CENTRO REGIONAL UNIVERSITARIO CÓRDOBA IUA

Diseño e implementación de un framework para el desarrollo de aplicaciones basadas en el método de los elementos finitos

TRABAJO FINAL PARA OBTENER EL TÍTULO DE GRADO EN INGENIERÍA AERONÁUTICA

AUTORES

Airaudo, Facundo Nicolás
Zúñiga Pérez, Marco Antonio

TUTOR

Ing. Germán Weht

Córdoba, 26 de enero de 2021

El presente trabajo lo dedicamos principalmente a Dios, por inspirarnos y darnos la fuerza necesaria para continuar en este proceso de obtener uno de los anhelos más deseados.

A nuestra familia, por su amor, trabajo y sacrificio en todos estos años, gracias a ustedes hemos logrado llegar hasta aquí y convertirnos en lo que somos.

A nuestros compañeros y amigos, por estar siempre presentes acompañándonos y por el apoyo moral que nos brindaron a lo largo de esta etapa de nuestras vidas.

A todas las personas que nos han apoyado y han hecho que el trabajo se realice con éxito, en especial a aquellos que nos abrieron las puertas y compartieron sus conocimientos.

Marco y Facundo

Personalmente también quiero dedicar este trabajo a una persona muy especial en mi vida, mi madre, Marietta. Quien falleció mientras desarrollábamos este trabajo y aunque no se encontraba físicamente conmigo, sé que en todo momento me acompañó para finalizarlo. Te amo y hasta luego, porque sé que algún día nos volveremos a encontrar.

Marco

Agradecimientos

Deseamos aprovechar estas líneas para agradecer el apoyo de todas las personas que han colaborado de algún modo para que este trabajo llegue a su esperado término.

A nuestro tutor, Ing. Germán Weht, por su tiempo en la revisión de este trabajo, los aportes que ayudaron a mejorarlo y a su completa disposición durante el desarrollo del mismo.

Resumen

La simulaciones numéricas son utilizadas ampliamente por ingenieros y científicos para analizar el comportamiento de una gran variedad de fenómenos físicos gobernados por ecuaciones diferenciales. Los avances en este campo han permitido conseguir un alto grado de confiabilidad, complementando e incluso substituyendo costosos modelos experimentales. Estas simulaciones se realizan a través de programas formados por un conjunto de algoritmos complejos en los que la prioridad de los desarrolladores se suele colocar en la funcionalidad y eficiencia, dejando de lado mejoras en portabilidad, extensibilidad, mantenibilidad y claridad del software. Estas últimas características cobran elevada importancia cuando se considera mejorar los requerimientos básicos de mantenimiento en programas existentes, la necesidad de implementar nuevos algoritmos a ellos o el desarrollo de nuevos códigos multidisciplinarios.

En las últimas décadas la Programación Orientada a Objetos (OOP según sus siglas en inglés) ha tomado una posición protagónica en el desarrollo de software. Este paradigma presenta una solución para los problemas de extensibilidad y mantenibilidad que los programas científicos suelen traer. La implementación orientada a objetos de un framework permite optimizar la reusabilidad y el mantenimiento al generar un conjunto de bloques de software prefabricado que los programadores pueden usar, extender o adecuar para aplicaciones específicas.

En este proyecto se describe el diseño y la implementación de un framework, mediante OOP, para el desarrollo de aplicaciones basadas en el método de los elementos finitos. Para su implementación se hace uso de Fortran como único lenguaje de programación, explotando las herramientas de programación moderna de las últimas versiones. Se describen las principales abstracciones y clases, además de las distintas estructuras generadas para el manejo de matrices, resolución de sistemas de ecuaciones e integración numérica y temporal. Por último se muestra la implementación concreta de una serie de aplicaciones a partir del framework con el fin de evaluar las características de generalidad y reusabilidad. Para cada aplicación se muestran también los distintos casos de prueba usados para validar su funcionamiento.

Palabras clave: Método de los elementos finitos (MEF), Programación orientada a objetos (POO), Framework, Computación científica, Fortran.

Abstract

Numerical simulations are widely used by engineers and scientists to analyze the behavior of a variety of physical phenomena governed by differential equations. Advancements in this field have allowed programs to achieve a high degree of reliability, complementing and sometimes even substituting expensive experimental models. These simulations are realized by developing code with a set of complex algorithms in which the priority is usually placed in functionality and efficiency, leaving behind improvements in portability, extensibility, maintainability and clarity in the software. These last characteristics gain extra importance when one intents to elevate the basic requirements of maintainability on some existing program; or when it is necessary to implement new algorithms to an old program; or even to develop new code for multi-disciplinary problems.

In the last decades Object Oriented Programming (OOP) has gained a lot of relevance in software development. This paradigm presents a solution to the extensibility and maintainability problems that scientific programs usually have. An Object Oriented Framework implementation allows us to optimize re-usability and maintenance by creating a set of premade software blocks that developers can choose to use, extend or adapt to the specific applications.

In this project we will describe the design and implementation of an Object Oriented Framework to develop application based on the Finite Element Method. For its development, we make use of Fortran as the only programming language, exploiting all the modern tools currently available with the latest versions. The main abstractions and classes are described, as well as the different structures generated for the handling of matrices, solution of systems of equations and numerical and temporal integration. In the last part we present the concrete implementation of a series of applications made from the Framework, and we evaluate their characteristics in terms of generality and re-usability. For each application we show some test cases in order to evaluate its functionality.

Key words: Finite element method (FEM), Object oriented programming (OOP), Framework, Scientific computing, Fortran.

Índice

Índice de figuras

Índice de tablas

Parte I
Definición del trabajo, conceptos y herramientas

1. Introducción

Las técnicas de simulación son usadas por ingenieros y científicos para predecir el comportamiento de una situación física bajo condiciones de contorno medidas o asumidas [1]. Las simulaciones por computadora han adquirido una enorme importancia en las últimas décadas. Esto se debe a múltiples razones, principalmente al hecho de que éstas técnicas presentan una alternativa viable al uso de experimentos, los cuales tienden a ser órdenes de magnitud más costosos.

Cuando se las usa de manera correcta, la confiabilidad de las simulaciones es muy alta. Esto, en conjunto con el costo reducido que presentan hacen que el uso de éstas técnicas sea común en la industria automovilística, aeroespacial, de construcción, entre otras.

1.1. Motivación

Las últimas décadas han presentado grandes avances en paradigmas y estándares de programación. Como consecuencia, la **Programación Orientada a Objetos** (**OOP** según sus siglas en inglés) ha tomado una posición protagónica en el desarrollo de software tanto en la industria como en áreas académicas. La programación científica, comúnmente realizada por científicos o ingenieros, está caracterizada por ser un área que no se adapta rápidamente a nuevos estándares de desarrollo de software. Esto se debe a justificadas razones, la prioridad de los ingenieros o científicos es obtener el resultado buscado de la manera más eficiente, dejando de lado mejoras en portabilidad, extensibilidad, mantenibilidad y claridad de software, las cuales son ventajas que traen las nuevas prácticas.

Es debido a esta diferencia en filosofía que los ingenieros tienden a desarrollar programas que, si bien cumplen su propósito, resultan extremadamente difíciles de revisar o plantear una extensión o adaptación para nuevos propósitos. Este problema se vuelve interesante cuando se considera el incremento en importancia en el desarrollo de códigos multidisciplinarios y en la necesidad de implementar nuevos algoritmos a un código existente.

El hecho de que los ingenieros hagan uso de códigos con más de 30 años de existencia es evidencia de que estos programas funcionan correctamente con una gran robustez. Esto lleva a pensar que para extender uno de estos programas a nuevas utilidades simplemente bastaría con agregar las rutinas necesarias sin modificar el código existente. Lamentablemente esta idea no siempre es viable ya que en un gran número de casos implicaría realizar una conversión en el formato de la información y muchas veces se generaría una duplicación de datos. Para casos como esos es necesario modificar directamente el código fuente de estos programas.

Es evidente que un desarrollo orientado a objetos con una cierta generalidad presentaría una solución al problema planteado anteriormente. Múltiples proyectos, como FEMPAR [2] y Kratos [3], han encontrado éxito con desarrollos de este tipo. No obstante, **es importante estudiar y tener presente que se estará sacrificando en términos de eficiencia y robustez para alcanzar los objetivos**.

1.2. Problema

Mucho esfuerzo se ha dedicado en las últimas décadas en tratar de combinar distintos análisis con métodos de optimización, mallas adaptativas, una interfaz de usuario única y la posibilidad de extender las soluciones implementadas a nuevos problemas, no solo aplicado al **Método de los Elementos Finitos** (**MEF** o **FEM** según sus siglas en inglés), sino también a otros métodos ampliamente utilizados.

El enfoque de juntar programas ya existentes en un nuevo programa maestro implementando algoritmos de interacción ha funcionado en el pasado, pero suele presentar problemas en eficiencia de

procesamiento y de memoria. Por otro lado este método no puede ser aplicado para problemas de acople monolítico o fuerte.

Nuevas implementaciones hacen uso de conceptos de ingeniería de software modernos para hacer sus programas más extensibles [4]. Usualmente se consigue una extensibilidad para la implementación de nuevos algoritmos pero no se considera la posibilidad de extender el programa a un nuevo dominio físico.

Usar la filosofía de programación orientado a objetos ayuda a mejorar la reutilizabilidad de códigos. Esto es considerado un punto clave para facilitar la implementación de módulos para la resolución de nuevos tipos de problemas. No obstante, muchos conceptos específicos de cada dominio restringen la utilización para otros módulos.

Algunos problemas comunes al trabajar con problemas multi-disciplina son:

- Grados de libertad por nodo predefinidos
- Lista de variables globales para todas las entidades
- Interfaces dependientes del dominio
- Input/Output (IO) restringido para la lectura de nuevos datos y la impresión de nuevos resultados
- Definición de algoritmos -dentro del código

Estas deficiencias requieren amplias modificaciones al código para extenderlo a nuevos campos.

Muchos programas tienen un número predefinido de grados de libertad por nodo. Tener este valor fijo permite a los desarrolladores optimizar los códigos y simplificar las implementaciones, aunque limite la capacidad de extender los programas a otras áreas con un número distinto de grados de libertad.

Otro problema surge cuando las estructuras de datos de un programa están diseñadas para contener la misma información para todas las entidades. Por ejemplo, querer agregar una variable nodal a la estructura de datos implicaría agregar esa variable a todos los nodos del problema.

Adicionalmente, en programas de propósito único, es común tener interfaces dependientes del dominio para aumentar la claridad del código. Por ejemplo, proponer que las propiedades de un elemento contengan una variable llamada `conductivity` sería muy útil para proveer claridad en el problema de transferencia de calor, pero hace la extensión a otros campos incompatible.

La entrada y salida de datos (IO) es otro problema a la hora de extender un programa a nuevos campos. Cada física contiene una serie diferente de datos de entrada y resultados. Usualmente no es simple manejar una nueva serie de datos con un único módulo de lectura/impresión. Esto implica considerables costos de implementación y mantenimiento en códigos de IO para cada nuevo problema.

Finalmente, introducir un nuevo algoritmo a un código ya existente, en general, requiere implementación interna. Esto implica que los desarrolladores externos a un proyecto interesados en implementar un nuevo algoritmo deben aprender la estructura interna del programa.

1.3. Objetivos

El objetivo general del presente trabajo consiste en **Diseñar, desarrollar e implementar una arquitectura de software orientada a objetos (object oriented framework architecture) para la resolución de ecuaciones diferenciales a través del Método de Elementos Finitos**. A su vez, también, se propone implementar algún número de aplicaciones para poner a prueba la funcionalidad y flexibilidad de la framework.

Generalidad es una de las prioridades del proyecto, debido a múltiples limitaciones no es posible explorar en detalle las aplicaciones más complejas, como ser los problemas de optimización multi-física. No obstante, se planteó el diseño con la idea de brindar una arquitectura compatible con nuevas implementaciones, ya sea en términos de algoritmos o dominios físicos.

Reusabilidad es otro de los objetivos de diseño. La metodología de elementos finitos posee muchos pasos que son similares entre los distintos algoritmos, incluso para distintos tipos de problemas. Un buen diseño permitirá al desarrollador de las distintas aplicaciones reutilizar un gran número de las estructuras de datos y algoritmos ya implementadas en la framework.

Finalmente, un objetivo muy importante de este trabajo es mantener un cierto nivel de **performance** en contraste con los programas de propósito único no orientados a objetos. Se conoce previamente que el hecho de realizar un desarrollo con visión general bajo el paradigma orientado a objetos tendrá como consecuencia una cierta pérdida en eficiencia, es decir, los programas aquí presentados serán *más lentos* resolviendo un problema que un programa de propósito único completamente optimizado para resolver ese mismo problema. Se buscará en este trabajo contrastar esa pérdida en eficiencia con las ganancias en extensibilidad y mantenibilidad del código

1.4. Organización del trabajo

La idea de este trabajo es dar una visión general de la arquitectura diseñada dividiéndola en sus partes principales, en las que luego se comentan las subpartes que las componen. De acuerdo a esto el trabajo se organiza de la siguiente manera:

- **Parte I:** Luego de la introducción y justificación del tema se explican y definen brevemente los conceptos principales brindándose las referencias correspondientes para una búsqueda más profunda de los mismos y, finalmente, se presentan las distintas herramientas usadas en el desarrollo del trabajo.
- **Parte II:** Se presenta la arquitectura de software propuesta para este proyecto incluyendo la estructura general, sus partes principales, las herramientas generadas, la implementación del método de los elementos finitos y el input/Output del software.
- **Parte III:** Se habla de como desarrollar una aplicación. Se enumera el total de las aplicaciones desarrolladas. Se explica en detalle el diseño e implementación de algunas de ellas, además de los test realizados para su validación.
- **Parte IV:** Presenta las conclusiones y los resultados obtenidos a lo largo del desarrollo del trabajo además de plantear algunos desarrollos futuros posibles a partir de la arquitectura generada.

2. Definición de conceptos

Para cumplir con el objetivo de este trabajo es necesario tener en cuenta los conceptos básicos que a continuación se detallan sobre análisis numérico, programación y el método de los elementos finitos.

2.1. Análisis numérico

2.1.1. Esquema de análisis numérico

Los distintos métodos de análisis numérico tienen en común un esquema global, el cual los hace similares en sus metodologías y en ciertos casos los hace intercambiables entre sí. Este esquema consiste de tres pasos principales:

- **Idealización** Definir un modelo matemático que refleje la naturaleza del sistema físico. En este paso las ecuaciones y las condiciones que gobiernan el sistema son transformadas en una forma general que permita resolverlas numéricamente. Generalmente es necesario hacer ciertas suposiciones para que el modelado sea posible. Estas suposiciones hacen que el modelo matemático sea diferente al problema físico, por lo que se introduce un *error de modelado* en la solución

- **Discretización** Convertir al modelo matemático con un número infinito de valores desconocidos, llamados *grados de libertad* (*dof*, por sus siglas en inglés), a un número finito de ellos. Esto es necesario ya que el modelo original con infinitas variables no puede ser resuelto numéricamente, mientras que el modelo discretizado podrá ser resuelto a través de métodos numéricos. Es importante notar que este paso implica una aproximación. Como consecuencia en este proceso surge el *error de discretización*, el cual depende mucho de la calidad de la discretización y de la metodología utilizada.
- **Resolución** Resolver el modelo discreto para obtener los grados de libertad desconocidos. Este paso introduce el *error de resolución* el cual surge por inexactitudes en los algoritmos utilizados para llegar a esta solución.

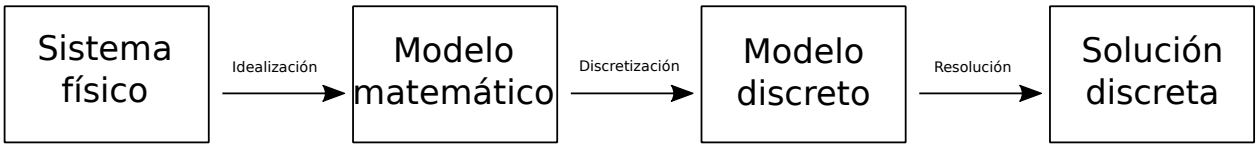


Figura 2.1: Los tres pasos del proceso de análisis numérico

La figura 2.1 representa el esquema global del análisis numérico. Es importante tener en cuenta los errores y las aproximaciones que se asumen con cada paso del proceso. La acumulación de estos errores afectará la validez de los resultados obtenidos por estos métodos. Reducir el error en cada uno de estos pasos es uno de los desafíos principales en el uso de métodos numéricos.

2.1.2. Idealización

La tarea principal de este paso es encontrar un modelo matemático adecuado para el problema físico.

Los modelos matemáticos están usualmente basados en diferentes suposiciones. Esto significa que sólo serán útiles para problemas en los cuales éstas suposiciones son correctas, o cercanas a la realidad. Por ésta razón, encontrar un buen modelo matemático no sólo requiere un buen conocimiento del problema a resolver sino que también de las suposiciones que se realizarán.

Hay diferentes formas de modelos matemáticos, los más comunes son: **forma fuerte**, **forma débil** y **forma variacional**. A continuación se las explica.

Forma fuerte

Se define el modelo matemático como un sistema de ecuaciones diferenciales y sus correspondientes condiciones de contorno. Considerando el dominio Ω con contorno Γ mostrado en la figura 2.2, esta forma define el modelo como una serie de ecuaciones sobre el dominio y el contorno de la forma:

$$\begin{cases} \mathcal{L}\left(u(x)\right) = p & x \in \Omega \\ \mathcal{S}\left(u(x)\right) = q & x \in \Gamma \end{cases} \tag{2.1}$$

Donde $u(x)$ es la función desconocida, $\mathcal{L}$ es el operador aplicado sobre el dominio Ω y $\mathcal{S}$ representa al operador aplicado sobre el contorno Γ. Un ejemplo de la ecuación (2.1) aplicado a un problema de transferencia de calor, según se puede ver en la figura 2.3 se modelará de la forma:

$$\begin{cases} \nabla^T \mathbf{k} \nabla \phi(x) = Q & x \in \Omega \\ \phi(x) = \phi_\Gamma & x \in \Gamma_\phi \\ q(x) = q_\Gamma & x \in \Gamma_q \end{cases} \tag{2.2}$$

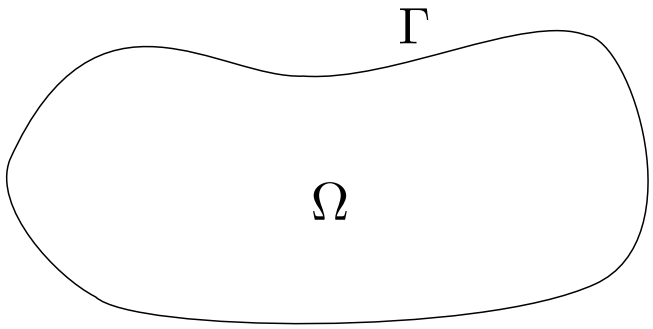


Figura 2.2: Dominio del problema y su contorno

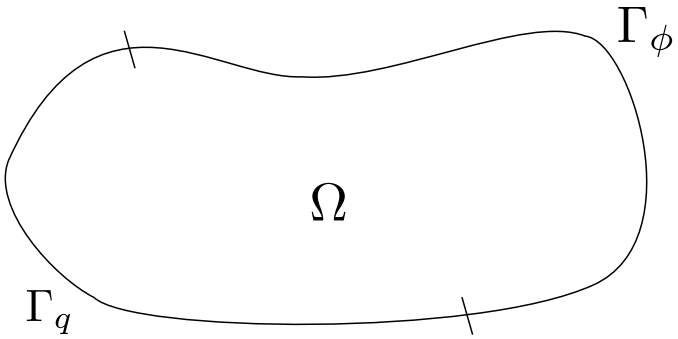


Figura 2.3: Dominio térmico Ω con temperatura fija en Γ_ϕ y flujo de calor fijo en Γ_q.

Donde $\phi(x)$ es la temperatura en Ω, q es el flujo de calor en el contorno, Q es la fuente de calor interna, Γ_ϕ es el contorno con temperatura fija ϕ_Γ, y Γ_q es el contorno con flujo fijo q_Γ.

Forma débil

Siendo V el espacio de Banach, considerando el problema de encontrar $u \in V$ de la ecuación:

$$\mathcal{L}(u) = p \qquad u \in V \tag{2.3}$$

Se puede verificar que el problema es equivalente a encontrar la solución $u \in V$ tal que para todos los $v \in V$ rige:

$$(\mathcal{L}(u), v) = (p, v) \qquad u \in V,\ \forall v \in V \tag{2.4}$$

Esta es conocida como la *formulación débil* del problema. La forma débil define al modelo matemático usando una formulación débil de la forma fuerte. Ahora usando el siguiente producto escalar:

$$\langle u, v \rangle = \int_\Omega uv d\Omega \tag{2.5}$$

resulta en la *forma integral*, la cual es la representación integral del modelo:

$$\int_\Omega \mathcal{L}(u) v d\Omega = \int_\Omega pv d\Omega \qquad u \in V,\ \forall v \in V \tag{2.6}$$

Esta formulación puede ser aplicada para incluir a las condiciones de contorno. Por ejemplo, el mismo problema representado en la ecuación (2.1) puede ser escrita de la siguiente forma:

$$\int_\Omega r(u) w d\Omega + \int_\Gamma \bar{r}(u) \bar{w} d\Gamma = 0 \qquad u \in V,\ \forall w, \bar{w} \in V \tag{2.7}$$

donde r y $\bar{r}$ son las funciones residuales definidas sobre el dominio y sobre el contorno respectivamente:

$$r(u) = \mathcal{L}(u) - p \tag{2.8}$$
$$\bar{r}(u) = \mathcal{S}(u) - q \tag{2.9}$$

y w y $\bar{w}$ son funciones de peso arbitrarias sobre el dominio y el contorno. Usualmente es conveniente realizar una integración por partes para reducir el orden máximo de las derivadas en las ecuaciones, al mismo tiempo aplicando derivadas sobre las funciones de peso. Realizando integración por partes sobre la ecuación (2.7) se obtiene:

$$\int_\Omega A(u)B(w)d\Omega + \int_\Gamma C(u)D(\bar{w})d\Gamma = 0 \qquad u \in V \; \forall w, \bar{w} \in V \tag{2.10}$$

Reduciendo el orden de las derivadas in A y C respectivamente a r y $\bar{r}$ permite un requerimiento menor en el orden de continuidad de la función elegida para representar a u. No obstante, implica un mayor requerimiento en el orden de continuidad de las funciones w y $\bar{w}$.

Forma variacional

La forma variacional surge usualmente de una cantidad fundamental del problema, como masa, momento o energía, de los cuales sus valores estacionarios son de interés. Se define el modelo matemático con una función de la forma siguiente:

$$\Pi(u) = \int_\Omega F(u)d\Omega \tag{2.11}$$

El estado estacionario de esta cantidad es requerido, por lo que toma la variación igual a cero, lo cual resulta en la siguiente ecuación:

$$\delta\Pi(u) = \int_\Omega \delta\left(F(u)\right)d\Omega = 0 \tag{2.12}$$

Derivar las ecuaciones variacionales a partir de leyes de conservación es atractivo para científicos ya que estas presentan las mismas características fundamentales del problema. Finalmente es importante mencionar que la forma débil puede ser derivada del estado estacionario de la forma variacional.

2.1.3. Discretización

Consiste en convertir el modelo continuo en uno discreto, con un número finito de valores desconocidos para encontrar.

Hay numerosas maneras de realizar esta conversión, lo cual resulta en diferentes métodos numéricos. El método apropiado de discretización dependerá no solo del tipo de problema a resolver, si no que también del modelo matemático elegido para describirlo. A continuación se brinda a una breve descripción de diferentes métodos de discretización.

Discretización de la forma fuerte

La forma fuerte es generalmente discretizada utilizando el *método de diferencias finitas*. Consiste simplemente en resolver las diferentes derivadas planteándolas como diferencias. Por ejemplo la primer derivada de la función $f(x)$ puede ser reemplazada por su forma discreta según:

$$\frac{df(x)}{dx} \approx \Delta_h^1(f,x) = \frac{1}{h}\left(f(x+h) - f(x)\right) \tag{2.13}$$

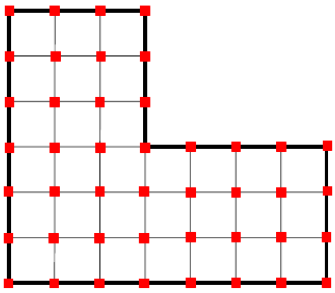
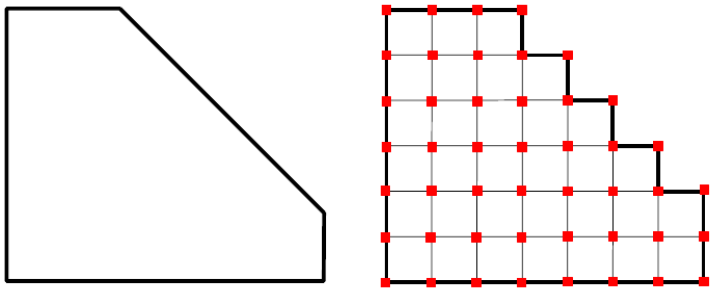


Figura 2.4: Dominio regular discretizado con una grilla de diferencias finitas

Donde el parámetro de discretización h es la distancia entre los puntos de la grilla. Este método también puede ser aplicado para resolver derivadas de alto orden:

$$\frac{d^n f(x)}{dx^n} \approx \Delta_h^n(f,x) = \sum_{i=0}^{n} (-1)^{n-k} \left(\frac{1}{h}\right)^n \binom{n}{i} f(x+ih) \tag{2.14}$$



(a) (b)

Figura 2.5: Geometría arbitraria y su modelo discreto de diferencias finitas

La discretización es simplemente una grilla cartesiana sobre el dominio. La figura 2.4 muestra un ejemplo de una grilla sobre un espacio bidimensional.

Este método ha sido utilizado de manera práctica en múltiples campos e implementado en muchas aplicaciones. Su metodología es simple y fácil de programar. Esto contribuyó a hacerlo uno de los métodos favoritos en el análisis numérico. No obstante, cuenta con algunos defectos. Uno de ellos es que, funciona bien para dominios regulares, pero encuentra dificultades para discretizar algunas geometrías y condiciones de contorno arbitrarias. Por ejemplo, el dominio irregular de la figura 2.5 (a) puede ser aproximado por el dominio discretizado de la figura 2.5 (b). Aquí puede verse fácilmente que esta discretización cambia los contornos del dominio para una geometría arbitraria. De todos modos, para estas situaciones, se puede utilizar la transformación del dominio en la aplicación del método, implicando un paso extra en la solución.

Otra desventaja es su solución aproximada, la cual puede ser obtenida sobre los puntos de la grilla, no habiendo ningún tipo de información sobre los otros puntos contenidos en la geometría.

Discretización de la forma débil

Considerando la forma débil en el espacio continuo V:

$$(\mathcal{L}(u), v) = (p, v) \qquad u \in V,\ \forall v \in V \tag{2.15}$$

Esta forma puede ser transformada al espacio discreto V_h para aproximar la solución con $u_h \in V$:

$$(\mathcal{L}(u_h), v_h) = (p, v_h) \qquad u \in V,\ \forall v_h \in V \tag{2.16}$$

Como se mencionó anteriormente la forma débil puede ser representada de manera integral con pesos de la forma:

$$\int_\Omega r(u) w d\Omega + \int_\Gamma \bar{r}(u)\bar{w} d\Gamma = 0 \qquad u \in V,\ \forall w, \bar{w} \in V \tag{2.17}$$

donde r y $\bar{r}$ son funciones residuales definidas sobre el dominio y sobre el contorno respectivamente. w y $\bar{w}$ son funciones arbitrarias de peso sobre el dominio y el contorno. Aquí seleccionando un espacio discreto V_h como nuestro espacio de trabajo resulta en el siguiente modelo discreto:

$$\int_\Omega r(u_h) w_h d\Omega + \int_\Gamma \bar{r}(u_h)\bar{w}_h d\Gamma = 0 \qquad u_h \in V_h,\ \forall w_h, \bar{w}_h \in V_h \tag{2.18}$$

donde r y $\bar{r}$ son funciones residuales. La ecuación (2.18) es una *integral de residuos ponderados*. Esta clase de aproximaciones es referida como *métodos de residuos ponderados*. Muchos métodos conocidos como el *Método de Elementos Finitos (FEM)*, *Método de Volumen Finito (FV)* y *Ajuste de Mínimos Cuadrados* son subclases de éste método.

Es común elegir un espacio discreto V_h formado por un arreglo conocido de *funciones de prueba* N_i y definir la solución discreta de la forma:

$$u_h \approx \sum_{j=1}^{n} a_j N_j \tag{2.19}$$

donde a_j son coeficientes desconocidos, N_j son funciones de prueba conocidas, y n es el número de desconocidos. Sustituyendo esto en la ecuación (2.18):

$$\int_\Omega r\left(\sum_{j=1}^{n} a_j N_j\right) w_h d\Omega + \int_\Gamma \bar{r}\left(\sum_{j=1}^{n} a_j N_j\right) \bar{w}_h d\Gamma = 0 \qquad \forall w_h, \bar{w}_h \in V_h \tag{2.20}$$

con

$$w_h = \sum_{i=1}^{n} \alpha_i w_i \quad ; \quad \bar{w}_h = \sum_{i=1}^{n} \alpha_i \bar{w}_i \tag{2.21}$$

donde α_i son coeficientes arbitrarios, w_i y $\bar{w}_i$ son funciones arbitrarias, y n es el número de desconocidos.

Expandiendo (2.20):

$$\sum_{i=1}^{n} \alpha_i \left[\int_\Omega r\left(\sum_{j=1}^{n} a_j N_j\right) w_i d\Omega + \int_\Gamma \bar{r}\left(\sum_{j=1}^{n} a_j N_j\right) \bar{w}_i d\Gamma\right] = 0 \qquad \forall \alpha_i, w_i, \bar{w}_i \in V_h \tag{2.22}$$

Como los coeficientes α_i son arbitrarios, todos los componentes de la suma anterior deben ser igual a cero para satisfacer la ecuación. Esto resulta en la siguiente serie de ecuaciones:

$$\int_\Omega r\left(\sum_{j=1}^{n} a_j N_j\right) w_i d\Omega + \int_\Gamma \bar{r}\left(\sum_{j=1}^{n} a_j N_j\right) \bar{w}_i d\Gamma \qquad \forall w_i, \bar{w}_i \in V_h\ , \qquad i = 1, 2, 3, ..., n \tag{2.23}$$

Hay múltiples tipos de funciones que pueden ser elegidas como funciones de peso. A continuación se listan algunas de las opciones más comunes:

a) Método de colocación Se utiliza la *delta de Dirac* δ_i como función de peso:

$$w_i = \delta_i \quad ; \quad \bar{w}_i = \delta_i \tag{2.24}$$

donde δ_i es una función tal que:

$$\int_\Omega f\delta_i d\Omega = f_i \tag{2.25}$$

Sustituyendo estas funciones de peso en la ecuación (2.23) resulta en el siguiente modelo discreto:

$$r_i \left(\sum_{j=1}^{n} a_j N_j \right) + \bar{r}_i \left(\sum_{j=1}^{n} a_j N_j \right) = 0 \ , \qquad i = 1, 2, 3, ..., n \tag{2.26}$$

Este método satisface la ecuación solo en el arreglo de puntos de colocación elegidos, y resulta en un sistema de ecuaciones similar al obtenido por el método de diferencias finitas.

b) Colocación en subdominios Es una extensión del método anterior. Usa funciones de peso w_i con valor identidad un subdominio Ω_i y cero en el resto del dominio:

$$w_i = \begin{cases} I & x \in \Omega_i \\ 0 & x \notin \Omega_i \end{cases} \quad , \quad \bar{w}_i = \begin{cases} I & x \in \Gamma_i \\ 0 & x \notin \Gamma_i \end{cases} \tag{2.27}$$

El modelo discreto puede ser obtenido sustituyendo estas funciones de peso en la ecuación general de la forma débil (2.23):

$$\int_{\Omega_i} r \left(\sum_{j=1}^{n} a_j N_j \right) d\Omega_i + \int_{\Gamma_i} \bar{r} \left(\sum_{j=1}^{n} a_j N_j \right) d\Gamma_i = 0 \ , \quad i = 1, 2, 3, ..., n \tag{2.28}$$

Este método presenta una aproximación uniforme para cada subdominio y establece una manera de dividir el dominio en subdominios para resolver el problema.

c) Método de mínimos cuadrados Este método usa el mismo operador aplicado a las funciones de prueba para las funciones de peso:

$$w_i = \delta\mathcal{L}(N_i) \quad , \quad \bar{w}_i = \delta\mathcal{S}(N_i) \tag{2.29}$$

El resultado de sustituir estas funciones de peso en la ecuación (2.23) es:

$$\int_\Omega r \left(\sum_{j=1}^{n} a_j N_j \right) \mathcal{L}(N_i) d\Omega + \int_\Gamma \bar{r} \left(\sum_{j=1}^{n} a_j N_j \right) \mathcal{S}(N_i) d\Gamma = 0 \ , \quad i = 1, 2, 3, ..., n \tag{2.30}$$

Se puede verificar que la ecuación resultante es equivalente a minimizar el cuadrado de el residuo global $\mathcal{R}$ sobre el dominio:

$$\delta\mathcal{R} = 0 \tag{2.31}$$

donde

$$\mathcal{R} = \int_\Omega r^2(u_h) d\Omega + \int_\Gamma \bar{r}^2(u_h) d\Gamma = 0 \tag{2.32}$$

d) Método de Garlekin Utiliza las funciones de prueba como funciones de peso:

$$w_i = N_i \quad , \quad \bar{w}_i = N_i \tag{2.33}$$

Sustituyendo en la ecuación (2.23):

$$\int_{\Omega} r \left(\sum_{j=1}^{n} a_j N_j \right) N_i d\Omega + \int_{\Gamma} \bar{r} \left(\sum_{j=1}^{n} a_j N_j \right) N_i d\Gamma = 0 \ , \quad i = 1, 2, 3, ..., n \tag{2.34}$$

Este método usualmente mejora el proceso de resolución ya que, generalmente, resulta en matrices simétricas con otras características útiles que lo hacen una metodología preferida y muy comúnmente utilizada en el método de elementos finitos.

2.1.4. Resolución

El último paso de la metodología numérica es la resolución del problema. Este modelo consiste en resolver el modelo discreto utilizando algoritmos adecuados para encontrar el valor de los grados de libertad y también poder calcular resultados adicionales. Este proceso incluye:

- **Cálculo de componentes** Todos los componentes de un modelo discreto son calculados. Esto puede ser derivadas en el método de diferencias finitas, integrales en el método de residuos ponderados o variacionales, entre otros valores.
- **Ensamblaje del sistema global** Los componentes del modelo discreto se unen para ensamblar un sistema global de ecuaciones que representa al sistema discreto completo.
- **Resolución del sistema global** Se resuelve para encontrar los desconocidos del problema. En algunos modelos esto implica la resolución de un sistema de ecuaciones lineales, para lo cual se utiliza un *solver* lineal.
- **Cálculo de resultados adicionales** En muchos problemas no basta con obtener el valor de los grados de libertad, si no que interesan otros resultados. Por ejemplo en problemas estructurales de la resolución del sistema global se obtienen desplazamientos, y esos desplazamientos se utilizan para calcular valores importantes como tensión y deformación.
- **Iteración** En muchos algoritmos es necesario iterar para obtener las incógnitas o para calcular una serie de diferentes valores desconocidos. Algunos casos donde iteración es necesaria son para resolver sistemas no lineales, calcular valores dependientes del tiempo y resolver problemas de optimización.

2.2. Método de los elementos finitos

El Método de Elementos Finitos (MEF, o FEM según sus siglas en inglés) hace uso de la forma integral del problema y utiliza en general polinomios como funciones de prueba. Hay una gran variedad de formulaciones pero la más utilizada es a través del método de Garlekin, el cual se usa a continuación para describir el FEM y sus pasos básicos.

2.2.1. Discretización

Considerando el problema continuo:

$$\begin{cases} \mathcal{L}\left(u(x)\right) = p & x \in \Omega \\ \mathcal{S}\left(u(x)\right) = q & x \in \Gamma \end{cases} \tag{2.35}$$

y su forma integral

$$\int_{\Omega} r(u) w d\Omega + \int_{\Gamma} \bar{r}(u) \bar{w} d\Gamma = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{2.36}$$

donde r y $\bar{r}$ son funciones residuales definidas sobre el dominio y el contorno respectivamente:

$$r(u) = \mathcal{L}(u) - p \tag{2.37}$$
$$\bar{r}(u) = \mathcal{S}(u) - q \tag{2.38}$$

Definiendo el espacio del elemento finito V_h como una composición de funciones polinómicas N_i:

$$x = \sum_{i=1}^{n} \alpha_i^x N_i \tag{2.39}$$

y transformando la ecuación (2.36) a este espacio resulta en:

$$\int_\Omega r(u_h) w_h d\Omega + \int_\Gamma \bar{r}(u_h) \bar{w}_h d\Gamma = 0 \qquad u_h \in V_h, \forall w_h, \bar{w}_h \in V_h \tag{2.40}$$

donde:

$$u_h = \sum_{i=1}^{n} \alpha_i N_i \tag{2.41}$$

$$w = \sum_{i=1}^{n} \beta_i N_i \tag{2.42}$$

$$\bar{w} = \sum_{i=1}^{n} \beta_i N_i \tag{2.43}$$

Expandiendo la ecuación (2.40) con estas definiciones resulta en:

$$\int_\Omega r \left(\sum_{j=1}^{n} \alpha_j N_j \right) \sum_{i=1}^{n} \beta_i N_i d\Omega + \int_\Gamma \bar{r} \left(\sum_{j=1}^{n} \alpha_j N_j \right) \sum_{i=1}^{n} \beta_i N_i d\Gamma = 0 \qquad \alpha, N \in V_h \ , \ \forall \beta \in V_h \tag{2.44}$$

alternativamente sacando el β_i de las integrales:

$$\sum_{i=1}^{n} \beta_i \left[\int_\Omega r \left(\sum_{j=1}^{n} \alpha_j N_j \right) N_i d\Omega + \int_\Gamma \bar{r} \left(\sum_{j=1}^{n} \alpha_j N_j \right) N_i d\Gamma \right] = 0 \qquad \alpha, N \in V_h \ , \ \forall \beta \in V_h \tag{2.45}$$

Como β_i es arbitrario, todos los componentes de la suma anterior deben ser cero para satisfacer la ecuación. Esto resulta en la siguiente serie de ecuaciones:

$$\int_\Omega r \left(\sum_{j=1}^{n} \alpha_j N_j \right) N_i d\Omega + \int_\Gamma \bar{r} \left(\sum_{j=1}^{n} \alpha_j N_j \right) N_i d\Gamma \qquad \alpha, N \in V_h \ , \ i = 1, 2, 3, ..., n \tag{2.46}$$

Se puede observar la equivalencia entre la forma discreta de la ecuación (2.46) y la ecuación (2.34) obtenida por el método de Garlekin en la sección 2.1.3.

Nodo y grado de libertad

En el método de los elementos finitos, cada valor desconocido es referido como un *grado de libertad (dof)* y es considerado como la solución de elementos finitos u_h en un punto del dominio llamado *nodo*.

$$\alpha_i = a_i \tag{2.47}$$

donde a_i es la solución aproximada u_h en el nodo i. Usando esta suposición, la serie de ecuaciones (2.46) se puede reescribir:

$$\int_\Omega r\left(\sum_{j=1}^{n} a_j N_j\right) N_i d\Omega + \int_\Gamma \bar{r}\left(\sum_{j=1}^{n} a_j N_j\right) N_i d\Gamma \qquad \alpha, N \in V_h\ ,\ i = 1, 2, 3, ..., n \tag{2.48}$$

Estas ecuaciones pueden ser resueltas para obtener directamente los desconocidos en cada nodo del dominio. Sustituyendo la ecuación (2.47) en (2.41) resulta:

$$u_h = \sum_{i=1}^{n} a_i N_i \tag{2.49}$$

la cual relaciona la solución aproximada u_h sobre el dominio con los valores nodales obtenidos de resolver el conjunto de ecuaciones de (2.48). De esta manera, la solución aproximada puede ser obtenida no solo en todos los nodos, si no también en todos los puntos del dominio.

Funciones de forma y elementos

Volviendo a la ecuación (2.48), una serie de funciones de prueba N_i son necesarias para definir el problema. En el método de los elementos finitos estas funciones son llamadas *funciones de forma*. Una correcta definición de las funciones de forma es muy importante para obtener una buena aproximación de la solución y sus propiedades.

La discretización introducida en la sección anterior reduce el infinito número de desconocidos de la ecuación (2.36) por un número finito n en (2.48). Este es un gran paso para obtener un modelo que pueda ser resuelto numéricamente, pero en la práctica no es suficiente. El problema surge por el hecho de que cada ecuación en (2.48) contiene una ecuación completa sobre el dominio y una relación completa entre los desconocidos de la ecuación, lo cual hace que la resolución sea muy costosa. Para evitar estos problema, se divide el dominio Ω en múltiples sub-dominios Ω^e como se puede ver:

$$\Omega^1 \cup \Omega^2 \cup ... \cup \Omega^e \cup ...\Omega^m = \omega \quad , \quad \Omega^1 \cap \Omega^2 \cap ... \cap \Omega^e \cap ...\Omega^m = \emptyset \tag{2.50}$$

donde cada partición Ω^e es llamada *elemento*. Dividiendo las integrales de (2.48) se obtiene:

$$\sum_{e=1}^{m}\left[\int_{\Omega^e} r\left(\sum_{j=1}^{n} a_j N_j\right) N_i d\Omega^e + \int_{\Gamma^e} \bar{r}\left(\sum_{j=1}^{n} a_j N_j\right) N_i d\Gamma^e\right] = 0\ , \quad i = 1, 2, 3, ..., n \tag{2.51}$$

donde Γ^e es la parte del contorno Γ correspondiente al elemento Ω^e según se puede ver en la figura 2.6. A continuación se definen las funciones de forma N_i según:

$$N_i = \begin{cases} N_i^e & x \in \Omega^e \cup \Gamma^e \\ 0 & x \notin \Omega^e \cup \Gamma^e \end{cases} \tag{2.52}$$

Sustituyendo la expresión (2.52) en (2.48) resulta en:

$$\sum_{e=1}^{m_i}\left[\int_{\Omega^e} r\left(\sum_{j=1}^{m_i} a_j N_j^e\right) N_i^e d\Omega^e + \int_{\Gamma^e} \bar{r}\left(\sum_{j=1}^{m_i} a_j N_j^e\right) N_i^e d\Gamma^e\right] = 0\ , \quad i = 1, 2, 3, ..., n \tag{2.53}$$

donde m_i es el número de elementos que contienen al nodo i. De esta manera la relación entre los desconocidos es reducida a una relación entre vecinos, y en cada ecuación la integración solo debe ser realizada sobre algunos elementos.

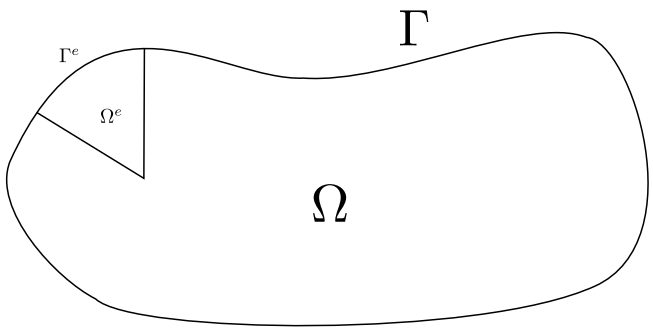


Figura 2.6: Elemento Ω^e y su contorno Γ^e

Condiciones de contorno

Considerando nuevamente el problema:

$$\begin{cases} \mathcal{L}\left(u(x)\right) = p & x \in \Omega \\ \mathcal{S}\left(u(x)\right) = q & x \in \Gamma \end{cases} \tag{2.54}$$

y asumiendo que $\mathcal{L}$ contiene como máximo derivadas de orden m. Las condiciones de contorno de tal problema se pueden dividir en dos categorías: *esenciales* y *naturales*.

