" then
    if first_ok then
        if (x0_fut = x and y0_fut = y1) or (x1_fut = x1 and y1_fut = y1) then
            giro ← 1
        else
            x0_temp ← x0
            y0_temp ← y0
            x0 ← x1
            y0 ← y1
            x1 ← x0_temp
            y1 ← y0_temp
            giro ← −1
        end
    else if x0 = x1_old and y0 = y1_old then
        giro ← 1
    else if x1 = x1_old and y1 = y1_old then
        x0_temp ← x0
        y0_temp ← y0
        x0 ← x1
        y0 ← y1
        x1 ← x0_temp
        y1 ← y0_temp
        giro ← −1
    end
end
if tipo_old = "CIRCLE" then
    first_ok ← true
end
if first_ok = true then
    region_id ← region_id + 1
    x_first ← x0
    y_first ← y0
    first_ok ← false
else if x1 = x_first and y1 = y_first then
    first_ok ← true
end
//Agregar operaciones geométricas al resultado
DXF_new.add(tipo, x0, y0, x1, y1, rx, ry, r, ars, are, id, region_id, giro)
//Asignación de variables anteriores
tipo_old ← tipo
x0_old ← x0
y0_old ← y0
x1_old ← x1
y1_old ← y1
```

```
    end
    return DXF_new
    end
end
```

Algoritmo 31. Distancia entre puntos (**distance**)

Inputs: $p1$, $p2$: puntos en el plano cartesiano en dos dimensiones.

```
    return ((p2[0].p1[0])^2 + (p2[1].p1[1])^2)^0.5
end
```

Algoritmo 32. Encontrar límites de una geometría (**find_limits**)

Inputs: ***points*** Arreglo de n puntos en dos dimensiones

```
    minX ← Infinity
    minY ← Infinity
    maxX ← −Infinity
    maxY ← −Infinity
    for each points as point do
        minX ← min(minX, point.x)
        minY ← min(minY, point.y)
        maxX ← max(maxX, point.x)
        maxY ← max(maxY, point.y)
    end
    return [minX, minY, maxX, maxY]
end
```

Anexo B. Algoritmos auxiliares para mallado y refinamiento

A continuación, se presentan los algoritmos auxiliares la generación de mallas.

Algoritmo 33. Relajación de Lloyd (**lloyd_relaxation**)

Inputs: ***points*** arreglo de vértices en dos dimensiones libres de la malla poligonal, ***fixedPoints*** arreglo de vértices en dos dimensiones fijos de la malla poligonal , $minX$ límite menor en la coordenada de eje horizontal, $minY$ límite menor en la coordenada del eje vertical, $maxX$ límite mayor de la coordenada del horizontal, $maxY$ límite mayor de la coordenada del eje vertical, ***contorno*** arreglo de vértices que describen el contorno exterior de la geometría en dos dimensiones, ***perforaciones*** arreglo de dos dimensiones con los vértices que describen el contorno de las perforaciones de la geometría en dos dimensiones, d distancia objetivo entre vértices, *mesh_type* tipo de malla, *points_have_fixed* variable booleana para definir si existen vértices inamovibles.

```
// Combinar puntos móviles y fijos para la triangulación de Delaunay
combinedPoints ← points_have_fixed ? points : points.concat(fixedPoints)
// Calcular la triangulación de Delaunay y teselación de Voronoi en el cuadro delimitado.
delaunay ← d3.Delaunay.from(combinedPoints)
voronoi ← delaunay.voronoi([minX − d, minY − d, maxX + d, maxY + d])
// Crear un "set" para una búsqueda rápida de vértices fijos
fixedPointsSet ← new Set(fixedPoints.map((point) → JSON.stringify(point)))
// Pasar polígonos para contornos y perforaciones formando objetos para la librería Turf.js
contourPolygon ← turf.polygon([contorno])
holePolygons ← perforaciones.map((hoyo) → turf.polygon([hoyo]))
// Aplicar la relajación de Lloyd sólo a los puntos móviles
foreach point, i of points do
    pointKey ← JSON.stringify(point)
    if fixedPointsSet.has(pointKey) then
        return point // Saltar puntos fijos
    end
    // Obtener la celda de Voronoi correspondiente
    polygon ← voronoi.cellPolygon(i)
    // Calcular el centroide de la celda de Voronoi
    centroid ← d3.polygonCentroid(polygon)
    // Crear punto de Turf.js para verificaciones espaciales
    centroidPoint ← turf.point([centroid.x, centroid.y])
    // Verificar si el centroide está dentro del contorno
    if !turf.booleanPointInPolygon(centroidPoint, contourPolygon) then
        return point // Fuera del contorno
    end
    // Verificar si el centroide está dentro de alguna perforación
    foreach hole of holePolygons do
        if turf.booleanPointInPolygon(centroidPoint, hole) then
            return point; // Dentro de la perforación
        end
    end
    // Verificar distancia mínima al contorno
    distContorno ← distance_to_polygon(centroid, contorno)
    if distContorno < d then
        return point // Demasiado cerca del contorno
    end
    // Verificar distancia mínima a las perforaciones
    for j = 0 to j < hoyos.length do
        distHoyo ← distance_to_polygon(centroid, hoyos[j])
        if distHoyo < d then
```

```
            return point // Demasiado cerca de una perforación
        end
    end
    // Verificar distancia mínima a todos los puntos fijos
    foreach fixedPoint of fixedPoints do
        distPuntoFijo ← hypot(
            centroid.x − fixedPoint.x,
            centroid.y − fixedPoint.y
        )
        if distPuntoFijo < d then
            return point // Demasiado cerca de un punto fijo
        end
    end
    // Verificar distancia mínima a otros puntos móviles (excluyéndose a sí mismo)
    foreach otherPoint of points do
        if otherPoint = point then continue; // Saltar así mismo
        distOtroPunto ← hypot(
            centroid.x − otherPoint.x,
            centroid.y − otherPoint.y
        )
        if distOtroPunto < d then
            return point // Demasiado cerca de otro punto móvil
        end
    end
    // Si todas las condiciones se cumplen, mover el punto al centroide
    return centroid
  end
  return relaxedPoints
end
```

Algoritmo 34. Polígonos de vértices (get_polygons_from_vertices)

```
Inputs: connect, vertexIndicesToRefine
polygonsToRefine ← [ ]
vertexSet ← new Set(vertexIndicesToRefine)
foreach (polygon, polygonIndex) of connect do
    for i = 0 to i < polygon.length do
        if vertexSet.has(polygon[i]) then
            polygonsToRefine.push(polygonIndex)
            break // No es necesario verificar más vértices en este polígono
        end
    end
```

```
    end
    return polygonsToRefine
end
```

Algoritmo 35. Extraer contornos desde estructura HE (**countours_from_HE**)

Inputs: ***halfEdgeMesh*** información HE de la malla triangular, ***groups*** arreglo de arreglos de identidades de polígonos que conforman grupos conectados por dos o más vértices, ***coords*** arreglo de coordenadas de los nodos de la malla.

```
{ vertices, halfEdges, faces } ← halfEdgeMesh
groupContours ← [ ]
// Construir un mapa de caras a grupos (A qué grupo pertenece cada cara)
faceIndexToGroupIndex ← new Map()
foreach group, groupIndex of groups do
    foreach faceIndex of group do
        faceIndexToGroupIndex.set(faceIndex, groupIndex)
    end
end
// Procesar cada grupo
for groupIndex = 0 to groupIndex < groups.length do
    group ← groups[groupIndex]
    boundaryEdges ← new Set()
    edgeConnections ← new Map()
    // Paso 1: Recolectar aristas de borde
    foreach faceIndex of group do
        face ← faces[faceIndex]
        edgeIndex ← face.edge
        startEdgeIndex ← edgeIndex
        while edgeIndex ≠ startEdgeIndex do
            he ← halfEdges[edgeIndex]
            twinIndex ← he.twin
            isBoundaryEdge ← false
            // Verificar si es una arista de contorno comprobando si la arista no tiene gemelo o si su
            gemelo pertenece a otro grupo
            if twinIndex = null then
                isBoundaryEdge ← true
            else
                twinHe ← halfEdges[twinIndex]
                adjacentFaceIndex ← twinHe.face.index
                adjacentGroupIndex ← faceIndexToGroupIndex.get(adjacentFaceIndex)
                if adjacentGroupIndex ≠ groupIndex then
```

```
            isBoundaryEdge ← true
        end
    end
    if isBoundaryEdge then
        boundaryEdges.add(edgeIndex)
        originIndex ← he.origin.index
        targetIndex ← he.target.index
        // Almacenar conexiones para construir contornos
        if !edgeConnections.has(originIndex) then
            edgeConnections.set(originIndex, new Set())
        end
        if !edgeConnections.has(targetIndex) then
            edgeConnections.set(targetIndex, new Set())
        end
        edgeConnections.get(originIndex).add(targetIndex)
        edgeConnections.get(targetIndex).add(originIndex)
    end
    edgeIndex ← he.next
  end
end
// Paso 2: Encontrar todos los contornos recorriendo componentes conectados
visitedNodes ← new Set()
visitedEdges ← new Set()
contoursInGroupIndices ← [ ]
for node of edgeConnections.keys() do
  if visitedNodes.has(node) then
    continue
  end
  contourIndices ← [ ]
  currentNode ← node
  startNode ← node
  while (true) do
    contourIndices.push(currentNode)
    visitedNodes.add(currentNode)
    neighbors ← edgeConnections.get(currentNode)
    nextNode ← null
    // Seleccionar un vecino con una arista no visitada
    for neighbor of neighbors do
      heIndex ← null
      // Buscar la HE correspondiente
```

