Programación de un código de elementos finitos para el cálculo de estructuras.

Trabajo final de grado.

Facultad de Náutica de Barcelona
Universitat Politécnica de Cataluña

Trabajo realizado por:
Arnau Fabró López

Dirigido por:
Xavier Martinez García

Grado en Ingeniería en Sistemas y Tecnología Naval.
Barcelona, 16/06/2020
Departamento de Ciencia e ingeniería Náuticas.

Agradecimientos.

Resumen

Antecedentes y objetivos: En el campo de la ingeniería, se usan prominentemente los métodos numéricos para resolver problemas complicados. Particularmente, el método de los elementos finitos (MEF) es usado para resolver problemas gobernados por ecuaciones diferenciales en derivadas parciales (EDP).

En este proyecto, el MEF se usa para resolver problemas estructurales, la PDE que gobierna este tipo de problemas es la ecuación de la conservación de cantidad de movimiento para sólidos continuos. El objetivo principal es la programación de un código MEF que pueda resolver una variedad de problemas estructurales.

Método: Para la programación del código este se subdivide en los módulos de pre-proceso/post-proceso y el Solver. Los módulos de proceso sirven como interfaz entre el usuario y el código propiamente dicho, permite introducir datos y leer los resultados eficientemente. El módulo de proceso se genera usando el programa GiD, el Solver se escribe en Matlab.

El código permite resolver problemas 2D y 3D, usando elementos triangulares, cuadriláteros y hexaédicos.

Resultados: El objetivo principal del proyecto se ha conseguido, el código funciona de forma eficiente resolviendo varios tipos de problemas. El código se ha testado con problemas para los que se conoce la solución analítica.

Conclusiones: Uno de los logros del proyecto es conseguir un código eficiente en Matlab con bajo tiempo de computación. Esto es posible escribiendo el código en términos de operaciones matriciales, evitando al máximo usar bucles.

El código puede ser ampliado fácilmente debido a su naturaleza modular. Posibles nuevos módulos pueden implementar el calculo dinámico, nuevos elementos, plasticidad, etc.

Abstract

Background and purpose: In the engineering field, the use of numerical methods to solve complicated problems is widespread. Particularly, the finite element method (FEM) provides an efficient and versatile way to solve partial differential equations (PDE).

In this project the FEM will be used to solve static structural problems, the PDE governing this kind of problem is the conservation of momentum equation for a continuous body. The principal purpose is to write a FEM code versatile enough to solve a large variety of problems, with different boundary conditions, elements and constitutive equations.

Method: The code is subdivided between the process module and the solver module. The process module is an interface that enables the user to introduce the problem data efficiently and to interpret the results of the problem once it has been solved. This module is based on the GiD platform. The solver module is an algorithm responsible of assembling and solving the problem. This module is programmed using Matlab functions.

The code is programmed to solve 2D and 3D problems, using triangular, quadrilateral and hexahedrical elements.

Results: The principal purpose of the project is accomplished, the code works efficiently and solves a variety of problems. The code has been tested with some simple problems.

Conclusions: One of the accomplishments of this project is to achieve an efficient code in Matlab, with relatively low computation times. This is possible given that Matlab is good making matrix operations, making the code matrix oriented instead of loop oriented saves lots of time.

The code accepts the programming of new modules, capable of solving dynamic problems or plasticity. Further development of the code might solve the issues with the triangle elements module or implement new modules: Dynamic analysis, plasticity, etc.

Índice general

Índice de tablas.

Índice de figuras

Capítulo 1

Introducción:

1.1. Introducción general al MEF.

En el campo de la ingeniería y la ciencia hay una gran variedad de problemas que, debido a su magnitud o características, no se pueden resolver de forma analítica. Por esta razón se han desarrollado los métodos numéricos, que son una herramienta para obtener soluciones a estos problemas. Estos métodos suelen requerir una gran cantidad de operaciones, es por ello que a menudo se ejecutan usando ordenadores.

Dentro de los métodos numéricos utilizados para la resolución de problemas de resistencia de materiales, se destaca el método de los elementos finitos o MEF como uno de los más extendidos, este se usa tanto para resolver problemas ingenieriles como para la simulación de modelos matemáticos.

El MEF permite obtener soluciones numéricas a fenómenos físicos gobernados por ecuaciones diferenciales en derivadas parciales (EDP). Para solucionar un problema, este método discretiza el dominio sobre el que este se plantea. Con esto se logra reescribir la EDP en forma de sistema de ecuaciones algebraicas.