Las condiciones de contorno esenciales $\mathcal{S}_D$ son condiciones que especifican los valores de la solución que necesita la frontera del dominio. Estas condiciones son también llamadas *condiciones de Dirichlet*. Por ejemplo un desplazamiento prescrito en un problema estructural es una condición de Dirichlet.

El resto de las condiciones de contorno son consideradas naturales $\mathcal{S}_N$. Estas también son referidas como *condiciones de Neumann*. Por ejemplo el flujo de calor fijo en un problema de transferencia de calor es una condición de Neumann.

Aplicando esta división a la ecuación (2.54) se obtiene:

$$\begin{cases} \mathcal{L}\left(u(x)\right) = p & x \in \Omega \\ \mathcal{S}_D\left(u(x)\right) = q_D & x \in \Gamma_D \\ \mathcal{S}_N\left(u(x)\right) = q_N & x \in \Gamma_N \end{cases} \tag{2.55}$$

donde $\mathcal{S}_D$ es la condición de Dirichlet aplicada al contorno Γ_D y $\mathcal{S}_N$ es la condición de Neumann aplicada al contorno Γ_N como se puede ver en la figura 2.7. Transformando la ecuación (2.55) a su forma integral resulta en:

$$\int_\Omega r(u) w d\Omega + \int_{\Gamma_D} \bar{r}_D(u) \bar{w} d\Gamma_D + \int_{\Gamma_N} \bar{r}_N(u) \bar{w} d\Gamma_N \qquad u \in V, \forall w, \bar{w} \in V \tag{2.56}$$

donde:

$$r(u) = \mathcal{L}(u) - p \tag{2.57}$$

$$\bar{r}_D(u) = \mathcal{S}_D(u) - q_D \tag{2.58}$$

$$\bar{r}_N(u) = \mathcal{S}_N(u) - q_N \tag{2.59}$$

Si se restringe la solución u a funciones que satisfacen las condiciones de Dirichlet en Γ_D, la integral sobre el contorno de dirichlet puede ser omitida restringiendo la elección de $\bar{w}$ a funciones que sean nulas sobre Γ_D. Usando esta restricción:

$$\begin{aligned} \int_\Omega r(u) w d\Omega + \int_{\Gamma_N} \bar{r}_N(u) \bar{w} d\Gamma_N = 0 \qquad & u \in V, \forall w, \bar{w} \in V \\ u = \bar{u} \qquad & x \in \Gamma_D \end{aligned} \tag{2.60}$$

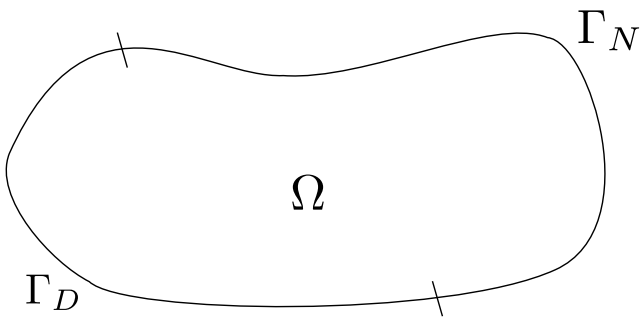


Figura 2.7: Un dominio y sus contornos de Dirichlet y Neumann

donde $\bar{u}$ es la solución sobre el dominio de Dirichlet. Convirtiendo el modelo anterior a su forma discreta usando el mismo proceso descrito anteriormente se obtienen las siguientes ecuaciones discretas:

$$\begin{aligned} \int_\Omega r \left(\sum_{j=1}^{n} a_j N_j \right) N_i d\Omega + \int_{\Gamma_N} \bar{r}_n \left(\sum_{j=1}^{n} a_j N_j \right) N_i d\Gamma_N = 0 \qquad & a, N \in V_h \ , \ i = 1, 2, ..., n \\ u = \bar{u} \qquad & x \in \Gamma_D \end{aligned} \tag{2.61}$$

2.2.2. Resolución

Cálculo de componentes

Aplicando la condición de contorno esencial a la ecuación (2.53) resulta en:

$$\begin{aligned} \sum_{e=1}^{m_i} \left[\int_{\Omega^e} r \left(\sum_{j=1}^{m_i} a_j N_j^e \right) N_i^e d\Omega^e + \int_{\Gamma_N^e} \bar{r} \left(\sum_{j=1}^{m_i} a_j N_j^e \right) N_i^e d\Gamma_N^e \right] = 0 \qquad & i = 1, 2, ..., n \\ u = \bar{u} \qquad & x \in \Gamma_D \end{aligned} \tag{2.62}$$

Si las ecuaciones diferenciales son lineales entonces se puede escribir la ecuación de la forma:

$$\mathbf{K}\mathbf{a} + \mathbf{f} = 0 \tag{2.63}$$

donde:

$$\mathbf{K}_{ij} = \sum_{e=1}^{m} \mathbf{K}_{ij}^e \tag{2.64}$$

$$\mathbf{f}_i = \sum_{e=1}^{m} \mathbf{f}_i^e \tag{2.65}$$

Esta paso consiste en calcular las funciones de forma y sus derivadas en cada elemento y luego realizar la integración para cada elemento.

La técnica usual consiste en calcular estos componentes en coordenadas locales al elemento y luego transformar el resultado a coordenadas globales. Comúnmente las funciones de forma están definidas en términos de coordenadas locales y sus valores y gradientes con respecto a dichas coordenadas son conocidos. No obstante, las matrices elementales contienen gradientes de las funciones de forma con respecto a las coordenadas globales. Estos gradientes deben ser calculados usando los locales y la inversa de una matriz $\mathbf{J}$ conocida como el *Jacobiano*. Considerando las coordenadas globales x, y, z y las locales

al elemento ξ, η, ζ, se puede ver que el gradiente de las funciones de forma con respecto a las coordenadas locales se puede escribir en términos de las globales de la forma:

$$\begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} \tag{2.66}$$

Ahora los gradientes de las funciones de forma con respecto a las coordenadas globales pueden ser calculadas según:

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix} \tag{2.67}$$

Luego de calcular los gradientes de las funciones de forma con respecto a las coordenadas globales, es posible integrarlos sobre los elementos. Esto puede ser realizado la integral sobre el dominio de coordenadas globales a locales según:

$$\int_{\Omega^e} f dx dy dz = \int_{\Omega^e} f det(\mathbf{J}) d\xi d\eta d\zeta \tag{2.68}$$

Ahora todas las matrices elementales puede ser calculadas utilizando coordenadas locales. La forma usual de calcular integrales sobre un elementos es utilizando el método de la *Cuadratura de Gauss*. Este método convierte la integración de una función sobre el dominio en una suma ponderada de funciones valuadas en ciertos puntos:

$$\sum_{a}^{b} f(x) dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{2.69}$$

En la sección 5.2 se explicará en detalle como se implementó este método para realizar integraciones en la framework.

Ensamble del sistema global

Como se mencionó anteriormente, para el caso de ecuaciones diferenciales lineales las ecuaciones de (2.62) pueden ser escritas como un sistema lineal de la forma:

$$\mathbf{K}\mathbf{a} = \mathbf{f} \tag{2.70}$$

donde

$$\mathbf{K}_{ij} = \underset{e=1}{\overset{m}{A}} \mathbf{K}_{ij}^e \tag{2.71}$$

$$\mathbf{f}_i = \underset{e=1}{\overset{m}{A}} \mathbf{f}_i^e \tag{2.72}$$

Teniendo las matrices elementales y los vectores K^e y f^e, el procedimiento de juntarlos para crear la ecuación del sistema global se llama **ensamble** y consiste en encontrar la posición de cada componente elemental en los sistemas de ecuaciones globales y sumarlo al valor en su posición. Se representa con el *assembly operator* A.

Este procedimiento primero asigna una numeración secuencial a todos los grados de libertad (dofs). A veces es útil separar los dofs restringidos con condiciones de Dirichlet de los otros. Esto se puede hacer fácilmente a la hora de asignar índices a dofs. Después de eso, el procedimiento va elemento por elemento y agrega sus matrices y vectores locales al sistema de ecuaciones globales.

Otra tarea a realizar cuando se construyen las ecuaciones del sistema global es la aplicación de condiciones de contorno esenciales. Este procedimiento se puede realizar sin reordenar las ecuaciones, pero es más conveniente separar las ecuaciones restringidas de otras para simplificar el proceso.

La matriz global del sistema obtenida por medio del MEF generalmente tiene muchos ceros en ella. Almacenar todos estos valores en una estructura de matriz que guarda todos los elementos, implica una gran sobrecarga en el uso de memoria debido a que se guardan también los elementos que son nulos. Hay varias estructuras alternativas para almacenar la porción útil de la matriz para resolver. Por ejemplo una estructura matricial en banda almacena una banda alrededor de la diagonal de una matriz y asume que todos los elementos fuera de la banda son ceros. Otro ejemplo son las estructuras matriciales sparse, como ser **compressed sparse row** ó **CRS** la cual almacena los elementos no ceros de cada fila con su correspondiente número de columna. Por último, otra estructura común es la estructura matricial simétrica que usa la propiedad de simetría de la matriz para almacenar aproximadamente la mitad de os elementos, esto puede ser combinado con una estructura sparse para guardar la mitad de los no ceros de la matriz.

2.2.3. Solución del sistema global

Luego de ensamblar y de aplicar las condiciones de contorno, el sistema esta listo para ser resuelto. Hay dos categorías de métodos para resolver un sistema de ecuaciones, los métodos *directos* y los métodos *iterativos*.

Los métodos **directos** resuelven el sistema por medio de la generación de las matrices triangular superior, inferior, diagonal o alguna descomposición de la forma superior inferior y calcular las incógnitas usando estas formas matriciales.

Los métodos **iterativos** comienzan con algún valor inicial de la incógnita y tratan de encontrar la solución correcta calculando el residuo y minimizándolo por iteración.

Para sistemas de ecuaciones chicos los métodos directos son muy rápidos ya que son menos dependientes del condicionamiento de la matriz, la única excepción es la existencia de un cero en la diagonal lo cual necesita un tratamiento especial, al contrario de los métodos iterativos que son altamente dependientes de la condición del sistema, lo que afecta considerablemente su convergencia. Usualmente para sistemas chicos se utilizan los métodos directos que son más rápidos y para sistemas de mediano a gran tamaño los métodos iterativos son más adecuados, dependiendo aún del condicionamiento del sistema.

2.3. Conceptos de programación

Diseñar e implementar un nuevo software es una tarea compleja. Hacer uso adecuado de soluciones de ingeniería de software conocidas y de técnicas de programación avanzada pueden incrementar significativamente la calidad del programa. A continuación se explican algunos de los conceptos de programación esenciales en el desarrollo de este proyecto.

2.3.1. Desarrollo modular y orientado a objetos

Las capacidades de un software para ser mantenido, extendido y modificado sin mayores complicaciones serán factores que se definirán en la etapa de diseño. El programador debe ser capaz de entender como operará el programa, como las distintas partes interactuarán entre sí y proponer un diseño que priorice claridad y eficiencia. Sin eficiencia el programa no será útil, sin claridad las tareas de mantenimiento o futuros cambios serán muy dificultosos, quizás hasta llegar a casos donde resultaría conveniente empezar un proyecto nuevo a modificar uno existente [5].

El desarrollo modular propone diseñar el programa pensándolo como un conjunto de bloques que interactúan entre sí. Cada bloque desarrolla una tarea específica, comunicándose con los otros a través de los datos que cada uno recibe y entrega.

El desarrollo orientado a objetos se basa en definir estructuras de datos, las cuales poseen una serie de atributos y rutinas a las que se podrá acceder una vez se declare una instancia de la misma dentro del

programa, en lo que normalmente se llama objeto. Estos objetos tendrán una gran cantidad de información y funcionalidades autocontenidas, permitiendo así mejorar la modularidad del programa.

Una de las primeras aplicaciones que viene a la mente para un problema como este es hacer uso de un objeto `Element`. Cada instancia de `Element` representará un elemento finito con el cual el resto del programa interactuará, obteniendo información de éste y haciendo uso de sus funcionalidades. No obstante, el resto del programa debería ser capaz de trabajar con `Element` independientemente de su naturaleza. Por ejemplo, para la rutina encargada de recibir un `Element` y pedirle su matriz de rigidez local para ensamblarla en la global, no será relevante la cantidad de nodos del mismo, o que orden tienen sus funciones de forma, ya que la implementación estaría autocontenida.

Esta idea es la más importante del desarrollo aquí presentado, la capacidad de en el futuro poder reemplazar un solver por otro, un modelo físico por otro, o quizás las rutinas de *Input/Output* para trabajar con otro pre/postprocesador sin que el resto del programa se vea modificado es un aspecto sumamente importante de la programación orientada a objetos.

Estos conceptos vienen siendo aplicados al método de los elementos finitos desde la década del 90' [6]. No obstante las variantes en el desarrollo orientado a objetos son ilimitadas y en este trabajo nos proponemos realizar un diseño propio.

Pilares de la programación orientada a objetos

Los pilares de la programación orientada a objetos están mencionados en la figura 2.8 y se los explica a continuación.

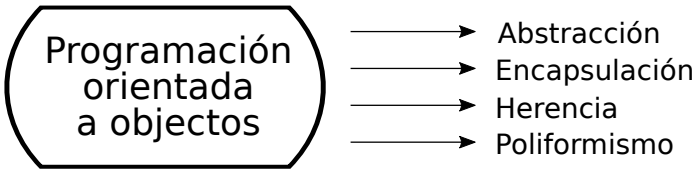


Figura 2.8: Pilares de la programación orientada a objetos

- **Abstracción** El usuario de un objeto solo verá la información que le es necesaria, quedando muchas de las implementaciones escondidas. Por ejemplo, se podría utilizar un objeto `shape`, el cual puede tomar cualquier forma. Para este objeto podría tener una rutina llamada `getArea`, por medio de la cual, internamente, se calcularía el área. El algoritmo utilizado para calcular el área no es relevante para el usuario, por lo que esta `shape` puede ser cualquier forma, y dependiendo de su forma el programa resolverá como obtener su área.

- **Encapsulación** Implica agrupar datos y métodos en una única unidad, la clase. El encapsulamiento incluye el concepto de *data hiding*. El programador de una clase elige que variables y rutinas serán privadas (aquellas que solo pueden ser invocadas desde el módulo en el cual se define la clase), y cuales serán públicas (aquellas que pueden ser invocadas en cualquier lugar a través del objeto). Una aplicación de este concepto es el uso de *getters* y *setters*, estas son rutinas que permiten modificar y acceder al valor de una variable privada. En este proyecto se hace uso de este concepto pero de una manera limitada ya que en ciertos casos no resultó conveniente hacer uso de variables privadas.

- **Herencia** Esta es la capacidad de crear una clase a partir de una existente. Este concepto le permite a la nueva clase *(clase hijo)* adquirir todas las variables y rutinas de la clase existente *(clase padre)*, esto es extremadamente útil para evitar reescribir código en el caso de necesitar clases con características generales, además permite utilizar código ajeno y personalizarlo en función de las necesidades propias.

- **Poliformismo** Le permite a un algoritmo trabajar con múltiples tipos de entidades siempre y cuando estas tengan una interfaz en común. Este concepto se aplica en múltiples oportunidades a lo largo de este proyecto cuando se definen rutinas que van a trabajar con un objeto `Element`. De la clase `Element` heredarán múltiples clases como `ThermalElement` o `StructuralElement`. No obstante esas rutinas podrán trabajar con objetos de ambas clases, gracias al polimorfismo.

2.3.2. Técnicas avanzadas de programación en Fortran

Performance y eficiencia de memoria son dos requerimientos cruciales para programas de elementos finitos. Es en esas áreas que Fortran se destaca con respecto a las alternativas. No obstante, Fortran históricamente se ha resistido a la implementación de conceptos de programación modernos que otros lenguajes han adoptado de manera natural.

Para el desarrollo de una arquitectura de software con las capacidades ya mencionadas será importante hacer uso de herramientas implementadas en las versiones de Fortran 2003 y 2008. En estas versiones se brinda soporte para el desarrollo orientado a objetos [7]. También se hará uso extensivo de punteros, para mantener un alto nivel de generalidad sin tener que duplicar información.

Derived Types

Derived type es la sintaxis utilizada en Fortran para referir a las estructuras de datos que ya se mencionaron anteriormente, las cuales contienen una serie de atributos y métodos. En C++ a estas estructuras se las llama *clases*, refiriendo a la plantilla para la creación de *objetos*, los cuales son *instancias* de dicha clase. Es importante tener en cuenta esta nomenclatura ya que se usará extensivamente a lo largo del proyecto.

En el fragmento de código 2.1 se puede ver la definición de un nuevo tipo o clase llamado `PointDT` (DT por Derived Type). En este ejemplo extremadamente simplificado se implementa una clase para representar puntos bidimensionales. Los atributos de `PointDT` son las variables reales (`x`, `y`) y su método es una subrutina para inicializar los valores. También se puede ver que se implementa un `constructor`, que se llamará con el nombre `point`, esto es útil para hacer una analogía directa con la metodología utilizada en C++ para inicializar objetos. Es decir, será posible inicializar puntos con la sintaxis:

```
myPoint = point(x, y)
```

Otro detalle a notar es que todas las rutinas que forman parte del objeto `PointDT` deben tener como argumento al mismo, ya sea con la sintaxis `type(PointDT)`, la cual es no polimórfica, o con usando `class(PointDT)` que permite entrar con `PointDT` o cualquiera de sus extensiones como argumento.

```
module PointM

  private
  public :: PointDT, point

  type :: PointDT
     real :: x, y
   contains
     procedure :: init
  end type PointDT

  interface point
     procedure :: constructor
  end interface point
```

```
contains

  type(PointDT) function constructor(x, y)
    implicit none
    real, intent(in) :: x, y
    call constructor%init(x, y)
  end function constructor

  subroutine init(this, x, y)
    implicit none
    class(PointDT), intent(inout) :: this
    real          , intent(in)    :: x, y
    this%x = x
    this%y = y
  end subroutine init

end module PointM
```

Fragmento de código 2.1: Implementación del tipo `PointDT` para puntos bidimensionales

Abstract Derived Types

Los tipos abstractos son aquellos que deben ser extendidos antes de ser usados. En Fortran cuando un tipo es una extensión de otro, éste toma todas las variables y rutinas implementadas en el *padre*, excepto por aquellas que se quiera reimplementar.

Hacer uso de *abstract types* facilita un buen uso de poliformismo, a través de las rutinas deferidas. En el fragmento de código 2.2 se puede ver que para una clase abstracta `ShapeDT`, se define la interface de la rutina `getArea` pero su implementación quedará pendiente para las formas que extiendan a `ShapeDT`.

```
module ShapeM
  use PointM

  private
  public :: ShapeDT

  type, abstract :: ShapeDT
     type(PointDT), dimension(:), allocatable :: point
   contains
     procedure(getAreaInterf), deferred :: getArea
  end type ShapeDT

  abstract interface
     real function getAreaInterf(this)
       use PointM
       import ShapeDT
       implicit none
       class(ShapeDT), intent(inout) :: this
     end function getAreaInterf
  end interface
```

```
end module ShapeM
```

Fragmento de código 2.2: Implementación de un tipo abstract ShapeDT

En el código 2.3 se puede ver un ejemplo de una extensión de ShapeDT. En este se agrega un constructor y se implementa la rutina getArea.

```
module TriangleM
  use PointM
  use ShapeM

  private
  public :: TriangleDT, triangle

  type, extends(ShapeDT) :: TriangleDT
   contains
     procedure :: init
     procedure :: getArea
  end type TriangleDT

  interface triangle
     procedure :: constructor
  end interface triangle

  integer, parameter :: NPOINT = 3

contains

  type(TriangleDT) function constructor(point)
    implicit none
    type(PointDT), dimension(NPOINT), intent(in) :: point
    call constructor%init(point)
  end function constructor

  subroutine init(this, point)
    implicit none
    class(TriangleDT)                    , intent(inout) :: this
    type(PointDT)     , dimension(NPOINT), intent(in)    :: point
    this%point = point
  end subroutine init

  real function getArea(this)
    implicit none
    class(TriangleDT), intent(inout) :: this
    getArea = this%point(1)%x * (this%point(2)%y-this%point(3)%y) &
         +    this%point(2)%x * (this%point(3)%y-this%point(1)%y) &
         +    this%point(3)%x * (this%point(1)%y-this%point(2)%y)
  end function getArea
```

```
end module TriangleM
```

Fragmento de código 2.3: Ejemplo de una extensión de `ShapeDT` para representar triángulos

Punteros y arreglos de punteros

En Fortran es posible implementar punteros de rutinas o de tipos, ya sea tipos intrínsecos como `real` o tipos derivados como el `PointDT` que se ejemplificó anteriormente.

Surge un problema cuando uno se propone trabajar con arreglos de punteros en Fortran. Esto se debe a que la *keyword* necesaria para definir un arreglo dinámico (`allocatable`) y la que es necesaria para definir un puntero (`pointer`) no pueden encontrarse juntas en la definición de una variable. Este es un problema puramente semántico, ya que el lenguaje no tiene problemas en tener un arreglo de punteros, simplemente es imposible definirlos de manera directa.

```
module PointPtrM

  private
  public :: PointPtrDT

  type :: PointPtrDT
     class(PointDT), pointer :: ptr
  end type PointPtrDT

end module PointPtrM
```

Fragmento de código 2.4: Solución utilizada en Fortran para poder definir punteros de `PointDT`

En el fragmento de código 2.4 se encuentra la solución a este problema, la cual es utilizada extensivamente en este proyecto. Ahora en vez de declarar un arreglo de punteros de `PointDT` deberé definir un arreglo de `PointPtrDT`, en donde se encuentran estos punteros.

Una desventaja obvia es el hecho de que para acceder a las rutinas y atributos del objeto original es necesario pasar por el puntero. Por ejemplo si tengo un `PointPtrDT` llamado `point`, accederé a su valor `x` escribiendo `point%ptr%x`.

2.3.3. Patrones de diseño

Generalmente diseñar un programa consiste en la toma de varias decisiones que afectarán las características de reusabilidad, flexibilidad y extensibilidad de su código. Sin embargo hay varios problemas clásicos conocidos que aparecen durante el diseño y pueden ser resueltos fácilmente por la aplicación de *patrones de diseño* [8] ya existentes. A continuación se presentan y explican brevemente aquellos que fueron usados en el diseño del programa.

Patrón de estrategia

Define una familia de algoritmos por encapsulación de cada uno en clases separadas y los hace intercambiables por medio de una interfaz uniforme establecida en su clase base.

En esta estructura **Strategy** declara la interfaz para todas *ConcreteStrategy*. **User** tiene una referencia al objeto **Strategy** y usa la interfaz para llamar al algoritmo.

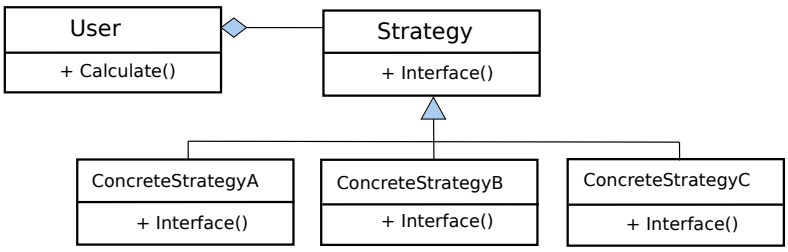


Figura 2.9: Estructura del patrón de estrategia

Existen muchos puntos en el diseño de un programa de elementos finitos donde este patrón puede ser usado. Para los métodos de solución de sistemas lineales, geometrías, elementos, condiciones, procesos, estrategias, etc.

Patrón de puente

Desacopla la abstracción y su implementación de forma que puedan cambiar independientemente. En este patrón **Abstraction** define la interfaz para el usuario y también tiene la referencia a *Implementor*. **AbstractionForm** crea un nuevo concepto y también puede extender la interfaz de **Abstraction**.

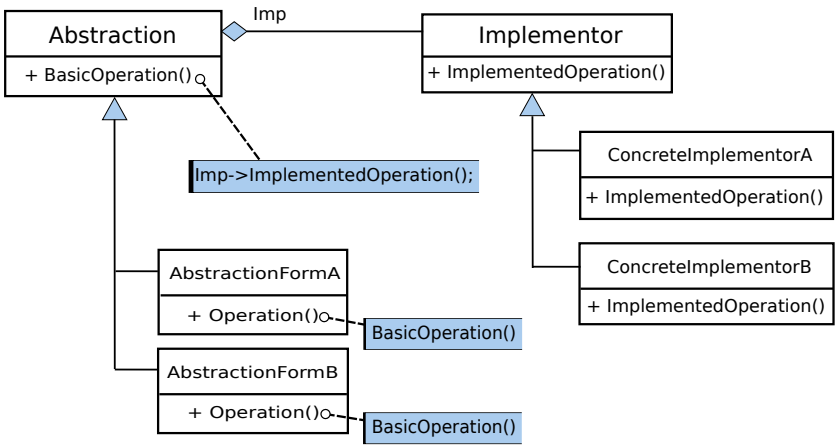


Figura 2.10: Estructura del patrón de puente

Implementor define la interfaz para la implementación que es usada por *Abstraction*. Cada *ConcreteImplementor* implementa la interfaz de implementación para un caso concreto.

Este patrón es útil para conectar conceptos con una estructura jerárquica. En un programa de elementos finitos se puede usar para conectar elementos a geometrías, los métodos de solución lineal con sus reordenadores, o conectar los métodos iterativos con los precondicionadores.

Patrón compuesto

Permite a los usuarios agrupar un conjunto de objetos en un objeto compuesto y tratar objetos individuales y composiciones de objetos de manera uniforme.

En este patrón, *Component* define la interfaz para las operaciones de objetos y para acceder y administrar los componentes secundarios. *Leaf* no tiene hijos y sólo implementa la operación, también implementa su operación mediante el uso de operaciones de sus hijos. Finalmente, *User* puede usar la interfaz de *Component* para trabajar con todos los objetos de la composición de manera uniforme.

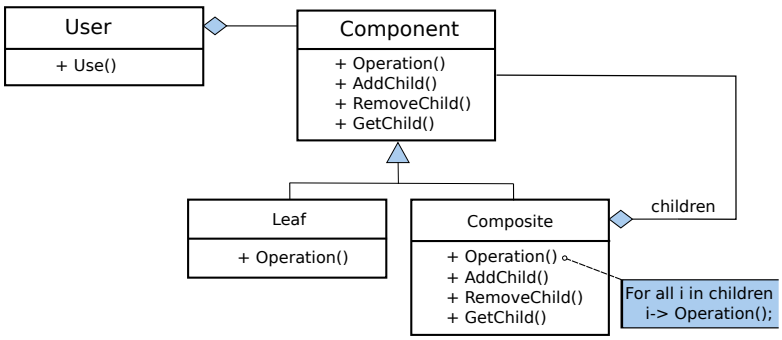


Figura 2.11: Estructura del patrón compuesto

Este patrón puede ser usado para diseñar procesos formados por un grupo de procesos, geometrías con posibilidad de agruparlas en una compuesta, o incluso elementos que agrupan diferentes elementos en uno y mezclan formulaciones.

3. Herramientas utilizadas

3.1. Compilador

Es un estándar común al comenzar un proyecto de programación al cual se le dedicará mucho tiempo y trabajo, elegir inicialmente una versión de compilador y mantenerla durante todo el desarrollo del proyecto. En principio se propuso hacer eso con Gfortran 6, no obstante luego de unos meses se encontró que ifort, el compilador de Intel, se adaptaría mejor a las necesidades del proyecto, por tener el mejor soporte a herramientas modernas de Fortran y muchas utilidades de depuración.

Actualmente el proyecto es compilado utilizando una de las versiones más actualizadas de ifort, específicamente ifort 19.1.0.166.

3.2. Pre y post procesador

Al ser una arquitectura de software de características general, sería ideal brindar soporte intrínseco para el trabajo con múltiples pre y post procesadores. Debido a las limitaciones de tiempo para desarrollar las aplicaciones se trabajo únicamente con GiD.

GiD [9] es una herramienta para pre y post proceso de datos para simulaciones numéricas en ciencia e ingeniería.

Permite como utilidades:

- Modelado geométrico (CAD)
- Generación de malla de elementos finitos
- Definición de datos del problema
- Visualización de resultados

3.3. Software de control de versión

Git [10] es un software de control de versiones, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

Parte II
Framework

4. Estructural general

En esta sección se presenta la arquitectura diseñada e implementada. Para comenzar se discuten los objetivos planteados, la metodología de diseño aplicada y las clases principales desarrolladas. Luego se detallan las herramientas básicas generadas y la implementación del método de los elementos finitos.

4.1. Requerimientos

Se buscó diseñar una *framework* para el desarrollo de aplicaciones basadas en el método de los elementos finitos. En otras palabras, esta arquitectura de software debía ofrecer las principales estructuras para el manejo de datos y algoritmos básicos para el desarrollo de *cualquier* problema basado en el método de los elementos finitos. A su vez, esto debía ser logrado brindando flexibilidad para la implementación de problemas multi-física y el agregado de nuevos algoritmos.

El programa debe presentar un alto nivel de reusabilidad. Las herramientas de elementos finitos disponibles deben ser capaces de ser utilizadas por todas las aplicaciones, buscando que la implementación de un nuevo problema de elementos finitos sea lo más simple posible.

Otro objetivo importante es presentar una buena performance y eficiencia de memoria. No obstante, es esperable que haya una pérdida de eficiencia por la naturaleza general del diseño. Se debe diseñar una arquitectura con las características mencionadas anteriormente la cual al mismo tiempo mantenga niveles de eficiencia aceptables.

Finalmente, se propone hacer el desarrollo puramente en Fortran. Hacer una demostración de las herramientas de programación moderna presentadas en éste lenguaje es otra de las finalidades del proyecto.

4.2. Uso

El uso de la framework se hace a través del desarrollo de aplicaciones. En este proyecto se implementaron una serie de aplicaciones, las cuales se presentarán en la parte III. El diseño de la framework se hizo en paralelo con las aplicaciones, ya que pensar en los requerimientos de las distintas aplicaciones permitió comprender que herramientas eran necesarias en el software base.

Con esta idea se puede diferenciar el desarrollo del proyecto entre la programación de la librería base o framework y las distintas aplicaciones.

La programación sobre la framework se debe desarrollar teniendo en mente generalidad y eficiencia, brindando la mayor cantidad de herramientas para facilitar el trabajo del desarrollador de aplicaciones.

El programador de aplicaciones utilizará las estructuras de datos que brinda la framework, agregando algoritmos y nuevas estructuras de datos según considere necesario. Es esencialmente en las aplicaciones donde se desarrollará el programa, cada una con el propósito de resolver una serie de problemas específicos.

4.3. Diseño orientado a objetos

El diseño orientado a objetos en programas de elementos finitos surge a principios de la década de 1990 [4] [6], pero incluso antes de eso ya se popularizaba un enfoque modular en el diseño. La necesidad de resolver problemas complejos y la dificulta de mantenerlos y extenderlos ha direccionado a muchos desarrolladores a adoptar una estrategia orientada a objetos.

La filosofía de la estructura orientada a objetos consiste en dividir el programa en múltiples objetos y definir sus interfaces. Hay infinitas maneras de realizar esta división, y el programa resultante dependerá intensamente de que elecciones se toman en esta fase de diseño.

En este proyecto se adopta la metodología orientada a objetos. Debido a esto muchos de los objetos serán esencialmente partes comunes de una estructura FEM, agregando objetos más abstractos necesarios para facilitar la implementación. En la figura 4.1 se muestran los objetos principales, los mismos se explican brevemente a continuación.

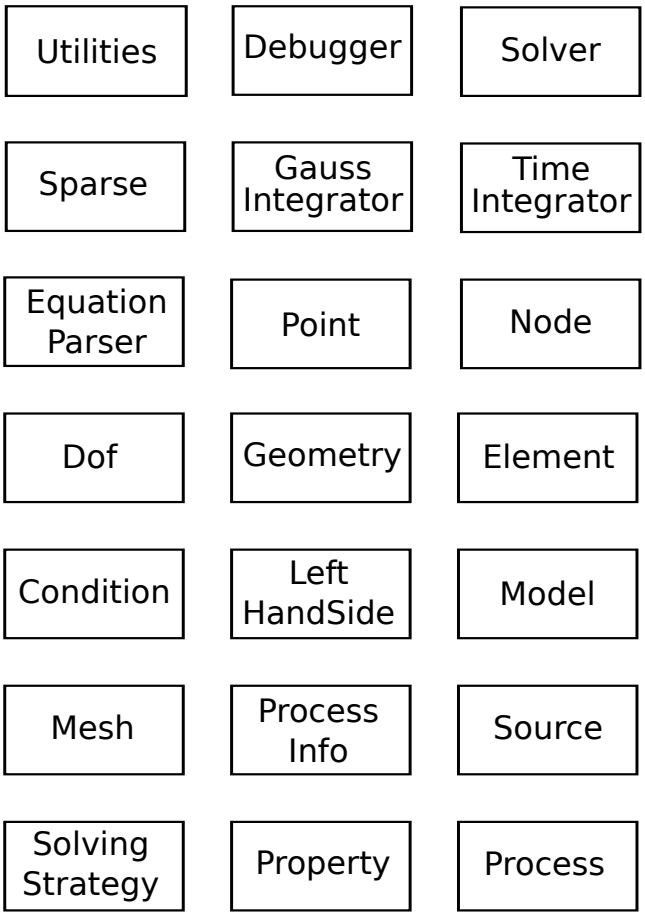


Figura 4.1: Clases principales de la framework

`Utilities` Define la precisión de las variables numéricas intrínsecas (`real`, `integer`) y define algunas rutinas útiles para el resto de los módulos.

`Debugger` Ofrece rutinas para reportar el progreso del programa en un archivo `'log.dat'`.

`Solver` Contiene una serie de métodos, directos e iterativos, para la resolución de sistemas lineales y no lineales de ecuaciones. Utiliza las rutinas de la librería MKL de Intel.

`Sparse` Permite definir una matriz sparse de formato CRS de manera eficiente. Contiene múltiples rutinas para su manipulación y operación.

`GaussIntegrator` Es una clase que permite definir los puntos de cuadratura de Gauss de cualquier geometría definida comúnmente en un problema de elementos finitos (actualmente: Línea, triángulo, cuadrilátero, tetraedro y hexaedro). Además almacena los valores de las funciones de forma y sus gradientes valuados en dichos puntos de Gauss.

TimeIntegrator Ofrece las herramientas para realizar integración numérica de ecuaciones diferenciales mediante esquemas de integración a partir de la definición del modelo físico del problema.

EquationParser Permite al usuario definir y utilizar una ecuación durante la ejecución del programa. En otros lenguajes realizar esto suele ser más simple, no obstante, en Fortran es necesario realizar una recompilación de parte del programa cada vez que se necesita definir una función. `EquationParser` encapsula este proceso y permite definir cualquier número de funciones con cualquier número de variables como miembros de una clase.

Point Define un punto en dominios uni, bi o tridimensionales.

Dof Representa un grado de libertad del problema. Contiene una variable lógica para chequear si su valor está fijado (condición de Dirichlet), en cuyo caso almacena el valor fijado. En caso de no estar fijado contiene un puntero a su valor en el vector global `dof`.

Node Clase que extiende a `Point`. A cada punto le agrega un `id` y una lista de `dofs`. Adicionalmente cada nodo puede tener una lista de `Sources`, para representar fuentes puntuales.

Geometry Define una serie de geometrías (triángulo, tetraedro, etc.) e implementa una serie de rutinas para acceder a su longitud, área o volumen, sus funciones de forma, su jacobiano, entre otras cosas. Cada geometría contiene un `GaussIntegrator` para valuar sus funciones de forma y gradientes en los puntos de integración correspondientes. Además cada geometría puede tener otra geometría como miembro, llamada `boundaryGeometry`. Así, por ejemplo, un tetraedro podrá acceder a su `BoundaryGeometry`, la cual es un triángulo. Al mismo tiempo éste triángulo podrá acceder a su `boundaryGeometry`, la cual será una línea.

Element Encapsula la formulación de un elemento finito y provee una interfaz para el cálculo de las matriz y vector locales necesarias para ensamblar el sistema global. El desarrollador de cada aplicación deberá extender esta clase implementando las rutinas diferidas `CalculateLocalSystem` y `CalculateResults`. Adicionalmente contiene una lista de punteros a sus nodos y un puntero a la geometría que representa.

Condition Permite definir condiciones de cualquier complejidad, posee una interfaz muy similar a `Element` y también implementa su funcionalidad a través de una rutina llamada `CalculateLocalSystem`.

LeftHandSide Clase desarrollada para todas las posibles matrices que pueden existir en el lado izquierdo de un sistema de ecuaciones, diferencial o algebráico.

Mesh Contiene una lista de `Nodes`, `Elements` y `Conditions` como punteros. Es decir que a través de mesh es posible acceder a todos sus componentes, aunque el objeto `mesh` no es el contenedor de estos datos, ya que estos son punteros.

ProcessInfo Es esencial para problemas que requieren integración en el tiempo, `ProcessInfo` es uno de los argumentos de las rutinas de `Element` y `Condition` donde se ensamblarán los distintos sistemas. Para el caso de problemas en los cuales es necesario volver a ensamblar en cada paso del tiempo, este objeto almacenará información importante para realizar esta operación.

Model Contiene a `Mesh` y `ProcessInfo`, además posee rutinas para inicializarlos y acceder a sus componentes.

Source Permite definir una fuente variable, la cual es esencialmente una función, que se atribuirá a cualquier `Node` o `Element`. Para que la función pueda ser definida durante la ejecución del programa cada `Source` contiene una o más instancias de `EquationParser`.

SolvingStrategy Ofrece la interfaz para que el desarrollador de cada aplicación defina el orden de las operaciones de ensamble del sistema global, la aplicación de condiciones de contorno y la resolución del sistema.

Property Permite definir un valor que se utilizará durante la ejecución del programa, éste puede ser una constante, que se definirá como `real` o un valor variable, para lo cual se utilizará `EquationParser`.

Process Presenta una interfaz para la implementación de nuevos algoritmos.

En las siguientes secciones se categorizarán las clases más importantes y se las explicará con más detalle.

5. Herramientas básicas

Un programa de elementos finitos tiene múltiples rutinas y estructuras de datos que pueden ser implementadas como herramientas básicas para ser usadas por otras partes del programa. Implementar herramientas básicas reduce el tiempo de implementación de las distintas herramientas más específicas removiendo duplicados de métodos o variables y al mismo tiempo incrementando la reusabilidad y la compatibilidad de las distintas partes del código.

En estas herramientas, por el hecho de que serán utilizadas en múltiples partes del programa, se coloca una gran importancia en eficiencia y performance.

5.1. SparseKit

Debido a la naturaleza del método, las matrices presentes en los problemas que se resolverán tienen una naturaleza *sparse*. Estas son matrices en las cuales la mayoría de los elementos serán nulos. Almacenar esos ceros implica un enorme costo de memoria innecesario, y tenerlos en cuenta en los cálculos significará un costo computacional elevado [11].

Para lidiar con este problema se desarrolló un módulo llamado `SparseKit` en el cual se define una estructura de datos llamada `Sparse`. Esta estructura representará una matriz de tipo sparse comprimida en sus filas (**CRS** según sus siglas en inglés). Para representarla esta clase posee como miembros tres listas:

- **A** La lista de valores no nulos, serán de tipo reales.
- **JA** Lista de enteros que indica a que columna le corresponde cada valor listado en A.
- **IA** Lista de enteros a los cuales se los suele llamar punteros. Cada valor de esta lista indica en que posición de A y JA comienza una nueva fila.

En adición el módulo contiene un gran variedad de rutinas para la manipulación de las matrices, estas se listan en la tabla 5.1.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`sparse`	**Interface**	PUBLIC	Interface del constructor del type `Sparse`

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`constructor`	**Type Procedure**	PRIVATE	Constructor del type `Sparse`. Sus argumentos son: Número estimado de no ceros y número de filas y columnas de la matriz.
`init`	**Type Procedure**	PRIVATE	Inicializador de los valores de `Sparse`. Rutina llamada por el constructor
`update`	**Type Procedure**	PUBLIC	Reasigna valores de `Sparse` para volver a trabajar con la misma variable
`append`	**Type Procedure**	PUBLIC	Agrega a un triplete un nuevo conjunto (valor + columna + fila), la cual se utilizará para formar la CRS
`makeCRS`	**Type Procedure**	PUBLIC	Convierte los valores introducidos por la rutina append a formato CRS
`appendPostCRS`	**Type Procedure**	PUBLIC	Permite agregar valores a la matriz sparse luego de haber llamado a la rutina `makeCRS`
`change`	**Type Procedure**	PUBLIC	Modifica un valor de la matriz luego de haber llamado a la rutina `makeCRS`
`setDirichlet`	**Type Procedure**	PUBLIC	Convierte todos los valores de una fila en cero excepto por el de la diagonal igual a uno
`handleDuplicates`	**Type Procedure**	PRIVATE	Utilizada por makeCRS para encontrar duplicados en el triplete, y sumarlos
`get`	**Type Procedure**	PUBLIC	Función para acceder a los valores de la Sparse. De la forma `A%get(i,j)`
`getnnz`	**Type Procedure**	PUBLIC	Función para acceder al valor calculado de no ceros.
`getn`	**Type Procedure**	PUBLIC	Función para acceder al orden de la matriz.
`printValue`	**Type Procedure**	PUBLIC	Imprime un valor deseado dentro de la Sparse.
`printNonZeros`	**Type Procedure**	PUBLIC	Imprime todos los no ceros de la matriz en formato de lista.
`printAll`	**Type Procedure**	PUBLIC	Imprime toda la matriz, incluyendo ceros.
`deleteRowAndCol`	**Type Procedure**	PUBLIC	Permite eliminar una fila y columna dada
`free`	**Type Procedure**	PUBLIC	Limpia memoria ocupada por la Sparse
`transpose`	**Module Procedure**	PUBLIC	Obtiene la transpuesta de una Sparse
`inverse`	**Module Procedure**	PUBLIC	Obtiene la inversa de una Sparse dada
`norm`	**Module Procedure**	PUBLIC	Obtiene la norma de Frobenius de una Sparse dada
`jacobiEigen`	**Module Procedure**	PUBLIC	Obtiene todos los autovectores y autovalores de la matriz
`trace`	**Module Procedure**	PUBLIC	Obtiene la traza de la matriz
`det`	**Module Procedure**	PUBLIC	Obtiene el determinante de la matriz
`inverse`	**Module Procedure**	PUBLIC	Obtiene la inversa de una Sparse dada
`id`	**Module Procedure**	PUBLIC	Obtiene la matriz identidad de orden n en tipo Sparse
Operador(*)	**Interface**	PUBLIC	Hace uso de las rutinas sparse_sparse_prod, sparse_vect_prod o coef_sparse_prod dependiendo de los argumentos

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`sparse_sparse_prod`	**Module Procedure**	PRIVATE	Realiza la multiplicación entre dos Sparse
`sparse_vect_prod`	**Module Procedure**	PRIVATE	Realiza la multiplicación entre una Sparse y un vector real
`coef_sparse_prod`	**Module Procedure**	PRIVATE	Realiza la multiplicación entre un scalar y una matriz sparse.
Operador(+)	**Interface**	PUBLIC	Llama a la rutina sparse_sparse_add
`sparse_sparse_add`	**Module Procedure**	PRIVATE	Realiza la suma entre dos Sparse dadas
Operador(-)	**Interface**	PUBLIC	Llama a la rutina sparse_sparse_sub
`sparse_sparse_sub`	**Module Procedure**	PRIVATE	Realiza la resta entre dos Sparse dadas

Tabla 5.1: Lista de subrutinas y funciones presentes en el módulo `SparseKit`

5.2. Integración numérica

Realizar una integración sobre el dominio o el contorno es una de las operaciones fundamentales del MEF. En cada `Element` y `Condition` se realizarán una o más integraciones numéricas. Esto pone un gran énfasis en diseñar e implementar una metodología eficiente, sin perder la generalidad deseada. Antes de discutir el diseño se hace una breve introducción a los métodos de integración numérica en programas de elementos finitos.