```
            for edgeIndex of boundaryEdges do
                he ← halfEdges[edgeIndex]
                if (he.origin.index = currentNode and he.target.index = neighbor) or
                 (he.origin.index = neighbor and he.target.index = currentNode) then
                    heIndex ← edgeIndex
                    break
                end
            end
            if heIndex ≠ null and !visitedEdges.has(heIndex) then
                visitedEdges.add(heIndex)
                nextNode ← neighbor
                break
            end
        end
        if nextNode = null then
            break
        end
        currentNode ← nextNode
        if currentNode = startNode then
            contourIndices.push(currentNode)
            break
        end
    end
    if contourIndices.length > 1 then
        contoursInGroupIndices.push(contourIndices)
    end
end
// Paso 3: Convertir índices a coordenadas
contoursCoordinates ← [ ]
for contourIndices of contoursInGroupIndices do
    if coords ≠ null then
        contourCoords ← contourIndices.map(index → coords[index])
        contoursCoordinates.push(contourCoords)
    else
        contoursCoordinates.push([ ])
    end
end
// Almacenar resultados del grupo
groupContours.
push({group, contoursIndices: contoursInGroupIndices, contoursCoordinates: contoursCoordinate
```

```
    return groupContours
end
```

Algoritmo 36. Triangulación de Delaunay (**triangulation**)

```
Inputs: points arreglo de puntos de la malla.
    delaunay ← Delaunator.from(points)
    triangles ← [ ]
    for i = 0 to i < delaunay.triangles.length do
        a ← delaunay.triangles[i]
        b ← delaunay.triangles[i + 1]
        c ← delaunay.triangles[i + 2]
        triangles.push([points[a], points[b], points[c]])
    end
    return ensure_CCW_triangle(triangles)
end
```

Algoritmo 37. Asegurar CCW en triángulos (**ensure_CCW_triangle**)

```
Inputs: triangles arreglo de triángulos que conforman la malla.
    return triangles.map((tri) → {
        det ← compute_det(tri)
        if det < 0 then
            // Triángulo en orden CW, intercambiar dos puntos para hacerlo CCW
            return [tri[0], tri[2], tri[1]]
        end
        return tri
    })
end
```

Algoritmo 38. Cálculo determinante (**compute_det**)

```
Inputs: triangle arreglo de coordenadas que conforman el triángulo
    [a, b, c] = triangle
    return (b[0] − a[0]) · (c[1] − a[1]) − (b[1] − a[1]) · (c[0] − a[0])
end
```

Algoritmo 39. Cálculo de centroide en triángulo (**compute_centroid**)

```
Inputs: triangle arreglo de coordenadas que conforman el triángulo
    [a, b, c] = triangle
    x ← (a[0] + b[0] + c[0]) / 3
```

$\boldsymbol{y} \leftarrow \frac{(\boldsymbol{a}[1] + \boldsymbol{b}[1] + \boldsymbol{c}[1])}{3}$
return $[\boldsymbol{x}, \boldsymbol{y}]$
end

Algoritmo 40. Punto está dentro del polígono (**point_in_polygon**)

Inputs: ***point*** punto en el plano bidimensional, ***polygon*** arreglo de coordenadas que conforman el polígono

$[x, y] = point$
$inside \leftarrow$ **false**
for $i = 0, j =$ ***polygon*****.length** - 1 **to** $i <$ ***polygon*****. length do**
  $x_i \leftarrow$ ***polygon***$[i][0], y_i \leftarrow$ ***polygon***$[i][1]$
  $x_j \leftarrow$ ***polygon***$[j][0], y_j \leftarrow$ ***polygon***$[j][1]$
  $intersect = y_i > y \neq y_j > y$ **and** $x < \frac{(x_j - x_i) \cdot (y - y_i)}{y_j - y_i} + x_i$
  if $intersect$ **then** $inside = !\, inside$
end
return $inside$
end

Algoritmo 41. Centroide dentro de dominio (**is_centroid_valid**)

Inputs: ***centroid*** centroide del polígono, ***contorno*** arreglo de puntos de contorno exterior, ***hoyos*** arreglo de arreglo de puntos que conforman perforaciones de la superficie del cuerpo en dos dimensiones

// Verificar si el centroide está dentro del contorno
if (! **point_in_polygon** (***centroid***, ***contorno***)) **return false**
// Verificar si el centroide está dentro de alguna perforación
for ***hoyo*** **of** ***hoyos*** **then**
  if (**point_in_polygon** (***centroid***, ***hoyo***)) **return false**
end
return true
end

Algoritmo 42. Datos de malla desde polígonos (**extract_mesh**)

Inputs: ***polygons*** arreglos de polígonos con puntos que conforman en dos dimensiones.

coords $\leftarrow$ [ ]
coordsMap $\leftarrow$ **new Map**()
$index \leftarrow 0$
connect $\leftarrow$ [ ]
foreach (***polygon***, $polygonIndex$) **of** ***polygons*** **do**

```
    element ← [ ]
    previousIndex ← null
    for point in polygon do
        key ← point.join(',')
        if !coordsMap.has(key) then
            coordsMap.set(key, index)
            coords.push(point)
            index++
        end
        pointIndex ← coordsMap.get(key)
        if pointIndex ≠ previousIndex then
            element.push(pointIndex)
        end
        previousIndex ← pointIndex
    end
end
return {coords, connect, mesh: polygons}
end
```

Algoritmo 43. Malla local a la malla global (to_global_*connect*)

Inputs: ***coords_origen***, arreglo de coordenadas originales de la malla, ***coords*** arreglo de coordenadas locales, ***connect*** matriz de conectividad del arreglo de coordenadas locales.

```
// Crear un mapa de índice local a índice global
indiceMap ← new Map()
foreach (coord, localIndex) of coords do
    globalIndex ← coords_origen.findIndex(
        globalCoord → globalCoord[0] = coord[0] and globalCoord[1] = coord[1]
    )
    if globalIndex ≠ −1 then
        indiceMap.set(localIndex, globalIndex)
    end
})
return connect.map(polygon →
    polygon.map(localIndex → indiceMap.get(localIndex))
)
end
```

Anexo C. Algoritmos auxiliares para el solucionador

A continuación, se presentan los algoritmos auxiliares del solucionador.

Algoritmo 44. Cálculo de parámetros del material (**material_parameters**)

Inputs: Ey módulo de elasticidad del material, nu coeficiente de Poisson, *plane_state* comportamiento elástico, *method* método de solución ya sea MEV o MEF.

$\boldsymbol{matProps} \leftarrow \{\ \}$
$\boldsymbol{matProps}.Ey = Ey$
$\boldsymbol{matProps}.nu = nu$
$\boldsymbol{matProps}.plane\_state = plane\_state$
if $plane\_state =$ "*plane_strain*" **then**
  $lam \leftarrow \dfrac{Ey \cdot nu}{(1+nu)\cdot(1-2\cdot nu)}$
  $mu \leftarrow \dfrac{Ey}{2\cdot(1+nu)}$
  $\boldsymbol{D} \leftarrow$ **numeric.mul**$(Ey/((1+nu)\cdot(1-2\cdot nu)), [$
    $[1-nu, nu, 0],$
    $[nu, 1-nu, 0],$
    $[0, 0, 2\ (1-2nu)],$
  $])$
  $nu\_bar \leftarrow \dfrac{nu}{1-nu}$
  $Ey\_bar \leftarrow \dfrac{Ey}{1-nu^2}$
else if $plane\_state =$ "*plane_stress*" **then**
  $lam \leftarrow \dfrac{Ey \cdot nu}{(1+nu)\cdot(1-2nu)}$
  $mu \leftarrow \dfrac{Ey}{2\cdot(1+nu)}$
  $\boldsymbol{D} \leftarrow$ **numeric.mul**$(Ey/(1-nu^2),$
  $[$
    $[1, nu, 0],$
    $[nu, 1, 0],$
    $[0, 0, 2(1-nu)],$
  $])$
  $nu\_bar \leftarrow nu$
  $Ey\_bar \leftarrow Ey$
end
$\boldsymbol{matProps}.D = D$
$\boldsymbol{matProps}.nu\_bar = nu\_bar$
$\boldsymbol{matProps}.Ey\_bar = Ey\_bar$
$\boldsymbol{matProps}.lam = lam$
$\boldsymbol{matProps}.mu = mu$
return ***matProps***
end

Algoritmo 45. Formato de malla para solución (**format_mesh**)

Inputs: ***mesh*** Información de la malla 2D, *neumann_nodes* Nodos seleccionados para condición de Neumann, *dirichlet_nodes* Nodos seleccionados para condición de Dirichlet.