Este método, es idóneo para implementarse computacionalmente. El uso de un ordenador permite programar algoritmos que discreticen el dominio, generen el sistema de ecuaciones anteriormente mencionado y lo resuelvan.

Actualmente este método se usa prominentemente en la ingeniería para resolver problemas gobernados por EDP's: estructurales, fluido-dinámicos, térmicos, eléctricos,etc.

Generalmente, un código de elementos finitos se subdivide en tres partes, el Pre-proceso, el Solver y el Post-proceso.

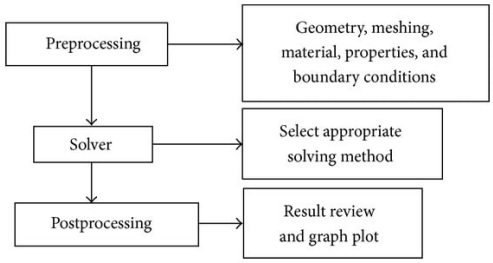


Figura 1.1: Etapas de un código MEF. De [1].

El Solver es la parte del código que realiza el análisis y los cálculos necesarios para resolver el problema. Sin embargo, la introducción de los datos de entrada y la interpretación de los datos de salida puede ser difícil, para ello se usan el Pre-proceso y el Post-proceso. Los módulos de proceso hacen de interfaz entre el usuario y el código.

1.2. Uso del MEF en la industria naval.

En la actualidad, en la industria naval el uso de estos métodos se hace cada vez más necesario debido a los nuevos retos que se presentan. En la arquitectura naval, dos de los problemas en los que más se aplica el MEF son el cálculo y optimización de la estructura y la optimización de las formas del casco.

Es tanta la importancia de estos métodos en la arquitectura naval que todas las grandes sociedades de clasificación requieren de su uso en algún momento del proyecto. Reglamentos como DNV-GL requieren y regulan el uso de estas herramientas para el cálculo estructural.

1.2.1. Guía de clase para el MEF en la industria naval: DNVGL-CG-0127.

Esta guía define como debe ser el uso del MEF en la industria naval para el cálculo estructural y es una herramienta que proporciona un manual para realizar este tipo de análisis en el proceso de diseño de un buque. Se ha tomado esta normativa como ejemplo, la mayoría de las SSCC tienen una guía de este tipo publicada.

A continuación se expone el contenido de dicha normativa:

1. Section 1: Finite element analysis. En esta sección se ofrece una introducción a la normativa y sus objetivos. También describe y clasifica diferentes tipos de MEF a usar.
2. Section 2: Global strength analysis. Esta sección se centra en análisis MEF a gran escala, donde se simula la totalidad del casco del buque. Se dan guías sobre el modelado del dominio a estudiar, el tipo y tamaño de malla adecuado y las cargas a aplicar. También se dan los criterios de análisis para este tipo de cálculo.
3. Section 3: Partial ship structural analysis. Esta sección esta enfocada a análisis en los que se simulan secciones del buque para estudiar los efectos de cargas locales (por ejemplo el efecto de la carga en un tanque). Se pueden encontrar en esta sección recomendaciones para modelar el problema, sus condiciones de borde y los criterios de análisis. Adicionalmente, se encuentran explicaciones sobre como modelar las cargas de presión tanto internas como externas. También se detalla como realizar análisis de buque viga en esta sección.
4. Section 4: Local structure strength analysis. Esta sección proporciona información para el análisis local de partes de la estructura, como por ejemplo de las cuadernas. Se puede encontrar información sobre los métodos de modelado y mallado, las zonas donde la malla debe ser más fina, el tipo de cargas a aplicar y los criterios de análisis.
5. Section 5: Beam analysis. En esta sección se muestra la información necesaria para realizar análisis de barras, que es un tipo de análisis MEF en el que se usa la teoría de vigas. Se muestran en esta sección los tipos de modelos adecuados y su aplicación, así como también los criterios de análisis.

El tipo de análisis en el que se centra este trabajo es el que se describe según DNV-GL como ”local structure strength analysis”.

Como se puede observar en la normativa, el uso de elementos tipo lámina es muy extendido en la industria naval, sobretodo para el modelado global del casco. Sin embargo, en este trabajo no se profundiza en este tipo elementos como se explicará en la siguiente sección.

1.3. Objetivos de este trabajo.

Este trabajo, se centra en el uso del MEF para el cálculo de estructuras.

El objetivo principal de este trabajo es la programación de un código de elementos finitos que permita resolver una amplia variedad de problemas estructurales. El resultado final de este trabajo por tanto es un código que realiza todos los procesos necesarios en un código MEF: Pre-proceso, Solver y Post-proceso. Concretamente, se hace hincapié en el Solver, que se programa en Matlab. El Pre-proceso y el Post-proceso se personalizará usando el soporte GiD.