5.2.1. Métodos de integración numérica

Integrar una función de manera analítica no siempre es posible o sencillo. Esto generó un interés en la integración numérica, también llamada cuadratura, durante los siglos XVIII y XIX. No obstante, no fue hasta que se empezaron a desarrollar las computadoras que éstos métodos tomaron relevancia en problemas reales.

Un enfoque clásico es el de aproximar la integral valuando la función en diferentes puntos, aplicar un peso para cada uno y sumar los valores:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{5.1}$$

Muchos métodos clásicos asumen que los puntos de prueba se encuentran todos a la misma distancia h:

$$x_i = x_0 + ih \tag{5.2}$$

Se aplican diferentes puntos de prueba y diferentes coeficientes de peso para obtener el resultado. A continuación se mencionan algunos ejemplos de métodos en esta categoría:

Formulas clásicas

Un caso muy simple es la regla del trapecio. Toma dos puntos puntos de prueba, los evalúa, conecta las funciones valuadas con una línea y obtiene el área del trapecio que se encuentra debajo de esta línea:

$$\int_a^b f(x)dx = \frac{h}{2}f(a) + \frac{h}{2}f(b)\mathcal{O}(h^3 f'') \tag{5.3}$$

Se puede ver fácilmente que esta integral es exacta para funciones lineales, para otras funciones es una aproximación:

$$\int_a^b f(x)dx \approx \frac{h}{2}f(a) + \frac{h}{2}f(b) \tag{5.4}$$

De acuerdo a la ecuación (5.1) se pueden definir puntos de prueba y pesos para este método:

$$x_1 = a,\ x_2 = b \tag{5.5}$$

$$w_1 = w_2 = \frac{h}{2} \tag{5.6}$$

Las famosas reglas de Simpson también se encuentran en esta categoría. Estas utilizan mas puntos de prueba para producir una aproximación de orden mayor. A continuación se escribe la primera:

$$\int_a^b f(x)dx = \frac{h}{3}f(a) + \frac{4h}{3}f\left(\frac{a+b}{2}\right) + \frac{h}{3}f(b) + \mathcal{O}\left(h^5 f^{(4)}\right) \tag{5.7}$$

la cual es exacta para un polinomio de orden 2. Mientras que es una mejor aproximación para otras funciones que la regla del trapecio:

$$\int_a^b f(x)dx \approx \frac{h}{3}f(a) + \frac{4h}{3}f\left(\frac{a+b}{2}\right) + \frac{h}{3}f(b) \tag{5.8}$$

Además se puede introducir este método en el formato global definiendo:

$$x_1 = a,\ x_2 = \frac{a+b}{2},\ x_3 = b \tag{5.9}$$

$$w_1 = w_3 = \frac{h}{3},\ w_2 = \frac{4h}{3} \tag{5.10}$$

Es posible extender estos métodos para aproximar funciones de orden mayor utilizando más puntos de prueba. En cualquier caso, para obtener la integral exacta de un polinomio de orden n es necesario utilizar $2n+1$ puntos. Esto hace que estos métodos sean muy costosos. Es posible conseguir los mismos resultados utilizando solo n puntos, se describen esos métodos a continuación.

Cuadratura de Gauss

En los métodos mencionados anteriormente se asumía que los puntos de prueba estaban todos a una misma distancia. Si se descarta esta restricción y se elige otra manera para elegir los puntos, es posible duplicar el orden de aproximación. Esta es la idea básica de las fórmulas de cuadratura de Gauss.

Para analizar como funciona este método, primero se define un producto escalar ponderado de dos funciones f y g:

$$\langle f, g\rangle = \int_a^b W(x)f(x)g(x)dx \tag{5.11}$$

Es posible encontrar una serie de polinomios que incluyan un p_j de orden j, donde para cada $j = 1, 2, 3, ...$ sea mutuamente ortogonal sobre una función de peso $W(x)$.

Se extiende la ecuación (5.1) a un caso más general:

$$\int_a^b W(x)f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{5.12}$$

aquí $W(x)$ es agregada como una función de peso conocida aplicada a la integral. Esta extensión sirve para utilizar una característica muy poderosa de la cuadratura de Gauss, en la cual se puede obtener la integral exacta de un polinomio multiplicada por una función de peso conocida $W(x)$. Esta función de peso permite descomponer una función integrable en un polinomio y una otra parte más compleja pero también integrable. Eligiendo esta función $W(x)$ resulta en la llamada integración de *Gauss-Legendre*.

Ahora es necesario aplicar el teorema fundamental de la cuadratura de Gauss. Para encontrar los pesos se utiliza una serie de polinomios ortogonales definidos según:

$$\begin{aligned} p_0(x) &\equiv 1 \\ p_1(x) &= \left[x - \frac{\langle xp_0, p_0\rangle}{\langle p_0, p_0\rangle}\right] p_0(x) \\ p_{i+1}(x) &= \left[x - \frac{\langle xp_i, p_i\rangle}{\langle p_i, p_i\rangle}\right] p_i(x) - \frac{\langle p_i, p_i\rangle}{\langle p_{i-1}, p_{i-1}\rangle} p_{i-1}(x) \end{aligned} \tag{5.13}$$

Se puede mostrar que cada polinomio $p_j(x)$ tiene exactamente j raíces distintas y que hay exactamente una raíz de $p_{j-1}(x)$ entre cada par adyacente de ellas. Esta propiedad es muy útil a la hora de encontrar raíces. Un esquema de localización de raíces puede ser aplicado, empezando por $p_1(x)$ y continuando a ordenes mayores, utilizando intervalos de raíces previas para encontrarlas todas. Finalmente la ecuación (5.12) puede ser utilizada para formar un sistema de ecuaciones para encontrar los pesos w_i. Considerando que la ecuación (5.12) debe dar la respuesta correcta para las integrales de los primeros $N-1$ polinomios:

$$\begin{aligned} a_{ij} w_j &= b_i \\ a_{ij} &= p_{i-1}(x_j) \\ b_i &= \int_a^b W(x) p_{i-1}(x) dx \\ i &= 1, 2, ..., N;\ j = 1, 2, ..., N \end{aligned} \tag{5.14}$$

También se puede mostrar que usando los mismos pesos w_i la cuadratura es exacta para polinomios de hasta orden $2N-1$. Además en la primer ecuación de (5.14) se puede verificar que b_i para $i = 2, 3, ..., N$ es igual a cero considerando la naturaleza ortogonal de los p_i.

Hay otra manera, más práctica de encontrar los pesos w_i, utilizando otra secuencia de polinomios $\phi_i(x)$

$$\begin{aligned} \phi_0(x) &\equiv 0 \\ \phi_1(x) &= p_1' \int_a^b W(x) dx \\ \phi_{i+1}(x) &= \left[x - \frac{\langle xp_i, p_i\rangle}{\langle p_i, p_i\rangle}\right] \phi_i(x) - \frac{\langle p_i, p_i\rangle}{\langle p_{i-1}, p_{i-1}\rangle} \phi_{i-1}(x) \end{aligned} \tag{5.15}$$

donde p_1' es la derivada de $p_1(x)$ con respecto a x. Utilizando estos polinomios calcular los pesos es directo:

$$w_i = \frac{\phi_N(x_i)}{p_N'(x_i)},\ i = 1, 2, ..., N \tag{5.16}$$

Además hay una fórmula sólo para el caso de Gauss-Legendre:

$$w_i = \frac{2}{(1 - x_i^2)[p_N'(x_i)]^2} \tag{5.17}$$

Integrales multidimensionales

Pasar de una integral unidimensional a una multidimensional consiste en multiplicar el número de puntos de prueba y la complejidad de los contornos. En esta parte se discutirá un caso simple. En el MEF usualmente se transforma la geometría a unas coordenadas locales y se reduce la integral a una

dimensionalidad menor utilizando los contornos de estas coordenadas locales y considerando la suavidad de la función. Para el caso tridimensional:

$$F \equiv \int_{x_1}^{x_2} \int_{y_1(x)}^{y_2(x)} \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)dxdydz \tag{5.18}$$

Se la puede reducir a una forma bidimensional:

$$F = \int_{x_1}^{x_2} \int_{y_1(x)}^{y_2(x)} G(x,y)dxdy \tag{5.19}$$

$$G \equiv \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)dz \tag{5.20}$$

Y finalmente a una forma unidimensional:

$$F = \int_{x_1}^{x_2} H(x)dx \tag{5.21}$$

$$H(x) \equiv \int_{y_1(x)}^{y_2(x)} G(x,y)dy \tag{5.22}$$

Se puede ver que para implementar esto es necesario calcular las integrales subdimensionales en cada punto de prueba.

Finalmente, utilizando cuadratura de Gauss resulta en:

$$F = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} w_i w_j w_k f(x_i, y_j, z_k) \tag{5.23}$$

5.2.2. Implementación

Múltiples enfoques pueden ser tomados a la hora de diseñar un módulo para la integración de funciones utilizando la cuadratura de Gauss. En definitiva, como ilustra la ecuación (5.23) el problema se reduce a multiplicar una lista de coeficientes $c_{ijk} = w_i w_j w_k$ por la función valuada en el punto correspondiente (x_i, y_j, z_k), y sumar esos productos.

El primer problema a resolver es como encontrar estos puntos y pesos, teniendo en cuenta que estos varían para cada geometría. Para el caso de cuadriláteros y hexaedros, es posible hacer una extensión directa sobre los puntos de Gauss correspondientes a un dominio unidimensional. Es decir que a partir de un algoritmo para generar puntos de Gauss para una línea, obtener los puntos correspondientes de cuadriláteros y hexaedros consiste simplemente en combinar estos valores para las nuevas dimensiones. La figura 5.1 ilustra ésta idea para un caso de orden 2.

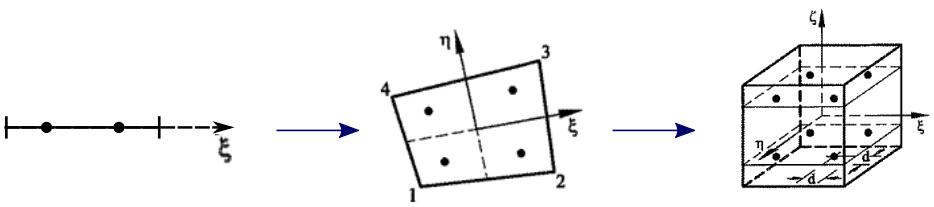


Figura 5.1: Extensión de puntos de gauss para integraciones del mismo orden en distintas dimensiones

Incrementa la dificultad para el caso de triángulos y tetraedros. Si bien es posible desarrollar algoritmos para encontrar los puntos y pesos correspondientes para cualquier orden n, resultó más conveniente escribir

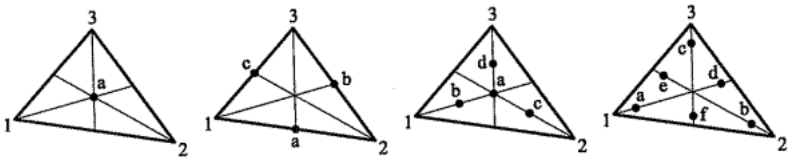


Figura 5.2: Puntos de Gauss en elementos triangulares

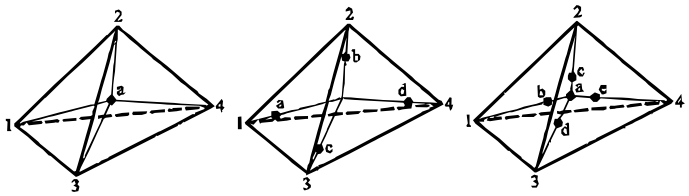


Figura 5.3: Puntos de Gauss en elementos tetraedros

a mano estos valores para $n \leq 3$ a partir de una fuente online [12]. En las figuras 5.2 y 5.3 se pueden ver los puntos de Gauss correspondientes a triángulos y tetraedros.

Se resolvió entonces generar una clase llamada `IntegratorDT`, la cual debe ser iniciada especificando el tipo de geometría que debe integrar. Dependiendo del tipo ingresado la estructura generará y almacenará las listas de pesos y puntos.

Otro problema en el diseño de éste módulo se encontraba en las dificultades de programar una rutina donde se realice la integración. Es decir una rutina que reciba el integrando como una función y devuelva el resultado. Se decidió que los inconvenientes de programar una subrutina de este tipo no valían la pena en comparación con la simple alternativa: El usuario de `IntegratorDT` podrá acceder a los valores de los pesos y los puntos de Gauss y los utilizará para realizar la integral de manera externa.

En el fragmento de código 5.1 se puede ver un ejemplo de como se utilizaría el módulo para realizar la siguiente integral:

$$I = \int_{-1}^{1} f(x)g(x)dx \tag{5.24}$$

```
program IntegrateTwoFunctions
  use UtilitiesM
  use Functions, only: f, g
  use IntegratorM
  implicit none
  integer(ikind)     :: iGauss
  real(rkind)        :: integral
  type(IntegratorDT) :: myIntegrator
  myIntegrator = integrator(2, 'line')
  integral = 0._rkind
  do iGauss = 1, myIntegrator%getIntegTerms()
     integral = integral + myIntegrator%getWeight(iGauss) &
          * f(myIntegrator%getGPoint(iGauss,1)) &
          * g(myIntegrator%getGPoint(iGauss,1))
  end do
  print'(A,E16.8)', 'Resultado integral: ', integral
end program IntegrateTwoFunctions
```

Fragmento de código 5.1: Programa que utiliza `IntegratorDT` para integrar dos funciones f y g en un dominio unidimensional

En el ejemplo se inicia el integrador para un dominio 1D ingresando el orden de integración deseado y la palabra clave *line* que describe la geometría. En el resto de las líneas se realiza la integración. Se puede ver que trabajar de esta manera resulta en códigos relativamente verbosos, pero muy claros y versátiles.

Finalmente, se notó que las funciones de forma y gradientes correspondientes a un geometría específica, por ejemplo un triángulo, toman siempre el mismo valor cuando se las valúa en los puntos de Gauss. Esto se debe a que éstas están escritas en coordenadas locales. Se identificó la necesidad de almacenar estos valores para no tener que calcularlos para cada elemento específico, así generando una carga de computación innecesaria. Se puede ver en la tabla 5.2 la lista de rutinas en el módulo de `IntegratorDT`.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`integrator`	**Interface**	PUBLIC	Interface del constructor del type `IntegratorDT`
`constructor`	**Type Procedure**	PRIVATE	Constructor del type `IntegratorDT`. Sus argumentos son: Orden de integración y tipo de geometría.
`init`	**Type Procedure**	PRIVATE	Inicializador de los valores de la clase. Rutina llamada por el constructor
`valueGPoints`	**Type Procedure**	PUBLIC	Valúa los puntos de gauss correspondientes a la geometría elegida
`getGaussOrder`	**Type Procedure**	PUBLIC	Devuelve el valor de orden de integración de Gauss
`getIntegTerms`	**Type Procedure**	PUBLIC	Devuelve el número de puntos de Gauss necesarios para una integración
`getWeight`	**Type Procedure**	PUBLIC	Devuelve uno de los pesos de integración
`getGPoint`	**Type Procedure**	PUBLIC	Devuelve una de las coordenadas de uno de los puntos de Gauss
`getShapeFunc`	**Type Procedure**	PUBLIC	Devuelve el valor de una de las funciones de forma valuada en los puntos de Gauss
`getDShapeFunc`	**Type Procedure**	PUBLIC	Devuelve el valor de una de las derivadas de las funciones de forma valuada en los puntos de Gauss
`getDDShapeFunc`	**Type Procedure**	PUBLIC	Devuelve el valor de una de las derivadas segundas de las funciones de forma valuada en los puntos de Gauss
`getWeightFull`	**Type Procedure**	PUBLIC	Devuelve todos los pesos de integración
`getGPointFull`	**Type Procedure**	PUBLIC	Devuelve todos los puntos de integracion
`getShapeFuncFull`	**Type Procedure**	PUBLIC	Devuelve todas las funciones de forma valuadas en los puntos de Gauss
`getDShapeFuncFull`	**Type Procedure**	PUBLIC	Devuelve todas las derivadas de las funciones de forma valuadas en los puntos de Gauss
`getDDShapeFuncFull`	**Type Procedure**	PUBLIC	Devuelve todas las derivadas segundas de las funciones de forma valuadas en los puntos de Gauss
`getG1D`	**Type Procedure**	PRIVATE	Encuentra los puntos y pesos de Gauss correspondientes a una línea 1D

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`getGTriangle`	**Type Procedure**	PRIVATE	Encuentra los puntos y pesos de Gauss correspondientes a un triángulo
`getGSquare`	**Type Procedure**	PRIVATE	Encuentra los puntos y pesos de Gauss correspondientes a un cuadrilátero
`getGTetrahedron`	**Type Procedure**	PRIVATE	Encuentra los puntos y pesos de Gauss correspondientes a un tetraedro
`getGHexahedron`	**Type Procedure**	PRIVATE	Encuentra los puntos y pesos de Gauss correspondientes a un hexaedro

Tabla 5.2: Lista de subrutinas y funciones presentes en el módulo `IntegratorM`

5.3. Métodos de resolución

Como ya se habló en la sección 2.2, mediante el método de los elementos finitos se construye un sistema de ecuaciones que representa el modelo y que es necesario resolver para encontrar los valores de los grados de libertad incógnita.

Estos sistemas generados pueden ser lineales o no lineales o, también, pueden requerir ser resueltos mediante integración en el tiempo. Es por esto que se crearon tres estructuras que permiten manipular los distintos métodos necesarios para cada caso.

A continuación se presentan por un lado las estructuras dedicadas a resolver sistemas de ecuaciones lineales y no lineales y por otro lado la estructura necesaria para la integración temporal. En todos los casos se utiliza la herramienta `SparseKit`, detallada en la sección 5.1, para el almacenamiento y las operaciones entre matrices.

5.3.1. Solución de sistemas lineales y no lineales

Para manejar estas tareas fueron creadas las clases `LinearSolverDT` y `NonLinearSolverDT` que funcionan como interfaz de usuario. Esto quiere decir que mediante la definición de una instancia de estos se pueden utilizar cualquiera de los métodos que se encuentren implementados o implementar uno nuevo y usarlo sin problema también a través de la interfaz.

Para el caso de los métodos no lineales la estructura propuesta posee un patrón de estrategia directo, como el de la figura 2.9, que permite encapsular distintos métodos extendiendo la clase abstracta `NonLinearSolversDT` que contiene la rutina diferida *useSolver* y utilizarlos mediante la interfaz. La tabla 5.3 muestra la lista de rutinas presentes en la clase `NonLinearSolverDT`.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`SetNonLinearSolver`	**Interface**	PUBLIC	Interfaz del constructor del type `NonLinearSolverDT`
`init`	**Type Procedure**	PUBLIC	Asigna el método no lineal a utilizar
`changeSolver`	**Type Procedure**	PUBLIC	Permite cambiar el método a usar en tiempo de ejecución
`solve`	**Type Procedure**	PUBLIC	Permite usar el método asignado

Tabla 5.3: Lista de subrutinas y funciones presentes en el módulo `NonLinearSolverM`

En el caso de los métodos lineales la estructura propuesta también se diseño bajo un patrón de estra-

tegia pero ligeramente diferente ya que en este caso para resolver sistemas lineales se dispone de métodos de solución iterativos y directos. Para esto se creó las clases abstractas `IterativeLinearSolverDT` y `DirectLinearSolverDT` las cuales contienen la rutina diferida *solveSystem* y permiten implementar los métodos iterativos y directos, respectivamente, extendiendo la clase correspondiente. Ambos tipos de métodos una vez implementados se pueden usar mediante la interfaz de métodos lineales de forma indistinta ya que utiliza rutinas genéricas. En la tabla 5.4 se puede ver la lista de rutinas que se encuentran en la clase `LinearSolverDT`.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`SetLinearSolver`	**Interface**	PUBLIC	Interfaz del constructor del type `LinearSolverDT`
`init`	**Type Procedure**	PUBLIC	Asigna el método lineal a utilizar de forma genérica para tipos iterativos y directos
`changeSolver`	**Type Procedure**	PUBLIC	Permite cambiar el método a usar
`solve`	**Type Procedure**	PUBLIC	Permite usar el método asignado

Tabla 5.4: Lista de subrutinas y funciones presentes en el módulo `LinearSolverM`

Las estructuras comentadas permiten cumplir con los objetivos de generalidad y encapsulación de los métodos necesarios para resolver los sistemas, sin embargo, hasta la fecha de escritura de este trabajo sólo se implementaron dos métodos debido a que los casos a resolver no requerían implementar nuevos. Los métodos implementados fueron: Gradiente Biconjugado [13] (Iterativo), y Pardiso de la librería MKL de intel [14] (Directo).

Además de esto también se agregó la posibilidad de implementar reordenadores y precondicionadores necesarios en algunos casos para resolver sistemas lineales. Ambas estructuras tienen configuraciones similares a las de `LinearSolverDT` y `NonLinearSolverDT`.

5.3.2. Esquemas de integración en el tiempo

La estructura propuesta para los esquemas de integración esta basada en la metodología de diseño mostrada en la referencia [7], la misma se basa en utilizar un patrón de puente como la mostrada en la figura 2.10, para generar el modelo físico del problema que permita integrarlo con los métodos implementados.

De esta estructura la clase abstracta `IntegrandDT` es la principal. Contiene el vector de estado (*state*), el paso de integración actual (*step*), punteros que permiten guardar estados anteriores para usar con los esquemas de integración multipaso (*previous*) y el esquema de integración elegido (*quadrature*).

Al extender `IntegrandDT` se deben definir las rutinas diferidas de las operaciones válidas para el modelo y la derivada del vector de estado respecto del tiempo que son utilizadas para la integración. En la tabla 5.5 se muestran las rutinas presentes en esta clase. Los tipos mostrados en color celeste son las rutinas diferidas que se deben implementar al extenderla.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`set_quadrature`	**Type Procedure**	PUBLIC	Asigna el método de integración a utilizar
`get_quadrature`	**Type Procedure**	PUBLIC	Función que devuelve el método de integración actualmente asignado
`integrate`	**Type Procedure**	PUBLIC	Llama al método de integración asignado

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`t`	Type Procedure	PUBLIC	Función que devuelve en un `IntegrandDT` la derivada del vector de estado respecto del tiempo
`add`	Module Procedure	PRIVATE	Realiza la suma de dos `IntegrandDT`
`multiply`	Module Procedure	PRIVATE	Realiza la multiplicación de una constante por un `IntegrandDT`
`assign`	Module Procedure	PRIVATE	Devuelve un `IntegrandDT`
`operator(+)`	Interface	PUBLIC	Hace uso de la rutina add
`operator(*)`	Interface	PUBLIC	Hace uso de la rutina multiply
`operator(=)`	Interface	PUBLIC	Hace uso de la rutina assign

Tabla 5.5: Lista de subrutinas y funciones presentes en el módulo `IntegrandM`

Para implementar un método de integración se debe extender la clase abstracta `BaseIntegrandDT` que posee la rutina diferida `Integrate` con el atributo *nopass* que indica que la rutina a definir no contiene un argumento ficticio de objeto pasado.

Para las aplicaciones desarrolladas hasta la fecha de escritura del trabajo se implementaron los métodos: Euler Explícito, Runge Kutta de orden 2 y de orden 4 y Adams Bashforth de orden 4.

5.4. Geometría

Resolver un problema de elementos finitos implica una discretización del dominio, lo cual implica dividir el modelo en muchas partes más pequeñas con una forma definida. En las geometrías se encapsulan estas formas, se maneja su información y se calculan sus parámetros. En esta sección se explicará el desarrollo y la implementación de geometrías en la framework.

5.4.1. Definiendo la geometría

Como ya se explicó en la sección 2.2 para resolver la forma integral es necesario definir para cada elemento una serie de funciones de forma. Es conveniente definir estas funciones en coordenadas locales. Esto significa que todas las funciones de forma correspondientes a una misma forma geométrica serán iguales, dado que están basadas en la misma geometría normalizada. Por ejemplo en la figura 5.4 se puede ver un triángulo normalizado con tres nodos.

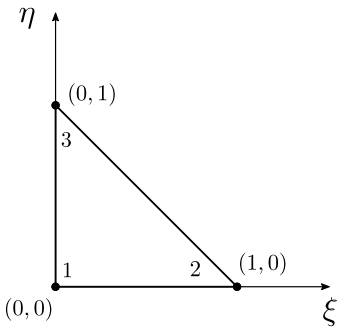


Figura 5.4: Triángulo en coordenadas naturales

Por supuesto para poder completar las integrales en el dominio real es necesario acceder a un jacobiano y el determinante de este jacobiano. Estos dependerán no solo de los coordenadas globales de cada nodo del elemento, si no que también de las derivadas de sus funciones de forma.

Se puede deducir que una estructura `GeometryDT` resultaría conveniente englobar todas las características que tendrán en común cada forma geométrica. Estas serán las funciones de forma, el jacobiano, el determinante del jacobiano y una longitud característica que dependerá de la dimensión (1D: Longitud, 2D: Área, 3D: Volumen). También resultaría conveniente contener los nodos que forman cada geometría, pero se decidió que no era eficiente. En consecuencia las rutinas requerirán como argumento que se ingrese la lista de nodos.

5.4.2. Requerimientos de `GeometryDT`

Primero que nada `GeometryDT` debe ser rápido en sus operaciones. Los elementos utilizarán esta clase para realizar calculos en bucles internos del código. La performance de las operaciones de las geometrías afectará a la performance de todo el código. También es importante que los valores que se utilicen múltiples veces sean calculados una vez y almacenados.

Otro requerimiento es que los usuarios sean capaces de combinar un elemento genérico con cualquier geometría sin tener que hacer cambios. Para lograr este objetivo es necesario que todas las geometrías posean una interfaz estándar. Hacer esto permitirá cambiar de una geometría a otra muy fácilmente.

En adición, agregar una nueva geometría debería ser posible sin modificar partes existentes del código. Este objetivo se logrará haciendo uso de una correcta estructura de clases.

5.4.3. Diseño de `GeometryDT`

Todas las geometrías específicas se definirán como clases únicas, las cuales extenderán un tipo abstracto: `GeometryDT`. Esta clase, cuyas rutinas se muestran en la tabla 5.6, definirá la interfaz que deberán tener todas las geometrías específicas que se deseen implementar. Los *type procedures* que se muestran en celeste son rutinas diferidas y deberán ser definidas cuando se extienda la clase.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`getnNode`	**Type Procedure**	PUBLIC	Devuelve la cantidad de nodos de la geometría
`getDim`	**Type Procedure**	PUBLIC	Devuelve la dimensión de la geometría
`getIntegrator`	**Type Procedure**	PUBLIC	Devuelve el `integrator` de la geometría
`getLenght`	**Type Procedure**	PUBLIC	Obtiene una magnitud característica dependiendo de la geometría (longitud, área o volumen)
`shapeFunc`	**Type Procedure**	PUBLIC	Función de forma para la geometría definida
`dShapeFunc`	**Type Procedure**	PUBLIC	Derivada de la función de forma para la geometría definida
`jacobian`	**Type Procedure**	PUBLIC	Jacobiano de la geometría
`jacobianAtGPoints`	**Type Procedure**	PUBLIC	Jacobiano de la geometría valuado en los puntos de Gauss
`jacobianDet`	**Type Procedure**	PUBLIC	Determinante del jacobiano de la geometría
`jacobianDetAtGPoints`	**Type Procedure**	PUBLIC	Determinante del jacobiano de la geometría valuado en los puntos de Gauss

Tabla 5.6: Lista de subrutinas y funciones presentes en el módulo `GeometryM`

6. Implementación de Elemento Finitos

6.1. Modelo

Cada aplicación extenderá aunque sea una vez a la clase `ModelDT`. Esta clase se propone como un medio para transferir la información de elementos finitos esencial: `MeshDT` y `ProcessInfoDT`.

Estas clases son necesarias para la comunicación entre las distintas partes del programa. Cada modelo podrá tener una o más `meshes` y un `processInfo`, y a su vez cada aplicación podrá tener uno o más `models`. Este diseño ofrece una alta flexibilidad para la implementación de problemas multidisciplina. A continuación se introducen las dos partes esenciales de un modelo.

6.1.1. `MeshDT`

Representa una malla de elementos finitos, a través de `mesh` es posible acceder a los elementos, nodos y condiciones del problema. Esto se implementa utilizando punteros, ya que los contenedores de `elements`, `nodes` y `conditions` se ubicarán en otra parte del programa.

Esta manera de acceder a los objetos de un problema de elementos finitos a través de `mesh` es necesaria para desarrollar `strategies` y `processes` que trabajen de manera general. Por ejemplo, para la implementación de una aplicación de transferencia de calor bidimensional será necesario extender `ConditionDT` para las distintas condiciones, como `FluxOnLines` y `ConvectionOnLines`. A pesar de tener dos extensiones con implementaciones completamente distintas, los procesos que utilicen estas condiciones podrán trabajarlas de manera abstracta ya que accederán a las condiciones a través de la `mesh` que las contiene.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`mesh`	**Interface**	PUBLIC	Interface del constructor del type `MeshDT`
`constructor`	**Type Procedure**	PRIVATE	Constructor del type `MeshDT`
`init`	**Type Procedure**	PRIVATE	Inicializador de los valores de la clase. Rutina llamada por el constructor
`addNode`	**Type Procedure**	PUBLIC	Agrega un nodo a la malla, con el ID especificado
`addElement`	**Type Procedure**	PUBLIC	Agrega un elemento a la malla, con el ID especificado
`addCondition`	**Type Procedure**	PUBLIC	Agrega una condición a la malla, con el ID especificado
`getnNode`	**Type Procedure**	PUBLIC	Devuelve la cantidad de nodos de la malla
`getnElement`	**Type Procedure**	PUBLIC	Devuelve la cantidad de elementos de la malla
`getnCondition`	**Type Procedure**	PUBLIC	Devuelve la cantidad de condiciones de la malla
`getID`	**Type Procedure**	PUBLIC	Devuelve la ID de la malla
`getNode`	**Type Procedure**	PUBLIC	Devuelve el nodo con la ID especificada
`getElement`	**Type Procedure**	PUBLIC	Devuelve el elemento con la ID especificada
`getCondition`	**Type Procedure**	PUBLIC	Devuelve la condición con la ID especificada
`removeNode`	**Type Procedure**	PUBLIC	Desasocia el puntero del nodo con la ID especificada
`removeElement`	**Type Procedure**	PUBLIC	Desasocia el puntero del elemento con la ID especificada

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`removeCondition`	**Type Procedure**	PUBLIC	Desasocia el puntero de la condición con la ID especificada
`free`	**Type Procedure**	PUBLIC	Libera la memoria utilizada por las listas de nodos, elementos y condiciones

Tabla 6.1: Lista de subrutinas y funciones presentes en el módulo `MeshM`

En la tabla 6.1 se puede ver la lista de subrutinas y funciones del módulo que contiene a `MeshDt`.

6.1.2. `ProcessInfoDT`

Este derived type es muy importante para el desarrollo de aplicaciones inestacionarias. Es valioso almacenar toda la información relevante al avance en el tiempo del problema en un solo objeto. El mismo podrá ser accedido por cada elemento y condición para utilizar y alterar la información según sea necesario.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`setTime`	**Type Procedure**	PUBLIC	Setea el tiempo actual del problema
`setDT`	**Type Procedure**	PUBLIC	Setea el paso de avance en el tiempo Δt.
`setMinimumDT`	**Type Procedure**	PUBLIC	Recibe un valor de Δt, lo compara con el actual y se queda con el mínimo
`setErrorTol`	**Type Procedure**	PUBLIC	Setea un valor de tolerancia para el error de la integración numérica
`setPrintStep`	**Type Procedure**	PUBLIC	Setea un valor que representa cada cuantos pasos se debe imprimir el resultado
`setT0`	**Type Procedure**	PUBLIC	Setea el valor de tiempo inicial t_0
`setStep`	**Type Procedure**	PUBLIC	Setea el paso de integración actual
`setMaxIter`	**Type Procedure**	PUBLIC	Setea el número de iteraciones máxima
`getTime`	**Type Procedure**	PUBLIC	Devuelve el tiempo actual del problema
`getDT`	**Type Procedure**	PUBLIC	Devuelve el paso de avance en el tiempo Δt.
`getErrorTol`	**Type Procedure**	PUBLIC	Devuelve la tolerancia para el error de la integración numérica
`getPrintStep`	**Type Procedure**	PUBLIC	Devuelve el valor que representa cada cuantos pasos se debe imprimir el resultado
`getT0`	**Type Procedure**	PUBLIC	Devuelve el valor de tiempo inicial t_0
`getStep`	**Type Procedure**	PUBLIC	Devuelve el paso de integración actual
`getMaxIter`	**Type Procedure**	PUBLIC	Devuelve el número de iteraciones máxima

Tabla 6.2: Lista de subrutinas y funciones presentes en el módulo `ProcessInfoM`

En la tabla 6.2 están las rutinas de la clase `ProcessInfoDT`.

6.2. Elemento

`Element` y `Condition` son los principales puntos de extensión de la framework. Es posible introducir nuevas formulaciones implementando un nuevo elemento con sus correspondientes condiciones. Esto hace que `Element` sea un objeto muy especial en el diseño.

6.2.1. Requerimientos de `ElementDT`

Una extensión de `ElementDT` es utilizada para introducir un nuevo problema. Agregar un elemento debería ser una tarea relativamente simple. Para esto es necesario proveer las herramientas para programar el ensamble de las matrices y vectores locales y los resultados.

`ElementDT` debe tener una interfaz muy flexible debido a la gran variedad de formulaciones que se pueden presentar y los distintos requerimientos que éstas tienen. Algunas formulaciones requieren recalcular todo el lado izquierdo y derecho de la ecuación en cada paso, mientras que otras requieren que se calcule el lado izquierdo una sola vez. Además, el uso de los objetos `LeftHandSide` y `ProcessInfo` como argumentos de las rutinas de ensamblaje presentan una solución para distintas dificultades que se podrían generar. Por ejemplo, algunos problemas poseen solo una matriz de rigidez en el lado izquierdo, mientras que otros posee múltiples matrices.

Una buena performance y eficiencia de memoria son puntos importantes para el diseño de `ElementDT`. Se debe tener en cuenta que las rutinas de esta clase serán llamadas múltiples veces en bucles internos del código. Cualquier ineficiencia dentro de estos objetos será un detrimento para todo el programa.

6.2.2. Diseño de `ElementDT`

Un `element` es un objeto que almacena datos y calcula las matrices y los vectores elementales para el ensamblaje global y posteriormente obtiene resultados adicionales específicos. El ensapsulamiento de estas funciones es importante para que los programas resulten claros y fáciles de mantener y extender.

La tabla 6.3 muestra la lista de subrutinas y funciones de la clase abstracta `ElementDT`. Extender esta clase será la tarea del desarrollador de las distintas aplicaciones específicas.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`assignGeometry`	**Type Procedure**	PUBLIC	Asigna la geometría dada al elemento
`assignNode`	**Type Procedure**	PUBLIC	Asigna el nodo dado al elemento
`assignSource`	**Type Procedure**	PUBLIC	Asigna la fuente dada al elemento
`getID`	**Type Procedure**	PUBLIC	Devuelve la ID del elemento
`getnNode`	**Type Procedure**	PUBLIC	Devuelve el número de nodos asignados al elemento
`getIntegrator`	**Type Procedure**	PUBLIC	Devuelve un `IntegratorPtrDT` correspondiente a la geometría del elemento
`getNode`	**Type Procedure**	PUBLIC	Devuelve el nodo `i`
`getNodeID`	**Type Procedure**	PUBLIC	Devuelve la ID del nodo `i`
`hasSource`	**Type Procedure**	PUBLIC	Devuelve una variable lógica que indica si el elemento tiene una fuente asignada o no
`calculateLocalSystem`	**Type Procedure**	PUBLIC	Calcula el `lhs` y `rhs` del sistema local del elemento
`calculateLHS`	**Type Procedure**	PUBLIC	Calcula el `lhs` del sistema local del elemento
`calculateRHS`	**Type Procedure**	PUBLIC	Calcula el `rhs` del sistema local del elemento
`calculateResults`	**Type Procedure**	PUBLIC	Calcula resultados especiales requeridos por la aplicación

Tabla 6.3: Lista de subrutinas y funciones presentes en el módulo `ElementM`

Cada elemento *contiene* un puntero hacia una geometría. Quizás un poco más perceptivo hubiera sido que cada elemento *sea* una geometría, lo cual en la práctica implica que `ElementDT` sea una extensión de `GeometryDT`. No obstante este enfoque no fue elegido ya que todos los datos que contiene una geometría

deberían ser multiplicados por el número de elementos en el problema, generando una carga de memoria innecesaria. En adición de esta forma es posible programar un solo elemento que sea capaz de trabajar con múltiples geometrías sin dificultades.

6.3. Condición

Se definirá una `Condition` para representar cualquier tipo de condición sobre el contorno, o posiblemente sobre el dominio también. Esta clase está diseñada de manera muy similar a `Element`.

6.3.1. Requerimientos de `ConditionDT`

De igual manera que con los elementos, extendiendo esta clase es como se introducen nuevas formulaciones. Es por eso que implementar una nueva condición debe ser una tarea relativamente sencilla. Para esto es necesario proveer las herramientas para programar las matrices y vectores locales.

`ConditionDT` debe tener una interfaz muy flexible por la gran variedad de formulaciones que se pueden presentar y los distintos requerimientos que éstas tienen. Es por eso que utiliza una interfaz casi equivalente a la de `ElementDT` y las rutinas deferidas `calculateLocalSystem`, `calculateLHS` y `calculateRHS` son las mismas.

Eficiencia de memoria y performance son importantes, pero no tanto como en el caso de `ElementDT`. Esto se debe a que hay menos instancias de condiciones que elementos.

6.3.2. Diseño de `ConditionDT`

`ConditionDT` es muy similar a `ElementDT`. Este debe almacenar información y calcular matrices y vectores locales que van a contribuir al ensamblaje global. Encapsular esta utilidad permite mantener una claridad en los programas que facilita el mantenimiento y la extensión de los mismos.

En la tabla 6.4 se puede ver la lista de rutinas de la clase abstracta `ConditionDT`. Para las distintas aplicaciones se extenderá esta clase. Por ejemplo para un problema de transferencia de calor en 2D posible extensiones serán `ConvectionOnLine` para definir una condición de convección sobre el contorno o `NormalFluxOnLine` para definir flujo de calor normal fijo sobre los bordes.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`assignGeometry`	**Type Procedure**	PUBLIC	Asigna la geometría dada a la condición
`assignNode`	**Type Procedure**	PUBLIC	Asigna el nodo dado a la condición
`getID`	**Type Procedure**	PUBLIC	Devuelve la ID de la condición
`getnNode`	**Type Procedure**	PUBLIC	Devuelve el número de nodos asignados a la condición
`getIntegrator`	**Type Procedure**	PUBLIC	Devuelve un `IntegratorPtrDT` correspondiente a la geometría de la condición
`getNode`	**Type Procedure**	PUBLIC	Devuelve el nodo `i`
`getNodeID`	**Type Procedure**	PUBLIC	Devuelve la ID del nodo `i`
`getAffectsLHS`	**Type Procedure**	PUBLIC	Devuelve una variable lógica que indica si la condición afecta el lado izquierdo del sistema global
`getAffectsRHS`	**Type Procedure**	PUBLIC	Devuelve una variable lógica que indica si la condición afecta el lado derecho del sistema global
`calculateLocalSystem`	**Type Procedure**	PUBLIC	Calcula el `lhs` y `rhs` del sistema local de la condición

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`calculateLHS`	Type Procedure	PUBLIC	Calcula el `lhs` del sistema local de la condición
`calculateRHS`	Type Procedure	PUBLIC	Calcula el `rhs` del sistema local de la condición

Tabla 6.4: Lista de subrutinas y funciones presentes en el módulo `ConditionM`

6.4. Estrategias de solución

En lo que respecta a la solución de los problemas, se puede decir que crear una aplicación de elementos finitos consiste en la implementación de una serie de algoritmos para resolverlos. En la práctica, cada conjunto de problemas tiene sus propios algoritmos de solución y estos resultan ser la parte principal del código por lo tanto de ellos depende la flexibilidad y robustez del mismo. Por esto se vuelve importante generar un diseño que permita manipular de forma genérica estos algoritmos en cada etapa que se necesiten.

Las estrategias de solución usualmente son distintas entre sí para distintas aplicaciones pero en general el algoritmo global es el mismo y sólo algunos pasos locales son diferentes. Por ejemplo el algoritmo global de estrategia de solución para problemas estacionarios suele ser similar para distintos casos y lo mismo pasa en problemas no estacionarios.

Teniendo en cuenta lo anterior y con el fin de dar generalidad se decidió dividir las estrategias en dos partes en las que se delegan los procesos a realizar. Para esto se crearon las clases `NewBuilderAndSolverDT` y `NewSchemeDT` con sus respectivas interfaces. La primera parte básicamente se encarga de resolver el problema mientras que la segunda se encarga de realizar los postprocesos.

Por otro lado se diseño una estructura mediante un patrón de estrategia, como el de la figura 2.9, con la clase `NewSolvingStrategyDT`, cuya lista de rutinas se puede ver en la tabla 6.5, y la clase abstracta `NewStrategy`. La primera es la interfaz que contiene una instancia de `NewBuilderAndSolverDT` y de `NewSchemeDT` y la segunda es la que usa estas instancias a través de la rutina diferida *useNewStrategy*.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`setStrategy`	Type Procedure	PUBLIC	Asigna la estrategia a utilizar
`changeStrategy`	Type Procedure	PUBLIC	Permite cambiar la estrategia a usar
`useStrategy`	Type Procedure	PUBLIC	Permite usar la estrategia elegida

Tabla 6.5: Lista de subrutinas y funciones presentes en el módulo `NewSolvingStrategyM`

6.4.1. Clase `NewBuilderAndSolverDT`

Esta clase se utiliza para realizar las operaciones de ensamble, aplicación de condiciones de contorno y solución del sistema.

La idea de juntar estas operaciones en una sola etapa mejora la posibilidad de paralelización del código, algo que para problemas estacionarios puede no tenerse en cuenta pero que para los problemas no estacionarios se vuelve un punto importante a trabajar para disminuir los tiempos de ejecución.

Para poder manipular los métodos de resolución, `NewBuilderAndSolverDT` contiene una instancia de `LinearSolverDT` y una de `NonLinearSolverDT`. De esta manera se puede seleccionar cualquiera de los métodos implementados para resolver los sistemas de ecuaciones de los problemas.

También se creó la clase `BuilderAndSolverDT` que en sí funciona como la interfaz de usuario. Esto

se hizo pensando en probar si es posible escribir configuraciones de `NewBuilderAndSolverDT` que se puedan reutilizar e intercambiar en algún caso. En la tabla 6.6 se puede ver la lista de rutinas de la clase `BuilderAndSolverDT`.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`SetBuilderAndSolver`	**Interface**	PUBLIC	Interfaz del constructor del type `BuilderAndSolverDT`
`init`	**Type Procedure**	PUBLIC	Asigna el método de ensamble y solución a utilizar
`change`	**Type Procedure**	PUBLIC	Permite cambiar el método a usar en tiempo de ejecución

Tabla 6.6: Lista de subrutinas y funciones presentes en el módulo `BuiderAndSolverM`

6.4.2. Clase `NewSchemeDT`

Simplemente encapsula las operaciones sobre la solución del sistema con las que se realizan los postprocesos. De esta forma los algoritmos no se mezclan o confunden con los del calculo principal permitiendo implementarlos y modificarlos de forma más simple.