```
coords ← mesh.content.coords.map(i → [i[0], i[1]])
connect ← [ ]
function set_boundary_DOFs(bound_nodes) {
    result ← new Array(2 · bound_nodes.length).fill(0)
    // Llenar el arreglo con los índices de los nodos
    for i = 0 to i < bound_nodes.length do
        idx1 ← 2 · i
        idx2 ← idx1 + 1
        // Asignar valores directamente
        result[idx1] = 2 · bound_nodes[i]
        result[idx2] = 2 · bound_nodes[i] + 1
    end
    return result
end
neumann_dofs ← set_boundary_DOFs(neumann_nodes)
dirichlet_dofs ← set_boundary_DOFs(dirichlet _nodes)
return {
    num_nodes: coords.length,
    coords,
    num_elements: mesh.content.connect.length,
    mesh.content.connect,
    neumann_dofs,
    neumann_nodes,
    dirichlet_dofs,
    dirichlet_nodes,
end
end
```

Algoritmo 46. Fórmulas de Neumann (**create_neumann_functions**)

Inputs: ***node_info*** Información de los nodos seleccionados.

```
neumannFunctions ← { }
for nodeId of node_info.cb_neu do
    nodeData ← node_info.cb_neu[nodeId]
    if nodeData.code_x then
        neumannFunctions[nodeId] = {
            f_x: new Function("x", "y", "arr_x", "arr_y", "matProps", nodeData.code_x),
        }
```

```
    end
    if nodeData.code_y then
        neumannFunctions[nodeId] = {
            ...neumannFunctions[nodeId],
            f_y: new Function("x", "y", "arr_x", "arr_y", "matProps", nodeData.code_y),
        }
    end
end
return neumannFunctions
end
```

Algoritmo 47. Fórmulas de Dirichlet (create_dirichlet_functions)

```
Inputs: node_info Información de los nodos seleccionados.
dirichletFunctions ← { }
for nodeId of node_info.cb_dir do
    nodeData ← node_info.cb_dir[nodeId]
    if nodeData.code_x then
        dirichletFunctions[nodeId] = {
            u_x: new Function("x", "y", "arr_x", "arr_y", "matProps", nodeData.code_x),
        }
    end
    if nodeData.code_y then
        dirichletFunctions[nodeId] = {
            ...dirichletFunctions[nodeId],
            u_y: new Function("x", "y", "arr_x", "arr_y", "matProps", nodeData.code_y),
        }
    end
end
return dirichletFunctions
end
```

Algoritmo 48. Fórmulas de fuerza nodal (**create_body_force_functions**)

```
Inputs: node_info Información de los nodos seleccionados.
body_force_info ← node_info.cb_corporal
b_x ← body_force_info.code_x or "return 0;"
b_y ← body_force_info.code_y or "return 0;"
return {
    b_x: new Function("x", "y", "arr_x", "arr_y", "matProps", b_x),
    b_y: new Function("x", "y", "arr_x", "arr_y", "matProps", b_y),
}
end
```

Algoritmo 49. Ensamble de matriz de rigidez (**assembly**)

Inputs: ***domainMesh*** Información de la malla, ***config*** Parámetros de configuración de la solución, ***props*** propiedades del material, ***body_force_fun_values*** Valores de fuerza corporal.

```
// Determina el módulo y método para el ensamblaje
if config.method = "VEM2D" then
    // Ensambla usando el método MEV para lineal en 2D
    {K_global, f_global}
    ← vem_assembly(domainMesh, props, body_force_fun_values, config)
else if config.method = "FEM2DT3" or config.method = "FEM2DQ4" then
    // Ensambla usando el método MEF con elementos triangulares o cuadriláteros
    {K_global, f_global}
    ← fem_assembly(domainMesh, props, body_force_fun_values, config)
end
// Retorna la matriz de rigidez global y el vector de fuerzas global
return { K_global, f_global }
end
```

Algoritmo 50. Ensamble de matriz de rigidez MEV (**vem_sparse_assembly**)

Inputs: ***domainMesh*** Información de la malla, ***matProps*** propiedades del material, ***body_force_fun*** Funciones de fuerza corporal, ***config*** Parámetros de configuración de la solución

```
num_nodes ← domainMesh.coords.length
numel ← domainMesh.connect.length
len_vector_K←0
len_vector_f←0
// Calcula la longitud total requerida para los vectores de ensamblaje
for e = 0 to e < numel do
    num_eldofs ← 2 · domainMesh.connect[e].length
    len_vector_K += num_eldofs^2
    len_vector_f += num_eldofs
end
// Inicializa los vectores para ensamblar la matriz y el vector globales
vector_K ← new Array(len_vector_K).fill(0)
indrow_K ← new Array(len_vector_K).fill(0)
indcol_K ← new Array(len_vector_K).fill(0)
vector_f ← new Array(len_vector_f).fill(0)
indrow_f ← new Array(len_vector_f).fill(0)
pos_vector_K ← 0
pos_vector_f ← 0
// Itera sobre cada elemento del dominio
for e = 0 to e < numel do
    nodes ← domainMesh.connect[e]
```

```
    verts ← nodes.map(node → domainMesh.coords[node])
    // Calcula los índices globales de los grados de libertad (DOFs) del elemento
    ind ← [ ]
    foreach (node, index) of nodes do
        ind[2 · index] = 2 · node
        ind[2 · index + 1] = 2 · node + 1
    end
    num_eldofs ← 2 · nodes.length
    // Calcula la matriz de rigidez local y el vector de fuerzas local para el elemento
    K_local ← vem_stiffness(verts, matProps, config)
    f_local ← vem_body_force (verts, body_force_fun, matProps)
    // Verifica si la matriz local contiene valores no numéricos
    if K_local.flat().some(value → isNaN(value)) then
        continue
    end
    // Ensambla la matriz de rigidez global agregando K_local en las posiciones correctas
    for i = 0 to i < num_eldofs do
        for j = 0 to j < num_eldofs do
            indrow_K[pos_vector_K] = ind[i]
            indcol_K[pos_vector_K] = ind[j]
            vector_K[pos_vector_K] = K_local[i][j]
            pos_vector_K++
        end
    end
    // Ensambla el vector de fuerzas global agregando f_local en las posiciones correctas
    for i = 0 to i < num_eldofs do
        indrow_f[pos_vector_f] = ind[i]
        vector_f[pos_vector_f] = f_local[i]
        pos_vector_f++
    end
end
// Construye la matriz y el vector globales en formato disperso
row_vector_f ← Array(len_vector_f).fill(0)
K_global ← sparse(indrow_K, indcol_K, vector_K)
f_global ← sparse(indrow_f, row_vector_f, vector_f)
return { K_global, f_global }
end
```

Algoritmo 51. Ensamble de matriz de rigidez local MEV (**vem_stiffness**)

Inputs: ***verts*** vértices en 2D, ***matProps*** Propiedades del material, ***config*** configuración de la solución.