Del mismo modo que con `BuilderAndSolverDT`, se creó la clase `SchemeDT` para que sirva de interfaz y poder probar si existe alguna generalidad que se pueda implementar a futuro. La tabla 6.7 muestra la lista de rutinas de la clase `SchemeDT`.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`SetScheme`	**Interface**	PUBLIC	Interfaz del constructor del type `SchemeDT`
`init`	**Type Procedure**	PUBLIC	Asigna el esquema a utilizar
`change`	**Type Procedure**	PUBLIC	Permite cambiar el método a usar en tiempo de ejecución

Tabla 6.7: Lista de subrutinas y funciones presentes en el módulo `SchemeM`

7. Input/Output

Es de mucha utilidad desarrollar una estructura de datos para el manejo de la entrada y salida de valores en el programa. Programar una clase de `I/O` es importante para facilitar la implementación de nuevas aplicaciones.

Por supuesto la comunicación de los programas de elementos finitos que resultarán de este proyecto será con un pre y postprocesador. Por lo que si interesa desarrollar una estructura para trabajar con estos programas es necesario definir a cuales se les brindará soporte.

No se desarrollaron clases especiales para el manejo de Input/Output en este proyecto. No obstante, al trabajar puramente con GiD, se desarrolló un módulo para facilitar la impresión de resultados independientemente del tipo de problema o su dimensión.

Parte III
Aplicaciones

Una de las finalidades de la creación de una arquitectura como la propuesta es el **Desarrollo de aplicaciones** para la resolución de problemas físicos gobernados por ecuaciones diferenciales. Es por esto que la creación de una aplicación a partir de la arquitectura desarrollada se convierte en una parte fundamental de explicar y ejemplificar en este trabajo.

A continuación se explica cómo crear una nueva aplicación a modo de guía especificando las clases a utilizar y las rutinas que es necesario implementar. Por último, se muestran como ejemplo algunas aplicaciones creadas, las implementaciones necesarias para obtenerlas y algunos resultados de distintos casos usados para verificar su correcto funcionamiento.

8. ¿Cómo crear una nueva aplicación?

Para desarrollar una aplicación, previo a empezar con el código, es fundamental tener claro para qué tipos de problemas va a estar orientada o que tipo de casos se va a querer resolver con ella. Además, se debe saber como calcular los sistemas elementales, como ensamblar el sistema global y como aplicar las condiciones de borde tanto de Dirichlet como de Neumann. Una vez obtenido el sistema global completo es necesario establecer que métodos de resolución se van a utilizar y que operaciones de postproceso se realizarán con la solución del problema.

Luego de tener claro todo lo anterior el primer paso es seleccionar una forma de entrada de información, usualmente un archivo de datos, ya sea mediante un preprocesador que lo genere o mediante la creación directa del mismo. Lo usual es seleccionar un pre y postprocesador y familiarizarse con la forma de manejo de datos para poder usarlo sin problemas. Una vez definido, se puede generar el input de la aplicación, la cual en general necesitará:

1. **Datos de la malla:**
   El número de elementos y de nodos, una lista de los nodos y sus coordenadas y una lista de los elementos y sus conectividades.

2. **Materiales:**
   Una lista de propiedades y sus valores para cada material presente, así como alguna forma de identificar los materiales que están aplicados en cada elemento si es necesario.

3. **Fuentes:**
   Una lista de fuentes que se aplican en el dominio del problema y a qué elementos o geometrías afectan.

4. **Condiciones de borde:**
   Una lista de condiciones de Dirichlet y de Neumann; y a qué elementos o geometrías afectan.

5. **Otros:**
   Cualquier otro dato específico necesario para resolver los problemas.

En lo que respecta al código de la aplicación se van a utilizar dos objetos principalmente. El primero es `ApplicationDT` el cual sirve de contenedor de las listas de nodos, elementos, condiciones, fuentes, materiales y los modelos a utilizar. Este objeto es creado para cada aplicación ya que no forma parte de la framework lo que le da mucha libertad al desarrollador para implementarlo de la forma más conveniente. El segundo es `NewSolvingStrategyDT` el cual sí esta disponible para usarlo desde la framework y es simplemente una interfaz de usuario para asignar, cambiar o usar las estrategias de resolución. En la figura

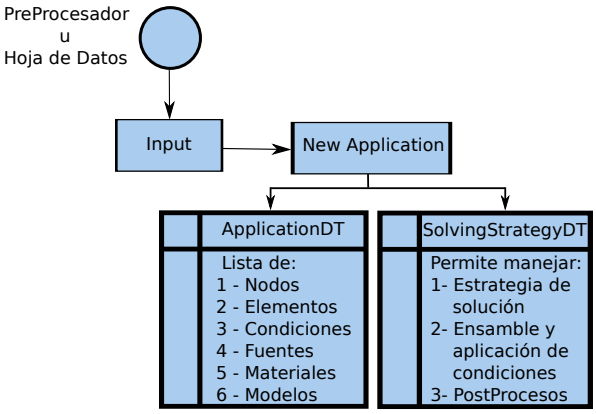


Figura 8.1: Diagrama de flujo inicial para el desarrollo de una aplicación

8.1 se muestra el diagrama de flujo de la etapa inicial de creación dejando a criterio del desarrollador la elección del preprocesador y el input a la aplicación.

En la figura 8.1 se puede ver que para `ApplicationDT` es necesario crear la clase material que tendrá como componentes las propiedades que se van a utilizar en los distintos problemas y si se desea también las rutinas de inicio y de acceso a estos valores. La clase `PropertyDT` de la framework podría utilizarse para fijar alguna propiedad mediante una función que se va a evaluar usando la librería *Fortran Parser*.

Una vez que se tiene el material se puede agregar a la clase `ElementDT` para crear el elemento que se va a usar en la aplicación con distintas geometrías que pueden ser las que se encuentran en la framework o alguna nueva que se implemente a partir de la clase abstracta `GeometryDT`. Dentro del elemento se especifica como armar el sistema elemental y como realizar los postprocesos mediante las rutinas diferidas `CalculateLocalSystem` y `CalculateResults` respectivamente. Dentro de la primer rutina están incluidos los aportes de los términos fuente al sistema.

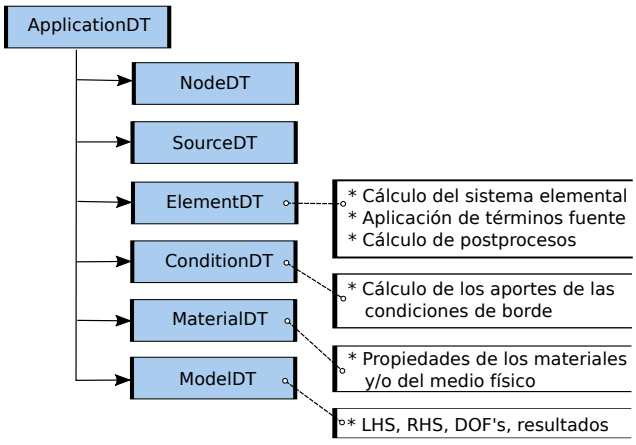


Figura 8.2: Diagrama de flujo para `ApplicationDT`

Las condiciones de borde se tratan de forma similar a los elementos pero usando la clase `ConditionDT`, esto permite que por medio de la geometría del elemento se pueda acceder a los contornos usando una lista de nodos que están afectados por la condición. Dentro de esta clase se encapsula el algoritmo de cálculo devolviendo los aportes nodales mediante las rutinas diferidas `CalculateLocalSystem`, `CalculateLHS` y `CalculateRHS`.

El modelo de la aplicación representa en conjunto la malla y el sistema global para una física en particular, esto permite separar por ejemplo en un problema acoplado Térmico-Estructural el modelo térmico del modelo estructural y de esta forma trabajarlos por separado. Se crea a partir de la clase `ModelDT` agregando el o los *RHS*, *LHS* y cualquier otra estructura de almacenamiento necesaria como por ejemplo las variables para los resultados que se van a mostrar en un PostProcesador.

En la figura 8.2 se muestran las partes que deberían formar una `NewApplicationDT` que se explicaron antes. Como se puede ver en la figura también forman parte de esta clase `NodeDT` y `SourceDT` las cuales no se describieron porque en general se pueden utilizar sin necesidad de realizar cambios en la clase básica. A través de `SourceDT` se evalúan los términos fuente mediante *Fortran Parser*, de esta forma se pueden usar funciones o constantes como fuentes. `NodeDT` es la abstracción de un nodo el cual tiene un número que lo identifica, una lista de grados de libertad o *Dof's* y la posibilidad de asignarle una fuente, a su vez por ser la extensión de un punto, posee coordenadas en el espacio. La clase `DofDT` permite fijar un valor como condición de Dirichlet ó darle un valor al momento de resolver el sistema y también asignar un nombre para poder diferenciarlo.

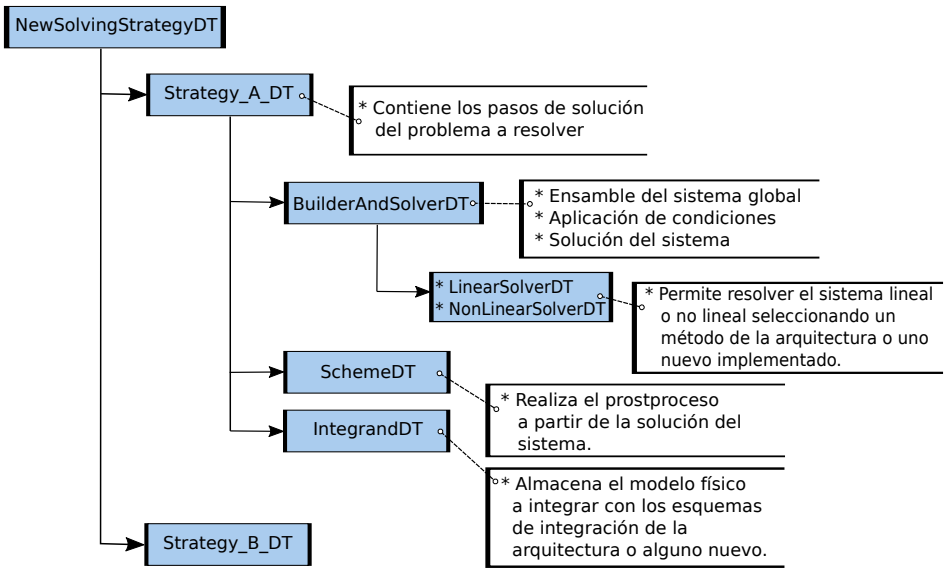


Figura 8.3: Diagrama de flujo para `NewSolvingStrategyDT`

Como ya se dijo, la clase `NewSolvingStrategyDT` es sólo una interfaz de usuario que permite usar una estrategia creada a partir de `NewStrategyDT` para resolver un problema y de ser necesario también sirve para crear una nueva interfaz pero que también permita usar las demás estrategias. Desde el punto de vista de qué pasos realizar para armar y resolver un problema existen distintas estrategias a utilizar según se trate de problemas estacionarios, no estacionarios, acoplados, etc y el objetivo de crear esta clase es poder reutilizarlas, hacerlas intercambiables entre sí y hasta combinables. En la figura 8.3 se muestra el diagrama de flujo de `NewSolvingStrategyDT`. La idea principal de la división de esta fase en distintas partes es poder optimizar cada una para obtener el mejor rendimiento global pudiendo trabajar cada una por separado.

La clase `NewStrategyDT` es desde donde se crean las distintas estrategias concretas para un tipo de problema. En general para un problema estacionario sólo va a llamar a las rutinas de `BuilderAndSolverDT` y `SchemeDT` para ensamblar el sistema, resolverlo y realizar el postproceso y, para un problema no estacionario se agrega a eso la clase `IntegrandDT` que permite el uso de los esquemas de integración de la framework.

Como se mencionó antes, la clase `BuilderAndSolverDT` contiene el ensamble del sistema y la solu-

ción por esto en general las rutinas que se encuentran ahí suelen ser `assembleSystem`, `applyNeumann`, `applyDirichlet`, `solve`, etc. Para resolver los sistemas ensamblados se pueden utilizar las clases que se encuentran en la framework `LinearSolverDT` o `NonLinearSolverDT` mediante las cuales se puede seleccionar el método para resolver un sistema lineal o no lineal respectivamente y sus precondicionadores y/o reordenadores. Los métodos implementados hasta el momento sólo han sido *Gradiente BiConjugado* y *Pardiso* de la libreria MKL.

La figura 8.4 muestra el diagrama de clases propuesto para `LinearSolverDT` y `NonLinearSolverDT`, ambos forman parte de `BuilderAndSolverDT` y son usados mediante la rutina diferida *solve* a través de una interfaz genérica. Más allá de que en la figura se puede ver las relaciones entre los métodos y los precondicionadores y reordenadores sólo fue creada la estructura necesaria para utilizarlos, no se implementó ningún método en especial para reordenar o precondicionar un sistema ya que no eran relevantes para los casos planteados.

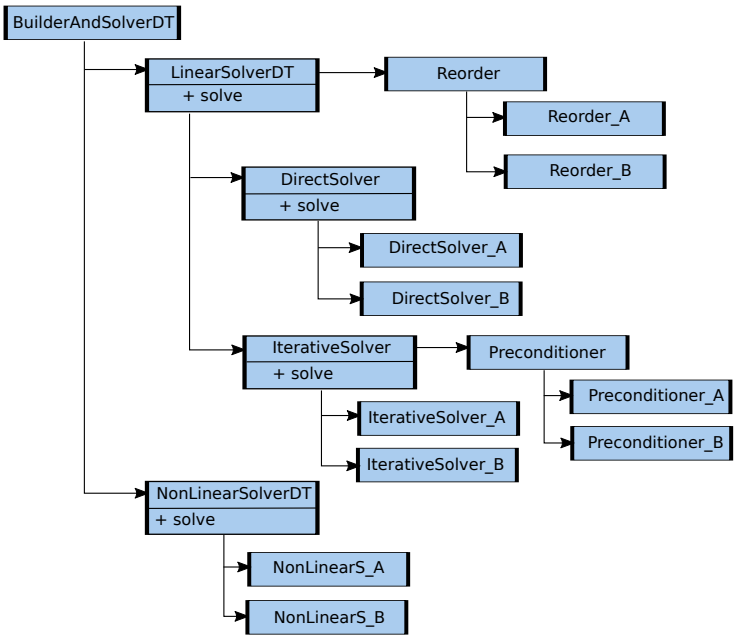


Figura 8.4: Diagrama de clases de `LinearSolverDT` y `NonLinearSolverDT`

La clase `IntegrandDT` se usa cuando se necesita utilizar un esquema de integración en el tiempo y a partir de él se genera el modelo físico a integrar. Se deben implementar las rutinas diferidas `t`, `add`, `multiply` y `assign`. La primera define la derivada respecto del tiempo del sistema y el resto definen las operaciones para poder integrarlo. En la figura 8.5 se muestra el diagrama de clases usado para los esquemas de integración aplicando un patrón de estrategia. De esta forma al extender un *ConcreteModel* se tiene acceso a los *IntegratorScheme's*, que estén implementados, a través de la rutina diferida *integrate*.

Finalmente en la clase `SchemeDT` se encuentran las rutinas que operan con el modelo del problema y generan los resultados de postproceso. Debido a que el cálculo explícito de los postprocesos está dentro del `NewElementDT` creado para una aplicación, sólo se realiza un *ensamble* de los resultados en las variables definidas para esto en el `NewModelDT`.

Hasta acá se ha desarrollado de forma un poco más profunda lo necesario para una nueva aplicación. Sin embargo es de mucha utilidad presentar ejemplos de las implementaciones que se comentaron, estos ejemplos son mostrados a continuación.

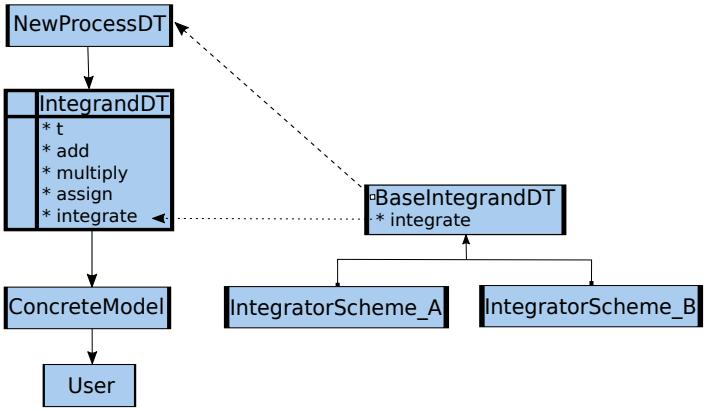


Figura 8.5: Diagrama de clases usado para los esquemas de integración temporal

9. Ejemplos

Para validar el funcionamiento de la arquitectura de software desarrollada y sus características de usabilidad, reutilizabilidad, extensibilidad, flexibilidad y mantenimiento, se implementaron hasta la fecha de escritura de este trabajo un total de siete aplicaciones, las cuales se pueden clasificar de la siguiente forma:

1. **Problemas Térmicos**

   a) **Thermal2D** Resuelve la ecuación de Poisson con condiciones de borde de Dirichlet y de Neumann para problemas de transferencia de calor en un dominio bidimensional de manera estacionaria. Presenta como resultados la distribución de temperatura y flujo de calor.

   b) **Thermal2DTransient** Idem a la aplicación anterior para resolver problemas de manera no estacionaria, pudiendo accederse a los resultados para cualquier paso de tiempo.

   c) **Thermal3D** Idem a **Thermal2D** para dominios tridimensionales.

2. **Problemas Estructurales**

   a) **Structural2D** Resuelve las ecuaciones de equilibrio de un sólido para problemas que cumplen con las hipótesis de elasticidad bidimensional de manera estacionaria. Los resultados que se dan son: el campo de desplazamientos, los campos de tensiones normales y de corte, y el campo de deformaciones.

   b) **Structural3D** Idem al anterior aplicado a dominios tridimensionales.

3. **Problemas acoplados**

   a) **ThermalStructural2D** Resuelve en tándem las ecuaciones de transferencia calor y de elasticidad lineal en dominios bidimensionales. Al cálculo del problema estructural se le agrega como deformación inicial las deformaciones que surgen como consecuencia de una diferencia de temperatura ΔT.

4. **Problemas de CFD**

   a) **CFD2D** Resuelve las ecuaciones de Euler para dominios bidimensionales y flujo compresible. Presenta como resultados el campo de velocidades y las distribuciones de presión, de mach, de temperatura, de energía interna y de densidad.

10. Problemas Térmicos: Thermal2D

Esta aplicación se diseñó para trabajar con todos los tipos de elementos bidimensionales implementados en la framework: Triángulos y cuadriláteros, lineales y cuadráticos. También se desarrolló un *Problem Type* para ejecutar los problemas de manera directa con GiD. A continuación se hablará de como se define el problema, como se estructura la aplicación y finalmente se presentarán algunos casos de verificación.

10.1. Definición del problema

El problema de transferencia de calor está dado por la ecuación de Poisson:

$$A(\phi) = \frac{\partial}{\partial x}\left(k_x \frac{\partial \phi}{\partial x}\right) + \frac{\partial}{\partial y}\left(k_y \frac{\partial \phi}{\partial y}\right) + Q = 0 \quad \text{en } \Omega \tag{10.1}$$

siendo $\phi(x, y)$ la función incógnita, en este caso temperatura, k_x y k_y son las conductividades en direcciones x e y respectivamente y $Q(x, y)$ es una fuente de calor distribuida sobre el domino Ω.

Ésta ecuación se puede obtener planteando el balance de flujos para un elemento diferencial, el cual se puede ver en la figura 10.1.

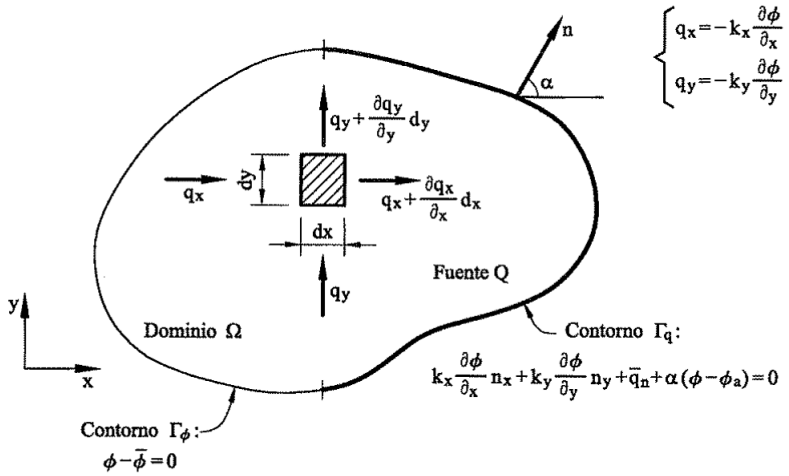


Figura 10.1: Definición del problema de Poisson en un dominio bidimensional

Las condiciones de contorno pueden ser de dos tipos:

$$B(\phi) = \begin{cases} \phi - \bar{\phi} = 0 & \text{en } \Gamma_\phi \\ q_n = \bar{q}_n + \alpha(\phi - \phi_a) & \text{en } \Gamma_q \end{cases} \tag{10.2}$$

Siendo la primera la condición de variable prescrita (Dirichlet) y la segunda la condición de flujo normal prescrito (Mixta).

El flujo normal se obtiene proyectando el flujo en el contorno normal. Así:

$$q_n = \mathbf{n}^T \mathbf{q} = n_x q_x + n_y q_y \tag{10.3}$$

Sustituyendo $q_x = -k_x \frac{\partial \phi}{\partial x}$ y $q_y = -k_y \frac{\partial \phi}{\partial y}$ en la ecuación (10.3) se obtiene la condición de flujo prescrito en el contorno en la forma usual:

$$k_x \frac{\partial \phi}{\partial x} n_x + k_y \frac{\partial \phi}{\partial x} n_y + \bar{q}_n + \alpha(\phi - \phi_a) = 0 \quad \text{en } \Gamma_q \tag{10.4}$$

Se debe tener en cuenta que:

$$\Gamma = \Gamma_\phi \cup \Gamma_q \tag{10.5}$$

La división de Γ en dos zonas Γ_ϕ y Γ_q es conceptual, ya que puede darse la situación de que estas condiciones se vayan alternando en los puntos sucesivos del contorno.

La discretización por elementos finitos se desarrolla exactamente como se explica en la sección 2.2 para resultar en el sistema:

$$\mathbf{Ka} = \mathbf{f} \tag{10.6}$$

Donde las ecuaciones para la matriz de rigidez $\mathbf{K}$ son:

$$\begin{aligned} K_{ij}^{(e)} = \int\int_{\Omega^{(e)}} \left(\frac{\partial N_i^{(e)}}{\partial x} k_x \frac{\partial N_j^{(e)}}{\partial x} + \frac{\partial N_i^{(e)}}{\partial y} k_y \frac{\partial N_j^{(e)}}{\partial y} \right) d\Omega + \\ + \int_{\Gamma_q^{(e)}} \alpha N_i^{(e)} N_j^{(e)} d\Gamma \end{aligned} \tag{10.7}$$

Y la expresión para el vector de cargas o *right hand side*:

$$f_i^{(e)} = \int\int_{\Omega^{(e)}} N_i^{(e)} Q d\Omega - \int_{\Gamma_q^{(e)}} N_i^{(e)} \left[\bar{q}_n - \alpha \phi_a \right] d\Gamma - \int_{\Gamma_q^{(e)}} N_i^{(e)} q_n d\Gamma \tag{10.8}$$

Es útil calcular los términos por separado de las matrices y el vector, según

$$K_{ij}^{(e)} = K_{d_{ij}}^{(e)} + K_{c_{ij}}^{(e)} \tag{10.9}$$

Siendo $K_{d_{ij}}^{(e)}$ la contribución de la conductividad y $K_{c_{ij}}^{(e)}$ la contribución de la convección.

$$f_i^{(e)} = f_{Q_i}^{(e)} + f_{c_i}^{(e)} + q_{n_i} \tag{10.10}$$

Donde

$$f_{Q_i}^{(e)} = \int\int_{\Omega^{(e)}} N_i^{(e)} Q d\Omega \tag{10.11}$$

es la contribución del término de fuente de calor sobre el dominio.

$$f_{c_i}^{(e)} = -\int_{\Gamma_q^{(e)}} N_i^{(e)} \left[\bar{q}_n - \alpha \phi_a \right] d\Gamma \tag{10.12}$$

es la contribución del término de flujo de calor a través del contorno de Neumann donde se prescribe el flujo saliente.

$$q_{n_i} = -\int_{\Gamma_q^{(e)}} N_i^{(e)} q_n d\Gamma \tag{10.13}$$

es el término de flujo reacción sobre el contorno de Γ_ϕ donde se prescribe el valor de ϕ. Dicho flujo se calcula "a posteriori", una vez obtenidos los valores nodales de ϕ.

10.2. Diseño de la aplicación

La aplicación se ejecutará desde el archivo `main.f90` donde se escribirán las instrucciones generales: Leer datos, ensamblar, resolver, realizar cálculos adicionales e imprimir resultados, entre otros. Aquí se declaran las instancias de las estructuras principales de la aplicación: `Thermal2DApplicationDT` y `SolvingStrategyDT`. A continuación se explican estas clases.

10.2.1. Thermal2DApplicationDT

El desarrollador de la aplicación debe elegir un objeto que funcione de contenedor para el resto de los objetos del programa. Si bien `ApplicationDT` no es parte de la framework se lo eligió como una forma de abstracción para generalizar el contenedor en las aplicaciones a crear. En este caso `Thermal2DApplicationDT` contendrá:

1. Lista de nodos (`NodeDT`)
2. Lista de elementos (`ThermalElementDT`)
3. Lista de condiciones (`FluxOnLineDT` y `ConvectionOnLineDT`)
4. Lista de fuentes de calor (`SourceDT`)
5. Lista de materiales (`ThermalMaterialDT`)
6. El modelo (`ThermalModelDT`)

Es de interés mencionar como se implementan algunas de estas clases.

ThermalMaterialDT

En todas las aplicaciones es necesario dejar fijas las propiedades relevantes al problema que se quiere resolver, para esto se crea un objeto que encapsule estas propiedades. En este caso la única propiedad necesaria es la conductividad térmica del material en dos dimensiones la cual se asume constante y en general es igual en ambas direcciones x e y. De todos modos la framework tiene disponible el uso de clase `PropertyDT` la cual permite ingresar funciones a través del uso de *Fortran Parser* en el caso de ser necesario.

```
type :: ThermalMaterialDT
   real(rkind), dimension(2) :: conductivity
 contains
   procedure :: init
end type ThermalMaterialDT
```

Fragmento de código 10.1: Declaración del type `ThermalMaterialDT`

En el fragmento de código 10.1 se muestra la declaración de la clase `ThermalMaterialDT` para la aplicación. En este caso el material sólo contiene un vector con los valores de conductividad en las direcciones x e y y la interfaz de inicio.

ThermalElementDT

Esta es la única extensión de `ElementDT` de la aplicación. En esta clase se agrega como miembro un puntero al material `ThermalMaterialDT`. Está diseñado para elementos triangulares o cuadriláteros de orden linear o cuadrático. Recordando que cada elemento tiene un puntero hacia una `GeometryDT`, es de interés tener inicializados estos cuatro tipos de geometrías para que cualquier puntero sea capaz de utilizar cualquiera de ellos. Esto se hace declarando estas geometrías como variables del módulo. En el inicializador de `ThermalElementDT`, cuando se ingresen los nodos, se decidirá a cual de estas geometrías apuntar basándose en la cantidad de nodos ingresados. Esto se puede ver en el fragmento de código 10.2.

```
subroutine init(this, id, node, material)
  implicit none
```

```
  class(ThermalElementDT)                , intent(inout) :: this
  integer(ikind)                         , intent(in)    :: id
  type(NodePtrDT)          , dimension(:), intent(in)    :: node
  class(ThermalMaterialDT), target       , intent(in)    :: material
  this%id = id
  this%node = node
  this%material => material
  if(size(node) == 3) then
     this%geometry => myTriangle2D3Node
  else if(size(node) == 4) then
     this%geometry => myQuadrilateral2D4Node
  else if(size(node) == 6) then
     this%geometry => myTriangle2D6Node
  else if(size(node) == 8) then
     this%geometry => myQuadrilateral2D8Node
  end if
  allocate(this%source(1))
end subroutine init
```

Fragmento de código 10.2: Rutina inicializadora de `ThermalElementDT`

El resto del módulo se dedica a implementar las rutinas diferidas `CalculateLocalSystem`, `CalculateLHS`, `CalculateRHS` y `CalculateResults`.

```
  do i = 1, nNode
     do j = 1, nNode
        lhs%stiffness(i,j) = 0._rkind
        do k = 1, integrator%getIntegTerms()
           bi = jacobian(k,2,2)*integrator%getDShapeFunc(k,1,i) &
                - jacobian(k,1,2)*integrator%getDShapeFunc(k,2,i)
           bj = jacobian(k,2,2)*integrator%getDShapeFunc(k,1,j) &
                - jacobian(k,1,2)*integrator%getDShapeFunc(k,2,j)
           ci = jacobian(k,1,1)*integrator%getDShapeFunc(k,2,i) &
                - jacobian(k,2,1)*integrator%getDShapeFunc(k,1,i)
           cj = jacobian(k,1,1)*integrator%getDShapeFunc(k,2,j) &
                - jacobian(k,2,1)*integrator%getDShapeFunc(k,1,j)

           lhs%stiffness(i,j) = lhs%stiffness(i,j)     &
                + integrator%getWeight(k)              &
                *(this%material%conductivity(1)*bi*bj  &
                + this%material%conductivity(2)*ci*cj) &
                / jacobianDet(k)
        end do
     end do
     if(this%node(i)%hasSource()) then
        val = this%node(i)%ptr%source(1)%evaluate &
             ((/this%node(i)%getx(), this%node(i)%gety()/))
        rhs(i) = rhs(i) + val
     end if
  end do
```

```
if(this%hasSource()) then
   allocate(valuedSource(integrator%getIntegTerms()))
   valuedSource = this%getValuedSource(integrator)
   do i = 1, nNode
      val = 0._rkind
      do j = 1, integrator%getIntegTerms()
         val = val + integrator%getWeight(j) &
              *integrator%ptr%shapeFunc(j,i) &
              *valuedSource(j)*jacobianDet(j)
      end do
      rhs(i) = rhs(i) + val
   end do
   deallocate(valuedSource)
   deallocate(jacobianDet)
end if
```

Fragmento de código 10.3: Cálculo de sistema local en `calculateLocalSystem` de `ThermalElementDT`

En el fragmento de código 10.3 se puede ver la implementación de la rutina `calculateLocalSystem`. En esta se desarrolla el ensamble de la matriz de rigidez y el vector de carga para un elemento térmico con cualquier geometría y cantidad de nodos. Se utiliza `jacobian` y `jacobianDet` obtenidos a partir de la geometría del elemento. `jacobian`, en combinación con las derivadas de las funciones de forma valúadas en los puntos de Gauss (`integrator%getDShapeFunc`), se utilizarán para calcular las derivadas de las funciones de forma con respecto a las coordenadas locales:

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} = \left[\mathbf{J}^{(e)}\right]^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \end{bmatrix} = \frac{1}{|\mathbf{J}^{(e)}|} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial y}{\partial \xi} \\ -\frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \eta} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \end{bmatrix} \tag{10.14}$$

esta expresión se puede simplificar definiendo los coeficientes b_i y c_i de manera que:

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} = \frac{1}{|\mathbf{J}^{(e)}|} \begin{bmatrix} b_i \\ c_i \end{bmatrix} \tag{10.15}$$

Estos serán los valores que se obtienen en el loop interno de integración de mostrado en el fragmento 10.3. Posteriormente a realizar la integración se chequea la presencia de `sources` en nodos o en el elemento. En el caso de tener presente una fuente se realiza la integración que contribuye al `rhs`.

FluxOnLineDT

Consiste en implementar el primer término de la ecuación (10.12). El inicializador recibirá una ID, el valor de flujo $\bar{q}_n$, la lista de nodos afectados por esta condición, y la geometría.

Al igual que `ThermalElementDT`, el resto del módulo se dedica a implementar `CalculateLocalSystem`, `CalculateLHS` y `CalculateRHS`. No obstante, como esta condición solo afecta al lado derecho, solo se debe implementar realmente esa rutina.

```
nNode = this%getnNode()
integrator = this%getIntegrator()
integrator%ptr => this%geometry%boundaryGeometry%integrator
allocate(rhs(nNode))
allocate(nodalPoints(nNode))
rhs = 0._rkind
```

```
do i = 1, nNode
   nodalPoints(i) = this%node(i)
end do
jacobianDet = this%geometry%boundaryGeometry%jacobianDetAtGPoints(nodalPoints)
do i = 1, nNode
   int = 0._rkind
   do j = 1, integrator%getIntegTerms()
      int = int + integrator%ptr%weight(j)*integrator%getShapeFunc(j,i) &
           * this%flux*jacobianDet(j)
   end do
   int = int*(-1._rkind)
   rhs(i) = rhs(i) + int
end do
```

Fragmento de código 10.4: Implementación de la integral de flujo impuesto en los bordes

En el fragmento de código 10.4 se puede ver el interior de la rutina diferida `calculateRHS` de `FluxOnLineDT`. Esencialmente el código debe realizar una integral sobre el contorno de la geometría. Para lograr esto simplemente se debe utilizar el `integrator` y `JacobianDet` del *contorno* de la geometría. Una vez obtenidos estos dos valores, se resuelve la integral como se explicó en la sección 5.2.2.

ConvectionOnLineDT

La condición de convección afecta tanto a la matriz de rigidez (segundo término de la ecuación (10.7)) y al vector de carga (segundo término de la ecuación (10.12)). La implementación es muy similar a la demostrada para `FluxOnLineDT`, con la diferencia de que se calculará tanto un `rhs` como un `lhs`.

ThermalModelDT

Extensión de `ModelDT`. Esta clase contiene las partes del sistema global (`lhs`, `rhs` y `dof`) y además almacena el resultado de postproceso `heatFlux`.

```
type, extends(modelDT) :: ThermalModelDT
   type(Sparse)                                  :: lhs
   real(rkind)     , dimension(:), allocatable :: rhs
   real(rkind)     , dimension(:), allocatable :: dof
   type(HeatFluxDT)                              :: heatFlux
   character(:)                  , allocatable :: tempDofName
 contains
   procedure, public :: init
   procedure, public :: freeSystem
end type ThermalModelDT
```

Fragmento de código 10.5: Declaración del type `ThermalModelDT`

Su declaración se puede encontrar en el fragmento de código 10.5. Se puede ver que también se almacena un character que contendrá el nombre del dof del problema. Esto se hace para aprovechar la opción de `DofDT` de asignarle a cada dof un nombre descriptivo, pero haciendo uso de punteros para no tener repetición de información.

10.2.2. `SolvingStrategyDT`

`SolvingStrategyDT` contiene a las tres clases encargadas de la organización global del problema, y su resolución: `ThermalStrategyDT`, `ThermalBuilderAndSolver` y `ThermalSchemeDT`.

`ThermalStrategyDT` simplemente se encarga de llamar a las rutinas relevantes de `ThermalSchemeDT` y `ThermalBuilderAndSolverDT`. A continuación se explican estas dos.

ThermalBuilderAndSolverDT

Esta clase contiene dos rutinas llamadas `buildAndSolve` y `solve`, la primera a su vez llama a tres rutinas: `assembleSystem`, `applyNewmann` y `applyDirichlet`. En la rutina `solve` se establece y se usa el método para resolver el sistema lineal, en este caso se utiliza la rutina *MKLPardiso*.

`assembleSystem` Se encarga de iterar sobre todos los elementos del modelo y obtener de cada uno su sistema local. Esa matriz `localLHS` y el vector `localRHS` se agregarán a la matriz y al vector global. Esto es sin duda una de las mayores ineficiencias del diseño de la framework. Este enfoque, si bien totalmente encapsulado, requiere alocatar la memoria correspondiente a todo el sistema local al elemento, para luego pasarlo componente por componente al sistema global. En un programa con eficiencia como único objetivo, cada componente se adheriría al sistema global apenas se calcula. No obstante esa técnica no podría ser encapsulada de la misma manera.

En el fragmento de código 10.6 se muestra el cuerpo de la rutina `assembleSystem`. Se puede ver que como el `LHS` del modelo es un `SparseDT` se hace uso de las rutinas `append` para agregar un nuevo valor, y `makeCRS` para establecer la matriz, sumando los repetidos, posteriormente a agregar todos los componentes.

```
nElem = model%getnElement()
model%rhs = 0._rkind
do iElem = 1, nElem
   element = model%getElement(iElem)
   nNode = element%getnNode()
   call element%calculateLocalSystem(model%processInfo, localLHS, localRHS)
   do i = 1, nNode
      row = element%getNodeID(i)
      do j = 1, nNode
         col = element%getNodeID(j)
         call model%LHS%append(val = localLHS%stiffness(i,j)  &
              , row = row                                     &
              , col = col                                     )
      end do
      model%rhs(row) = model%rhs(row) + localRHS(i)
   end do
   call localLHS%free()
   deallocate(localRHS)
end do
call model%lhs%makeCRS()
```

Fragmento de código 10.6: Implementación de la rutina `assembleSystem`

`applyNewmann` Implementa las condiciones de Newmann de flujo fijo sobre los bordes (a través de `FluxOnLineDT`) y de convección en los bordes (utilizando `ConvectionOnLineDT`). En el fragmento de

código 10.9 se puede ver que consiste en iterar sobre el número de condiciones del modelo y acceder a la condición `iCond`, la cual puede ser cualquiera de las definidas, ya que son todas extensiones de `ConditionDT`. A la condición `iCond` se le solicita su sistema local de ecuaciones, el cual según necesario se agregará al sistema global, utilizando las variables lógicas `affectsLHS` y `affectsRHS`.

```
nCond = model%getnCondition()
do iCond = 1, nCond
   condition = model%getCondition(iCond)
   nNode = condition%getnNode()
   call condition%calculateLocalSystem(model%processInfo, localLHS, localRHS)
   if(condition%getAffectsLHS() == .true.) then
      do i = 1, nNode
         row = condition%getNodeID(i)
         do j = 1, nNode
            col = condition%getNodeID(j)
            call model%LHS%appendPostCRS(val = localLHS%stiffness(i,j)  &
                 , row = row                                            &
                 , col = col                                            )
         end do
      end do
      call localLHS%free()
   end if
   if(condition%getAffectsRHS() == .true.) then
      do i = 1, nNode
         row = condition%getNodeID(i)
         model%rhs(row) = model%rhs(row) + localRHS(i)
      end do
      deallocate(localRHS)
   end if
end do
```

Fragmento de código 10.7: Implementación de la rutina `applyNewmann`

`applyDirichlet` Aplica las condiciones de Dirichlet, que en este caso será para temperatura fija sobre los nodos. En esta aplicación cada nodo tendrá un solo dof: La temperatura. Es por eso que la implementación del fragmento de código 10.8 resulta ser la más simple y eficiente. En esta se accederá al `dof(1)` de cada nodo del modelo, revisando si ha sido fijado su valor. En caso de estar fijado, se hace uso de una rutina de `SparseDT` llamada `setDirichlet` para poder anular todos los valores de la fila correspondiente a ese grado de libertad, excepto por el valor de la diagonal, al cual se lo valúa en 1. En el `rhs` se coloca el valor fijado.

En nuestro proyecto se optó por una implementación como la del fragmento 10.7, identificando el dof por su nombre. Este algoritmo es muy similar, solo que admite la presencia de múltiples dofs en un nodo, buscando en todos ellos el que posea el nombre 'TEMPERATURE'. Esto se elegió así teniendo en mente el desarrollo de una aplicación de acople térmico-estructural estacionaria, en la cual iba a ser posible utilizar estas rutinas para el ensamble de la parte térmica, pero ahora los nodos tendrán tres grados de libertad (temperatura y dos desplazamientos).

```
nNode = model%getnNode()
do i = 1, nNode
```

```
    node = model%getNode(i)
    if(node%ptr%dof(1)%isFixed) then
        nodeID = node%ptr%getID()
       call model%lhs%setDirichlet(nodeID)
       model%rhs(nodeID) = node%ptr%dof(1)%fixedVal
    end if
 end do
```

Fragmento de código 10.8: Implementación de la rutina `applyDirichlet` utilizando numeración del dof

```
nNode = model%getnNode()
do i = 1, nNode
   node = model%getNode(i)
   do j = 1, node%getnDof()
      if(node%ptr%dof(j)%getName() == 'TEMPERATURE') then
          if(node%ptr%dof(j)%isFixed) then
             nodeID = node%ptr%getID()
            call model%lhs%setDirichlet(nodeID)
            model%rhs(nodeID) = node%ptr%dof(j)%fixedVal
         end if
       end if
    end do
end do
```

Fragmento de código 10.9: Implementación de la rutina `applyDirichlet` utilizando nombres de dofs

ThermalSchemeDT

Esta clase posee una única rutina llamada `calculateFlux`. Esta esencialmente se encarga de iterar por elemento invocando la rutina `calculateResults` e interpretar el output `resultMat`, asignando cada componente correspondiente al objeto que almacena los resultados: `heatFlux`.

10.3. Tests

A continuación se detalla la ejecución y los resultados para dos problemas de transferencia de calor en dominios bidimensionales. Como pre y post procesador se utiliza GiD.

10.3.1. Transferencia de calor con convección en una placa rectangular

Este ejemplo esta basado en el benchmark T4 extraído de la referencia [15]. El modelo es un rectángulo de $0{,}60[m] \times 1{,}00[m]$ de espesor uniforme, como se ve en la figura 10.2. El borde AB tiene una temperatura prescrita de $100°C$, el borde AD esta completamente aislado. Hay convección con una temperatura de referencia de $0°C$, a lo largo de los bordes BC y CD con un coeficiente de conducción superficial $K = 750\left[\frac{W}{m^{2}°C}\right]$. No hay generación de calor interna. La conductividad del material es $k = 52\left[\frac{W}{m^{2}°C}\right]$. El valor fijado de temperatura en el punto E es $18{,}3°C$.

Luego de generar la geometría y seleccionar el *Problem Type* **Thermal2D**, se deben cargar las condiciones de contorno según se muestra en la figura 10.2. Se selecciona el material, en este caso no hay fuentes generadoras de calor por lo cual no se modifica en *Problem Data*. El problema se resuelve con un orden de integración de cuadratura de gauss $n = 3$.

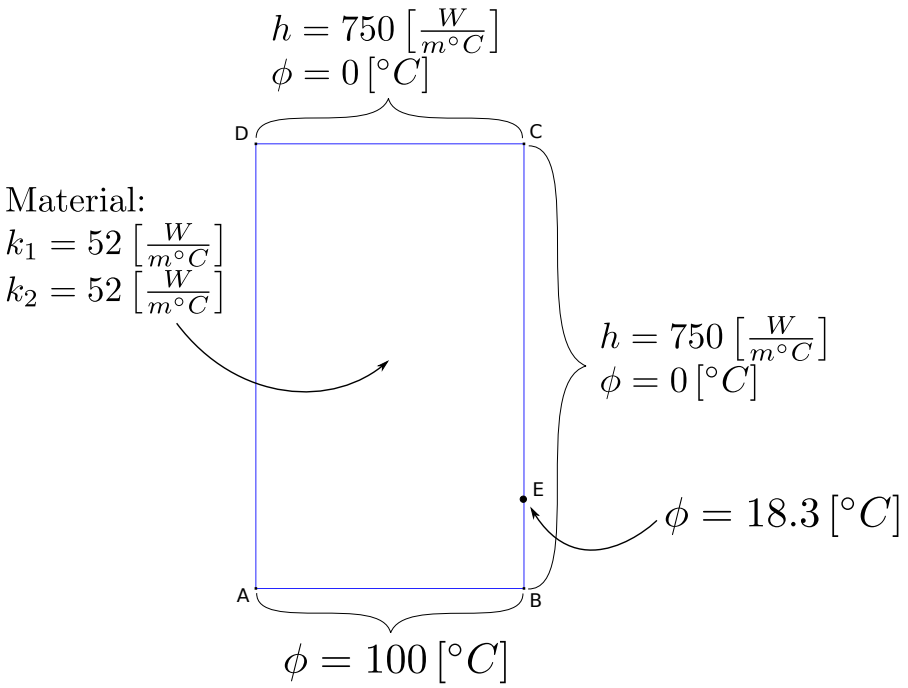


Figura 10.2: Geometría del problema y condiciones de contorno

Mallado

Una vez cargadas las condiciones, materiales, fuentes y modificado el orden de integración y la cantidad de fuentes presentes en el problema, se debe generar la malla de elementos finitos para resolver. Es importante notar que cada vez que se modifique alguna condición, se debe volver a generar la malla.

Para este caso, se decide resolver con tres tipos de malla distintos, como se muestra en la figura 10.3, para poder observar las diferencias que se presenten en las distintas soluciones.

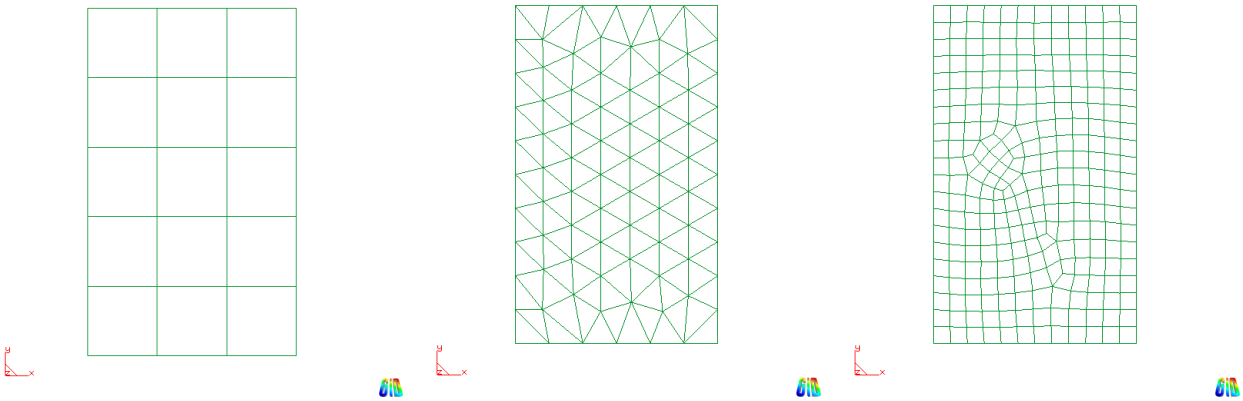
- Malla 1: 15 elementos cuadriláteros lineales estructurados.
- Malla 2: 132 elementos triangulares lineales no estructurados.
- Malla 3: 260 elementos cuadriláteros cuadráticos no estructurados.

Distribución de temperatura

Después de generada la malla, para resolver, se debe ejecutar la opción *Calcular*. Una vez terminada la ejecución, se va al post-proceso, e inmediatamente se abrirá una ventana con información general de la ejecución del mismo. Si no hubo ningún problema en la ejecución aparecerá en la ventana la hora de finalización de la ejecución.

En las figuras 10.4, 10.5 y 10.6 se muestran los resultados de las corridas para los tres tipos de malla. Como era de esperarse, la solución tiene un comportamiento más suave a medida que se aumenta la cantidad de elementos. En la solución de 15 elementos, las curvas tienen cambios de pendiente abruptas debido al tamaño de los elementos, contrario a lo que ocurre para 260 elementos cuadráticos en los cuales se observa curvas casi completamente suaves.

Los tiempos de ejecución en los tres casos resultan comparables y no significativos. Es importante notar que los resultados con 15 elementos brindan una solución numéricamente no muy alejada del caso con más elementos. Esto sirve para mostrar que en algunos casos no es necesario aumentar significativamente la



(a) Malla de 15 elementos y 24 nodos, con elementos lineales estructurados

(b) Malla de 132 elementos y 83 nodos, con elementos no estructurados lineales

(c) Malla de 260 elementos y 845 nodos, con elementos no estructurados cuadráticos

Figura 10.3: Mallas seleccionadas para resolver el ejercicio

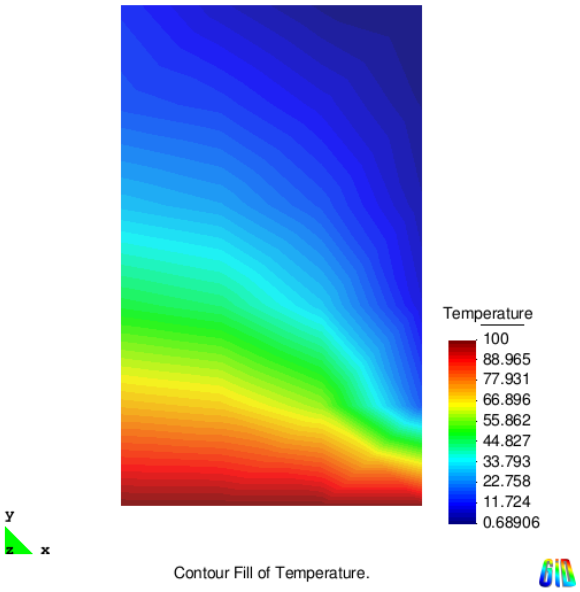


Figura 10.4: Temperaturas para la malla de cuadriláteros estructurados

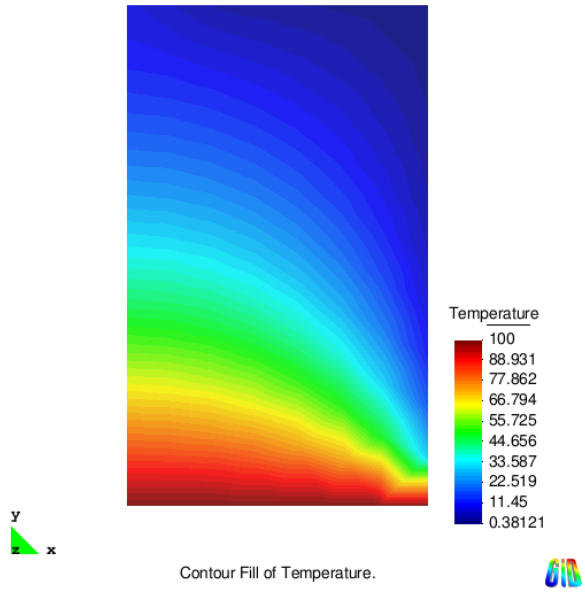


Figura 10.5: Temperaturas para la malla de triángulos

cantidad de elementos, lo cual puede seguramente aumentar mucho el tiempo de ejecución, para obtener una solución aceptable.

Flujo de calor

El flujo de calor en los problemas térmicos es un post-proceso realizado a partir de la distribución de temperatura obtenida. Debido a que el flujo de calor es la derivada de la distribución de temperatura, mientras más puntos conozcamos en el dominio, mejor va a ser la solución observada. Los resultados son muy similares para los tres mallados elegidos, por lo que para mostrar los resultados con mayor claridad

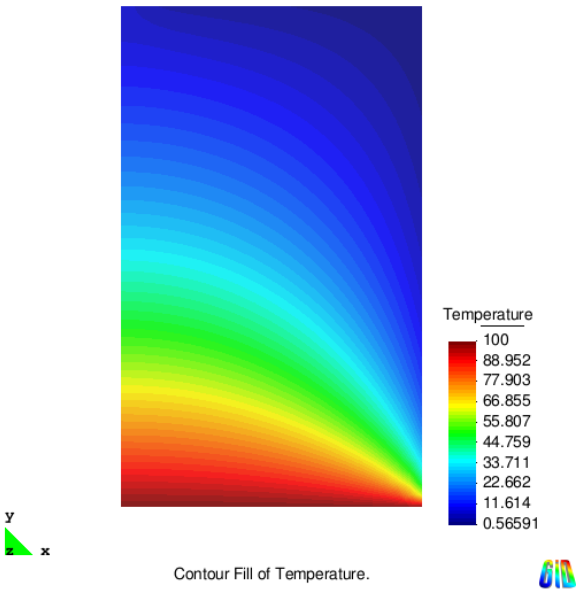


Figura 10.6: Temperaturas para la malla de cuadriláteros no estructurados

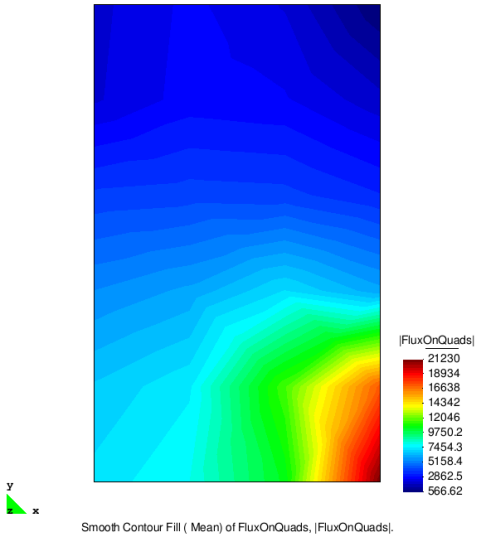


Figura 10.7: *Smooth Contour fill* del flujo de calor para el primer mallado

solo se ejemplificó uno de ellos. En la figura 10.7 se puede ver la distribución de las magnitudes del flujo de calor para el mallado de cuadriláteros estructurados. En la figura 10.8 se muestran las direcciones de esos flujos.

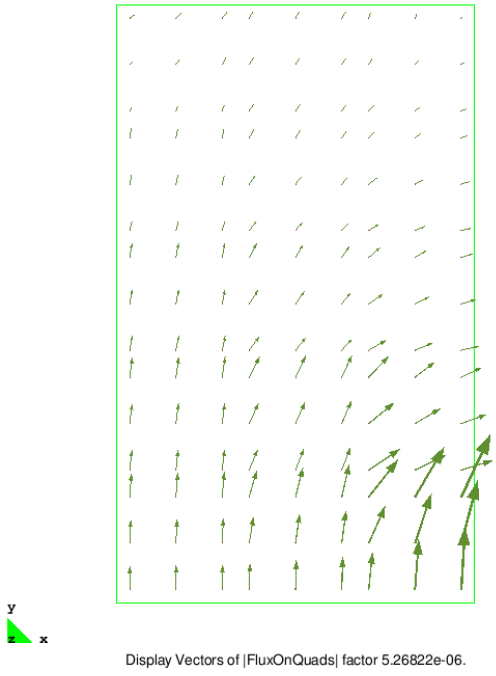


Figura 10.8: Dirección del flujo de calor para el primer mallado

10.3.2. Calle con cables calentadores (fuente de calor puntual)

Este ejemplo está basado en el *case study* de la referencia [16]. Aquí se plantea una situación real en la cual, para calentar una calle, se sitúan una serie de cables como se puede ver en la figura 10.9. Como muestra la figura, se hace uso de una sección representativa de $2cm$ de ancho, dado que los cables están separados $4cm$.

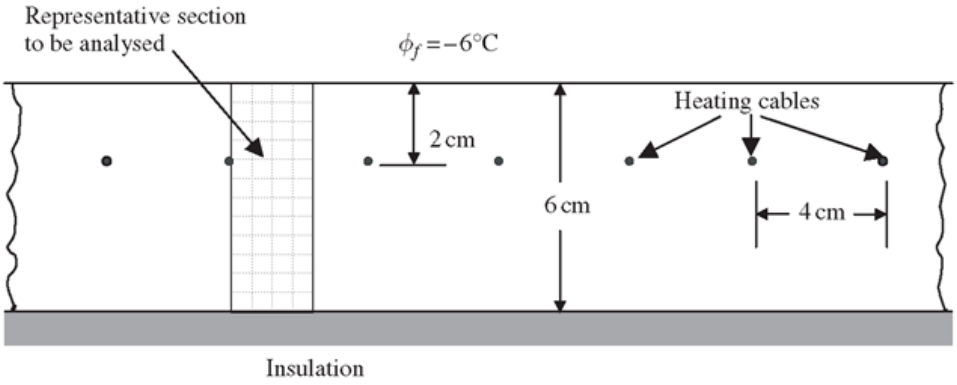


Figura 10.9: Sección transversal de la calle con los cables calentadores

Condiciones de contorno

Como muestra la figura 10.10 las condición de contorno del problema es la de convección sobre la línea que representa la calle y se coloca una fuente puntual sobre el punto que representa al cable.

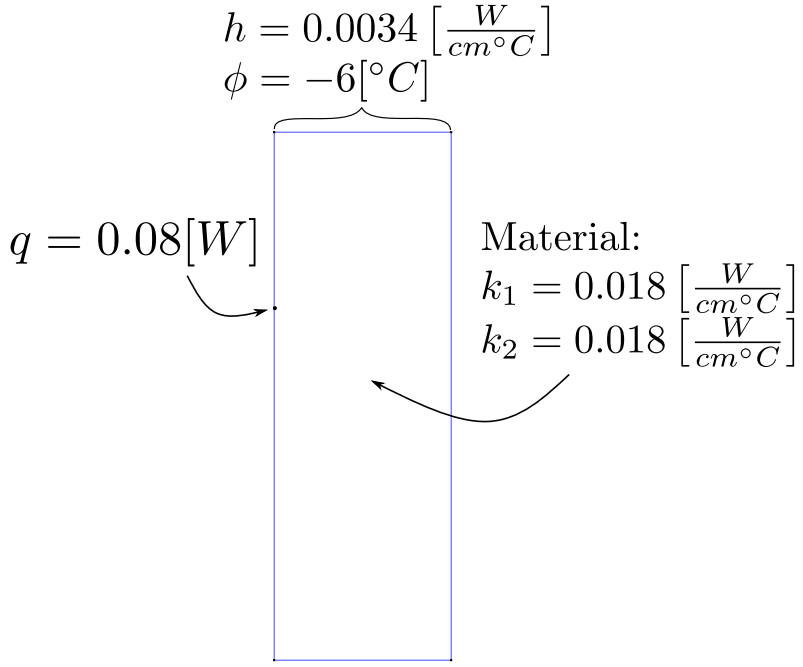
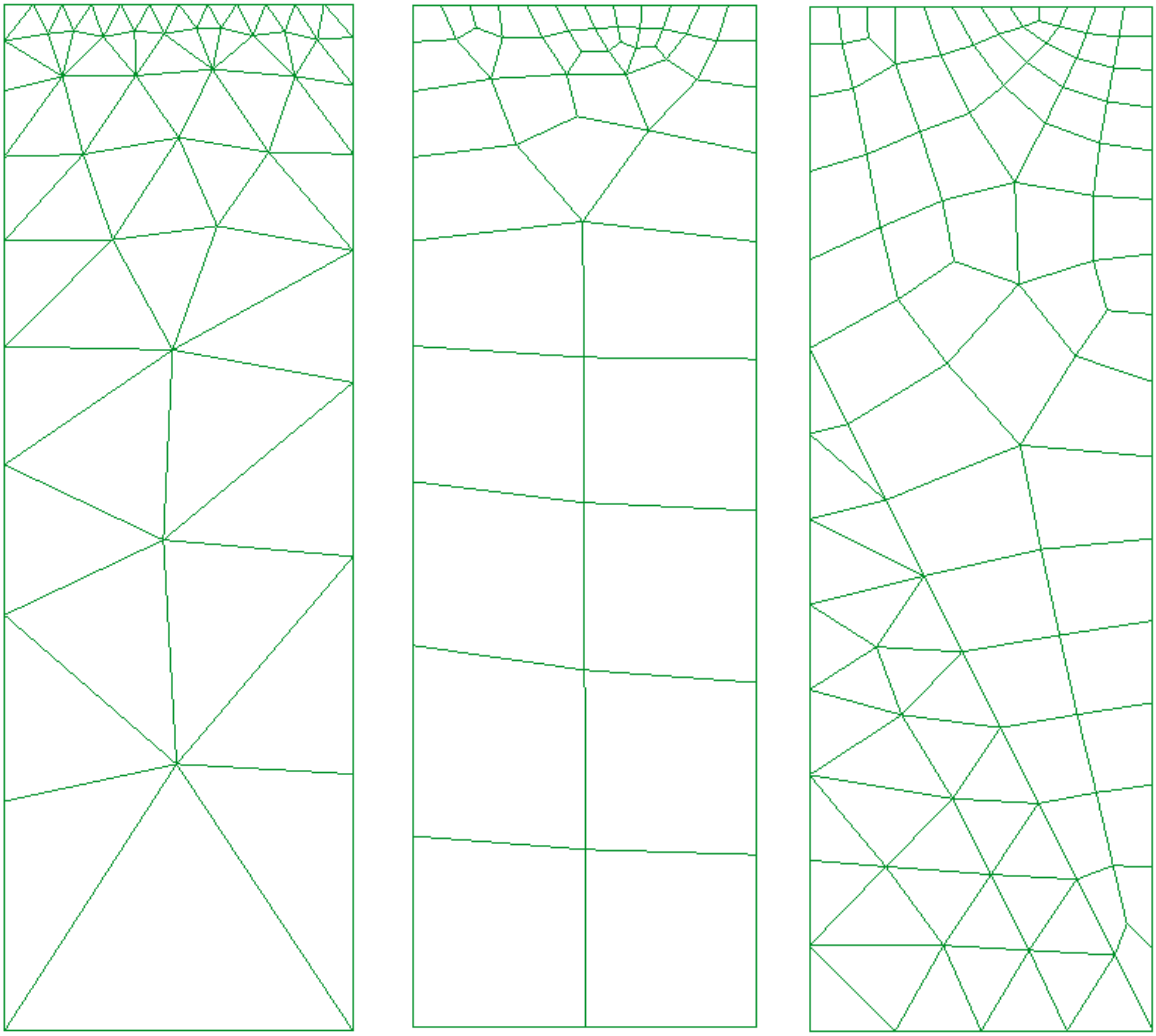


Figura 10.10: Condiciones y material del problema

Para setear correctamente este programa en GiD se debe acceder a Problem Data y colocar que el número de fuentes puntuales es igual a 1.

Mallado

Se eligió para poner a prueba este programa una serie de mallas no muy refinadas de todos los tipos disponibles, también se eligió combinar elementos triangulares y cuadriláteros en el dominio. Esto se realizó como una prueba, y no porque presente una ventaja en particular. Las tres mallas elegidas se pueden ver en la figura 10.11.



a) Mallado de 72 triángulos

b) Mallado de 44 cuadriláteros

c) Mallado de 33 triángulos y 62 cuadriláteros

Figura 10.11: Mallas utilizadas para resolver el problema

Resultados

Interesa analizar los valores en la línea superior del rectángulo, la cual corresponde a la calle. En el *case study* de la referencia se propone que la fuente de calor debería ser tal que la temperatura sobre la calzada será mayor a $0°C$. Para comparar el resultado se toman 5 puntos equiespaciados sobre la calle (figura 10.12).

La presencia de esos puntos agregará fineza en el mallado en la zona cercana a la calle, como se puede

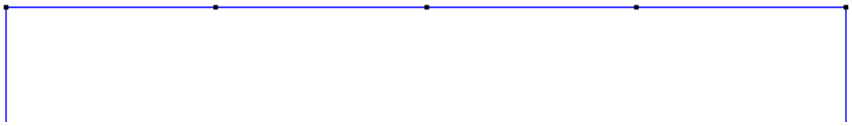


Figura 10.12: 5 puntos que se tomaron para comparar resultados

ver en la figura 10.11.

Con las condiciones dadas y el mallado realizado se procedió a correr el programa para los siguientes casos:

- 72 triángulos lineales
- 72 triángulos cuadráticos
- 44 cuadriláteros lineales
- 44 cuadriláteros cuadráticos
- 33 triángulos cuadráticos y 62 cuadriláteros cuadráticos

Todos los casos se pueden ver en la figuras 10.13, 10.14, 10.15, 10.16 y 10.17.

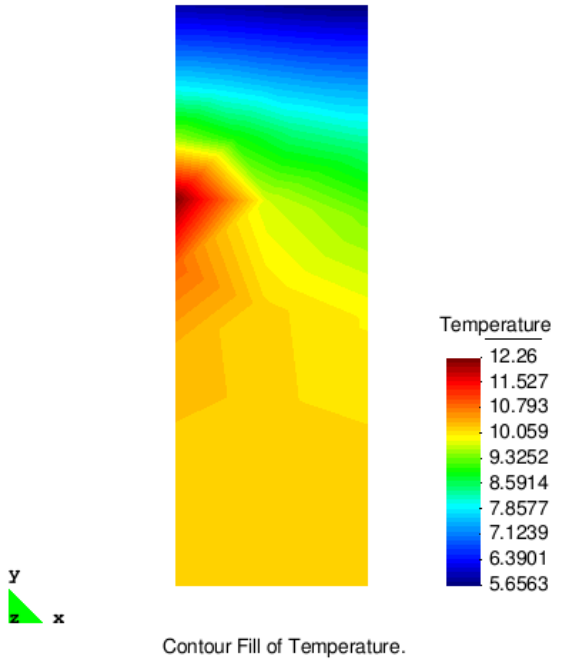


Figura 10.13: Temperaturas para triángulos lineales

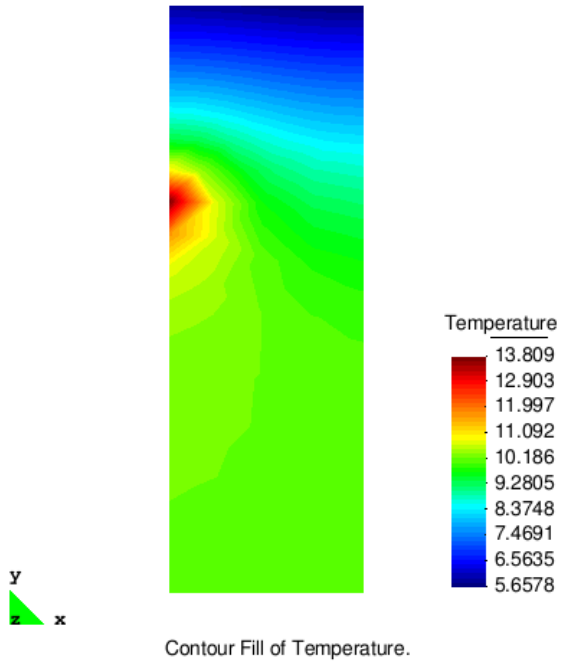


Figura 10.14: Temperaturas para triángulos cuadráticos

Como se puede ver todas las discretizaciones comparten la misma 'forma' y alcanzan resultados similares.

Es importante tener en cuenta que la fuente de calor usada, del tipo puntual, no es algo que existe en la realidad. No obstante, como se puede ver, es una herramienta muy útil para analizar casos de este tipo, pero hay que considerar que los valores de temperatura en la proximidad absoluta de la fuente no serán confiables.

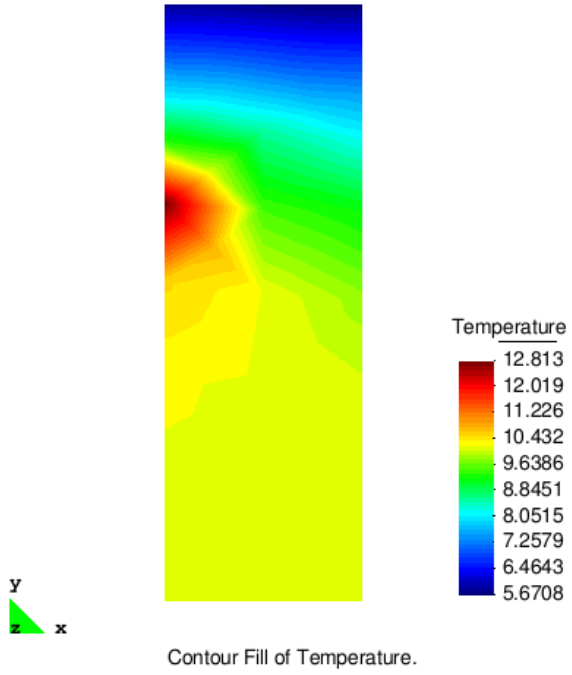


Figura 10.15: Temperaturas para cuadriláteros lineales

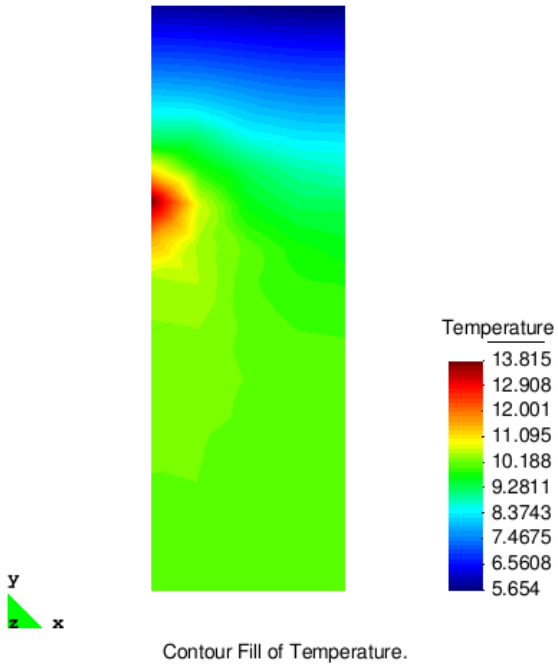


Figura 10.16: Temperaturas para cuadriláteros cuadráticos

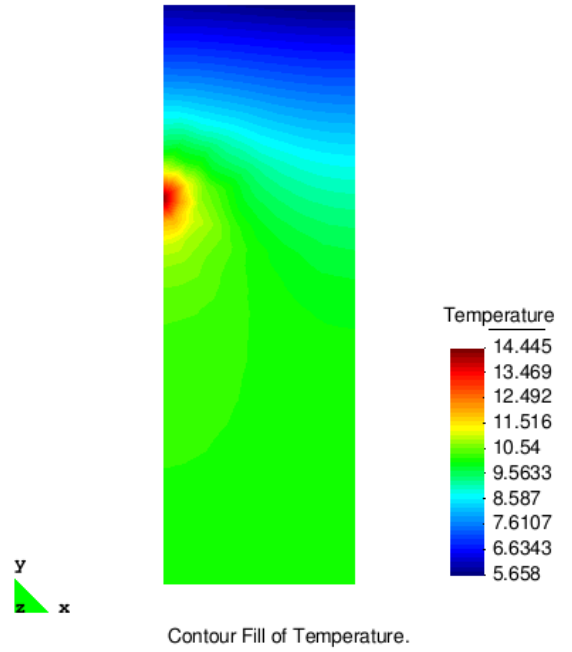


Figura 10.17: Temperaturas para triángulos y cuadriláteros cuadráticos

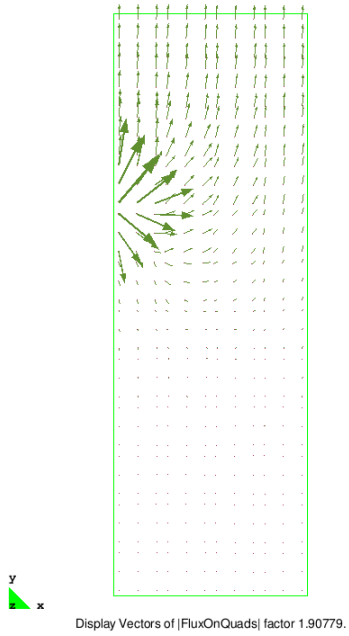


Figura 10.18: Flujo de calor para la malla con cuadriláteros

Es de interés observar el comportamiento del flujo de calor, se eligió el caso de la malla con cuadriláteros para representar el resultado (figura 10.18), pero todas las mallas devolvieron un resultado similar. Como se puede ver la corriente de calor sale de la fuente de calor y se va dispersando por el material, yéndose la mayor cantidad de calor por la superficie de la calle, como era de esperar.

En la siguiente subsección se realizará una comparación en detalle de los puntos de interés del caso.

Comparación de resultados

Utilizando los puntos de la figura 10.12 se registró el resultado para cada malla, el cual se puede ver en la tabla 10.1.

ElementType	nElem	nNode	T_1	T_2	T_3	T_4	T_5
TriangLin	72	52	5.8804	5.8414	5.7615	5.6879	5.6563
TriangQuad	72	175	5.8764	5.8419	5.7623	5.6875	5.6578
QuadLin	44	61	5.8581	5.8327	5.7655	5.6955	5.6708
QuadQuad	44	165	5.8806	5.8453	5.7619	5.6841	5.6540
HybridQuad	33 & 62	298	5.8760	5.8418	5.7622	5.6876	5.6580
HybridLin(Fuente)	6 & 32	49	5.8610	5.8320	5.7640	5.6970	5.6690

Tabla 10.1: Comparación en los 5 puntos de interés para los distintos mallados

Es notable que la diferencia entre los valores para todos los mallados no supera el $1\,\%$. Lo que es más, se puede deducir, analizando la gráfica de la figura 10.19, que posiblemente el mallado de la referencia [16] sea menos preciso que los utilizados en ésta evaluación, aunque sea para el caso de los cuadráticos.

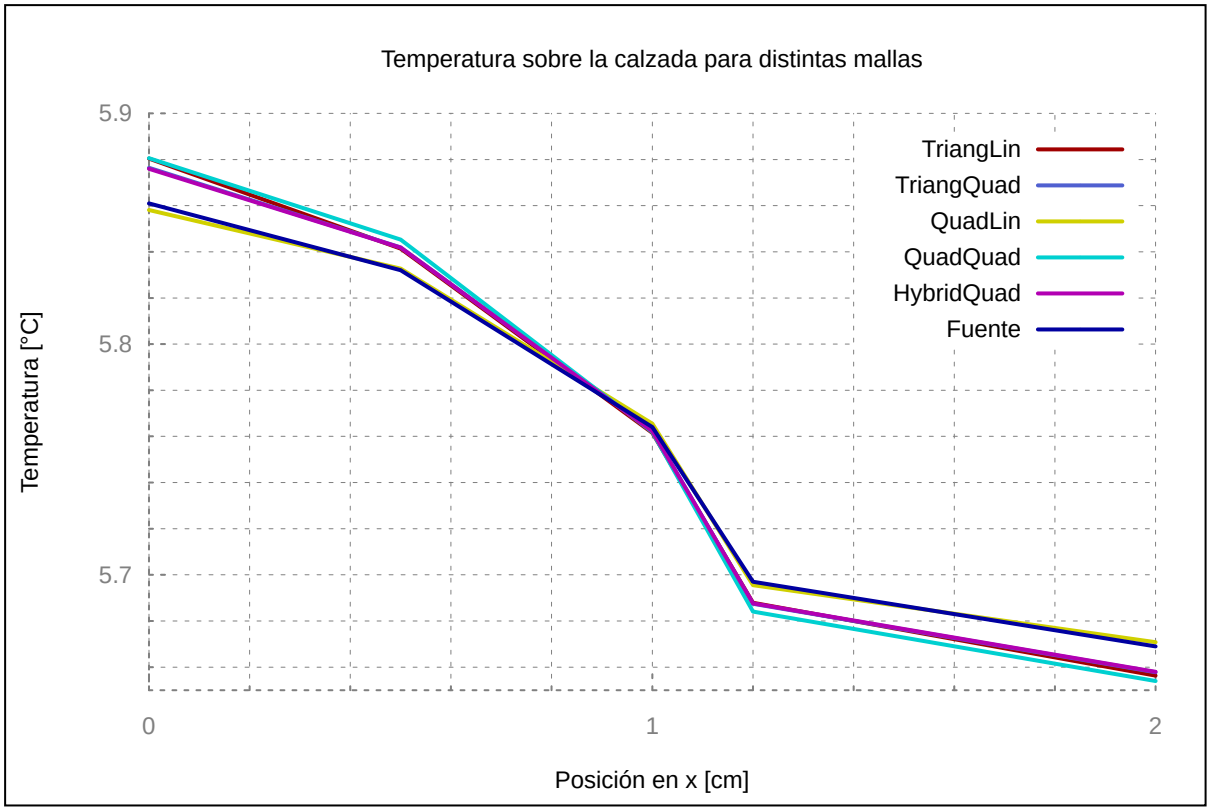


Figura 10.19: Comparación de los resultados dados por las distintas mallas en los puntos de la calzada

11. Problemas Estructurales: Structural2D

Esta aplicación fue desarrollada para resolver problemas de elasticidad bidimensional de manera estacionaria. De igual manera que con **Thermal2D** se buscó que trabaje con todos los tipos de elementos

bidimensionales soportados por la framework y también se desarrolló un *Problem Type* para trabajar con GiD. A continuación se definirá el problema específico a resolver y se mencionará como se implementó.

11.1. Definición del problema

La forma diferencial para problemas de elasticidad bidimensional está dada por:

Ecuaciones de equilibrio

$$\left.\begin{array}{rl} \frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + b_x & = 0 \\ \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + b_y & = 0 \end{array}\right\} \text{ en } \Omega \tag{11.1}$$

donde b_x y b_y son las fuerzas másicas según las direcciones x e y, respectivamente (figura 11.1).

Condiciones de contorno

Contorno donde las fuerzas están prescritas:

$$\left.\begin{array}{l} \sigma_x n_x + \tau_{xy} n_y = \bar{t}_x \\ \tau_{xy} n_x + \sigma_y n_y = \bar{t}_y \end{array}\right\} \text{ en } \Gamma_t \tag{11.2}$$

donde n_x y n_y son los cosenos directores y $\bar{t}_x$ y $\bar{t}_y$ las fuerzas de superficie que actúan sobre el contorno Γ_t.

Contorno donde las variables están prescritas:

$$\mathbf{u} = \bar{\mathbf{u}} \quad \text{en } \Gamma_u \tag{11.3}$$

Siendo $\mathbf{u}$ en este problema el vector de grados de libertad:

$$\mathbf{u}(x,y) = \begin{bmatrix} u(x,y) \\ v(x,y) \end{bmatrix} \tag{11.4}$$

La discretización por elementos finitos se desarrolla según lo explicado en la sección 2.2 para resultar en el sistema:

$$\mathbf{K}\mathbf{a} = \mathbf{f} \tag{11.5}$$

con

$$\mathbf{f}^{(e)} = \mathbf{f}_{\epsilon_0}^{(e)} + \mathbf{f}_{\sigma_0}^{(e)} + \mathbf{f}_b^{(e)} + \mathbf{f}_t^{(e)} \tag{11.6}$$

En (11.5):

$$\mathbf{K}^{(e)} = \int\int_{\Omega^{(e)}} \left[\mathbf{B}^{(e)}\right]^T \mathbf{D}\mathbf{B}^{(e)}\, d\Omega \tag{11.7}$$

es la matriz de rigidez del elemento. Definido de manera vectorial, teniendo en cuenta que la matriz $\mathbf{B}$ es la de las derivadas de las funciones de forma:

$$\mathbf{B} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \begin{bmatrix} N_1^{(e)} & 0 & N_2^{(e)} & 0 & \cdots & N_n^{(e)} & 0 \\ 0 & N_1^{(e)} & 0 & N_2^{(e)} & \cdots & 0 & N_n^{(e)} \end{bmatrix} \tag{11.8}$$

y siendo $\mathbf{D}$ la matriz de las propiedades del material, la cual depende del tipo de problema:

Tensión plana

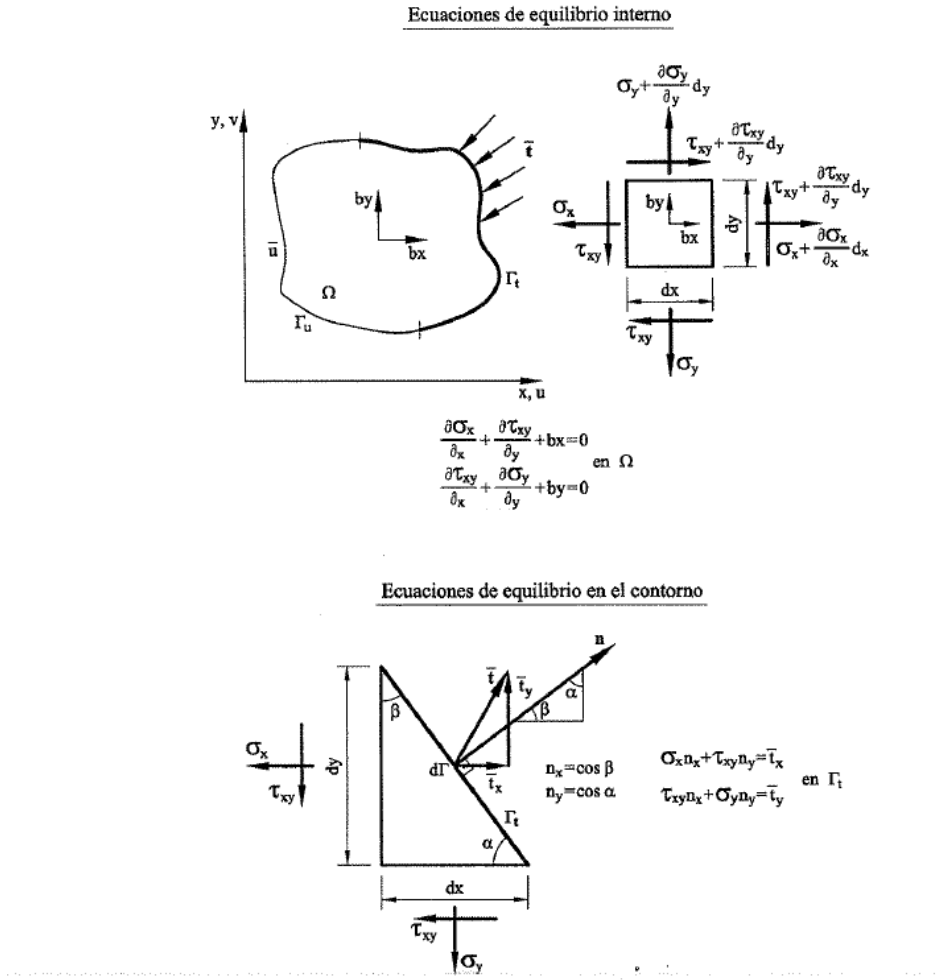


Figura 11.1: Equilibrio interno y en el contorno de un sólido bidimensional

$$\mathbf{D} = \frac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \tag{11.9}$$

Deformación plana

$$\mathbf{D} = \frac{E}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \tag{11.10}$$

De los vectores de carga:

$$\mathbf{f}_{\epsilon_0}^{(e)} = \int\int_{\Omega^{(e)}} \left[\mathbf{B}^{(e)}\right]^T \mathbf{D}\boldsymbol{\epsilon}_0^{(e)}\, d\Omega \tag{11.11}$$

es el vector de las fuerzas nodales equivalentes del elementos debido a las deformaciones iniciales.

$$\mathbf{f}_{\sigma_0}^{(e)} = \int\int_{\Omega^{(e)}} \left[\mathbf{B}^{(e)}\right]^T \boldsymbol{\sigma}_0\, d\Omega \tag{11.12}$$

es el vector de las fuerzas nodales equivalentes del elemento debido a las tensiones iniciales, el cual no se implementó en el programa.

$$\mathbf{f}_b^{(e)} = \int \int_{\Omega^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{b}\, d\Omega \tag{11.13}$$

es el vector de las fuerzas nodales equivalentes del elemento debido a las fuerzas másicas.

$$\mathbf{f}_t^{(e)} = \int_{\Gamma_t^{(e)}} \left[\mathbf{N}^{(e)}\right]^T \mathbf{t}\, d\Gamma \tag{11.14}$$

es el vector de las fuerzas nodales equivalentes del elemento debido a las fuerzas de superficie.

11.2. Diseño de la aplicación

Al igual que **Thermal2D** la aplicación se ejecutará desde el archivo `main.f90` donde se encuentran las instrucciones generales y se declaran las instancias de las estructuras principales, en este caso: `Structural2DApplicationDT` y `SolvingStrategyDT`.

11.2.1. Structural2DApplicationDT

En este caso contiene:

1. Lista de nodos (`NodeDT`)
2. Lista de elementos (`StructuralElementDT`)
3. Lista de condiciones (`PressureDT`)
4. Lista de fuentes (`SourceDT`)
5. Lista de materiales (`StructuralMaterialDT`)
6. El modelo (`StructuralModelDT`)

Las implementaciones relevantes para esta aplicación son:

StructuralMaterialDT

Para los problemas de elasticidad bidimensional se propone que el módulo de Young, el coeficiente de Poisson, el coeficiente de expansión térmica, el área, el espesor y los elementos de la matriz D de las ecuaciones 11.9 y 11.10 sean parte de esta clase.

```
type :: StructuralMaterialDT
   real(rkind) :: young
   real(rkind) :: poissonCoef
   real(rkind) :: thermalCoef
   real(rkind) :: area
   real(rkind) :: thickness
   real(rkind) :: d11, d12, d21, d22, d33
 contains
   procedure :: init
end type StructuralMaterialDT
```

Fragmento de código 11.1: Declaración del type `StructuralMaterialDT`

En el fragmento de código 11.1 se muestra la declaración de la clase `StructuralMaterialDT` donde se pueden ver las propiedades que la componen.

StructuralElementDT

Es la única extensión de `ElementDT` de la aplicación. En esta clase se agrega como miembro nuevamente el tipo de material `StructuralMaterialDT`. Está diseñado para elementos triangulares o cuadriláteros de orden linear o cuadrático. La rutina de inicialización es igual que la implementada en **Thermal2D**.

El resto del módulo se dedica a implementar las rutinas diferidas `CalculateLocalSystem`, `CalculateLHS`, `CalculateRHS` y `CalculateResults`.

```
nNode = this%getnNode()
nDof = this%node(1)%getnDof()
integrator = this%getIntegrator()
lhs = leftHandSide(0, 0, nNode*nDof)
allocate(rhs(nNode*nDof))
allocate(nodalPoints(nNode))
rhs = 0._rkind
do i = 1, nNode
   nodalPoints(i) = this%node(i)
end do
jacobian = this%geometry%jacobianAtGPoints(nodalPoints)
jacobianDet = this%geometry%jacobianDetAtGPoints(jacobian)
do i = 1, nNode
   do j = 1, nNode
      ii = nDof*i-1
      jj = nDof*j-1
      lhs%stiffness(ii,jj)     = 0._rkind
      lhs%stiffness(ii+1,jj)   = 0._rkind
      lhs%stiffness(ii,jj+1)   = 0._rkind
      lhs%stiffness(ii+1,jj+1) = 0._rkind
      do k = 1, integrator%getIntegTerms()
         bi = jacobian(k,2,2)*integrator%getDShapeFunc(k,1,i) &
              - jacobian(k,1,2)*integrator%getDShapeFunc(k,2,i)
         bj = jacobian(k,2,2)*integrator%getDShapeFunc(k,1,j) &
              - jacobian(k,1,2)*integrator%getDShapeFunc(k,2,j)
         ci = jacobian(k,1,1)*integrator%getDShapeFunc(k,2,i) &
              - jacobian(k,2,1)*integrator%getDShapeFunc(k,1,i)
         cj = jacobian(k,1,1)*integrator%getDShapeFunc(k,2,j) &
              - jacobian(k,2,1)*integrator%getDShapeFunc(k,1,j)

         Kij(1,1) = bi*bj*this%material%d11 + ci*cj*this%material%d33
         Kij(1,2) = bi*cj*this%material%d12 + bj*ci*this%material%d33
         Kij(2,1) = ci*bj*this%material%d21 + bi*cj*this%material%d33
         Kij(2,2) = bi*bj*this%material%d33 + ci*cj*this%material%d22

         lhs%stiffness(ii,jj)     = &
```

```
                lhs%stiffness(ii,jj)     + integrator%getWeight(k)&
                *Kij(1,1)/jacobianDet(k)
            lhs%stiffness(ii,jj+1)   = &
                lhs%stiffness(ii,jj+1)   + integrator%getWeight(k)&
                *Kij(1,2)/jacobianDet(k)
            lhs%stiffness(ii+1,jj)   = &
                lhs%stiffness(ii+1,jj)   + integrator%getWeight(k)&
                *Kij(2,1)/jacobianDet(k)
            lhs%stiffness(ii+1,jj+1) = &
                lhs%stiffness(ii+1,jj+1) + integrator%getWeight(k)&
                *Kij(2,2)/jacobianDet(k)
         end do
      end do
      if(this%node(i)%hasSource()) then
         val1 = this%node(i)%ptr%source(1)%evaluate(1, (/this%node(i)%getx()&
              , this%node(i)%gety()/))
         val2 = this%node(i)%ptr%source(1)%evaluate(2, (/this%node(i)%getx()&
              , this%node(i)%gety()/))
         rhs(nDof*i-1) = rhs(nDof*i-1) + val1
         rhs(nDof*i)   = rhs(nDof*i)   + val2
      end if
   end do
   lhs%stiffness = lhs%stiffness * this%material%thickness
   if(this%hasSource()) then
      allocate(valuedSource(2,integrator%getIntegTerms()))
      call this%setupIntegration(integrator, valuedSource, jacobianDet)
      do i = 1, nNode
         val1 = 0._rkind
         val2 = 0._rkind
         do j = 1, integrator%getIntegTerms()
            val1 = val1 + integrator%getWeight(j)*integrator%ptr%shapeFunc(j,i) &
                 *valuedSource(1,j)*jacobianDet(j)
            val2 = val2 + integrator%getWeight(j)*integrator%ptr%shapeFunc(j,i) &
                 *valuedSource(2,j)*jacobianDet(j)
         end do
         rhs(i*nDof-1) = rhs(i*nDof-1) + val1
         rhs(i*nDof)   = rhs(i*nDof)   + val2
      end do
      deallocate(valuedSource)
   end if
   deallocate(jacobian)
   deallocate(jacobianDet)
```

Fragmento de código 11.2: Cálculo de sistema local en `calculateLocalSystem` de `StructuralElementDT`

En el fragmento de código 11.2 se puede ver la implementación de la rutina `calculateLocalSystem`. En esta se desarrolla el ensamble de la matriz de rigidez y el vector de carga para un elemento térmico con cualquier geometría y cantidad de nodos.