```
area ← polyarea(verts) // Calcula el área del elemento
// Calcula las matrices específicas del MEV para el elemento
Wc ← vem_Wc(verts)
Wr ← vem_Wr (verts)
Hc ←  vem_Hc (verts)
Hr ←  vem_Hr (verts)
Pp
← numeric.add(numeric.dot(Hr, numeric.transpose(Wr)), numeric.dot(Hc, numeric.transp
I_2N ← eye(2 · verts.length)
if config.stability_type = 0 then
    // Si no se aplica estabilización [Implementación fuera de los alcances]
else
    if config.stability_type = 1 then
        // Estabilización según Gain et al.
        gamma ← 1.0
        alpha = gamma · area · math.trace(matProps.D) / math.trace(numeric.dot(numeric.transpose(Hc), Hc))
        Se ← numeric.mul(alpha, I_2N)
    else if config.stability_type = 2 then
        // Estabilización D-recipe [Implementación fuera de los alcances]
    else if config.stability_type = 3 then
        // Estabilización D-recipe modificada [Implementación fuera de los alcances]
    end
    // Calcula el primer término de la matriz de rigidez
    Wc^⊤ ← numeric.transpose(Wc)
    part1 ← numeric.mul(area, numeric.dot(Wc, numeric.dot(matProps.D, Wc^⊤)))
    // Calcula el segundo término de la matriz de rigidez (estabilización)
    part2
    ← numeric.dot(numeric.transpose(numeric.sub(I_2N, Pp)), numeric.dot(Se, numeric.sub(I
    // Suma ambos términos para obtener la matriz de rigidez local
    Kvem ← numeric.add(part1, part2)
end
return Kvem
end
```

Algoritmo 52. Cálculo $\boldsymbol{Wc}$ para MEV (**vemWc**)

Inputs: $\boldsymbol{verts}$ Vertices en plano 2D.

$area \leftarrow \textbf{polyarea}(\boldsymbol{verts})$
$\boldsymbol{v1} \leftarrow [\ldots \boldsymbol{verts}, \boldsymbol{verts}[0]]$
$\boldsymbol{n} \leftarrow \textbf{normals\_to\_edges}(\boldsymbol{verts})$
$\boldsymbol{n1} \leftarrow \boldsymbol{n}.\textbf{map}((\boldsymbol{val}) \rightarrow \boldsymbol{val}[0])$
$\boldsymbol{n2} \leftarrow \boldsymbol{n}.\textbf{map}((\boldsymbol{val}) \rightarrow \boldsymbol{val}[1])$
$\boldsymbol{n11} \leftarrow [\boldsymbol{n}[\boldsymbol{n}.\textbf{length} - 1][0], \ldots \boldsymbol{n1}.\textbf{slice}(0, -1)]$
$\boldsymbol{n22} \leftarrow [\boldsymbol{n}[\boldsymbol{n}.\textbf{length} - 1][1], \ldots \boldsymbol{n2}.\textbf{slice}(0, -1)]$
// Calcula las longitudes de los bordes
$len \leftarrow \boldsymbol{verts}.\textbf{map}((v, i) \rightarrow \textbf{hypot}(\boldsymbol{v1}[i][0] - \boldsymbol{v1}[i+1][0], \boldsymbol{v1}[i][1] - \boldsymbol{v1}[i+1][1]))$
$\boldsymbol{len1} \leftarrow [\boldsymbol{len}[len.\textbf{length} - 1], \ldots len.\textbf{slice}(0, -1)]$
// Calcula los valores de $\boldsymbol{Wc}$ para cada vértice
$\boldsymbol{Wc} \leftarrow [\ ]$
for $i = 0$ **to** $i < verts.\textbf{length}$ **do**
  $q1a \leftarrow \dfrac{0.25 \cdot (\boldsymbol{len1}[i] \cdot \boldsymbol{n11}[i] + \boldsymbol{len}[i] \cdot \boldsymbol{n1}[i])}{area}$
  $q2a \leftarrow \dfrac{0.25 \cdot (\boldsymbol{len1}[i] \cdot \boldsymbol{n22}[i] + \boldsymbol{len}[i] \cdot \boldsymbol{n2}[i])}{area}$
  $\boldsymbol{Wc\_a} \leftarrow [[2 \cdot q1a, 0, q2a], [0, 2 \cdot q2a, q1a]]$
  $\boldsymbol{Wc}.\textbf{push}(\ldots \boldsymbol{Wc\_a})$
end
return $\boldsymbol{Wc}$
end

Algoritmo 53. Cálculo de normales de aristas (**normals_to_edges**)

Inputs: $\boldsymbol{verts}$ Vertices en plano 2D.

$\boldsymbol{v1} \leftarrow [\ldots \boldsymbol{verts}, \boldsymbol{verts}.x]$ // Añade el primer vértice al final
$\boldsymbol{n1} \leftarrow [\ ]$
for $i = 0$ **to** $i < verts.\textbf{length}$ **do**
  $\boldsymbol{w} \leftarrow [\boldsymbol{v1}[i+1][0] - \boldsymbol{v1}[i][0], \boldsymbol{v1}[i+1][1] - \boldsymbol{v1}[i][1]]$ // Calcula el vector de borde
  $\boldsymbol{n} \leftarrow [\boldsymbol{w}[1], -\boldsymbol{w}[0]]$ // Rota el vector de borde para obtener el vector normal
  // Normaliza el vector normal
  $norm \leftarrow \textbf{sqrt}(\boldsymbol{n}[0] \cdot \boldsymbol{n}[0] + \boldsymbol{n}[1] \cdot \boldsymbol{n}[1])$
  $\boldsymbol{n1}.\textbf{push}\left(\left[\dfrac{\boldsymbol{n}[0]}{norm}, \dfrac{\boldsymbol{n}[1]}{norm}\right]\right)$
end
return $\boldsymbol{n1}$
end

Algoritmo 54. Cálculo $\boldsymbol{Wr}$ para MEV (**vemWr**)

Inputs: $\boldsymbol{verts}$ Vertices en plano 2D.

$area \leftarrow \textbf{polyarea}(\boldsymbol{verts})$
$\boldsymbol{v1} \leftarrow [\ldots \boldsymbol{verts}, \boldsymbol{verts}[0]]$
// Calcula las normales a los bordes del elemento
$\boldsymbol{n} \leftarrow \textbf{normals\_to\_edges}(\boldsymbol{verts})$
$\boldsymbol{n1} \leftarrow \boldsymbol{n}.\textbf{map}((\boldsymbol{val}) \rightarrow \ \boldsymbol{val}[0])$
$\boldsymbol{n2} \leftarrow \boldsymbol{n}.\textbf{map}((\boldsymbol{val}) \rightarrow \ \boldsymbol{val}[1])$
$\boldsymbol{n11} \leftarrow [\boldsymbol{n}[\boldsymbol{n}.\textbf{length} - 1][0], \ldots \boldsymbol{n1}.\textbf{slice}(0, -1)]$
$\boldsymbol{n22} \leftarrow [\boldsymbol{n}[\boldsymbol{n}.\textbf{length} - 1][1], \ldots \boldsymbol{n2}.\textbf{slice}(0, -1)]$
// Calcula las longitudes de los bordes
$\boldsymbol{len} \leftarrow \boldsymbol{verts}.\textbf{map}\big((v, i) \rightarrow \textbf{hypot}(\boldsymbol{v1}[i][0] \ - \ \boldsymbol{v1}[i+1][0], \boldsymbol{v1}[i][1] \ - \ \boldsymbol{v1}[i+1][1])\big)$
$\boldsymbol{len1} \leftarrow [\boldsymbol{len}[\boldsymbol{len}.\textbf{length} - 1], \ldots \boldsymbol{len}.\text{slice}(0, -1)]$
// Calcula los valores de $\boldsymbol{Wr}$ para cada vértice
$\boldsymbol{Wr} \leftarrow [\ ]$
for $i = 0$ **to** $i < verts.\textbf{length}$ **do**
  $q1a \leftarrow \dfrac{0.25 \cdot (\boldsymbol{len1}[i] \cdot \boldsymbol{n11}[i] + \ \boldsymbol{len}[i] \cdot \boldsymbol{n1}[i])}{area}$
  $q2a \leftarrow \dfrac{0.25 \cdot (\boldsymbol{len1}[i] \cdot \boldsymbol{n22}[i] + \ \boldsymbol{len}[i] \cdot \boldsymbol{n2}[i])}{area}$
  $\boldsymbol{Wr\_a} \leftarrow \left[\left[\dfrac{1}{verts.\textbf{length}}, 0, q2a\right], \left[0, \dfrac{1}{verts.\textbf{length}}, -q1a\right]\right]$
  $\boldsymbol{Wr}.\textbf{push}(\ldots \boldsymbol{Wr\_a})$
end
return $\boldsymbol{Wr}$
end

Algoritmo 55. Cálculo $\boldsymbol{Hc}$ para MEV (**vemHc**)

Inputs: $\boldsymbol{verts}$ Vertices en plano 2D.

// Obtiene las coordenadas $\boldsymbol{x}$ e $\boldsymbol{y}$ de los vértices
$\boldsymbol{verts1} \leftarrow \boldsymbol{verts}.\textbf{map}((\boldsymbol{val}) \rightarrow \boldsymbol{val}[0])$
$\boldsymbol{verts2} \leftarrow \boldsymbol{verts}.\textbf{map}((\boldsymbol{val}) \rightarrow \boldsymbol{val}[1])$
$n \leftarrow verts.\textbf{length}$
// Calcula el centroide del elemento
$\boldsymbol{x\_bar} \leftarrow [\boldsymbol{verts1}.\textbf{reduce}((acc, val)$
$\rightarrow \ acc + val, 0)/n \ , \boldsymbol{verts2}.\textbf{reduce}((acc, val) \rightarrow \ acc + val, 0)/n \ ]$
// Calcula la matriz $\boldsymbol{Hc}$ para cada vértice
$\boldsymbol{Hc} \leftarrow [\ ]$
for $i = 0$ **to** $i \ < \ n$ **do**
  $\boldsymbol{Hc\_a} \leftarrow [$
    $[\boldsymbol{verts1}[i] \ - \ \boldsymbol{x\_bar}[0], 0, \boldsymbol{verts2}[i] \ - \ \boldsymbol{x\_bar}[1]],$
    $[0, \boldsymbol{verts2}[i] \ - \ \boldsymbol{x\_bar}[1], \boldsymbol{verts1}[i] \ - \ \boldsymbol{x\_bar}[0]],$
  $]$

```
    Hc.push(...Hc_a)
  end
  return Hc
end
```

Algoritmo 56. Cálculo ***Hr*** para MEV (**vemHr**)

Inputs: ***verts*** Vertices en plano 2D.