PressureDT

Esencialmente es igual a la implementación del flujo para el caso térmico (fragmento de código 10.4). Se realizar una integral sobre el contorno de la geometría con los valores de la presión en las direcciones x e y.

StructuralModelDT

Extensión de `ModelDT`. Esta clase contiene las partes del sistema global (`lhs`, `rhs` y `dof`) y además almacena los resultados de postproceso `normalStress`, `shearStress` y `strain`.

Su declaración es similar a la del fragmento de código 10.5, con las variables de postproceso mencionadas y dos character que contendrán el nombre de los dof's del problema.

11.2.2. `SolvingStrategyDT`

`SolvingStrategyDT`, `StructuralStrategyDT` y `StructuralSchemeDT` están conformados de igual manera que en **Thermal2D**. Aunque `StructuralBuilderAndSolverDT` es también similar es importante mostrar las rutinas que ensamblan y donde se aplican las condiciones ya que en este caso se trabaja con dos grados de libertad.

StructuralBuilderAndSolverDT

Resulta interesante mostrar las rutinas `assembleSystem`, `applyNewmann` y `applyDirichlet` para ver como se implementan con dos grados de libertad.

`assembleSystem` Se encarga de iterar sobre todos los elementos del modelo y obtener de cada uno su sistema local. Esa matriz `localLHS` y el vector `localRHS` se agregarán a la matriz y al vector global.

En el fragmento de código 11.3 se muestra el cuerpo de la rutina `assembleSystem`. Se puede ver que como el `LHS` del modelo es un `SparseDT` se hace uso de las rutinas `append` para agregar un nuevo valor, y `makeCRS` para establecer la matriz, sumando los repetidos, posteriormente a agregar todos los componentes. También es importante notar el bucle interno que se realiza según el número de grados de libertad del problema, en este caso igual a 2.

```
nElem = model%getnElement()
nDof = 2
do iElem = 1, nElem
   element = model%getElement(iElem)
   nNode = element%getnNode()
   call element%calculateLocalSystem(model%processInfo, localLHS, localRHS)
   do iNode = 1, nNode
      iNodeID = element%getNodeID(iNode)
      do jNode = 1, nNode
         jNodeID = element%getNodeID(jNode)
         do iDof = 1, nDof
            do jDof = 1, nDof
               call model%LHS%append(
                    val = localLHS%stiffness(iNode*nDof-(nDof-iDof)&
                    ,jNode*nDof-(nDof-jDof)) &
                    , row = iNodeID*nDof-(nDof-iDof)
                    , col = jNodeID*nDof-(nDof-jDof)
            end do
          end do
```

```
        end do
        model%rhs(iNodeID*nDof-1) = model%rhs(iNodeID*nDof-1) &
             + localRHS(iNode*nDof-1)
        model%rhs(iNodeID*nDof)   = model%rhs(iNodeID*nDof)   &
             + localRHS(iNode*nDof)
    end do
    call localLHS%free()
    deallocate(localRHS)
end do
call model%lhs%makeCRS()
```

Fragmento de código 11.3: Implementación de la rutina `assembleSystem`

`applyNewmann` Implementa las condiciones de Newmann debidas a cargas o presión sobre los bordes o en puntos (a través de `PressureDT`). En el fragmento de código 11.5 se puede ver que consiste en iterar sobre el número de condiciones del modelo y acceder a la condición `iCond`. A la condición `iCond` se le solicita su sistema local de ecuaciones, el cual según necesario se agregará al sistema global, accediendo a las variables lógicas `affectsLHS` y `affectsRHS`. Como se puede ver también el bucle interno se aplica a los dos grados de libertad al mismo tiempo.

```
nCond = model%getnCondition()
do iCond = 1, nCond
    condition = model%getCondition(iCond)
    nNode = condition%getnNode()
    call condition%calculateRHS(model%processInfo, localRHS)
    do iNode = 1, nNode
        iNodeID = condition%getNodeID(iNode)
        model%rhs(iNodeID*2-1) = model%rhs(iNodeID*2-1) + localRHS(iNode*2-1)
        model%rhs(iNodeID*2)   = model%rhs(iNodeID*2)   + localRHS(iNode*2)
    end do
    deallocate(localRHS)
end do
```

Fragmento de código 11.4: Implementación de la rutina `applyNewmann`

`applyDirichlet` Al igual que **Thermal2D** aplica las condiciones de Dirichlet, en este caso desplazamientos fijos sobre los nodos. En esta aplicación cada nodo tiene dos dofs: DISPLACEMENT_X y DISPLACEMENT_Y. En el fragmento de código 11.4 se puede ver la implementación usando los nombres de los grados de libertad.

```
nNode = model%getnNode()
nDof = 2
do i = 1, nNode
    node = model%getNode(i)
    do j = 1, node%getnDof()
        if(node%ptr%dof(j)%getName() == 'DISPLACEMENT_X') then
            if(node%ptr%dof(j)%isFixed) then
                nodeID = node%ptr%getID()
```

```
            call model%lhs%setDirichlet(nodeID*nDof-1)
            model%rhs(nodeID*nDof-1) = node%ptr%dof(j)%fixedVal
          end if
        else if(node%ptr%dof(j)%getName() == 'DISPLACEMENT_Y') then
          if(node%ptr%dof(j)%isFixed) then
            nodeID = node%ptr%getID()
            call model%lhs%setDirichlet(nodeID*nDof)
            model%rhs(nodeID*nDof) = node%ptr%dof(j)%fixedVal
          end if
        end if
      end do
    end do
```

Fragmento de código 11.5: Implementación de la rutina `applyDirichlet` utilizando nombres de dofs

11.3. Tests

11.3.1. Perfil L de acero

El siguiente caso se basa en un problema propuesto en un tutorial de Ansys, extraído de la referencia [17]. El ejercicio plantea calcular los desplazamientos y las tensiones resultantes de una distribución de presión, como se ve en la figura 11.2. Para el problema, $E = 2 \times 10^{11}[Pa]$, $\nu = 0{,}3$ y $t = 3{,}125 \times 10^{-3}[m]$.

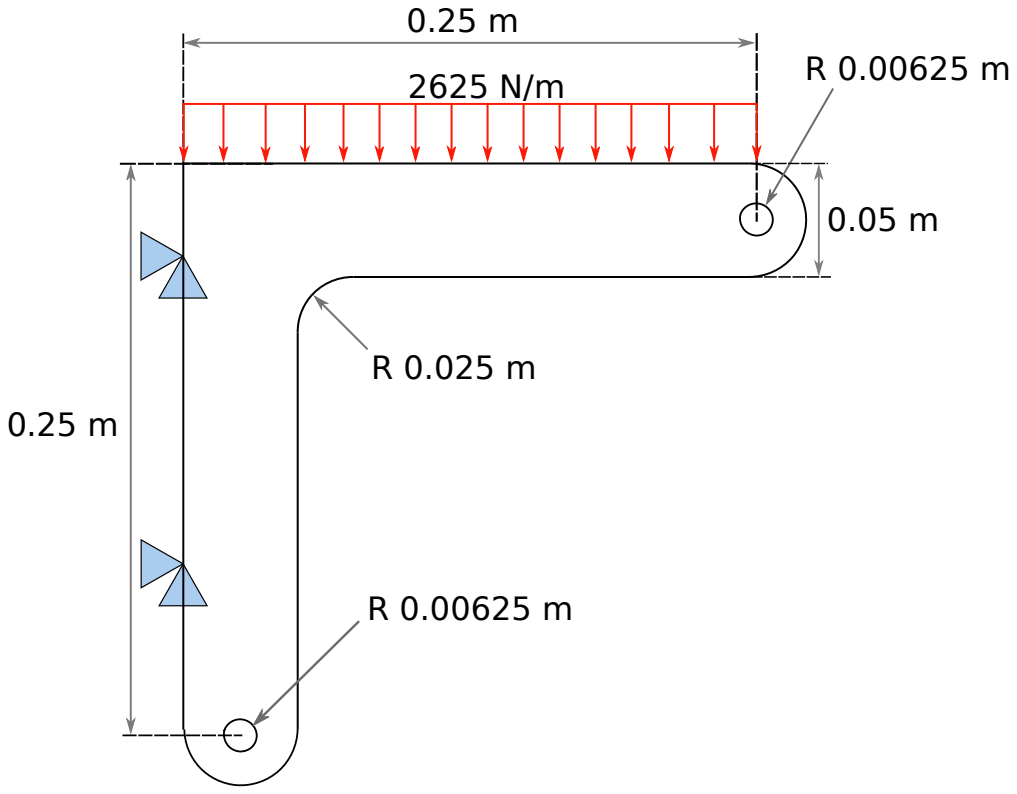
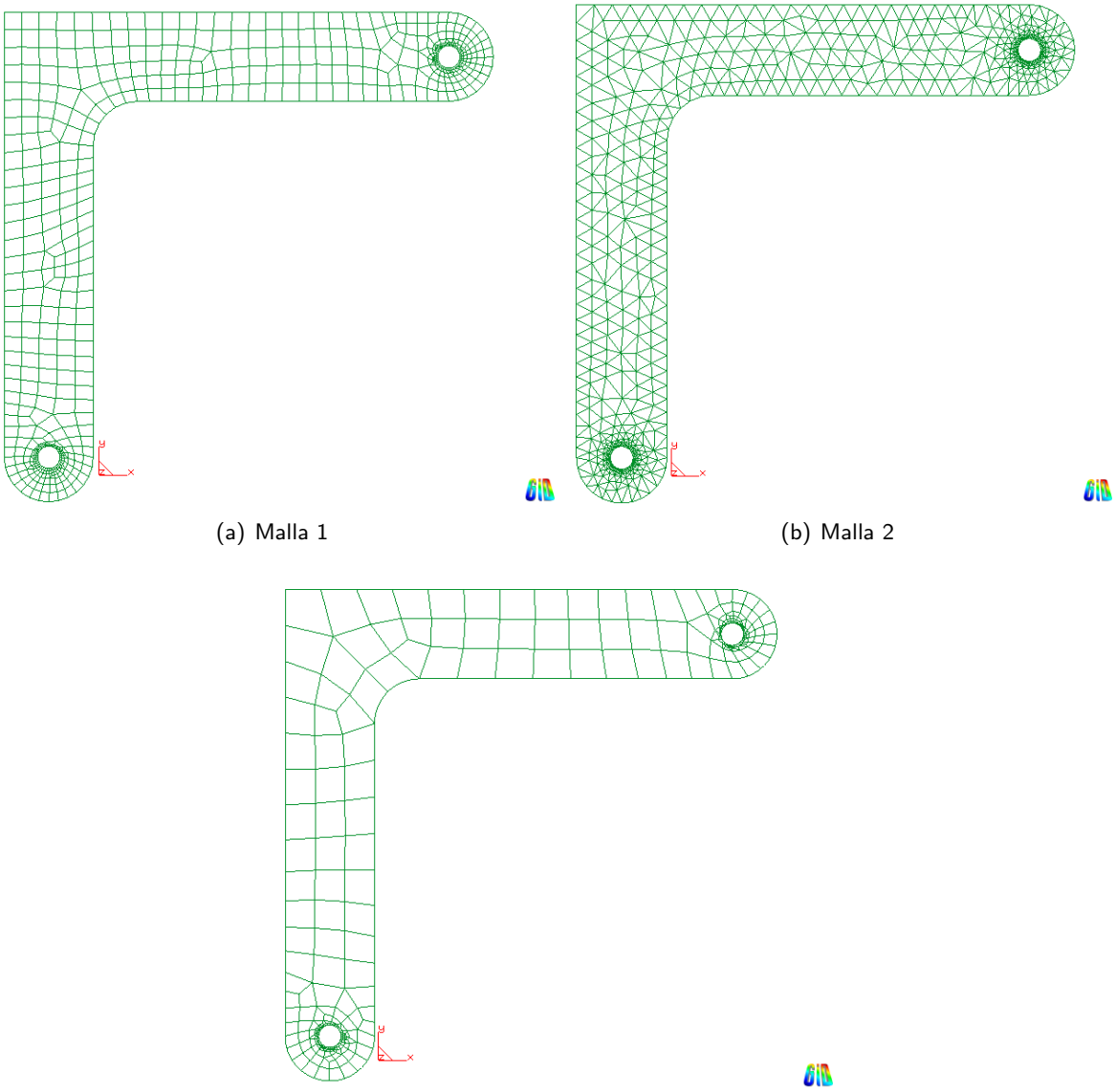


Figura 11.2: Geometría generada para resolver el ejercicio propuesto y condiciones de contorno

Luego de generar la geometría y seleccionar el *Problem Type* **Structural2D**, se deben cargar las condiciones de contorno según se muestra en la figura 11.2. Se selecciona el material modificando los valores y se agrega la condición de presión en la parte superior de la geometría. El problema se resuelve con un orden de integración de cuadratura de Gauss $n = 3$.

Mallado

Una vez cargadas las condiciones, materiales, cargas y modificado el orden de integración de ser necesario, se debe generar la malla de elementos finitos para resolver. Es importante notar que cada vez que se modifique alguna condición, se debe volver a generar la malla.



(a) Malla 1

(b) Malla 2

(c) Malla 3

Figura 11.3: Comparación de distintos mallados sobre la geometría

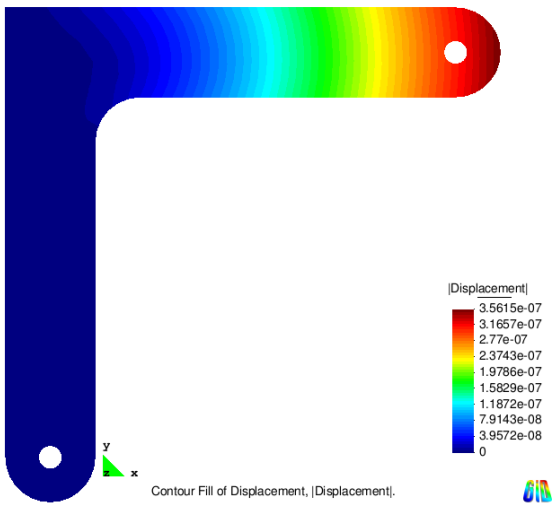
Para este caso, se decide resolver con tres tipos de malla distintos, como se muestra en la figura 11.3, para poder observar las diferencias que se presenten en las distintas soluciones. Además se agrega la condición de que en los orificios las lineas estén divididas en 40 segmentos para los 3 casos.

Las mallas utilizadas fueron:

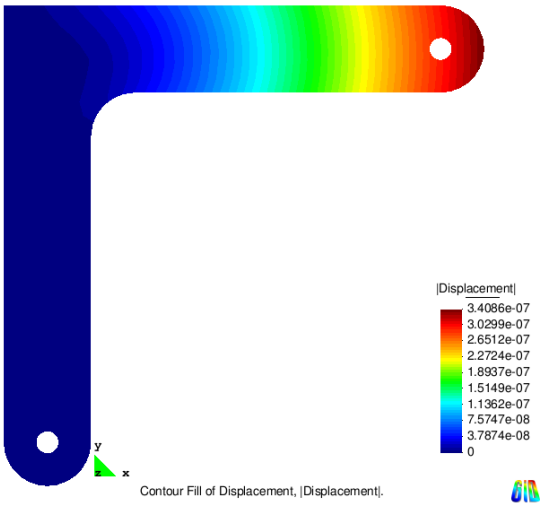
- Malla 1: Tamaño de elementos: 0.01, 625 elementos cuadriláteros lineales y 732 nodos.
- Malla 2: Tamaño de elementos: 0.01, 1208 elementos triangulares lineales y 711 nodos.
- Malla 3: Tamaño de elementos: 0.02, 353 elementos cuadriláteros cuadráticos y 1212 nodos.

Desplazamientos

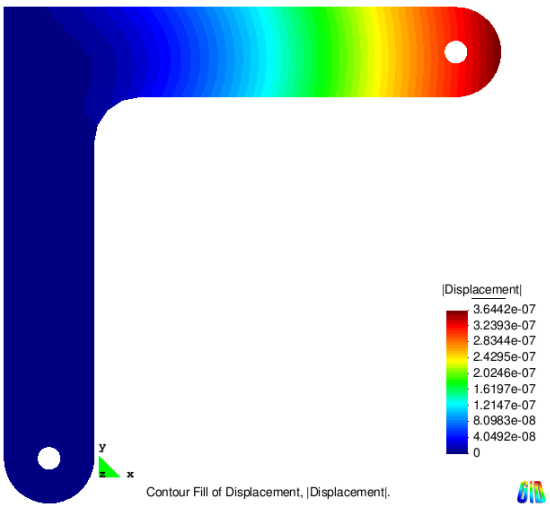
En la figura 11.4 se muestran los resultados de las corridas para los tres tipos de malla. A simple vista se puede observar que las soluciones son similares. Los tiempos de ejecución en los tres casos resultan comparables y no significativos.



(a) Malla 1



(b) Malla 2

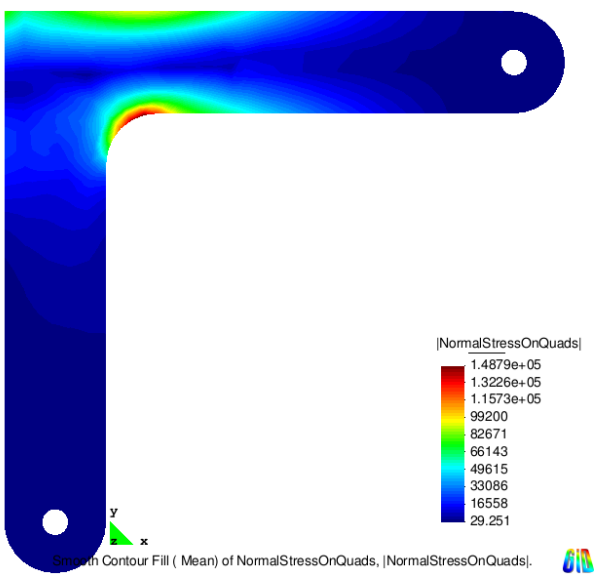


(c) Malla 3

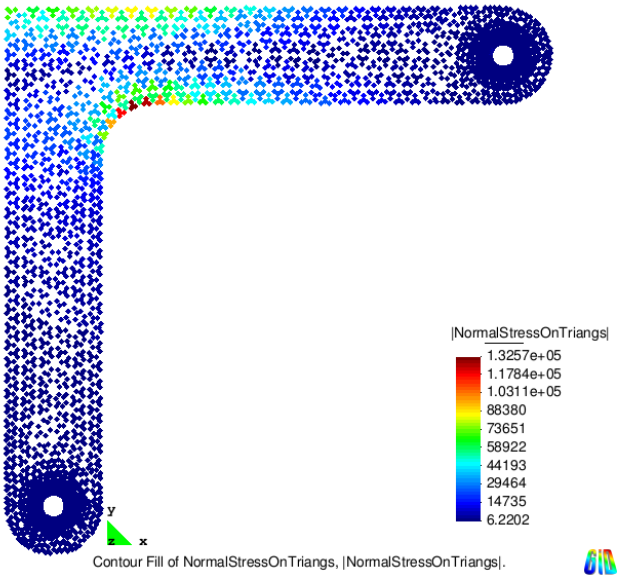
Figura 11.4: Desplazamientos resultantes para distintos mallados

Tensiones

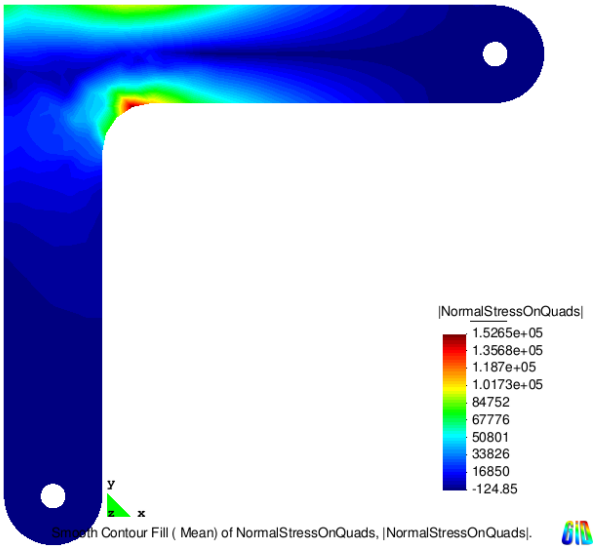
Las tensiones son fáciles de calcular como un post-proceso realizado a partir de los desplazamientos. Es por esto que, como se observa en la figura 11.5, la forma de la solución obtenida depende fuertemente de la cantidad de datos obtenidos ó de la cantidad de nodos. Debido a que es aproximadamente la derivada del campo de desplazamientos, mientras más puntos conozcamos en el dominio, mejor va a ser la solución observada. Los resultados mostrados en la malla 3 en este caso se ven con un patrón que responde a la cantidad de puntos de Gauss que se usó, los cuales no alcanzan para que el postprocesador interpole los resultados dentro de los elementos y que se muestren como en las mallas 1 y 2.



(a) Malla 1



(b) Malla 2



(c) Malla 3

Figura 11.5: Tensiones resultantes para los distintos mallados

11.3.2. Soporte de acero

El siguiente caso se basa en un problema propuesto en un tutorial de Ansys, extraído de la referencia [18]. El ejercicio plantea calcular los desplazamientos resultantes de una carga vertical $F = -2000[N]$ aplicada en el centro del círculo principal, como se ve en la figura 11.6. Para el problema se usó, $E = 2 \times 10^{11}[Pa]$, $\nu = 0{,}3$ y $t = 3{,}125 \times 10^{-3}[m]$.

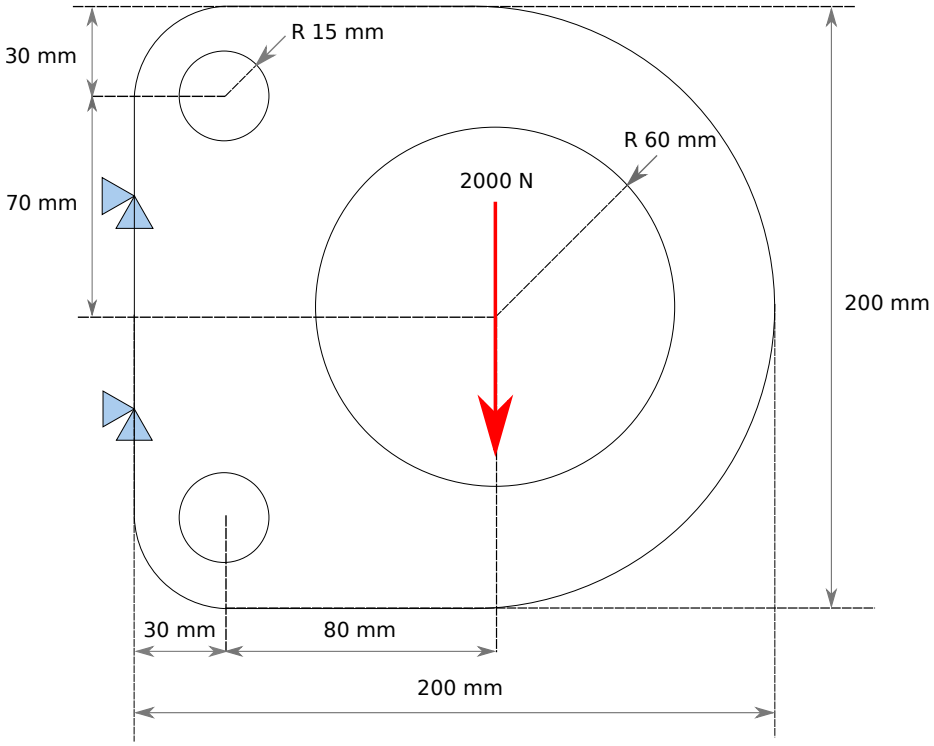


Figura 11.6: Geometría generada para resolver el ejercicio propuesto y condiciones de contorno

Luego de generar la geometría y seleccionar el *Problem Type* **Structural2D**, se deben cargar las condiciones de contorno según se muestra en la figura 11.6. En este caso al estar la carga en el centro del círculo se decidió dividirlo en dos y aplicarla en la semicircunferencia inferior, luego se selecciona el material modificando los valores con los del problema. El problema se resuelve para 3 puntos de Gauss.

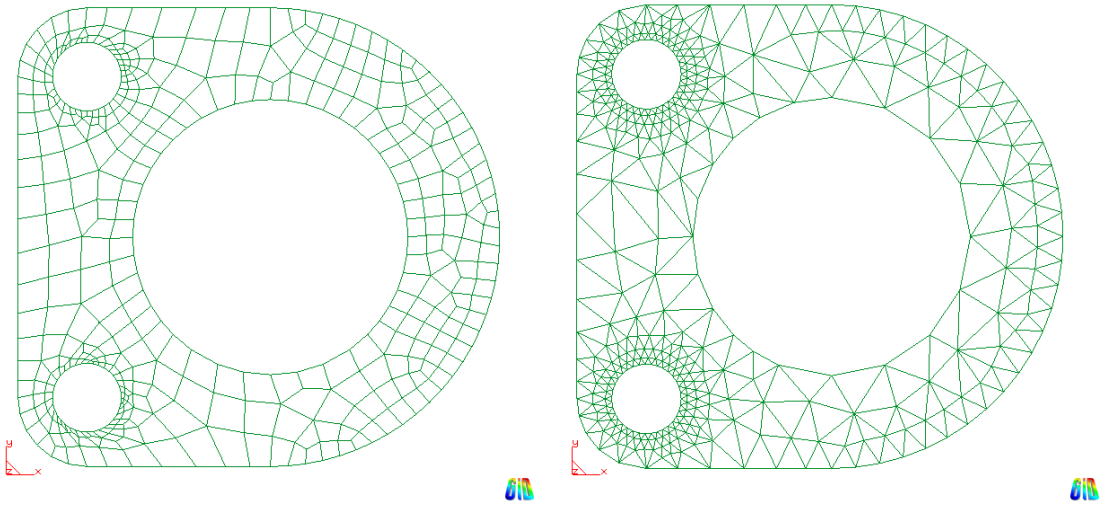
Mallado

Una vez cargadas las condiciones, materiales, cargas y modificado el orden de integración de ser necesario, se debe generar la malla de elementos finitos para resolver.

Para este caso se decide resolver con tres tipos de malla distintos como se muestra en la figura 11.7 para poder observar las diferencias que se presenten en las distintas soluciones. Además se divide en 36 partes las lineas de los orificios menores para las 3 mallas.

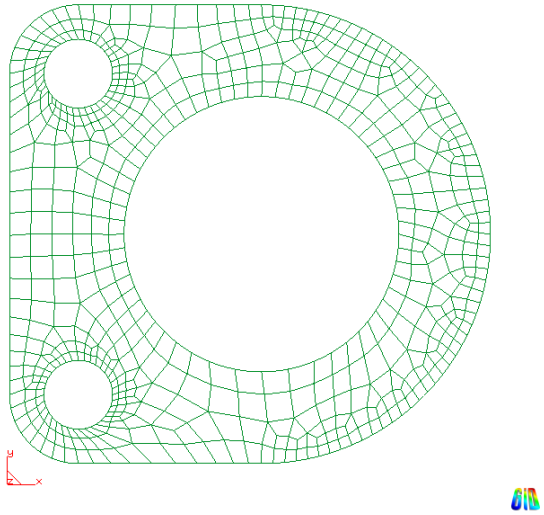
Las mallas utilizadas fueron:

- Malla 1: Tamaño de elementos: 0.15, 403 elementos cuadriláteros lineales y 499 nodos.
- Malla 2: Tamaño de elementos: 0.20, 685 elementos triangulares lineales y 417 nodos.
- Malla 3: Tamaño de elementos: 0.10, 616 elementos cuadriláteros cuadráticos y 2092 nodos.



(a) Malla 1

(b) Malla 2

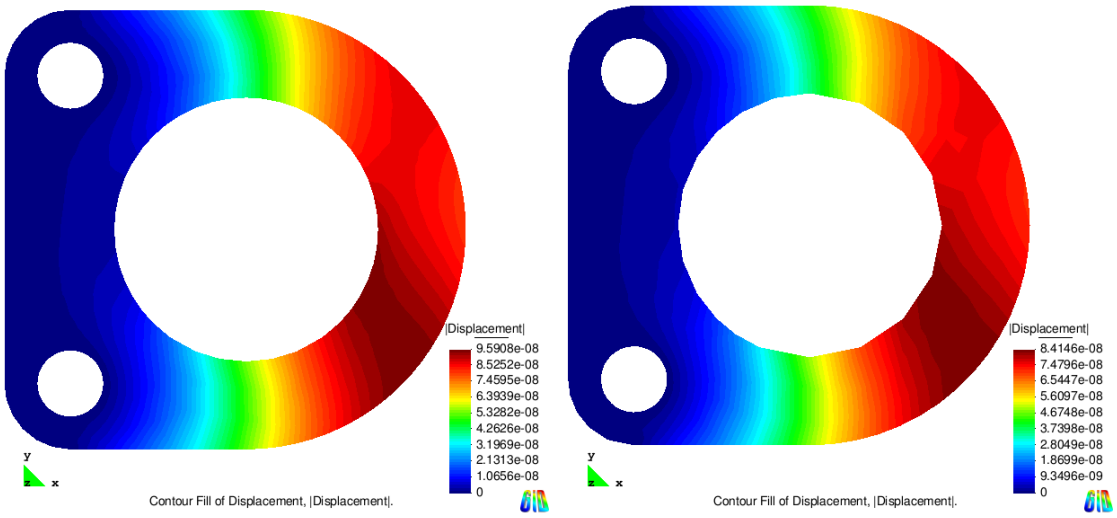


(c) Malla 3

Figura 11.7: Comparación de distintos mallados sobre la geometría

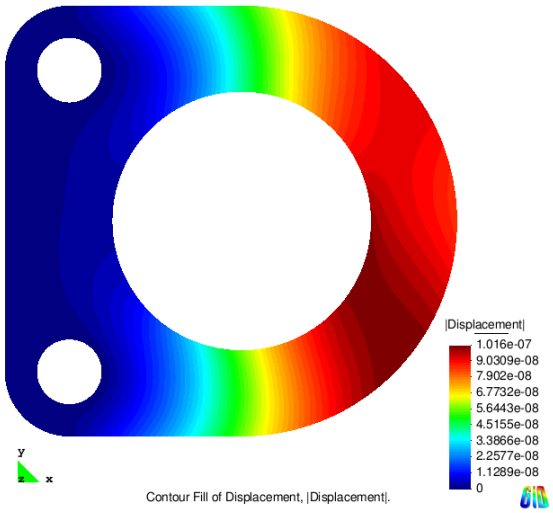
Desplazamientos

En la figura 11.8 se muestran los resultados de las corridas para los tres tipos de malla. Los tiempos de ejecución en los tres casos resultan comparables y no significativos. Se puede ver que los resultados obtenidos son similares y que en el caso de la malla 3 el comportamiento es más suave que en las otras dos.



(a) Malla 1

(b) Malla 2



(c) Malla 3

Figura 11.8: Desplazamientos resultantes para distintos mallados

12. Problemas Acoplados: ThermalStructural2D

Esta aplicación fue desarrollada para resolver problemas de elasticidad bidimensional con condiciones de temperatura fijadas en el dominio de manera estacionaria. Trabaja con todos los tipos de elementos bidimensionales soportados por la framework y también se desarrolló un *Problem Type* para trabajar con GiD. A continuación se definirá el problema específico a resolver y se mencionará su implementación.

12.1. Definición del problema

El acoplamiento térmico estructural resulta ser una extensión del problema estructural planteado para **Structural2D** al que se llega en la ecuación 11.5. Esta extensión implica primero resolver un problema térmico para conocer los valores de temperatura en el dominio lo cual ya fue comentado en **Thermal2D**.

Una vez calculada la distribución de temperatura se puede determinar su aporte al vector de cargas o fuerzas nodales f de la ecuación 11.6 mediante la ecuación 12.1. En esta ecuación ϵ_T representa la

deformación térmica unitaria la cual se asume proporcional al coeficiente de expansión térmica α y a la temperatura ($\epsilon_T = \alpha T$).

$$\mathbf{f}_T^{(e)} = \int \int_{\Omega^{(e)}} \left[\mathbf{B}^{(e)}\right]^T \mathbf{D}\boldsymbol{\epsilon}_T^{(e)} \, d\Omega \tag{12.1}$$

De esta forma el problema térmico estructural se reduce simplemente a la resolución de un problema térmico y luego uno estructural en el que se agrega el aporte de la carga térmica.

12.2. Diseño de la aplicación

Del mismo modo que las demás aplicaciones se ejecutará desde el archivo `main.f90` donde se encuentran las instrucciones generales y se declaran las instancias de las estructuras principales, en este caso: `ThermalStructural2DApplicationDT` y `SolvingStrategyDT`.

12.2.1. ThermalStructural2DApplicationDT

En este caso contiene:

1. Lista de nodos (`NodeDT`)
2. Lista de elementos (`ThermalStructuralElementDT` y `ThermalElementDT`)
3. Lista de condiciones (`PressureDT`, `convectionOnLineDT` y `FluxOnLineDT`)
4. Lista de fuentes (`SourceDT`)
5. Lista de materiales (`StructuralMaterialDT`)
6. El modelo (`StructuralModelDT` y `ThermalModelDT`)

Las implementaciones relevantes para esta aplicación son:

StructuralMaterialDT

Para los problemas de elasticidad bidimensional se propone que el módulo de Young, el coeficiente de Poisson, el coeficiente de expansión térmica, el área, el espesor, la temperatura inicial y los elementos de la matriz D de las ecuaciones 11.9 y 11.10 sean parte de esta clase la cual es extendida de `ThermalMaterial` de la aplicación **Thermal2D**.

```
type, extends(ThermalMaterialDT) :: StructuralMaterialDT
   real(rkind) :: young
   real(rkind) :: poissonCoef
   real(rkind) :: thermalCoef
   real(rkind) :: area
   real(rkind) :: thickness
   real(rkind) :: d11, d12, d21, d22, d33
   real(rkind) :: stableTemp
 contains
   procedure :: initThermalStructMat
end type StructuralMaterialDT
```

Fragmento de código 12.1: Declaración del type `StructuralMaterialDT`

En el fragmento de código 12.1 se muestra la declaración de la clase `StructuralMaterialDT` donde se pueden ver las propiedades que la componen.

ThermalStructuralElementDT

Junto con `ThermalElementDT`, que ya se explicó, son las dos extensiones de `ElementDT` que usa la aplicación. En esta clase se agrega como miembro nuevamente el tipo de material `StructuralMaterialDT`. Está diseñado para elementos triangulares o cuadriláteros de orden linear o cuadrático y la rutina de inicialización es igual que la implementada en **Thermal2D**.

El resto del módulo se dedica a implementar las rutinas diferidas `CalculateLocalSystem`, `CalculateLHS`, `CalculateRHS` y `CalculateResults`.

```
do i = 1, nNode
   do j = 1, nNode
      ii = nDof*i-1
      jj = nDof*j-1
      lhs%stiffness(ii,jj)     = 0._rkind
      lhs%stiffness(ii+1,jj)   = 0._rkind
      lhs%stiffness(ii,jj+1)   = 0._rkind
      lhs%stiffness(ii+1,jj+1) = 0._rkind
      do k = 1, integrator%getIntegTerms()
         bi = jacobian(k,2,2)*integrator%getDShapeFunc(k,1,i) &
              - jacobian(k,1,2)*integrator%getDShapeFunc(k,2,i)
         bj = jacobian(k,2,2)*integrator%getDShapeFunc(k,1,j) &
              - jacobian(k,1,2)*integrator%getDShapeFunc(k,2,j)
         ci = jacobian(k,1,1)*integrator%getDShapeFunc(k,2,i) &
              - jacobian(k,2,1)*integrator%getDShapeFunc(k,1,i)
         cj = jacobian(k,1,1)*integrator%getDShapeFunc(k,2,j) &
              - jacobian(k,2,1)*integrator%getDShapeFunc(k,1,j)

         Kij(1,1) = bi*bj*d11 + ci*cj*d33
         Kij(1,2) = bi*cj*d12 + bj*ci*d33
         Kij(2,1) = ci*bj*d21 + bi*cj*d33
         Kij(2,2) = bi*bj*d33 + ci*cj*d22

         lhs%stiffness(ii,jj)     = &
              lhs%stiffness(ii,jj)     + integrator%getWeight(k)&
              *Kij(1,1)/jacobianDet(k)
         lhs%stiffness(ii,jj+1)   = &
              lhs%stiffness(ii,jj+1)   + integrator%getWeight(k)&
              *Kij(1,2)/jacobianDet(k)
         lhs%stiffness(ii+1,jj)   = &
              lhs%stiffness(ii+1,jj)   + integrator%getWeight(k)&
              *Kij(2,1)/jacobianDet(k)
         lhs%stiffness(ii+1,jj+1) = &
              lhs%stiffness(ii+1,jj+1) + integrator%getWeight(k)&
              *Kij(2,2)/jacobianDet(k)
      end do
   end do
   if(this%node(i)%hasSource()) then
      val1 = this%node(i)%ptr%source(2)%evaluate(1&
```

```
            , (/this%node(i)%getx(), this%node(i)%gety()/))
       val2 = this%node(i)%ptr%source(2)%evaluate(2&
            , (/this%node(i)%getx(), this%node(i)%gety()/))
       rhs(nDof*i-1) = rhs(nDof*i-1) + val1
       rhs(nDof*i)   = rhs(nDof*i)   + val2
    end if
    temp = this%node(i)%ptr%dof(1)%val
    !Deformación plana
    strain = (1+this%material%poissonCoef)*this%material%thermalCoef&
         *(temp-this%material%stableTemp)
    !Tensión plana
    !strain = this%material%thermalCoef*(temp-this%material%stableTemp)
    val1 = 0._rkind
    val2 = 0._rkind
    do j = 1, integrator%getIntegTerms()
       bi = jacobian(j,2,2)*integrator%getDShapeFunc(j,1,i) &
            - jacobian(j,1,2)*integrator%getDShapeFunc(j,2,i)
       ci = jacobian(j,1,1)*integrator%getDShapeFunc(j,2,i) &
            - jacobian(j,2,1)*integrator%getDShapeFunc(j,1,i)
       val1 = val1 + integrator%getWeight(j)*bi*(d11*strain+d12*strain)&
            *thickness
       val2 = val2 + integrator%getWeight(j)*ci*(d21*strain+d22*strain)&
            *thickness
    end do
    rhs(i*nDof-1) = rhs(i*nDof-1) + val1
    rhs(i*nDof)   = rhs(i*nDof)   + val2
 end do
 lhs%stiffness = lhs%stiffness * thickness
 if(this%hasSource()) then
    allocate(valuedSource(2,integrator%getIntegTerms()))
    call this%setupIntegration(integrator, valuedSource, jacobianDet)
    do i = 1, nNode
       val1 = 0._rkind
       val2 = 0._rkind
       do j = 1, integrator%getIntegTerms()
          val1 = val1 + integrator%getWeight(j)*integrator%ptr%shapeFunc(j,i) &
               *valuedSource(1,j)*jacobianDet(j)
          val2 = val1 + integrator%getWeight(j)*integrator%ptr%shapeFunc(j,i) &
               *valuedSource(2,j)*jacobianDet(j)
       end do
       rhs(i*nDof-1) = rhs(i*nDof-1) + val1
       rhs(i*nDof)   = rhs(i*nDof)   + val2
    end do
    deallocate(valuedSource)
 end if
```

Fragmento de código 12.2: Cálculo de sistema local en `calculateLocalSystem` de `ThermalStructuralElementDT`

En el fragmento de código 12.2 se puede ver la implementación de la rutina `calculateLocalSystem`.

En esta se desarrolla el ensamble de la matriz de rigidez y el vector de carga para un elemento estructural con cualquier geometría y cantidad de nodos de igual forma que en **Structural2D** y se agrega al vector de carga la deformación inicial generada por la temperatura aplicada.

12.2.2. `SolvingStrategyDT`

Para este caso se agregan como miembros `ThermalModelDT` y `StructuralModelDT` a la clase `SolvingStrategyDT`. De esta forma se pueden resolver por separado los modelos con las mismas estrategias generadas para las aplicaciones **Thermal2D** y **Structural2D**.

12.3. Tests

12.3.1. Caso de prueba de acople térmico-estructural

Para poder validar la aplicación de acople se realizó la solución de un problema simple 2D sacado de la referencia [19]. El problema consiste en calcular los desplazamientos resultantes en los vértices de un triángulo de acero, de espesor unitario, cargado en un lado con una presión de $-2000\ psi$ y sometido a una temperatura uniforme en la superficie de $30°F$.

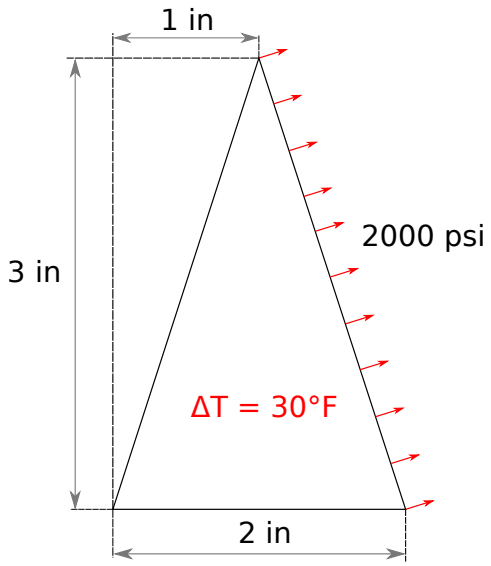


Figura 12.1: Datos iniciales y geometría del problema

En la figura 12.1 se puede observar el planteo del problema a resolver donde se usó $E = 30 \times 10^6$ [Psi], $\nu = 0{,}25$, $t = 1$ [In] y $\alpha = 7 \times 10^{-6}$ [in/in $F°$].

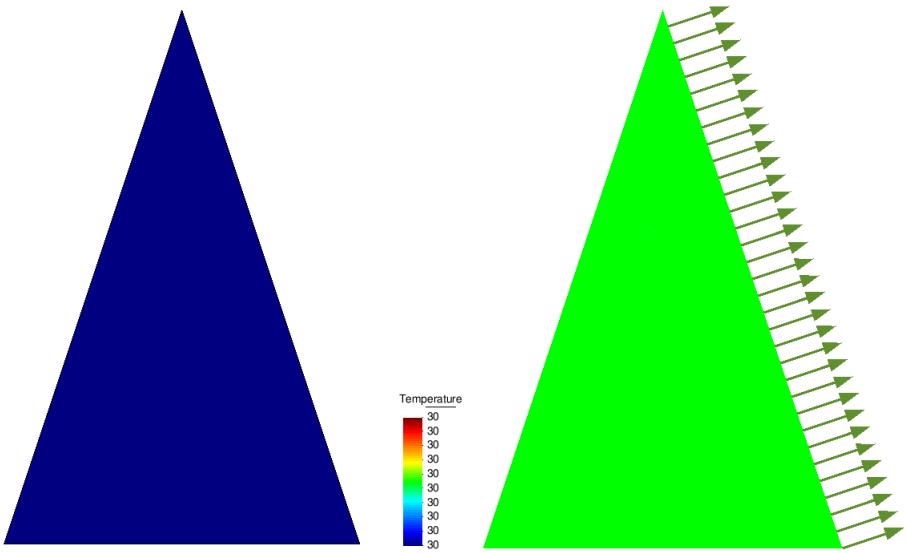
Mallado

Una vez generada la geometría, cargadas las condiciones y materiales se decide resolver dos casos para comparar las soluciones, uno con un solo elemento triangular lineal y otro con un elemento triangular cuadrático. Por su simplicidad no se muestran las mallas de los casos.

Resultados

Al resolver un problema de este tipo, lo primero que se resuelve es el caso térmico. En este problema si bien la temperatura es uniforme en la estructura, se aplicó como condición de Dirichlet sobre el lado

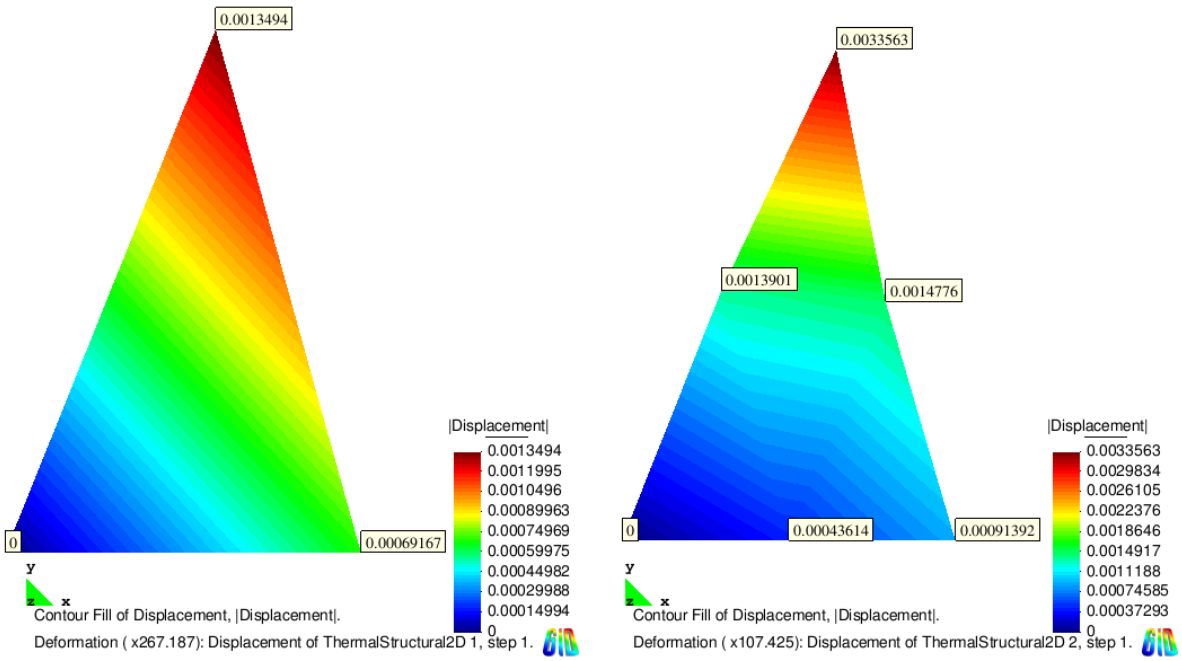
izquierdo y se dejó que se calculen los demás valores. Los resultados obtenidos en esta primera parte y la distribución de presión usada se muestran en la figura 12.2.



(a) Distribución de temperatura en el dominio (b) Presión aplicada sobre la geometría

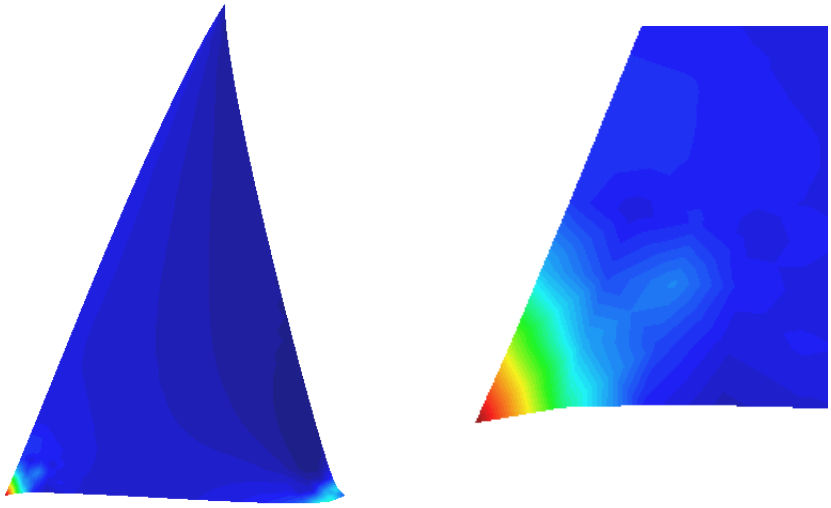
Figura 12.2: Temperatura y presión aplicadas sobre la geometría

Una vez obtenida la distribución de temperatura se calculan los desplazamientos de los nodos. Estos resultados se muestran para los dos casos en la figura 12.3 donde se muestra el módulo de los desplazamientos en los nodos con la geometría en la posición deformada.



(a) Malla de un elemento triangular lineal (b) Malla de un elemento triangular cuadrático

Figura 12.3: Resultados obtenidos para las dos mallas planteadas mostrados en la posición deformada de la estructura.



(a) (b)

Figura 12.4: Tensiones resultantes para los desplazamientos obtenidos

En la figura 12.4 se pueden observar las tensiones resultantes para el campo de desplazamientos obtenido en una malla con más elementos encontrándose los valores máximos en los apoyos. Los datos de esta malla no se muestran ya que el caso sólo se corrió para verificar la convergencia de la solución y la ubicación de las tensiones máximas. En la misma figura se muestra el detalle del vértice izquierdo, fijo en ambas direcciones.

13. Problemas de CFD: CFD2D

Esta aplicación es una adaptación de la implementación de esquema explícito para la simulación de flujos compresibles viscosos y no viscosos en dominios bidimensionales de fluidos newtonianos que se puede consultar en la referencia [20]. Para la discretización espacial se utiliza el método de los elementos finitos y para la discretización temporal un algoritmo mixto basado en los métodos de Runge-Kutta y Adams Bashforth ambos de cuarto orden. Para resolver los problemas de estabilidad numérica se utiliza una variante de la formulación *"Streamline-Upwind/Pretov-Galerkin"* (SUPG), de alto orden[21]. Este método es acompañado por un término de captura de choque que provee estabilidad en las discontinuidades.

13.1. Definición del problema

13.1.1. Ecuaciones de la Dinámica de Fluidos

Las ecuaciones de Navier Stokes de la dinámica de fluidos representan un sistema de leyes de conservación no lineal, expresando la conservación de la masa, cantidad de movimiento y energía definidos por las siguientes ecuaciones:

$$\frac{\partial U}{\partial t} + \frac{\partial F_i}{\partial x_i} - \frac{\partial G_i}{\partial x_i} = 0 \quad \text{, En } \Omega \ \forall \ \text{t} \in (0, t) \tag{13.1}$$

Donde Ω es el dominio con contorno Γ, siendo U el vector de las variables de campo o variables conservativas, F_i es el vector de flujo advectivo y términos de presión y G_i el vector de flujo difusivo,

definidos de la siguiente manera:

$$U = \begin{bmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho E \end{bmatrix}; F_i = \begin{bmatrix} \rho u_i \\ \rho u_1 u_i + p\delta_{1i} \\ \rho u_2 u_i + p\delta_{2i} \\ \rho H u_i \end{bmatrix}; G_i = \begin{bmatrix} 0 \\ \tau_{1i} \\ \tau_{2i} \\ (\tau_{ij}.u_j) + k.\frac{\partial T}{\partial x_i} \end{bmatrix} \tag{13.2}$$

con $\tau_{ij} = \mu\left[\left(\frac{\partial u_i}{\partial x_j} - \frac{\partial u_j}{\partial x_i}\right) - \delta_{ij}\frac{2}{3}\frac{\partial u_k}{\partial x_i}\right]$

donde ρ, u, p, E y $H = E + p/\rho$, son la densidad, velocidad, presión, energía total específica, y entalpía de estancamiento respectivamente. Considerando un gas ideal politrópico, se tiene la siguiente ecuación de estado, la cual brinda una relación adicional necesaria para que el sistema de ecuaciones quede determinado:

$$p = (\gamma - 1)\rho\left(E - \frac{\mid u \mid^2}{2}\right) \tag{13.3}$$

donde γ es la relación de calores específicos. Se adopta $\gamma = 1{,}4$ para aire. Siguiendo la metodología de [22], expresando como un sistema en forma cuasi-lineal se tiene:

$$\frac{\partial U}{\partial t} + A_i\frac{\partial U}{\partial x_i} - \frac{\partial}{\partial x_i}\left(K_{ij}\frac{\partial U}{\partial x_j}\right) = 0 \tag{13.4}$$

donde los $A_i = \frac{\partial F_i}{\partial U}$ se denominan matrices de Jacobianos advectivos y $K_{ij} = \frac{\partial G_i}{\partial U}$ matrices de Jacobianos difusivos. Las mismas se pueden consultar en detalle en la referencia [20].

El problema queda definido adicionando a la ecuación 13.4 las condiciones iniciales y de borde adecuadas al problema analizado.

13.1.2. Implementación de elementos finitos y discretización temporal

De las ecuaciones de la dinámica de fluidos en forma conservativa 13.4, considerando los términos asociados a la formulación de estabilidad y captura de discontinuidades, se obtiene la siguiente ecuación resultado de la formulación adoptada:

$$\frac{\partial U}{\partial t} + A_i\frac{\partial U}{\partial x_i} - \tau A_i\frac{\partial}{\partial x_i}\left(A_j\frac{\partial U}{\partial x_j} - \vartheta\right) - \mu_a\frac{\partial}{\partial x_i}\left(\frac{\partial U}{\partial x_j}\right) - \frac{\partial}{\partial x_i}\left(K_{ij}\frac{\partial U}{\partial x_j}\right) = 0 \quad \text{, En } \Omega \ \forall \ \text{t} \in (0,t) \tag{13.5}$$

Dada la ecuación 13.5, aplicando el método de los residuos ponderados de Galerkin y debilitando la ecuación resultante, se obtiene la siguiente expresión:

$$\int_\Omega N^T\frac{\partial U}{\partial t}d\Omega + \int_\Omega N^T A_i\frac{\partial U}{\partial x_i}d\Omega + \int_\Omega \frac{\partial N^T}{\partial x_i}\tau A_i\left(A_j\frac{\partial U}{\partial x_j} - \vartheta\right)d\Omega + \int_\Omega \frac{\partial N^T}{\partial x_i}\mu_a\left(\frac{\partial U}{\partial x_j}\right)d\Omega + \int_\Omega \frac{\partial N^T}{\partial x_i}K_{ij}\left(\frac{\partial U}{\partial x_j}\right)d\Omega = 0 \tag{13.6}$$

las matrices τ, μ_a, A_i y K_{ij} son la matriz de estabilización advectiva, la matriz de términos de captura de choque, los términos advectivos y los términos difusivos respectivamente. N es una matriz que contiene las funciones de forma.

Para la discretización temporal, se decide utilizar un esquema explícito Runge-Kutta de cuarto orden de mínimo almacenamiento [1] e intercalar los pasos con un esquema de Adams-Bashforth:

Runge-Kutta:

$$\Delta U^{n+i} = \alpha_i \Delta t r(U^n + \Delta U^{n+i+1}) \quad , i = 1, ..., 4,\ \Delta U^0 = 0,\ \alpha_i = 1/i \tag{13.7}$$

Adams-Bashforth:

$$U^{n+4} = U^{n+3} + \Delta t \left(\frac{55}{24} r(U^{n+3}) - \frac{59}{24} r(U^{n+2}) + \frac{37}{24} r(U^{n+1}) - \frac{9}{24} r(U^n) \right) \tag{13.8}$$

La ventaja de esta combinación se debe a la mayor rapidez del esquema de Adams-Bashforth, ya que luego de las primeras iteraciones, solo necesita una evaluación del miembro derecho $r(U)$ por cada paso. La parte pesada del algoritmo se centra en las sucesivas evaluaciones del vector $r(U)$, como se verá mas adelante.

13.2. Diseño de la aplicación

El diseño de esta aplicación es muy similar al de las presentadas anteriormente. La principal diferencia está en el hecho de que los problemas serán no estacionarios. Es por eso que se deben usar las estructuras de integración en el tiempo presentadas en la sección 5.3.2. Adicionalmente, para obtener un programa relativamente eficiente, fue necesario implementar rutinas adicionales, las cuales se explicarán a continuación.

13.2.1. `CFD2DApplicationDT`

En este caso `CFD2DApplicationDT` contendrá:

1. Lista de nodos (`NodeDT`)
2. Lista de elementos (`CFDElementDT`)
3. Lista de condiciones (`NormalVelocityDT`)
4. Lista de fuentes (`SourceDT`)
5. Lista de materiales (`CFDMaterialDT`)
6. El modelo (`CFDModelDT`)

Es de interés mencionar como se implementan algunas de estas clases y que diferencias poseen en relación a las aplicaciones anteriormente presentadas.

CFDMaterialDT

Se elige `CFDMaterialDT` como la clase contenedora de todas las propiedades del problema. Se puede ver en su declaración en el fragmento de código 13.1 que este type no solo contendrá las propiedades del material ($R, \gamma, \mu, k, \rho, C_v$), si no también propiedades físicas del problema ($V_{\infty_x}, V_{\infty_y}, P_\infty, T_\infty, M_\infty$) y propiedades numéricas del problema (Factor de seguridad para la integración numérica `fSafe` y constante de captura de choque `constant`).

```
type :: CFDMaterialDT
   real(rkind) :: R
   real(rkind) :: gamma
   real(rkind) :: mu
```

```
    real(rkind) :: k
    real(rkind) :: Vx_inf
    real(rkind) :: Vy_inf
    real(rkind) :: T_inf
    real(rkind) :: rho
    real(rkind) :: P_inf
    real(rkind) :: Cv
    real(rkind) :: mach
    real(rkind) :: Vc
    real(rkind) :: fSafe
    real(rkind) :: constant
  contains
    procedure :: init
end type CFDMaterialDT
```

Fragmento de código 13.1: Declaración del type `CFDMaterialDT`

CFDElementDT

Implementar `CFDElementDT` presentó un desafío debido a que en estos tipos de problemas es necesario reensamblar gran parte del sistema en cada iteración. Es por esto que si el ensamble del sistema local de cada elemento es ineficiente, los tiempos de resolución pueden llegar a ser muy largos.

En adición a las rutinas diferidas, era necesario implementar un número de rutinas que se llamarían condicionalmente. En la tabla 13.1 se puede ver la lista de subrutinas y funciones que se implementaron para el módulo `CFDElementM`. A continuación se explican algunas de estas rutinas.

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
`sparse`	**Interface**	PUBLIC	Interface del constructor del type `CFDElementDT`
`constructor`	**Type Procedure**	PRIVATE	Constructor del type `CFDElementDT`
`init`	**Type Procedure**	PRIVATE	Inicializador de los valores de `CFDElementDT`. Rutina llamada por el constructor
`calculateLHS`	**Type Procedure**	PUBLIC	Rutina vacía debido a que solo se ensambla un `rhs` en este problema
`calculateRHS`	**Type Procedure**	PUBLIC	Ensambla el `rhs` local del elemento
`calculateLocalSystem`	**Type Procedure**	PUBLIC	Llama a `calculateRHS`
`calculateResults`	**Type Procedure**	PUBLIC	Convierte los dofs de ρ, ρu, ρv y ρe a ρ, u, v y e.
`calculateDT`	**Type Procedure**	PUBLIC	Calcula el Δt para el elemento y se lo pasa a `processInfo` a través de la rutina `setMinimumDT`
`calculateMass`	**Type Procedure**	PUBLIC	Calcula los componentes de la matriz de masa concentrada para el elemento y se los agrega a la matriz global
`calculateStabMat`	**Type Procedure**	PUBLIC	Calcula la matriz de estabilización del elemento, y los τ y μ_a
`getValuedSource`	**Type Procedure**	PUBLIC	Valua una fuente para su integración

Lista de subrutinas y funciones

Nombre	Tipo $_{Sub-Prog}$	Kind	Descripción
vectorCalculus	Type Procedure	PRIVATE	Guarda valores del elemento que se usan múltiples veces pero cuyo valor no cambiará
initGeometries	Module Procedure	PUBLIC	Inicializa todas las geometrías que puede tomar un elemento
allocateMass	Module Procedure	PUBLIC	Alocata la matriz de masa global, la cual se almacena en el módulo
allocateStabMat	Module Procedure	PUBLIC	Alocata la matriz de estabilización, la cual se almacena en el módulo
zeroStabMat	Module Procedure	PUBLIC	Valúa la matriz de estabilización en cero.

Tabla 13.1: Lista de subrutinas y funciones presentes en el módulo CFDElementM

En **calculateDT** cada elemento calculará su propio Δt e invocará a la rutina setMinimumDT de processInfo, la cual se hará cargo de compararlo con el actual y almacenar el valor mínimo. Esto es necesario realizarlo en cada iteración para asegurar la convergencia.

Como era necesario obtener una matriz de masa, pero la misma solo debía ser calculada una vez y no en cada iteración, se resolvió implementarla aparte. **calculateMass** calcula las componentes de la matriz de masa concentrada para cada elemento y se las adiciona a la matriz global. Esta matriz se decidió almacenar adentro del módulo CFDElementM.

En **calculateStabMat** se calcula la matriz de estabilización y los coeficientes de estabilización τ y de captura de choque μ_a. En la selección de la estrategia de resolución global se decidirá si es necesario invocar esta rutina en cada iteración o cada cierto número de iteraciones. Es por eso que resulta útil separar estos cálculos de calculateLocalSystem.

NormalVelocityDT

La condición que se busca implementar es la de velocidad normal nula en los bordes. Para aplicar esta condición es necesario identificar el vector normal del contorno y utilizarlo para, nodo a nodo, quedarse con la componente tangencial de la velocidad, anulando la normal.

Resultó práctico utilizar esta clase NormalVelocityDT para obtener las componentes de la normal del contorno de cada elemento. Dejando la transformación de la velocidad para implementar en una rutina de BuilderAndSolver.

```
do i = 1, nNode
   nodalPoints(i) = this%node(i)
   pointToValue(i) = point(this%node(i)%getx(), this%node(i)%gety())
end do
do i = 1, nNode
   jacobian = this%geometry%boundaryGeometry%jacobian(pointToValue(i), nodalPoints)
   jacobianDet = this%geometry%boundaryGeometry%jacobianDet(jacobian)
   nx = jacobian(1,2)/jacobianDet
   ny = -jacobian(1,1)/jacobianDet
   rhs(2*i-1) = nx
   rhs(2*i)   = ny
end do
```

Fragmento de código 13.2: Obtención de la normal del contorno de un elemento en NormalVelocityDT

Se puede ver en el fragmento de código 13.2 como se obtiene la normal del contorno. Consiste simplemente en utilizar los jacobianos de la `boundaryGeometry` de cada elemento.

13.2.2. `SolvingStrategyDT`

En `SolvingStrategyDT` se almacenarán las clases encargadas de la organización global del problema y su resolución: `CFDStrategyDT`, `CFDBuilderAndSolverDT` y `CFDSchemeDT`. En adición se implementará la clase `NavierStokesDT`, extensión de `IntegrandDT`, esta será responsable de la integración numérica en el tiempo.

En `CFDStrategyDT` se encuentra la rutina que contendrá el bucle de integración en el tiempo. También se encontrarán en el mismo módulo las subrutinas que iterarán sobre todos los elementos del dominio para ensamblar la matriz de masa (`calculateMass`) y para obtener el Δt mínimo (`calculateDT`).

En `CFDBuilderAndSolver` se encontrarán las rutinas `build`, `update` y `applyDirichlet`. `build` se encargará de, iterando elemento por elemento, obtener los valores de estabilización y ensamblar el sistema global. `update` solo ensamblará el sistema global. `applyDirichlet` aplicará las condiciones de contorno. Es en esa rutina que se utilizará `NormalVelocityDT` para obtener las normales de los contornos adonde se coloca velocidad normal nula y con ellos se transformarán los grados de libertad correspondientes. En el fragmento de código 13.3 se puede ver el algoritmo que toma las normales de un contorno y convierte las velocidad a su componente tangencial. Se hace uso de un vector `position` para no realizar esta operación dos veces en el mismo nodo.

```
allocate(position(nNode))
position = 0
do iCond = 1, nCond
   condition = app%model%getCondition(iCond)
   nNodeCond = condition%getnNode()
   call condition%calculateRHS(app%model%processInfo, localRHS)
   do iNode = 1, nNodeCond
      iNodeID = condition%getNodeID(iNode)
      position(iNodeID) = position(iNodeID) + 1
      if (position(iNodeID) .le. 1) then
          Vx  = app%model%dof(iNodeID*4-2)
         Vy  = app%model%dof(iNodeID*4-1)
         nx  = localRHS(iNode*2-1)
         ny  = localRHS(iNode*2  )
         app%model%dof(iNodeID*4-2) = -ny*(-ny*Vx+nx*Vy)
         app%model%dof(iNodeID*4-1) =  nx*(-ny*Vx+nx*Vy)
      end if
   end do
   deallocate(localRHS)
end do
deallocate(position)
```

Fragmento de código 13.3: Transformación de los grados de libertad en los contornos con velocidad normal nula

En `CFDSchemeDT` se llamará al `calculateResults` de cada elemento para obtener los valores de ρ, u, v, e, T, M y P en cada nodo.

13.3. Tests

Se resolvieron dos problemas de aerodinámica supersónica en dominios bidimensionales. Se usó la solución analítica para comparar los resultados obtenidos. Como pro y post procesador se utiliza GiD.

13.3.1. Cuña simple

La formulación de este ejemplo se puede ver en la figura 13.1. La introducción de una cuña, en este caso representada como una elevación en el terreno, generará una onda de choque. Se buscará estudiar el ángulo de esta onda de choque, y obtener de manera analítica y numérica los valores de Mach y presión antes y después de la onda.

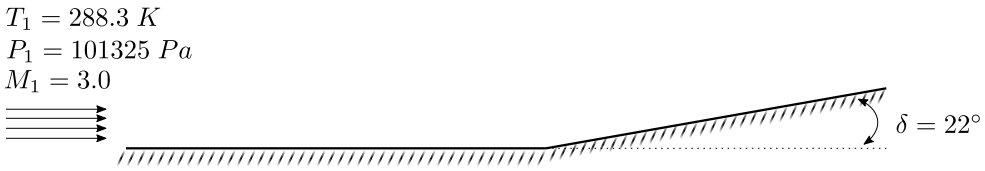


Figura 13.1: Problema de una cuña simple en régimen supersónico

Solución analítica

En este problema se generará una onda de choque oblicua, por lo tanto resulta útil hacer uso de las gráficas de la referencia [23] para obtener el ángulo de choque σ. En la figura 13.2 se puede ver las relaciones entre las velocidades antes y después de la onda, y sus componentes normales y tangenciales. Ya que la componente tangencial no cambia, el problema está en encontrar M_{n_1} y M_{n_2}. Este es un problema de onda de choque normal, el cual es muy simple de resolver analíticamente.

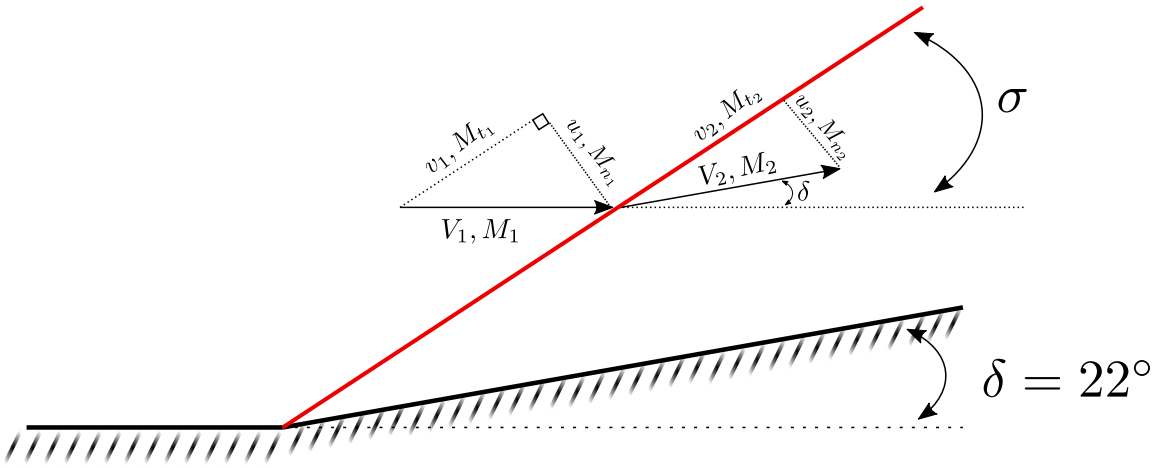


Figura 13.2: Onda de choque oblícua generada por la cuña

Entonces, según las gráficas de la referencia [23]:

$$\sigma = 40°$$

con el cual se obtiene:

$$M_{1_n} = M_1 \sin(\sigma) = 1{,}9284$$

Haciendo uso de las tablas de choque normal para $\sigma = 1{,}4$:

$$M_{2_n} \simeq 0{,}59$$

$$\frac{P_2}{P_1} \simeq 4{,}179$$

Con lo cual puedo obtener los valores de interés:

$$M_2 = \frac{M_{2_n}}{\sin(\sigma - \delta)} \simeq 1{,}91$$

$$P_2 = 419000Pa$$

Solución numérica

Para encarar el problema numéricamente se deben identificar las condiciones de contorno y las propiedades del material. Estos valores se presentan en la figura 13.3.

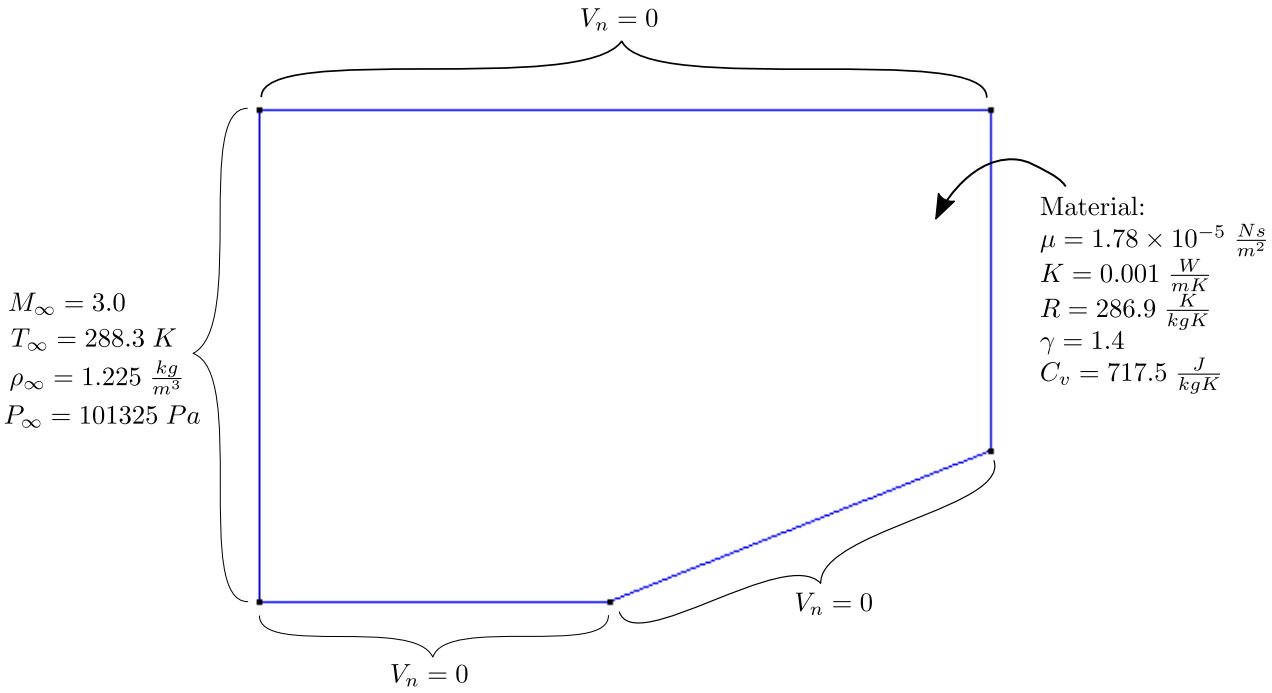


Figura 13.3: Geometría del problema y condiciones de contorno

Si bien tanto P_∞, ρ_∞ y T_∞ se listan como condiciones de contorno, en la realidad solo es necesario especificar dos de estos valores. El tercero será calculado a partir de la ecuación de los gases ideales.

Mallado

Se decidió correr el problema con dos mallas de elementos finitos, una de triángulos lineares y otra de cuadriláteros lineales. El tamaño de línea promedio de ambas mallas es equivalente, siendo $0{,}05m$ en la zona en la que se espera que se genere la onda de choque, y $0{,}1m$ en el resto de la malla. Las mallas resultantes se pueden ver en las figuras 13.4 y 13.5.

Se puede ver que la malla de cuadriláteros es bastante menos atractiva que la de triángulos. Principalmente en la zona donde se producirá la onda de choque se espera que las irregularidades en la malla generen irregularidades en los resultados.

Propiedades numéricas

Interesa alcanzar el resultado más preciso en el menor número de iteraciones. Se utilizan dos coeficientes numéricos en la operación del problema.

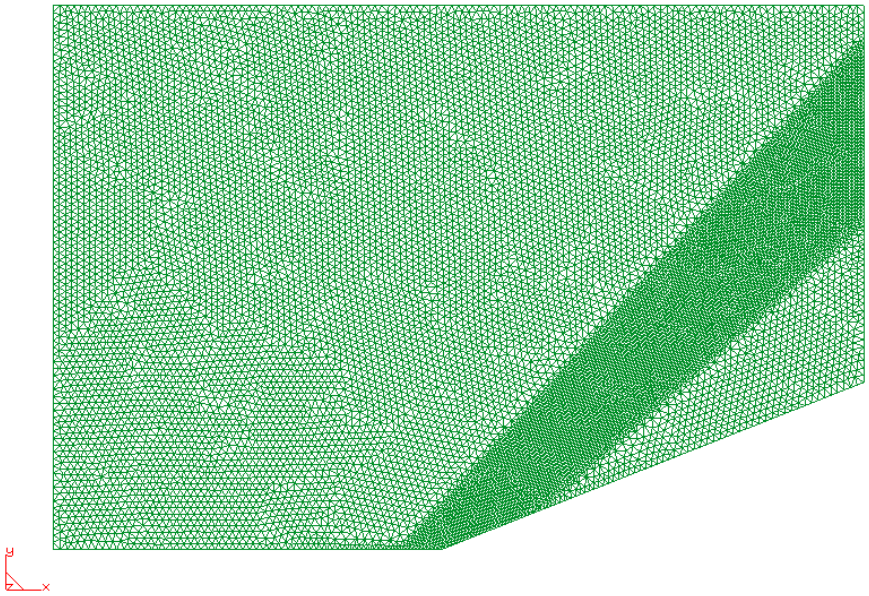


Figura 13.4: Malla de 13776 nodos y 27115 elementos triangulares lineales

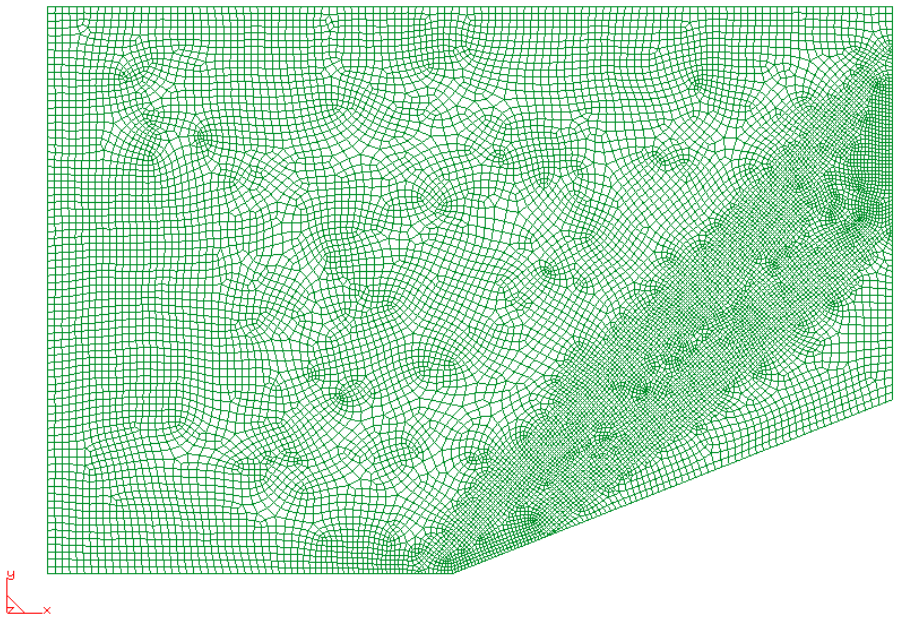


Figura 13.5: Malla de 14134 nodos y 13915 elementos cuadriláteros lineales

Para asegurar la convergencia se hace uso de un factor llamado $FSAFE$, el cual irá a multiplicar al paso de tiempo Δt para disminuir la velocidad de avance. Se eligió para ambas corridas un factor $FSAFE = 0{,}01$ para las primeras 1000 iteraciones. Posteriormente se aumentó este valor a $FSAFE = 0{,}1$ para el resto de las iteraciones.

Para el término de captura de choque es necesario obtener un coeficiente μ_a como se presentó en la sección 13.1.2. En el cálculo de este valor se puede introducir un factor entre 0 y 1 ingresado por el usuario. Se sabe que entre mayor sea este factor más precisamente se capturará la onda, pero más inestable será el resultado en la zona cercana a esta. Se eligió un factor de captura de onda de $0{,}9$, con el objetivo de obtener una onda de choque bien definida.

Otro valor importante de carácter numérico es el orden de cuadratura de Gauss que se utilizará para las integrales de ensamblaje del sistema. Se eligió $n = 2$.

Resultados

Interesa obtener 3 resultados: Ángulo de la onda de choque, distribución de número de Mach y distribución de presión.

Para el primer resultado es necesario acceder a cualquiera de las gráficas de postproceso, ya que en todas habrá una discontinuidad equivalente. Se elige por ejemplo la distribución de Mach de las figuras 13.6 y 13.7. En esta utilizando las herramientas de medición de GiD es posible comprobar que el ángulo de la onda es $\sigma = 40°$.

El número de Mach despues de la onda debería ser $M_2 = 1{,}91$ según el cálculo analítico. Es fácil comprobar en las figuras 13.6 y 13.7 que el Mach después de la onda no será exactamente constante, principalmente en el caso de cuadriláteros. No obstante esa no homogeneidad se debe a inestabilidades numéricas, y afortunadamente todos los valores se encuentran muy cerca del resultado esperado de $1{,}91$.

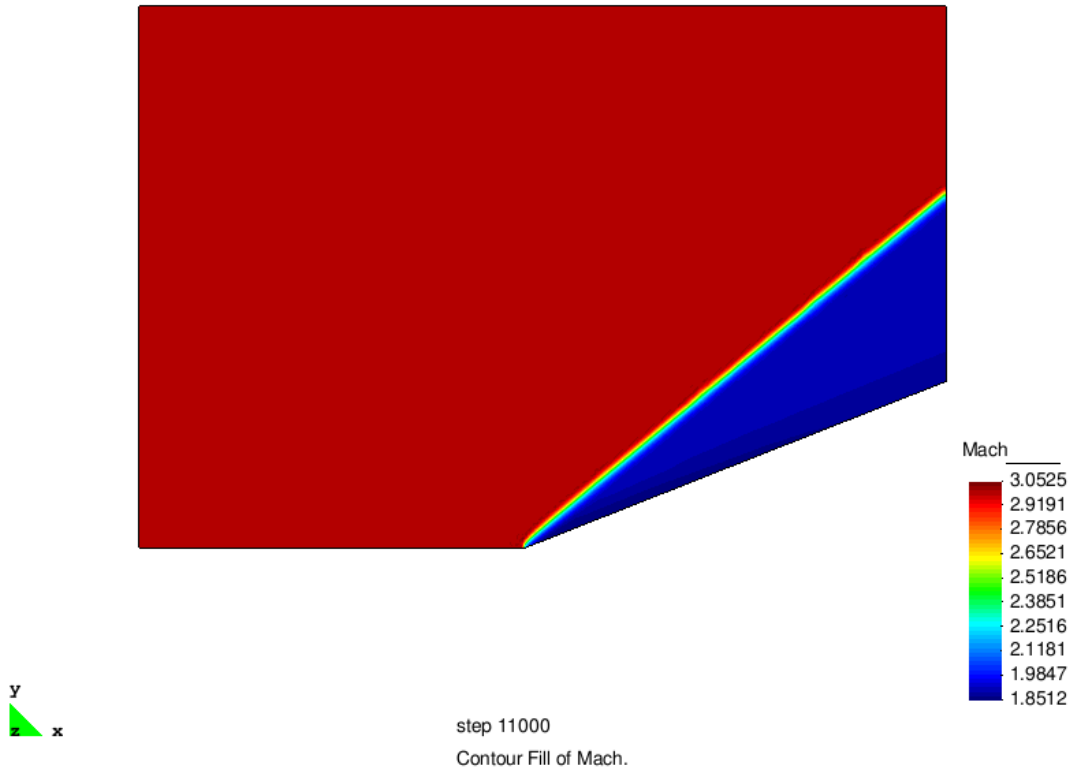


Figura 13.6: Resultados de distribución de número de Mach en malla de triángulos

El tercer resultado de interés es la presión. Analíticamente se espera que después de la onda haya una presión de $419000Pa$. Se puede ver en las figuras 13.8 y 13.9 que el valor obtenido en ambos casos es extremadamente cercano al esperado.

Se nota que con tan solo 11000 iteraciones el mallado de triángulos alcanza un resultado satisfactorio, mientras que para el mallado de cuadriláteros se usaron 201000. Quizás no eran necesarias tantas iteraciones, pero fue evidente durante las corridas que para los cuadriláteros la velocidad de convergencia era notablemente menor. Esto se debe a ineficiencias en el diseño de la aplicación, las cuales no pudieron ser identificadas por limitaciones de tiempo.

Finalmente, resultó de interés graficar la distribución de presión a lo largo de una línea horizontal del dominio. En la figura 13.10 se puede ver como se manifiesta el salto de presión. Se nota que la

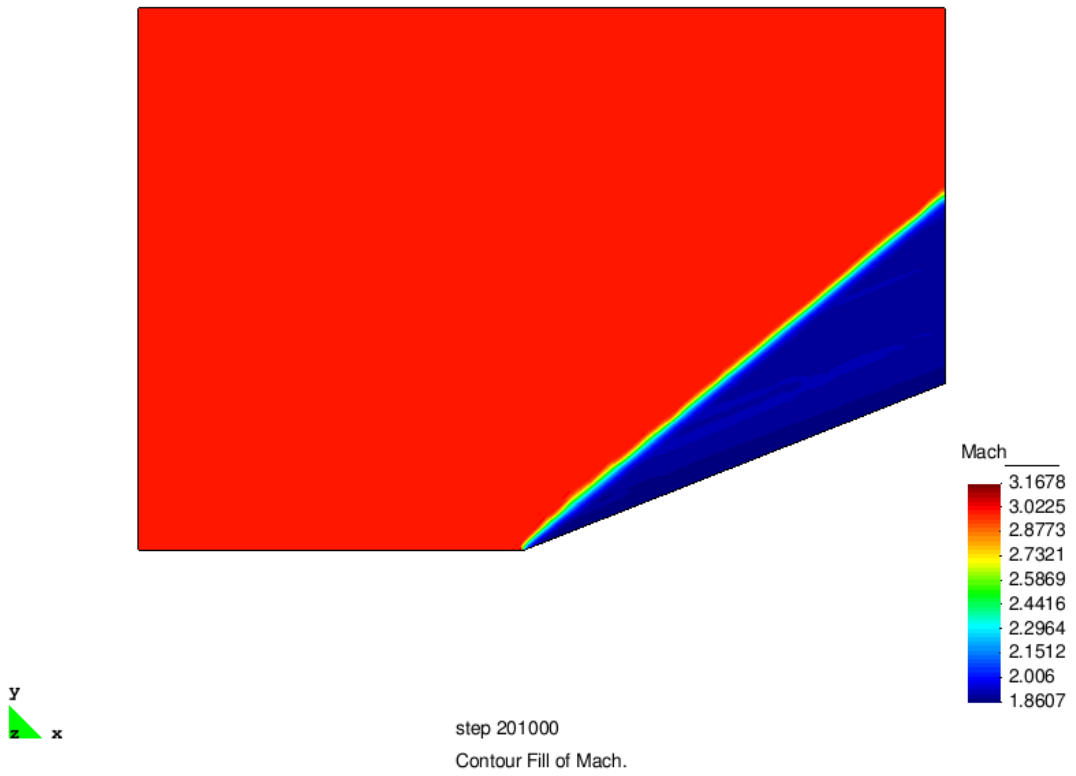


Figura 13.7: Resultados de distribución de número de Mach en malla de cuadriláteros

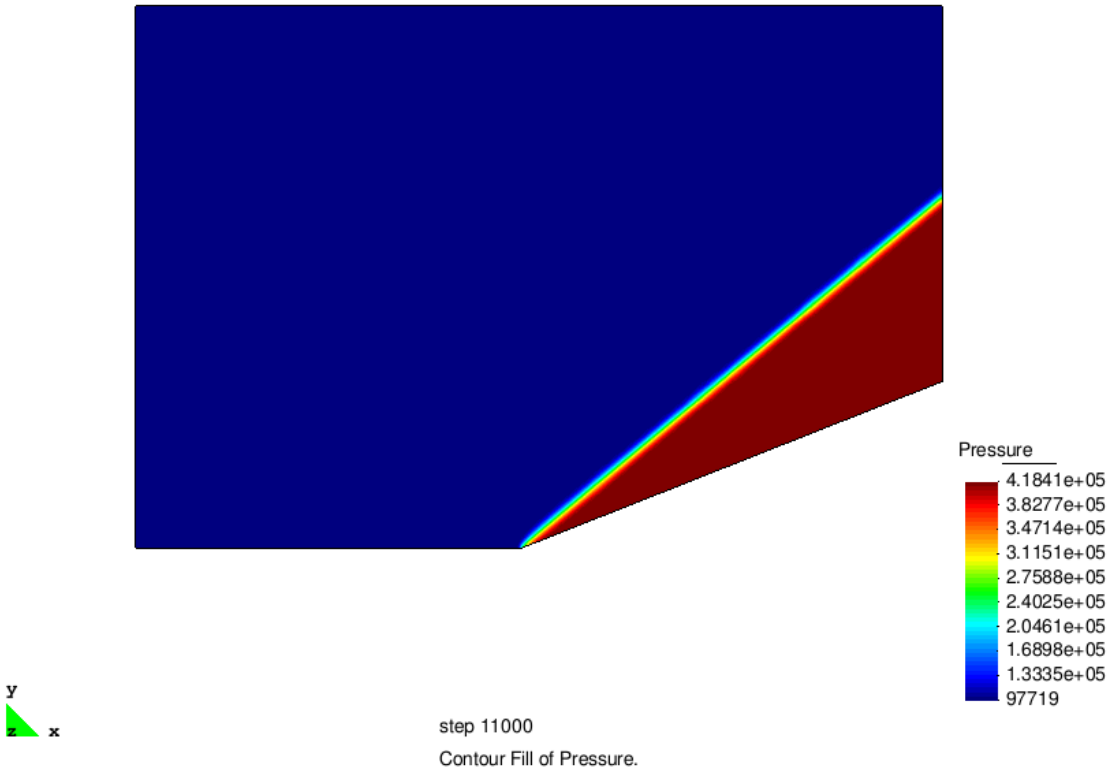


Figura 13.8: Resultados de distribución de presión en malla de triángulos

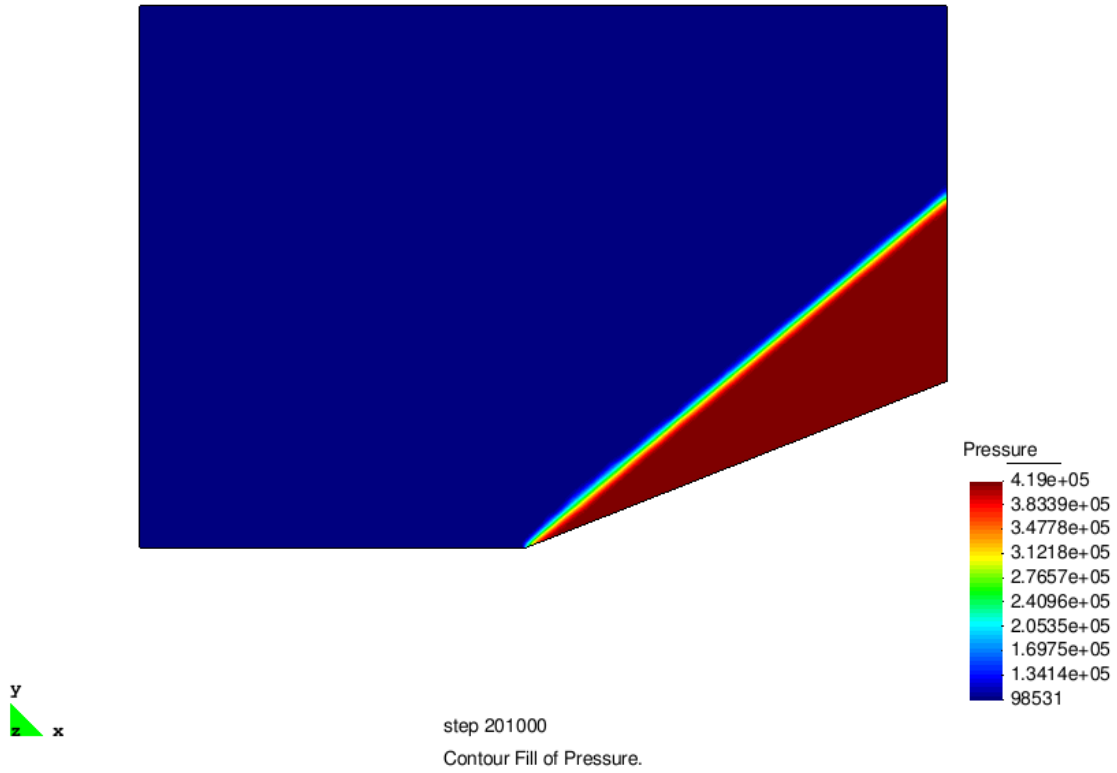


Figura 13.9: Resultados de distribución de presión en malla de cuadriláteros

discontinuidad que representa la onda de choque tiene una longitud notable, que ocupa varios elementos. También se nota un corrimiento entre la malla de triángulos y la de cuadriláteros, esto puede deberse a las irregularidades en la malla de cuadriláteros.