```
// Obtiene las coordenadas x e y de los vértices
verts1 ← verts.map((val) → val[0])
verts2 ← verts.map((val) → val[1])
n ← verts.length
// Calcula el centroide del elemento
x_bar ← [verts1.reduce((acc, val) → acc + val, 0)/n , verts2.reduce((acc, val)
                → acc + val, 0)/n ]
// Calcula la matriz Hr para cada vértice
Hr ← [ ]
for i = 0 to i < n do
  Hr_a ← [[1, 0, verts2[i] − x_bar[1]], [0, 1, −(verts1[i] − x_bar[0])]]
  Hr.push(...Hr_a)
end
return Hr
end
```

Algoritmo 57. Cálculo de fuerzas corporales (**vem_body_force**)

Inputs: ***verts*** Vertices en plano 2D, ***body_force_fun*** Funciones de fuerzas corporales, ***matProps*** Propiedades del material

```
area ← polyarea(verts) // Calcula el área del elemento
n ← verts.length // Número de vértices del elemento
// Calcula las fuerzas de cuerpo en los nodos del elemento
bf ← body_force_function (verts, body_force_fun, matProps)
// Calcula las fuerzas de cuerpo medias en el elemento
mean_bfx ← 0
mean_bfy ← 0
for i = 0 to i < n do
  mean_bfx += bf[2 · i] / n
  mean_bfy += bf[2 · i + 1] / n
end
mean_bf ← [mean_bfx, mean_bfy]
// Calcula el vector de fuerzas de cuerpo local para el elemento MEV
fb ← new Array(2 · n).fill(0)
```

```
    for i = 0 to i < n do
        fb[2 · i] = area · (1/n) · mean_bf[0]
        fb[2 · i + 1] = area · (1/n) · mean_bf[1]
    end
    return fb
end
```

Algoritmo 58. Calcula las fuerzas de cuerpo (body_force_function)

Inputs: ***eval_coords*** Vertices en plano 2D, ***body_force_fun*** Funciones de fuerzas corporales, ***matProps*** Propiedades del material

```
num_nodes ← eval_coords.length
bf ← new Array(num_nodes · 2).fill(0)
// Calcula las fuerzas de cuerpo en cada punto de evaluación
for i = 0 to i < num_nodes do
    x ← eval_coords[i][0]
    y ← eval_coords[i][1]
    // Calcula las componentes de la fuerza de cuerpo b_x y b_y
    bf[2 · i] ← body_force_fun.b_x? body_force_fun.b_x(x, y, matProps): 0
    bf[2 · i + 1] ← body_force_fun.b_y? body_force_fun.b_y(x, y, matProps): 0
end
return bf
end
```

Algoritmo 59. Construye matriz dispersa (sparse)

Inputs: ***i*** Vector de índice de filas, ***j*** Vector de índice de columnas, ***v*** Vector de índice de valores

```
result ← {fila: [ ], columna: [ ], valor: [ ]}
posiciones ← { }
for index = 0 to index < i.length do
    if v[index] ≠ 0 then
        key ← i[index] + "-" + j[index]
        if posiciones.hasOwnProperty(key) then
            result.valor[posiciones[key]] += v[index]
        else
            posiciones[key] = result.fila.length
            result.fila.push(i[index])
            result.columna.push(j[index])
            result.valor.push(v[index])
        end
    end
```

```
    end
    index_order ← result.fila
        .map(fila, index → {
            fila,
            columna: result.columna[index],
            valor: result.valor[index],
        })
        .sort(a, b →  a.columna − b.columna or a.fila − b.fila)
    result.fila = index_order.map(el → el.fila)
    result.columna = index_order.map(el → el.columna)
    result.valor = index_order.map(el → el.valor)
    return result
end
```

Algoritmo 60. Ensamble de matriz de rigidez para MEF (**fem_assembly**)

Inputs: ***domainMesh*** Información de la malla, ***config*** configuración de la solución, ***matProps*** Propiedades del material, ***body_force_fun*** Funciones de fuerzas corporales

```
num_nodes ← domainMesh.coords.length<
numel ← domainMesh.connect.length
len_vector_K ← 0
len_vector_f ← 0
// Calcula la longitud total requerida para los vectores de ensamblaje disperso
for e = 0 to e  <  numel do
    num_eldofs ← 2 · domainMesh.connect[e].length
    len_vector_K +=  num_eldofs²
    len_vector_f += num_eldofs
end
// Inicializa los vectores para el ensamblaje de la matriz dispersa
vector_K ← new Array(len_vector_K).fill(0)
indrow_K ← new Array(len_vector_K).fill(0)
indcol_K ← new Array(len_vector_K).fill(0)
vector_f ← new Array(len_vector_f).fill(0)
indrow_f ← new Array(len_vector_f).fill(0)
pos_vector_K ← 0
pos_vector_f ← 0
// Itera sobre todos los elementos para ensamblar K_global y f_global
for e = 0 to e  <  numel do
    nodes ← domainMesh.connect[e]
    verts ← nodes.map((node) → domainMesh.coords[node])
    // Determina los índices globales de los grados de libertad del elemento actual
    ind ← [ ]
    foreach(node, index) of nodes do
        ind[2 · index] = 2 · node
```

```
    ind[2 · index + 1] = 2 · node + 1
  end
  num_eldofs ← 2 · nodes.length
  // Calcula la matriz de rigidez local y el vector de fuerzas local
  K_local ← fem_stiffness(verts, matProps, config)
  f_local ← fem_body_force(verts, config, body_force_fun, matProps)
  // Determina las posiciones en los vectores de ensamblaje
  ind_vector_K ← Array.from({ length: num_eldofs^2 }, (_, i) → i + pos_vector_K)
  ind_vector_f ← Array.from({ length: num_eldofs }, (_, i) → i + pos_vector_f)
  // Aplana y almacena los valores de la matriz de rigidez local
  for i = 0 to i < num_eldofs^2 do
    vector_K[ind_vector_K[i]] = K_local.flat()[i]
  end
  // Almacena los valores del vector de fuerzas local
  for i = 0 to i < num_eldofs do
    vector_f[ind_vector_f[i]] = f_local[i]
  end
  // Crea matrices de índices para el ensamblaje disperso
  indcol_K_ ← new Array(ind.length).fill().map(() → ind.slice())
  indrow_K ← numeric.transpose(indcol_K_)
  // Asigna los índices de fila y columna para K_global
  for i = 0 to i < num_eldofs^2 do
    indrow_K[ind_vector_K[i]] = indcol_K_.flat()[i]
  end
  for i = 0 to i < num_eldofs^2 do
    indcol_K[ind_vector_K[i]] = indrow_K_.flat()[i]
  end
  // Asigna los índices de fila para f_global
  foreach (value, i) of ind_vector_f do
    indrow_f[value] = ind[i % ind.length]
  end
  // Actualiza las posiciones en los vectores de ensamble
  pos_vector_K += num_eldofs^2
  pos_vector_f += num_eldofs
end
row_vector_f ← Array(len_vector_f).fill(0)
K_global ← sparse(indrow_K, indcol_K, vector_K, )
f_global ← sparse(indrow_f, row_vector_f, vector_f)
return { K_global, f_global }
end
```

Algoritmo 61. Calcula la matriz de rigidez local para MEF (fem_stiffness)

Inputs: ***verts*** Vértices en plano 2D, ***config*** configuración de la solución, ***matProps*** Propiedades del material

```
if config.method = "FEM2DT3" then
    // Calcula el área del elemento triangular
    area ← polyarea(verts)
    // Extrae las coordenadas x e y de los vértices
    x ← verts.map(function (v) { return v[0] })
    y ← verts.map(function (v) {return v[1] })
    // Construye la matriz de deformación B para elementos triangulares lineales
    B ← numeric.mul(1/(2 · area) ,[
        [y[1] − y[2], 0, y[2] − y[0], 0, y[0] − y[1], 0],
        [0, x[2] − x[1], 0, x[0] − x[2], 0, x[1] − x[0]],
        [
            x[2] − x[1],
            y[1] − y[2],
            x[0] − x[2],
            y[2] − y[0],
            x[1] − x[0],
            y[0] − y[1],
        ],
    ])
    // Ajusta la matriz constitutiva D según las definiciones de MEF
    D ← matProps.D
    D[2][2] = D[2][2] / 4
    // Calcula la matriz de rigidez local Kfem
    Kfem
    ← numeric.mul(area, numeric.dot(numeric.transpose(B), numeric.dot(D, B)) )
    return Kfem
else if config.method = "FEM2DQ4" then
    // Fuera de los alcances del desarrollo
end
end
```

Algoritmo 62. Calcula el vector de fuerzas local para MEF (fem_body_force)

Inputs: ***verts*** Vértices en plano 2D, ***config*** configuración de la solución, ***matProps*** Propiedades del material ***body_force_fun*** Funciones de fuerzas corporales

```
if config.method = "FEM2DT3" then
    // Calcula el área del elemento triangular
    area ← polyarea(verts)
    // Obtiene las fuerzas de cuerpo en los vértices
    bf ← body_force_function(verts, body_force_fun, matProps)
```

```
    // Extrae las componentes de las fuerzas de cuerpo en cada vértice
    bx1 ← bf[0]
    bx2 ← bf[2]
    bx3 ← bf[4]
    by1 ← bf[1]
    by2 ← bf[3]
    by3 ← bf[5]
    // Calcula el vector de fuerzas nodales equivalentes debido a las fuerzas de cuerpo
    fb ← numeric.mul(area/12, [
      2 · bx1 + bx2 + bx3,
      2 · by1 + by2 + by3,
      bx1 + 2 · bx2 + bx3,
      by1 + 2 · by2 + by3,
      bx1 + bx2 + 2 · bx3,
      by1 + by2 + 2 · by3,
    ])
    return fb
  else if config.method = "FEM2DQ4" then
    // Fuera de los alcances del desarrollo.
  end
end
```

Algoritmo 63. Cálculo CBs Neumann (compute_Neumann_BCs)

Inputs: ***domainMesh*** Información de la malla, ***config*** configuración de la solución, ***matProps*** Propiedades del material ***NeumannBoundaryNodes*** Nodos de condiciones de borde de Neumann seleccionados, ***NeumannBoundaryDofs*** Grados de libertad asociados a los nodos en condición de borde de Neumann, ***NeumannFunValues*** Funciones asociados a los nodos seleccionados en condición de borde de Neumann.

```
if config.method = "VEM2D" then
  return vem_compute_Neumann_BCs (
    domainMesh,
    NeumannBoundaryNodes,
    NeumannBoundaryDofs,
    NeumannFunValues,
    matProps
  )
else if (config.method = "FEM2DT3" or config.method = FEM2DQ4 then
  return fem_compute_Neumann_BCs (
    domainMesh,
    NeumannBoundaryNodes,
    NeumannBoundaryDofs,
    NeumannFunValues,
    matProps
```

```
    )
  end
end
```

Algoritmo 64. Cálculo CBs Neumann MEV (vem_compute_Neumann_BCs)

Inputs: ***domainMesh*** Información de la malla, ***NeumannBoundaryNodes*** Nodos de condiciones de borde de Neumann seleccionados, ***NeumannBoundaryDofs*** Grados de libertad asociados a los nodos en condición de borde de Neumann, ***Neumann_fun_value*** Funciones asociados a los nodos seleccionados en condición de borde de Neumann, ***matProps*** Propiedades del material.

```
coords ← domainMesh.coords
nodes ← sort_boundary_nodes(coords, NeumannBoundaryNodes)
x ← nodes.map(node → coords[node][0])
y ← nodes.map(node →  coords[node][1])
N_bar ← [[0.5, 0.0], [0.0, 0.5]]
num_nodes ← nodes.length
num_edges ← num_nodes − 1
Neumann_BCs ← {
  values: new Array(2 · num_nodes).fill(0),
  indexes: NeumannBoundaryDofs,
}
for edge = 0 to edge < num_edges do
  actual_node ← nodes[edge]
  first_node ← edge
  second_node ← edge + 1
  x1 ← x[first_node]
  y1 ← y[first_node]
  x2 ← x[second_node]
  y2 ← y[second_node]
  edge_length ← hypot(x1 − x2, y1 − y2)
  tx_first_node ← Neumann_fun_value[actual_node].fx(x1, y1, x, y, matProps)
  ty_first_node ← Neumann_fun_value[actual_node].fy((x1, y1, x, y, matProps)
  tx_second_node ← Neumann_fun_value[actual_node].fx(x2, y2, x, y, matProps)
  ty_second_node ← Neumann_fun_value[actual_node].fy(x2, y2, x, y, matProps)
  traction_first_node ← [tx_first_node, ty_first_node]
  traction_second_node ← [tx_second_node, ty_second_node]
  mean_traction ← traction_first_node.map((val, index)
                  → (val  +  traction_second_node[index])/2 )
  range1 ← [2 · first_node, 2 · first_node + 1]
  range2 ← [2 · second_node, 2 · second_node + 1]
  Nt ← numeric.transpose(N_bar)
  mult1 ← numeric.mul(edge_length, numeric.dot(Nt, mean_traction))
  foreach(i, index) of range1 do
    Neumann_BCs.values[i] += mult1[index]
```

```
    end
    mult2 ← numeric.mul(edge_length, numeric.dot(Nt, mean_traction))
    foreach(i, index) of range2 do
        Neumann_BCs.values[i] += mult2[index]
    end
end
return Neumann_BCs
end
```

Algoritmo 65. Cálculo CBs Neumann MEF (fem_compute_neumann_BCs)

Inputs: ***domainMesh*** Información de la malla, ***NeumannBoundaryNodes*** Nodos de condiciones de borde de Neumann seleccionados, ***NeumannBoundaryDofs*** Grados de libertad asociados a los nodos en condición de borde de Neumann, ***Neumann_fun_value*** Funciones asociados a los nodos seleccionados en condición de borde de Neumann, ***matProps*** Propiedades del material

```
coords ← domainMesh.coords
nodes ← sort_boundary_nodes(coords, NeumannBoundaryNodes)
x ← nodes.map(node → coords[node][0])
y ← nodes.map(node → coords[node][1])
N ← [[0.5, 0.0, 0.5, 0.0], [0.0, 0.5, 0.0, 0.5]]
num_nodes ← nodes.length
num_edges ← num_nodes − 1
Neumann_BCs ← {
    values: new Array(2 · num_nodes).fill(0),
    indexes: NeumannBoundaryDofs,
}
for edge = 0 to edge < num_edges do
    actual_node ← nodes[edge]
    first_node ← edge
    second_node ← edge + 1
    x1 ← x[first_node]
    y1 ← y[first_node]
    x2 ← x[second_node]
    y2 ← y[second_node]
    edge_length ← hypot(x1 − x2, y1 − y2)
    xg ← 0.5 · x1 + 0.5 · x2 //Mapeo isoparamétrico en xi = 0: xg = N1(0) · x1 + N2(0) · x2
    yg ← 0.5 · y1 + 0.5 · y2 //Mapeo isoparamétrico en xi = 0: yg = N1(0) · y1 + N2(0) · y2
    tx ← Neumann_fun_value[actual_node].fx(xg, yg, x, y, matProps)
    ty ← Neumann_fun_value[actual_node].fy(xg, yg, x, y, matProps)
    traction ← [tx, ty]
    range1 ← [2 · first_node, 2 · first_node + 1]
    range2 ← [2 · second_node, 2 · second_node + 1]
    range ← range1.concat(range2)
    Nt ← numeric.transpose(N)
```

```
    mult ← numeric.mul(edge_length, numeric.dot(Nt, traction))
    foreach (i, index) of range do
        Neumann_BCs.values[i] = Neumann_BCs.values[i] + mult[index]
    end
  end
  return Neumann_BCs
end
```

Algoritmo 66. Sumar vector con vector disperso (**sumar_vector_vectord**)

```
Inputs: f_global Vector de fuerzas nodales, neu vector disperso
  // Itera a través de cada índice en el vector disperso neu
  for i = 0 to i < neu.indexes.length do
    index ← neu.indexes[i]
    value ← neu.values[i]
    // Encuentra el índice en f_global que coincida con el índice actual de neu
    fIndex ← f_global.fila.findIndex(fila → fila = index)
    // Si se encuentra una coincidencia, suma el valor de neu a su valor en f_global
    if fIndex ≠ −1 then
      newValue ← f_global.valor[fIndex] + value
      // Solo actualiza si el nuevo valor no es cero
      if newValue ≠ 0 then
        f_global · valor[fIndex] = newValue
      else
        // Si el nuevo valor es cero, elimina el índice y el valor correspondiente
        f_global.fila.splice(fIndex, 1)
        f_global.valor.splice(fIndex, 1)
        f_global.columna.splice(fIndex, 1)
      end
    else
      // Si no se encuentra el índice y el valor no es cero, se agrega el índice y el valor a f_global
      if value ≠ 0 then
        f_global.fila.push(index)
        f_global.valor.push(value)
        f_global.columna.push(0);
      end
    end
  end
  return f_global
end
```

Algoritmo 67. Cálculo CBs Dirichlet (compute_dirichlet_BCs)

Inputs: ***domainMesh*** Información de la malla, ***Dirichet_boundary_nodes*** Nodos seleccionados con la CB de Dirichlet, ***Dirichet_boundary_dofs*** Grados de libertad de los nodos de Dirichlet, ***Dirich_fun*** Funciones asociadas a los nodos de Dirichlet, ***matProps*** Propiedades del material

```
coords ← domainMesh.coords
nodes ← Dirichet_boundary_nodes
Dirichlet_DOFs ← { values: [ ], indexes: [ ] }
x_arr ← nodes.map(node → coords[node][0])
y_arr ← nodes.map((ode → coords[node][1])
for i = 0 to i < nodes.length do
    actual ← nodes[i]
    x ← coords[actual][0]
    y ← coords[actual][1]
    uxVal ← Dirich_fun[actual].ux? Dirich_fun[actual].ux(x, y, x_arr, y_arr, matProps)
                : null
    uyVal ← Dirich_fun[actual].uy?  Dirih_fun[actual].uy(x, y, x_arr, y_arr, matProps)
                :  null
    if uxVal ≠ null then
        Dirichlet_DOFs.values.push(uxVal)
        Dirichlet_DOFs.indexes.push(Dirichet_boundary_dofs[2 · i])
    end
    if uyValues ≠ null then
        Dirichlet_DOFs.values.push(uyVal)
        Dirichlet_DOFs.indexes.push(Dirichet_boundary_dofs[2 · i + 1])
    end
end
return Dirichlet_DOFs
end
```

Algoritmo 68. Multiplicar vector con matriz dispersa (mult_sparse_vector)

Inputs: ***matrix*** matriz dispersa, ***vector*** vector disperso

```
result ← new Array(vector.length).fill(0)
for i = 0 to i < matrix.valor.length do
    row ← matrix.fila[i]
    col ← matrix.columna[i]
    value ← matrix.valor[i]
    if col < vector.length then
        result[row] += value · vector[col]
    end
end
return result
end
```

Algoritmo 69. Resta vector a matriz dispersa (**subtract_sparse_vector**)

```
Inputs: matrix matriz dispersa, vector vector disperso
    vec ← to_dense(matrix)
    result ← [ ]
    i ← 0
    foreach val of vec do
        result.push(val − vector[i])
        i++
    end
    while i < vector.length do
        result.push(−vector[i]) // Elementos adicionales como si se restaran con 0
        i++
    end
    return result
end
```

Algoritmo 70. De matriz dispersa a matriz densa (**to_dense**)