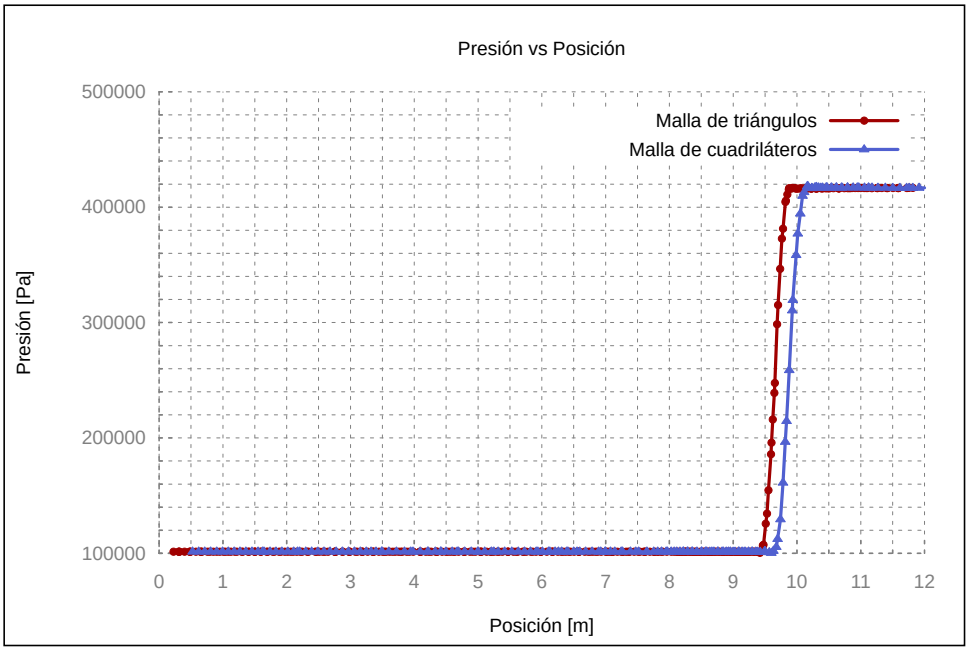


Figura 13.10: Gráfica de distribución de presión para ambas mallas

13.3.2. Perfil aerodinámico supersónico

Este ejemplo propone obtener la distribución de velocidad y presiones sobre un perfil supersónico a Mach 2.5. Las características del mismo se pueden ver en la figura 13.11. Nuevamente se realizará un cálculo análitico para luego comparar los valores obtenidos utilizando la aplicación.

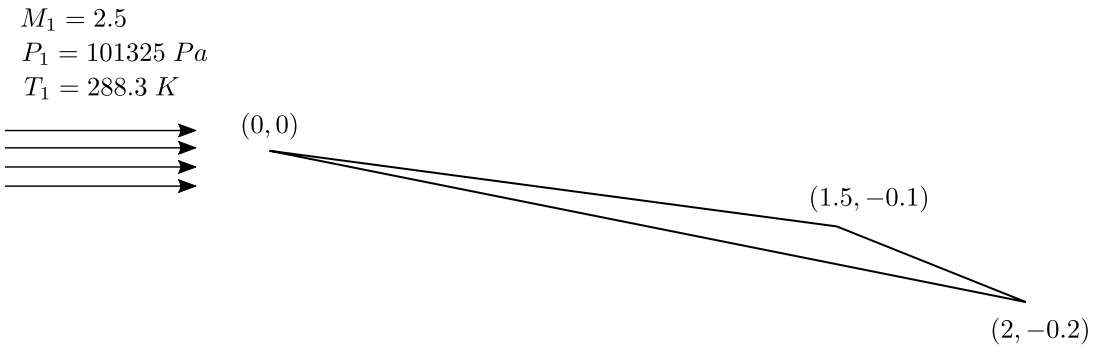


Figura 13.11: Problema de un perfil aerodinámico supersónico

Solución analítica

En este problema la incidencia de la corriente libre de aire generará sobre la geometría dos abanicos de expansión y una onda de choque oblicua, según se puede ver en la figura 13.12.

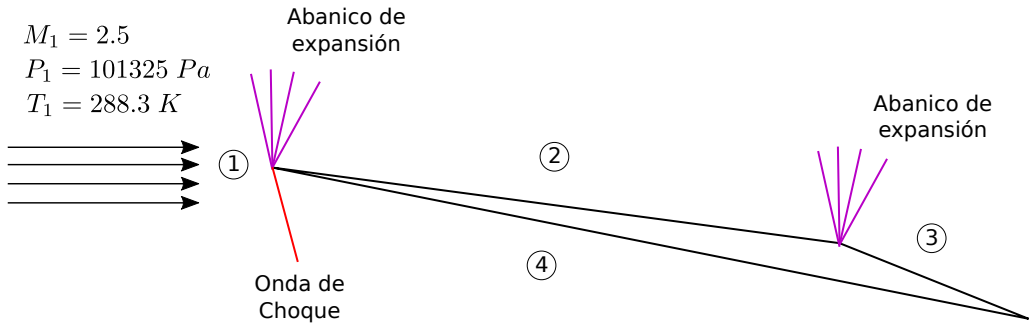


Figura 13.12: Abanicos de expansión y ondas de choque sobre el perfil

Se puede dividir el dominio en cuatro secciones adonde se espera que las velocidades y presiones serán relativamente constantes. Entonces resulta útil analizar cada pasaje de sección por separado.

Para el abanico de 1 a 2, se conoce:

$$\theta_1 = 0$$

$$M_1 = 2{,}5$$

y se calcula:

$$\theta_2 = \arctan\left(\frac{-0{,}1}{1{,}5}\right) = -3{,}814^\circ$$

Haciendo uso de la tabla de Flujo de Prandtl-Meyer se obtiene:

$$\nu_1 = 39{,}124^\circ$$

y utilizando:

$$\theta_1 + \nu_1 = \theta_2 + \nu_2 \tag{13.9}$$

se despeja:

$$\nu_2 = 42{,}938^\circ$$

con este valor se vuelve a la tabla para obtener:

$$M_2 \simeq 2{,}67$$

Finalmente con:

$$P_0 = P(1 + 0{,}2M_2^2)^{3{,}5} \tag{13.10}$$

y teniendo en cuenta que en un abanico de expansión se cumple:

$$P_{0_1} = P_{0_2} \tag{13.11}$$

se encuentra:

$$P_2 \simeq 77870\ Pa$$

Para la expansión de 2 a 3 se debe hacer un análisis análogo. ahora utilizando:

$$\theta_3 = \arctan\left(\frac{-0{,}1}{0{,}1}\right) = -11{,}31^\circ$$

Se obtiene:

$$M_3 \simeq 3{,}035$$

$$P_3 \simeq 44724\ Pa$$

Finalmente para la onda de choque de 1 a 4 se utiliza:

$$\delta = \arctan\left(\frac{0{,}2}{2}\right) = 5{,}71^\circ$$

Con lo cual realizando el mismo procedimiento que se realizó para la cuña simple se obtiene:

$$M_4 \simeq 2{,}27$$

$$P_4 \simeq 144933\ Pa$$

Solución numérica

Se debe definir la geometría e identificar las condiciones de contorno y las propiedades del material. Se puede ver la definición del problema en la figura 13.13.

Mallado

Nuevamente se eligieron dos mallas distintas, una con elementos triangulares lineales y otra con elementos cuadriláteros lineales. El tamaño de línea promedio de ambas mallas es de $0{,}04m$ en la superficie próxima al perfil y de $0{,}1m$ en el resto de la malla. Las mallas se presentan en las figuras 13.14 y 13.15.

Propiedades numéricas

De igual manera que con el problema anterior interesa definir dos valores para ajustar la estabilidad y la precisión con la cual se capturarán las discontinuidades.

Se elige $FSAFE = 0{,}01$ para las primeras 5000 iteraciones de ambos mallados. Posteriormente se toma $FSAFE = 0{,}1$ para el resto de las iteraciones. En el caso del mallado de cuadriláteros hizo falta luego setear $FSAFE = 0{,}5$ para las últimas 50000 iteraciones para acelerar la convergencia hacia un resultado estacionario.

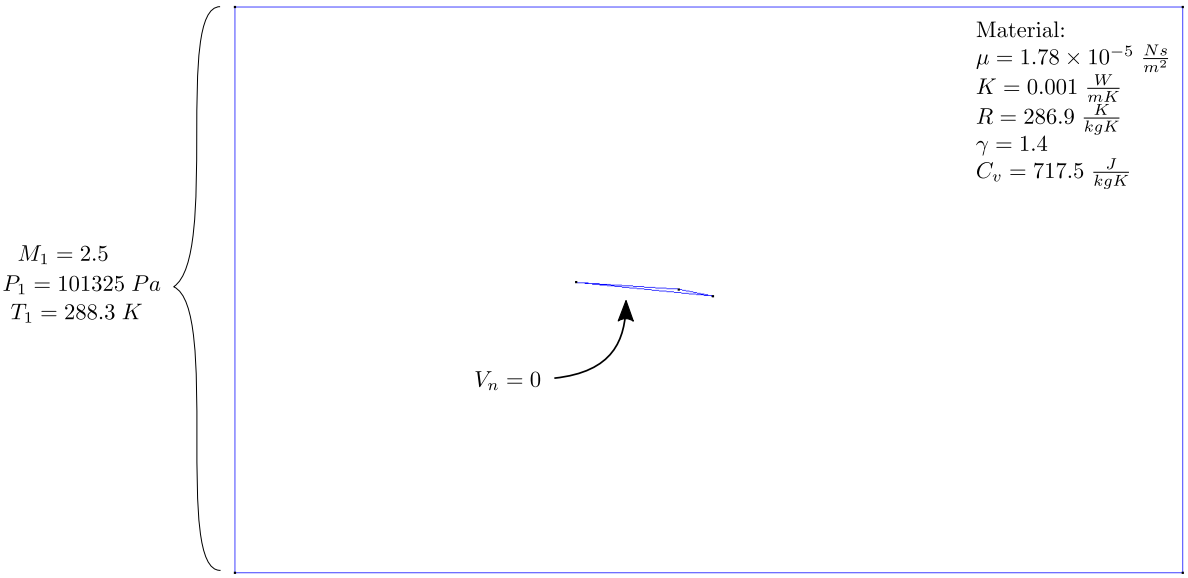


Figura 13.13: Geometría del problema y condiciones de contorno

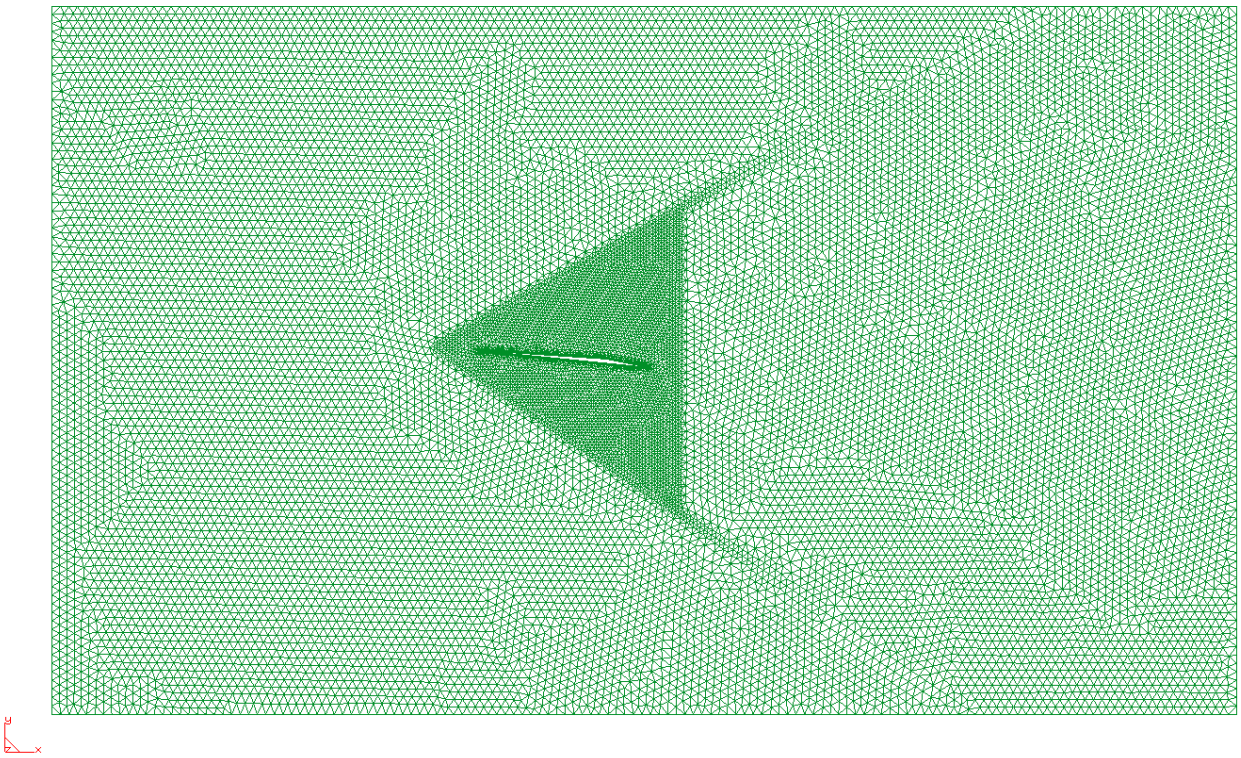


Figura 13.14: Malla de 17650 nodos y 34454 elementos triangulares lineales

Para ambos casos se toma una constante de captura de choque de $0{,}9$.
En adición se eligió un orden de cuadratura de gauss $n = 2$.

Resultados

Para este análisis solo se buscará comparar los valores de Mach M_2, M_3 y M_4 y de presión P_2, P_3 y

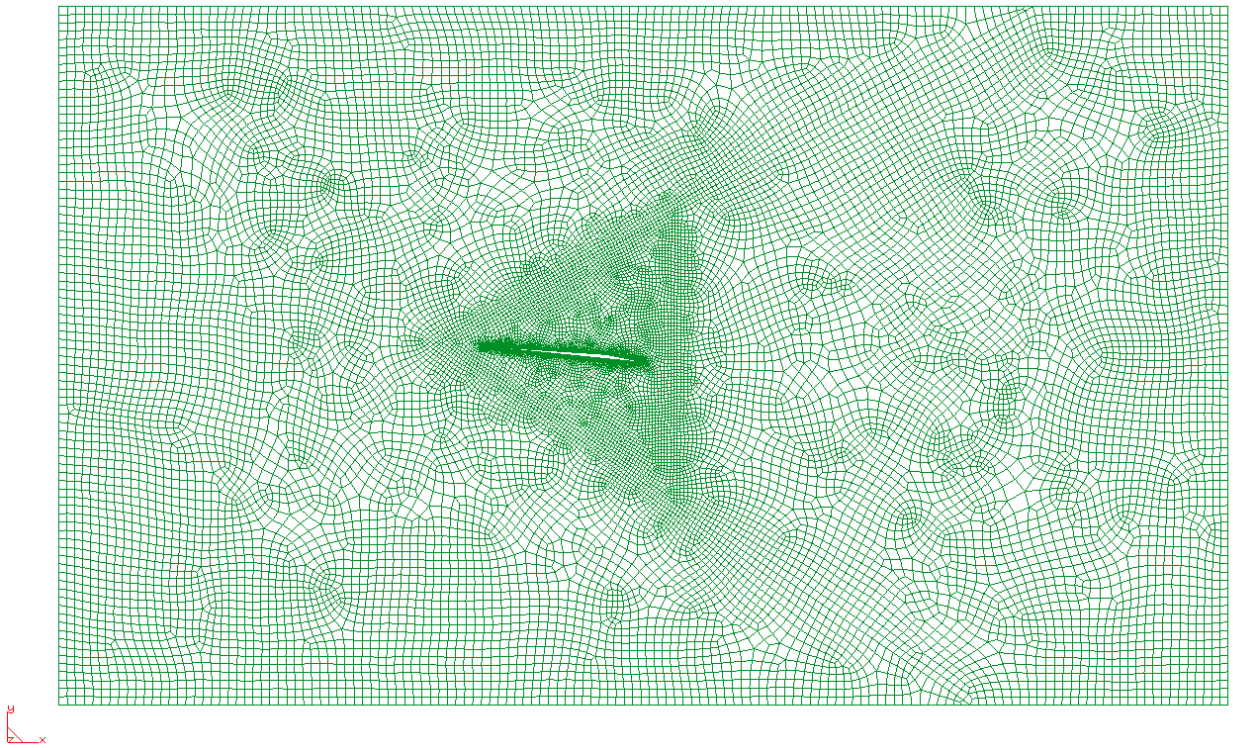


Figura 13.15: Malla de 20173 nodos y 19749 elementos cuadriláteros lineales

P_4 con los obtenidos analíticamente.

Se puede ver en las figuras 13.16, 13.17, 13.18 y 13.19 que las cuatro secciones pueden ser fácilmente diferenciables. Para asegurar que los valores obtenidos sean los esperados se tomaron valores en múltiples nodos de las distintas secciones y se promediaron sus valores. Los resultados pueden verse en las tablas 13.2 y 13.3 para el número de Mach y la presión respectivamente.

	M_2	M_3	M_4
Analítico	2,67	3,04	2,27
Triángulos	2,67	3,00	2,26
Cuadriláteros	2,66	3,00	2,26

Tabla 13.2: Comparación de valores de número de Mach en las distintas secciones del dominio

	P_2 [Pa]	P_3 [Pa]	P_4 [Pa]
Analítico	77870	44724	144933
Triángulos	78067	44000	146110
Cuadriláteros	78020	44728	146100

Tabla 13.3: Comparación de valores de presión en las distintas secciones del dominio

Se puede ver que los resultados obtenidos por **CFD2D** son extremadamente similares a los obtenidos analíticamente.

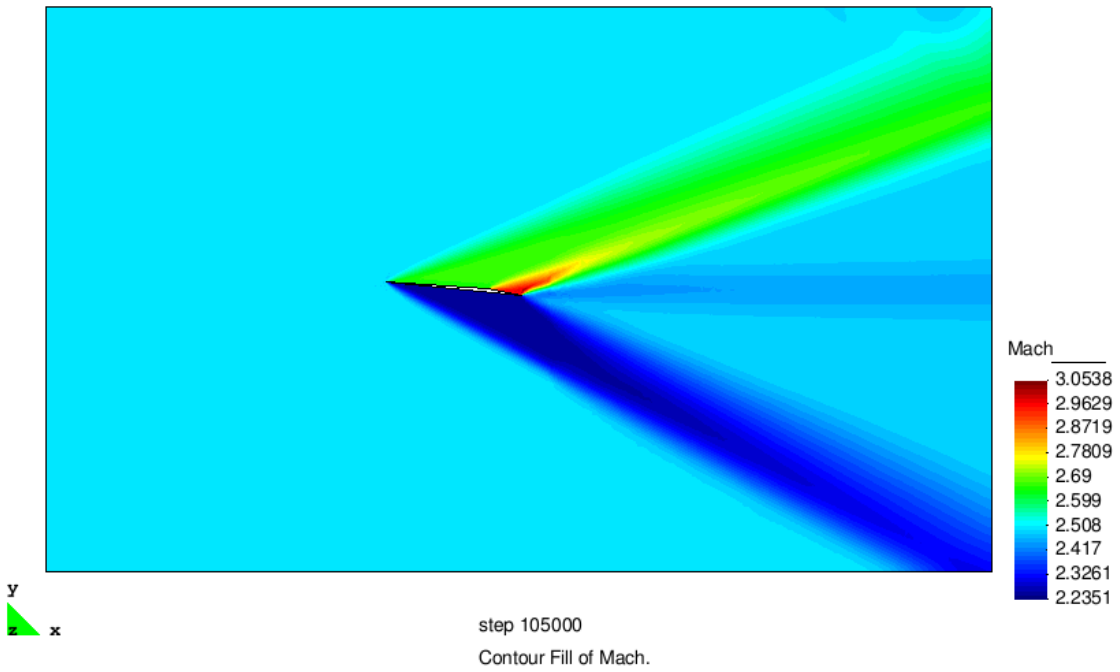


Figura 13.16: Resultados de distribución de número de Mach para la malla de triángulos

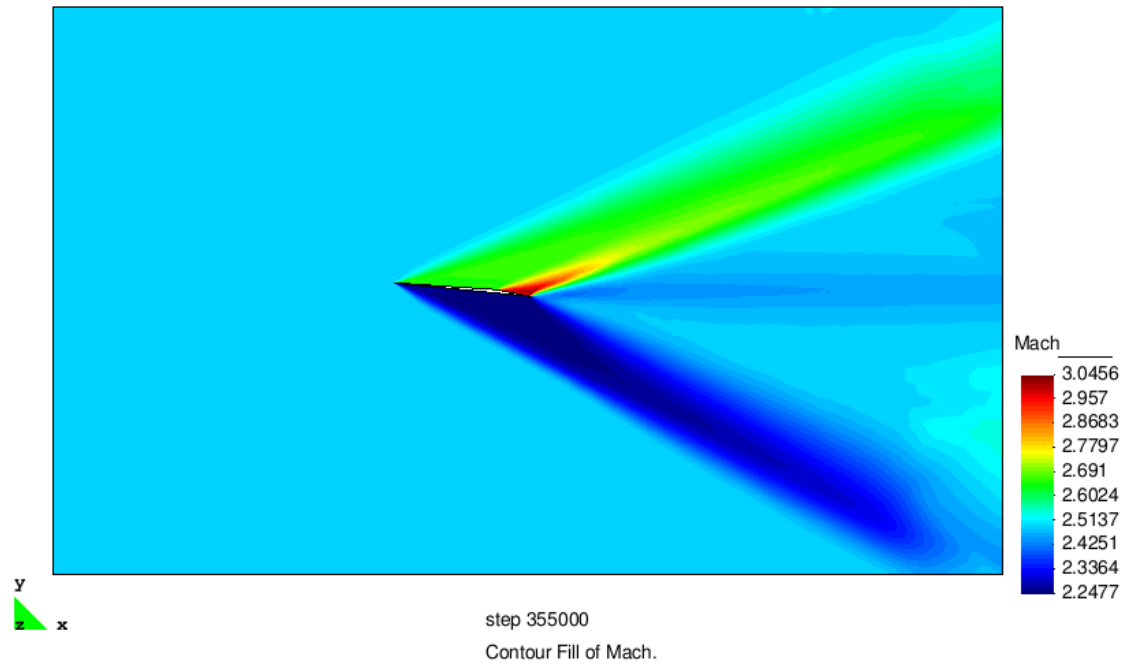


Figura 13.17: Resultados de distribución de número de Mach para la malla de cuadriláteros

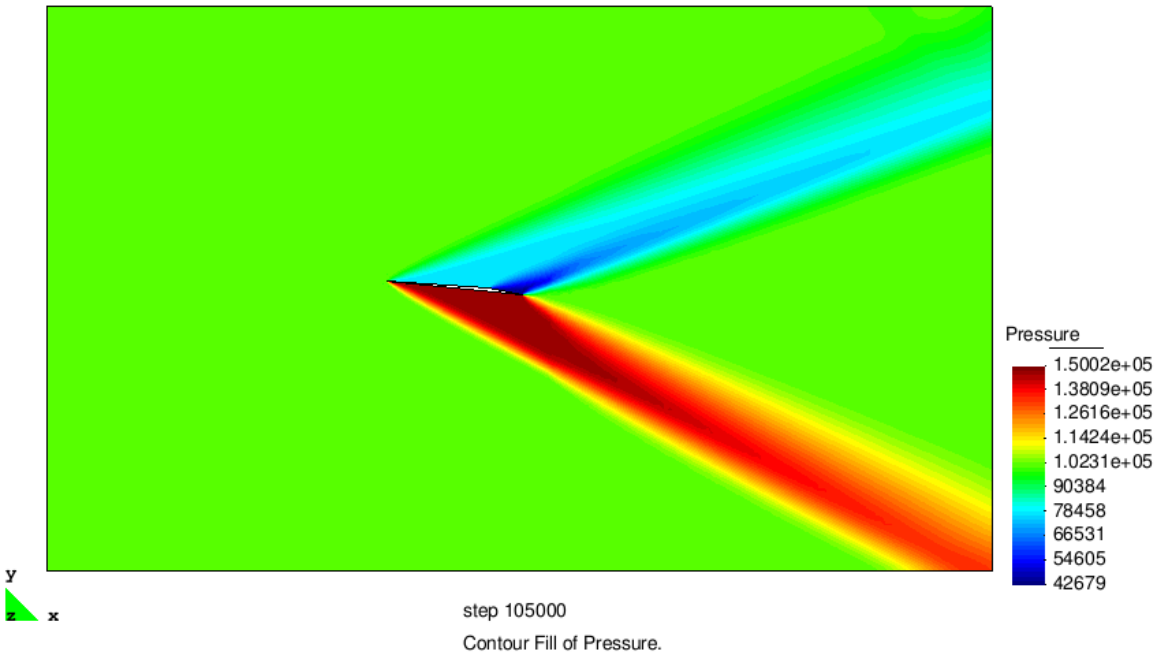


Figura 13.18: Resultados de distribución de presión para la malla de triángulos

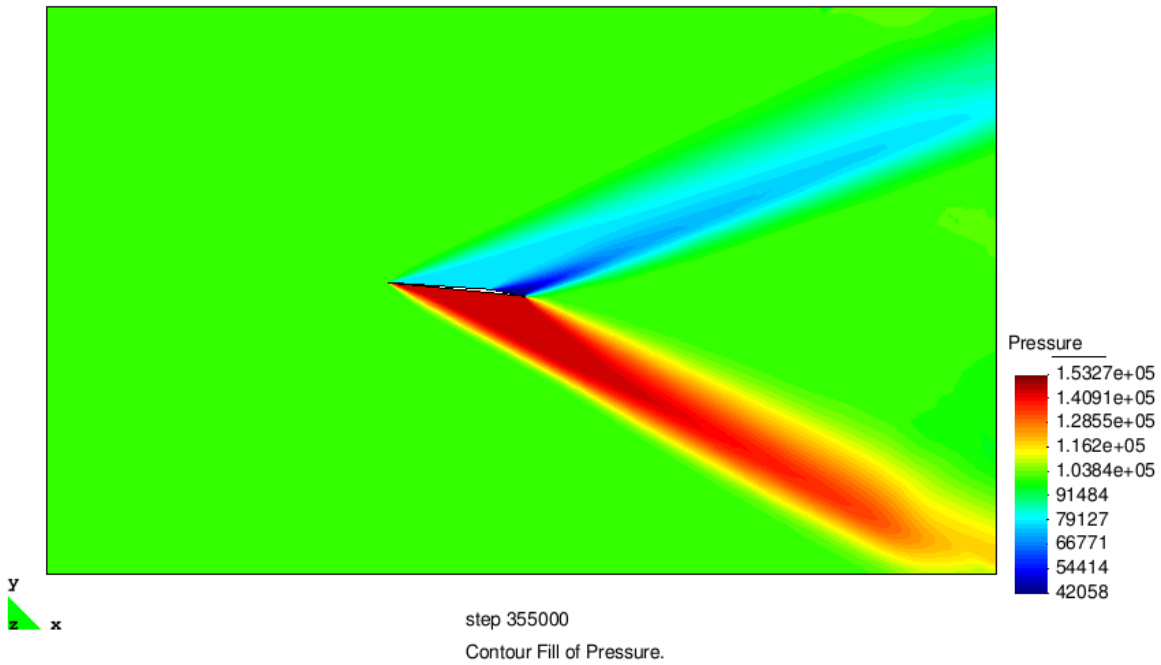


Figura 13.19: Resultados de distribución de presión para la malla de cuadriláteros

Parte IV
Conclusiones y Trabajos futuros

14. Conclusiones

La realización de este trabajo nos llevó a aplicar y profundizar distintos temas vistos durante el desarrollo de la carrera y además, nos permitió incorporar nuevos conocimientos que resultaron necesarios a medida que se avanzaba. A partir de todos estos conocimientos se logró diseñar e implementar un entorno que provee las herramientas básicas necesarias para crear una aplicación de elementos finitos y que se encuentra disponible para su uso libre en el repositorio de la referencia [24].

Crear esta framework implicó aprender valiosas lecciones referidas al mundo de la computación científica, tanto en el área del desarrollo de software como en la física aplicada a los distintos problemas que se quisieron resolver, además de todo lo referido al método de los elementos finitos en sí mismo. Desde el punto de vista del desarrollo de software fue necesario no solo incursionar en temas como el paradigma de la programación orientada a objeto aplicado al desarrollo de software y patrones de diseño, sino que también fue necesario aprender a utilizar las herramientas disponibles con las últimas versiones de Fortran para poder implementar estos temas a través de este lenguaje.

Un tema muy importante abordado en el comienzo del trabajo es el relacionado a las estructuras de almacenamiento de datos usado para matrices dispersas o `Sparse`. En este sentido se generó la herramienta `SparseKit` de la que se habló en la sección 5.1 y que también se encuentra disponible para su uso en el repositorio de la referencia [25].

A lo largo del desarrollo de este trabajo nos encontramos con múltiples obstáculos, muchos de los cuales se pudieron solucionar y otros que se tuvieron que dejar, en general, por temas de tiempo. El principal problema enfrentado fue que en un principio no se contaba con una idea clara de lo que implica desarrollar una *arquitectura de software* de elementos finitos.

Aunque es variada la bibliografía para consultar por el tema, son pocos los casos en los que se presta atención a la construcción de los programas, o las estructuras que permiten su construcción, y en general se presume que el lector tiene un cierto nivel de conocimiento, o más bien experiencia, en el tema. Se sabe que es mucha la diferencia que existe entre comprender, en principio, que hacer y hacerlo concretamente y más aún para los que no cuentan con esa experiencia necesaria.

Es por esto que para poder adquirir un poco de esta experiencia en el comienzo este trabajo consistió en la creación de múltiples programas individuales con varias herramientas en común. Sin embargo no fue hasta que no se replanteó todo el diseño, en base a lo aprendido de estos programas, que se pudo realizar una *framework* apropiada, capaz de brindar soporte al desarrollo de cualquier tipo de aplicación.

Consideramos que uno de los resultados más valiosos ha sido sin duda la experiencia y la práctica obtenidas de este trabajo. Esto nos permite tener una mirada completamente distinta del tema respecto de la que teníamos en un principio.

Pero más allá de este beneficio personal, otro de los resultados valiosos obtenido se refiere al problema mencionado antes y es que este trabajo puede ser aprovechado por tres tipos de personas interesadas en el tema. Para una persona que necesite resolver un problema en particular, sirve como una herramienta de solución si se cuenta con la aplicación necesaria; para una persona interesada en crear aplicaciones mediante el método de los elementos finitos, sirve como una framework que ofrece las herramientas necesarias y por último, para una persona interesada en desarrollar una arquitectura similar, sirve como una base o guía con un nivel de complejidad menor a las alternativas disponibles (como los ejemplos de las referencias [2] y [3]).

Las herramientas desarrolladas en este trabajo son sencillas y tienen el potencial de ser realmente útiles para emplearse en problemas reales de ingeniería. Sin embargo, es importante reflexionar sobre la calidad de las mismas y sobre cómo estas favorecen o perjudican el desarrollo de una aplicación para

poder trabajar en mejorarlas.

Seria muy simple pensar que algo es perfecto solo por el hecho de que funciona, y más sabiendo que todo lo referido a cómo resolver un problema tiene miles de soluciones posibles y todas son válidas si se obtienen los resultados correctos. De todos modos existen mejores o peores formas de realizar algo si comparamos estos resultados en base a que tan bien se cumplen los objetivos propuestos desde un principio. Por este motivo a continuación se realiza una critica de la arquitectura generada en base a que tanto se cumplen o no cada uno de los objetivos propuestos.

14.1. En cuanto a generalidad

Como ya se ha mencionado, las herramientas programadas son útiles para un gran número de aplicaciones y esto se pudo comprobar mediante las distintas aplicaciones creadas. Las estructuras usadas en forma general resultaron aquellas que permiten definir geometrías, asignar grados de libertad y fuentes, realizar integraciones numéricas y temporales, y manipular métodos para resolver sistemas.

Las herramientas mencionadas se utilizaron tanto para problemas estacionarios como para los no estacionarios y de igual modo para los casos de un grado de libertad como para los de múltiples. Sin embargo es importante notar que por más generalidad que se logre, siempre, cada aplicación nueva que se busque implementar va a traer una nueva serie de características las cuales pueden o no ser implementadas sin modificar estas estructuras.

Las aplicaciones creadas mostraron que gran parte de lo referido al ensamble de sistemas se podrían haber pensado para realizarlo de forma general ya que las implementaciones resultaron similares en todos los casos. Otro cambio posible se refiere al *input* de las aplicaciones. Es difícil generar una estructura que permita manipular el total de las variables que podrían ser ingresadas en las distintas aplicaciones pero de todas formas existen algunas que son comunes y tal vez trabajando con una estructura de almacenamiento de datos esto se podría llegar a generalizar.

14.2. En cuanto a reusabilidad

Esta es una de las principales bases de la Programación Orientada a Objetos. Permite no tener que empezar desde cero con cada nueva aplicación utilizando componentes disponibles como puntos de partida para el diseño. En esto se trabajó continuamente para diseñar clases o estructuras que resulten útiles, sin embargo también representó una dificultad el hecho de no tener muchos conocimientos previos en la programación bajo este paradigma.

Respecto a esto hay varios puntos que no han sido tenidos en cuenta entre los cuales están el `input` que ya se mencionó que debe ser implementado completamente por parte del desarrollador de la aplicación, tampoco se ha trabajado en un contenedor específico para el almacenamiento sino que se ha propuesto generar una clase `ApplicationDT` que cumpla con esta función, algo similar ocurre con la clase `SolvingStrategyDT` la cual si está disponible pero es necesario definir la totalidad de sus rutinas, las cuales resultaron similares en todas las aplicaciones. Dichas rutinas podrían estar disponibles en la framework.

14.3. En cuanto a performance

El nivel de performance logrado fue aceptable sin haber sido un objetivo principal durante el desarrollo del trabajo. Dentro de lo que se pudo evaluar en ninguno de los casos de prueba el almacenamiento de memoria fue mayor a la utilizada por un programa `ad hoc` para resolver el mismo problema. Aunque si se notó en estas pruebas un aumento en los tiempos de proceso principalmente para problemas no estacionarios, algo que en cierto modo era esperado debido al uso de la programación orientada a objeto.

Esto podría haberse mejorado utilizando técnicas de cálculo en paralelo mediante directivas OpenMP o MPI, en sus distintos niveles de aplicación, para disminuir los tiempos de ejecución. Por otro lado

también se podría intentar crear una estructura de almacenamiento de datos que forme parte de una clase contenedora, que posea rutinas de manejo de datos (referidas al almacenamiento y búsqueda) y que no permita la duplicación de información.

15. Trabajos Futuros

Son muchos los trabajos que se pueden realizar tomando como base lo logrado hasta acá. Estos se podrían organizar según el nivel de complejidad de su desarrollo en tres clases de trabajos.

Los primeros, y dentro de todo más simples, deberían estar dirigidos a realizar más casos de prueba o tests de las aplicaciones ya creadas, con el fin de evaluar las características relacionadas al manejo de memoria, tiempo de procesamiento y capacidades de paralelización de las aplicaciones.

En segundo lugar seria necesario implementar más y más complejas aplicaciones concretas para poder evaluar en mayor profundidad las características de la framework en cuanto a la flexibilidad y reusabilidad de sus componentes. Interesa principalmente desarrollar aplicaciones de análisis modal para sistemas estructurales, y continuar con CFD, implementando problemas de acople fluido-estructural los cuales involucren movimiento de malla. También se podrían realizar mejoras en las aplicaciones actuales en base a los resultados obtenidos de los tests mencionados antes.

Por último,y ya con un mayor nivel de complejidad, se pueden implementar cambios en la propia framework con el fin de simplificar y mejorar la implementación de las aplicaciones en general.

Además de los cambios descriptos, también se pueden tomar como futuros trabajos individuales la implementación de algoritmos a la framework para mejorar las herramientas disponibles, algunos de estos podrían ser algoritmos que permitan trabajar con movimiento de malla, suavizado de malla, métodos concretos de solución de sistemas lineales y no lineales, reordenadores, precondicionadores e integradores temporales.

Parte V
Referencias

16. Referencias

[1] Rainald Lohner. *Applied computational fluid dynamics techniques : an introduction based on finite element methods*. John Wiley Sons, Chichester, England Hoboken, NJ, 2008.

[2] Alberto F. Martín Santiago Badia and Javier Principe. Fempar: An object-oriented parallel finite element framework. `https://github.com/fempar`.

[3] Pooyan Dadvand. *A framework for developing finite element codes for multi-disciplinary applications*. CIMNE, Barcelona, 2007.

[4] R. J. Mackie. Object-oriented programming of the finite element method. Technical report, Woyson Bridge Research Unit, Department of Civil Engineering, The University, Dundee, 1992.

[5] Robert C Martin. *Clean code : a handbook of agile software craftsmanship*. Prentice Hall, Upper Saddle River, NJ, 2009.

[6] Dubois-Pèlerin Y. Zimmermann, T. and P. Bomme. *Object-oriented finite element programming: I. Governing principles*. Swiss Federal Institute of Technology, Switzerland, 1991.

[7] Jim Xia Rouson, Damian and Xiaofeng. *Scientific software design : the object-oriented way*. Cambridge University Press, New York, 2011.

[8] Richard; Johnson Ralph; Vlissides John Gamma, Erich; Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Assison-Wesley, Zurich, Switzerland, 1997.

[9] CIMNE. Gid. `https://www.gidhome.com/`.

[10] Wikipedia. Git. `https://es.wikipedia.org/wiki/Git`.

[11] Sergio Pissanetzky. *Sparse Matrix Technology*. Academic Press, Bariloche, Argentina, 1984.

[12] Ansys. Integration point locations. `https://www.mm.bme.hu/~gyebro/files/ans_help_v182/ans_thry/thy_et1.html`.

[13] Wikipedia. Biconjugate gradient method. `https://en.wikipedia.org/wiki/Biconjugate_gradient_method`.

[14] Intel. Intel math kernel library. `https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html`.

[15] NAFEMS. *The Standard NAFEMS Benchmarks.* National Agency for Finite Element Methods and Standards (NAFEMS), 1990.

[16] In Depth Tutorials what-when how and Information. Fem for heat transfer problems. `http://what-when-how.com/the-finite-element-method/fem-for-heat-transfer-problems-finite-element-method-part-4/`.

[17] Carnegie Mellon. Structural n°3: Analysis of a steel bracket. `https://www.andrew.cmu.edu/course/24-ansys/problems.htm`.

[18] Carnegie Mellon. Structural test n°2: Analysis of a steel bracket. https://www.andrew.cmu.edu/course/24-ansys/problems.htm.

[19] Tai-Ran Hsu. *Thermal Stress Analysis of Solid Structures Using Finite Element Method*. Department of Mechanical Engineering, San Jose State University, 2017.

[20] Carlos G. Sacco Mario A. D'Errico Germán Weht, Juan Pablo Giovacchini. Método de los elementos finitos aplicado a flujo compresible con gas en equilibrio. Technical report, Asociación Argentina de Mecánica computacional (Vol XXX, págs 547-562), 2011.

[21] Tayfun E. Tezduyar and Masayoshi Senga. Stabilization and shock-capturing parameters in supg formulation of compressible flows. Technical report, Computer Methods in Applied Mechanics and Engineering, 2006.

[22] Toro E. *Riemann Solvers and Numerical Methods for Fluids Dynamics: A Practical Introduction*. Springer, 1999.

[23] AMES Research Staff. Report 1135: Equations, tables and charts for compressible flow. Technical report, National Advisory Committe for Aeronautics, 1953.

[24] Airaudo Facundo and Zuñiga Marco. Github repository. https://github.com/ponfo/Project790.

[25] Airaudo Facundo and Zuñiga Marco. Github repository. https://github.com/ponfo/SparseKit.