```
Inputs: sparseMatrix Matriz dispersa.
    if sparseMatrix.fila and sparseMatrix.columna and sparseMatrix.valor then
        numRows = sparseMatrix.fila.reduce((max, val) → max(max, val), −Infinity) + 1
        numCols = sparseMatrix.columna.reduce((max, val)
                    → max(max, val), −Infinity) + 1
    else
        // Extraer las dimensiones
        keys ← Object.keys(sparseMatrix)
        dimensions ← keys.map(key → key.split("_").map(Number))
        numRows = dimensions.reduce((max, dim) → max(max, dim[0]), −Infinity) + 1
        numCols = dimensions.reduce((max, dim) → max(max, dim[1]), −Infinity) + 1
    end
    // Crear una matriz
    denseMatrix ← new Array(numRows).fill(null).map(( )
                    → new Array(numCols).fill(0))
    if sparseMatrix.fila and sparseMatrix.columna and sparseMatrix.valor then
        foreach (fila, i) of sparseMatrix.fila do
            columna ← sparseMatrix.columna[i]
            valor ← sparseMatrix.valor[i]
            denseMatrix[fila][columna] = valor
        end
    else
        // Formato de objeto para L y U
        foreach key of Object.keys(sparseMatrix) do
            [fila, columna] = key.split("_").map(Number)
            valor ← sparseMatrix[key]
```

```
        denseMatrix[fila][columna] = valor
    end
end
return denseMatrix
end
```

Algoritmo 71. Filtra matriz dispersa (filter_sparse)

```
Inputs: A Matriz dispersa, free_dofs nodos libres
filteredMatrix ← { fila: [ ], columna: [ ], valor: [ ]}
// Filtrar free_dofs tanto en fila como en columna
for i = 0 to i < A.valor.length do
    filaIndex ← free_dofs.indexOf(A.fila[i])
    colIndex ← free_dofs.indexOf(A.columna[i])
    if filaIndex ≠= −1 and colIndex ≠= −1 then
        filteredMatrix.fila.push(filaIndex)
        filteredMatrix.columna.push(colIndex)
        filteredMatrix.valor.push(A.valor[i])
    end
end
return filteredMatrix
end
```

Algoritmo 72. Filtrar grados de libertad libres (**filter_vector**)

```
Inputs: f_global Vector de fuerzas nodales globales, free_dofs grados de
libertad libres.
return free_dofs.map((dof) → f_global[dof])
end
```

Algoritmo 73. Opciones para invertir matriz (solve_options)

```
Inputs: solver Tipo de algoritmo de inversión de matriz, A matriz de rigidez, b
vector de fuerzas nodales.
solution_aux ← null
if solver = "LU_dense" then
    A_dense ← to_dense(A)
    solution_aux ← lu(A_dense, b)
else if solver = "cholesky_dense" then
    A_dense ← to_dense(A)
    solution_aux ← cholesky(A_dense, b)
else if solver = "CGS" then
    A_dense ← to_dense(A)
    solution_aux ← cgs(A_dense, b)
else if solver = "numeric_js" then
```

```
    A_dense ← to_dense(A)
    solution_aux ← numeric.solve(A_dense, b)
  end
  return solution_aux
end
```

Algoritmo 74. Rellenar con ceros (rellenar_soluciones)

Inputs: ***solution*** Solución del problema, ***freeDof*** Grados de libertad libres, ***total_dofs*** Grados de libertad de todos los nodos

```
todasSoluciones ← new Array(total_dofs).fill(0)
for i = 0 to i < freeDof.length do
  todasSoluciones[freeDof[i]] = soluciones[i]
end
return todasSoluciones
end
```

Algoritmo 75. Esfuerzo y deformación en MEF (fem_stress_and_strain)

Inputs: ***solution*** solución del problema, ***domainMesh*** Información de la malla, ***matProps*** Propiedades del material

$$mu \leftarrow \frac{matProps.Ey}{2 \cdot (1 + matProps.nu)}$$

$$kappa \leftarrow \frac{matProps.Ey}{3 \cdot (1 - 2 \cdot matProps.nu)}$$

$$lambda \leftarrow kappa - \frac{2 \cdot mu}{3}$$

```
// Datos de la malla
coords ← domainMesh.coords
connect ← domainMesh.connect
num_elem ← connect.length
stress ← { s11: [ ], s12: [ ], s22: [ ], s33: [ ], s1: [ ], s2: [ ], s3: [ ], vm: [ ], p: [ ]}
strain ← { e11: [ ] , e12: [ ], e22: [ ], e33: [ ], e1: [ ], e2: [ ], e3: [ ]}
// Inicio de recursividad
for i = 0 to i < num_elem do
  nodes ← connect[i]
  elem_coord ← nodes.map(node → coords[node])
  dofs ← nodes.flatMap(node → [2 · node, 2 · node + 1])
  uh_elem_column ← dofs.map(dof → solution[dof])
  area ← polyarea(elem_coord)
  // Matriz de deformación
  x ← elem_coord.map(coord → coord[0])
  y ← elem_coord.map(coord → coord[1])
  B ← calculate_B_matrix(x, y, area)
  // Deformación
```

```
str_h ← numeric.dot(B, uh_elem_column)
if matProps.plane_state = 'plane_stress' then
    eten_h ← [
      [str_h[0], str_h[2] / 2, 0],
      [str_h[2] / 2, str_h[1], 0],
      [0, 0, −matProps.nu · (str_h[0] + str_h[1])] //Definición del vector de deformaciones
      (e11, e22, e33, 2·e12, 2·e23, 2·e13) otra literatura donde se usa (e11, e22, e33, e12, e23,
      e13)
    ]
else if matProps.plane_state = 'plane_strain' then
    eten_h ← [
      [str_h[0], str_h[2] / 2, 0],
      [str_h[2] / 2, str_h[1], 0],
      [0, 0, 0]
    ]
end
eigenvalues ← math.eigs(eten_h).values
pstrain_h ← eigenvalues.sort((a, b) → b − a) // Orden descendente
strain.e11.push(eten_h[0][0])
strain.e12.push(eten_h[0][1])
strain.e22.push(eten_h[1][1])
strain.e33.push(eten_h[2][2])
strain.e1.push(pstrain_h[0])
strain.e2.push(pstrain_h[1])
strain.e3.push(pstrain_h[2])
// Esfuerzo MEF
D ← matProps.D
D[2][2] = D[2][2]/4 // Notas VEMLAV: "To avoid troubles, come back to the standard FEM
definition of D"
svec_h ← numeric.dot(D, [eten_h[0][0], eten_h[y][y], 2 · eten_h[0][1]])
if matProps.plane_state = 'plane_stress' then
    sten_h ← [
      [svec_h[0], svec_h[2], 0], [0][1]
      [svec_h[2], svec_h[1], 0],
      [0, 0, 0]
    ]
else if matProps.plane_state = 'plane_strain' then
    sten_h ← [
      [svec_h[0], svec_h[2], 0],
      [svec_h[2], svec_h[1], 0],
      [0, 0, matProps.nu · (svec_h[0] + svec_h[1])]
    ]
end
eigenvalues_2 ← math.eigs(sten_h).values
pstress_h ← eigenvalues_2.sort((a, b) → b − a)
```

$sxx\_h \leftarrow \boldsymbol{sten\_h}[0][0]$
$syy\_h \leftarrow \boldsymbol{sten\_h}[1][1]$
$szz\_h \leftarrow \boldsymbol{sten\_h}[2][2]$
$sxy\_h \leftarrow \boldsymbol{sten\_h}[0][1]$
$sxz\_h \leftarrow 0.0$
$syz\_h \leftarrow 0.0$
$VM\_h \leftarrow \frac{\textbf{sqrt}((sxx_h - syy_h)^2 + (syy_h - szz_h)^2 + (szz_h - sxx_h)^2 + 6 \cdot (sxy_h{}^2 + syz_h{}^2 + sxz\_h^2))}{\textbf{sqrt}(2)}$
$p\_h \leftarrow \frac{1}{3}(sxx\_h + syy\_h + szz\_h)$ // $-lambda \cdot (\boldsymbol{eten\_h}[0][0] + \boldsymbol{eten\_h}[1][1])$ traza del tensor de esfuerzo.
$\boldsymbol{stress}.\boldsymbol{s11}.\textbf{push}(\boldsymbol{sten\_h}[0][0])$
$\boldsymbol{stress}.\boldsymbol{s12}.\textbf{push}(\boldsymbol{sten\_h}[0][1])$
$\boldsymbol{stress}.\boldsymbol{s22}.\textbf{push}(\boldsymbol{sten\_h}[1][1])$
$\boldsymbol{stress}.\boldsymbol{s33}.\textbf{push}(\boldsymbol{sten\_h}[2][2])$
$\boldsymbol{stress}.\boldsymbol{s1}.\textbf{push}(\boldsymbol{pstress\_h}[0])$
$\boldsymbol{stress}.\boldsymbol{s2}.\textbf{push}(\boldsymbol{pstress\_h}[1])$
$\boldsymbol{stress}.\boldsymbol{s3}.\textbf{push}(\boldsymbol{pstress\_h}[2])$
$\boldsymbol{stress}.\boldsymbol{vm}.\textbf{push}(VM\_h)$
$\boldsymbol{stress}.\boldsymbol{p}.\textbf{push}(p\_h)$
end
return { $\boldsymbol{stress}, \boldsymbol{strain}$}
end

Algoritmo 76. Cálculo de matriz $\boldsymbol{B}$ (calculate_B_matrix)

Inputs: $\boldsymbol{x}$ Vector eje x, $\boldsymbol{y}$ Vector eje y, $area$ área del triángulo

$$
\boldsymbol{B} \leftarrow \left[
\begin{array}{l}
\left[
\begin{array}{l}
\frac{\boldsymbol{y}[1] - \boldsymbol{y}[2]}{2 \cdot area}, 0, \\
\frac{\boldsymbol{y}[2] - \boldsymbol{y}[0]}{2 \cdot area}, 0, \\
\frac{\boldsymbol{y}[0] - \boldsymbol{y}[1]}{2 \cdot area}, 0
\end{array}
\right], \\
\left[
\begin{array}{l}
0, \frac{\boldsymbol{x}[2] - \boldsymbol{x}[1]}{2 \cdot area}, \\
0, \frac{\boldsymbol{x}[0] - \boldsymbol{x}[2]}{2 \cdot area}, \\
0, \frac{\boldsymbol{x}[1] - \boldsymbol{x}[0]}{2 \cdot area}
\end{array}
\right], \\
\left[
\begin{array}{l}
\frac{\boldsymbol{x}[2] - \boldsymbol{x}[1]}{2 \cdot area}, \frac{\boldsymbol{y}[1] - \boldsymbol{y}[2]}{2 \cdot area}, \\
\frac{\boldsymbol{x}[0] - \boldsymbol{x}[2]}{2 \cdot area}, \frac{\boldsymbol{y}[2] - \boldsymbol{y}[0]}{2 \cdot area},
\end{array}
\right.
\end{array}
\right.
$$

$$\frac{\boldsymbol{x}[1]-\boldsymbol{x}[0]}{2\cdot area},\frac{\boldsymbol{y}[0]-\boldsymbol{y}[1]}{2\cdot area}$$

]

]

return $\boldsymbol{B}$

end

Algoritmo 77. Esfuerzo y deformación en MEV (vem_stress_and_strain)

Inputs: $\boldsymbol{solution}$ solución del problema, $\boldsymbol{domainMesh}$ Información de la malla, $\boldsymbol{matProps}$ Propiedades del material, $\boldsymbol{config}$ configuración de la solución

$mu \leftarrow \frac{\boldsymbol{matProps}.Ey}{2\cdot(1+\boldsymbol{matProps}.nu)}$

$kappa \leftarrow \frac{\boldsymbol{matProps}.Ey}{3\cdot(1-2\cdot\boldsymbol{matProps}.nu)}$

$lambda \leftarrow kappa - \frac{2\cdot mu}{3}$

// Matriz de deformación

$\boldsymbol{coords} \leftarrow \boldsymbol{domainMesh}.\boldsymbol{coords}$

$\boldsymbol{connect} \leftarrow \boldsymbol{domainMesh}.\boldsymbol{connect}$

$num\_elem \leftarrow \boldsymbol{connect}.\mathbf{length}$

$\boldsymbol{stress} \leftarrow \{\ \boldsymbol{s11}$: [ ], $\boldsymbol{s12}$: [ ], $\boldsymbol{s22}$: [ ], $\boldsymbol{s33}$: [ ], $\boldsymbol{s1}$: [ ], $\boldsymbol{s2}$: [ ], $\boldsymbol{s3}$: [ ], $\boldsymbol{vm}$: [ ], $\boldsymbol{p}$: [ ]$\}$

$\boldsymbol{strain} \leftarrow \{\ \boldsymbol{e11}$: [ ] , $\boldsymbol{e12}$: [ ], $\boldsymbol{e22}$: [ ], $\boldsymbol{e33}$: [ ], $\boldsymbol{e1}$: [ ], $\boldsymbol{e2}$: [ ], $\boldsymbol{e3}$: [ ]$\}$

// Recursión sobre índices

for $i = 0$ **to** $i < num\_elem$ **do**

$\boldsymbol{nodes} \leftarrow \boldsymbol{connect}[i]$

$\boldsymbol{elem\_coord} \leftarrow \boldsymbol{nodes}.\mathbf{map}(node \rightarrow [\boldsymbol{coords}[node][0], \boldsymbol{coords}[node][1]])$

$\boldsymbol{dofs} \leftarrow \boldsymbol{nodes}.\mathbf{flatMap}(node \rightarrow [2\cdot node, 2\cdot node+1])$

// Solución nodal MEV

$\boldsymbol{uh\_elem\_column} \leftarrow \boldsymbol{dofs}.\mathbf{map}(dof \rightarrow \boldsymbol{solution}[dof])$

// Matriz MEV

$\boldsymbol{Wc} \leftarrow \mathbf{vem\_Wc}(\boldsymbol{elem\_coord})$

// Deformación MEV

$\boldsymbol{pic\_grad\_uh} \leftarrow \mathbf{math}.\mathbf{multiply}(\mathbf{math}.\mathbf{transpose}(\boldsymbol{Wc}), \boldsymbol{uh\_elem\_column})$

if $\boldsymbol{matProps}.plane\_state = 'plane\_stress'$ **then**

$\boldsymbol{eten\_h} \leftarrow$ [

$[\boldsymbol{pic\_grad\_uh}[0], \boldsymbol{pic\_grad\_uh}[2], 0]$,

$[\boldsymbol{pic\_grad\_uh}[2], \boldsymbol{pic\_grad\_uh}[1], 0]$,

$[0, 0, -\boldsymbol{matProps}.nu\cdot(\boldsymbol{pic\_grad\_uh}[0] + \boldsymbol{pic\_grad\_uh}[1])]$

]

else if $\boldsymbol{matProps}.plane\_state = 'plane\_strain'$ **then**

$\boldsymbol{eten\_h} \leftarrow$ [

$[\boldsymbol{pic\_grad\_uh}[0], \boldsymbol{pic\_grad\_uh}[2], 0]$,

$[\boldsymbol{pic\_grad\_uh}[2], \boldsymbol{pic\_grad\_uh}[1], 0]$,

$[0, 0, 0]$

]

```
end
pstrain_h ← math.eigs(eten_h).values.sort((a, b) → b − a)
strain.e11.push(eten_h[0][0])
strain.e12.push(eten_h[0][1])
strain.e22.push(eten_h[1][1])
strain.e33.push(eten_h[2][2])
strain.e1.push(pstrain_h[0])
strain.e2.push(pstrain_h[1])
strain.e3.push(pstrain_h[2])
// Esfuerzo MEV
D ← matProps.D
D[2][2] = D[2][2] / 4
svec_h ← math.multiply(D, [eten_h[0][0], eten_h[1][1], 2 · eten_h[0][1]])
if matProps.plane_state = 'plane_stress' then
  sten_h ← [
    [svec_h[0], svec_h[2], 0],
    [svec_h[2], svec_h[1], 0],
    [0, 0, 0]
  ]
else if matProps.plane_state = 'plane_strain' then
  sten_h ← [
    [svec_h[0], svec_h[2], 0],
    [svec_h[2], svec_h[1], 0],
    [0, 0, matProps.nu · (svec_h[0] + svec_h[1])]
  ]
end
pstress_h ← math.eigs(sten_h).values.sort((a, b) → b − a)
sxx_h ← sten_h[0][0]
syy_h ← sten_h[1][1]
szz_h ← sten_h[2][2]
sxy_h ← sten_h[0][1]
sxz_h ← 0.0
syz_h ← 0.0
VM_h ← sqrt((sxx_h − syy_h)^2 + (syy_h − szz_h)^2 + (szz_h − sxx_h)^2 + 6 · (sxy_h^2 + syz_h^2 + sxz_h^2)) / sqrt(2)
p_h ← 1/3 · (sxx_h + syy_h + szz_h) //−lambda · (eten_h[0][0] + eten_h[1][1])
stress.s11.push(sten_h[0][0])
stress.s12.push(sten_h[0][1])
stress.s22.push(sten_h[1][1])
stress.s33.push(sten_h[2][2])
stress.s1.push(pstress_h[0])
stress.s2.push(pstress_h[1])
stress.s3.push(pstress_h[2])
stress.vm.push(VM_h)
stress.p.push(p_h)
```

```
    end
    return { stress, strain }
end
```

Algoritmo 78. Ordenar nodos de frontera (sort_boundary_nodes)

Inputs: ***coords*** Coordenadas de la malla en 2D, ***nodes*** Nodos de condiciones de borde de Neumann seleccionados

```
    // Extraer coordenadas de los nodos
    points ← nodes.map(node → {
        node,
        x: coords[node][0],
        y: coords[node][1]
    })
    // Encontrar el nodo inicial
    start ← points.reduce((min, p) →
        p.x < min.x or (p.x = min.x and p.y < min.y) ? p : min
                                                    )
    // Encontrar nodo más cercano
    ordered ← [start.node]
    remaining ← points.filter(p → p.node ≠ start.node)
    while (remaining.length > 0) {
        lastPoint ← points.find(p => p.node = ordered[ordered.length − 1])
        // Buscar nodo más cercano del último
        next ← remaining.reduce((min, p) → {
            dist ← hypot(p.x − lastPoint.x, p.y − lastPoint.y)
            return dist < min.dist ? { point: p, dist: dist } : min
        }, { point: null, dist: Infinity }).point
        ordered.push(next.node)
        remaining ← remaining.filter(p → p.node ≠ next.node)
    end
    return ordered
end
```