Se trata de hacer que el código MEF sea lo más general posible. Algunos de los módulos que tiene el programa son: Problemas 3D, Problemas de tensión plana 2D, problemas de deformación plana 3D, discretización con elementos triangulares, discretización con elementos cuadriláteros, discretización con elementos hexaédricos, cargas puntuales, cargas de superficie, cargas volumétricas.

Se puede observar que los módulos del código desarrollado no se centran específicamente en la ingeniería naval, aunque se puede usar el código para análisis local en buques. Esto es debido a que los elementos más óptimos para análisis de buques (láminas y vigas) no son los más generales en el MEF y el trabajo se ha centrado en usar formulación MEF general. Sin embargo, se plantea el desarrollo del código de forma modular para que se puedan implementar funciones adicionales como por ejemplo el uso de elementos tipo lámina.

Capítulo 2

Bases teóricas:

2.1. Ecuación de gobierno en el MEF.

Para este capítulo se han tomado como referencia los siguientes documentos: [9],[10],[11].

Como se ha explicado anteriormente, el MEF discretiza la ecuación que gobierna el problema para poder resolverlo. En la resolución de problemas estructurales, la ecuación que se discretiza es el principio de los trabajos virtuales (PTV).

2.2. Principio de los trabajos virtuales.

Esta ecuación se obtiene al reescribir la ecuación de la conservación de la cantidad de movimiento en lo que se conoce como su formulación débil o integral. Adicionalmente, es necesario el uso de algunas ecuaciones constitutivas (Ley de Hooke) para obtener el PTV en función de las deformaciones unitarias: $\bar{\bar{\epsilon}}$.

Ecuación de la conservación de la cantidad de movimiento:

$$\boldsymbol{\nabla} \cdot \bar{\bar{\sigma}} + \bar{f} = \rho(\frac{d\bar{v}}{dt} + (\nabla(\bar{v}))\bar{v}) \tag{2.1}$$

Ley de Hooke:

$$\bar{\bar{\sigma}} = C\bar{\bar{\epsilon}} \tag{2.2}$$

Deformación unitaria para elasticidad lineal:

$$\bar{\bar{\epsilon}} = \frac{1}{2}(\nabla(\bar{u}) + \nabla(\bar{u})^T) \tag{2.3}$$

Donde:

$\bar{\bar{\sigma}}$ es el campo de tensiones sobre el dominio estudiado.

$\bar{\bar{\epsilon}}$ es el campo de deformaciones unitarias sobre el dominio estudiado.

$\bar{a}$ es el campo de deformaciones sobre el dominio estudiado: (u,v,w).

$\bar{v}$ es el campo de velocidades sobre el dominio estudiado: (v_x, v_y, v_z).

ρ es el campo de densidades sobre el dominio estudiado.

$\bar{f}$ son las fuerzas externas aplicadas sobre el dominio estudiado.

C es la matriz de rigidez del material.

2.2.1. Ley de gobierno para análisis estático:

Para análisis estático, se supone que las deformaciones ocurren de forma muy lenta. Es decir, las velocidades son despreciables y se estudia el problema de forma estacionaria. Luego, la ecuación 2.1 se puede reescribir como:

$$\boldsymbol{\nabla} \cdot \bar{\bar{\sigma}} + \bar{f} = 0 \tag{2.4}$$

2.2.2. Deducción de la formulación débil:

Para expresar la ecuación 2.3 en su formulación débil, se multiplican ambos lados por $\partial\bar{a}$:

$$(\boldsymbol{\nabla} \cdot \bar{\bar{\sigma}} + \bar{f}) \cdot \partial\bar{a} = 0 \tag{2.5}$$

Donde $\partial\bar{a}$ es el denominado desplazamiento virtual. Este es un desplazamiento cinemáticamente admisible, es decir, que cumple las condiciones de borde aplicadas al problema.

A continuación se integra la ecuación sobre el volumen del cuerpo deformable que se quiere estudiar:

$$\int_V \boldsymbol{\nabla} \cdot \bar{\bar{\sigma}} \cdot \partial\bar{a}\ dv + \int_V \bar{f} \cdot \partial\bar{a}\ dv = 0 \tag{2.6}$$

Se usan las siguientes propiedades para reescribir la ecuación 2.6:

$$\boldsymbol{\nabla} \cdot (\bar{\bar{\sigma}}\partial\bar{a}) = \partial\bar{a} \cdot \boldsymbol{\nabla} \cdot \bar{\bar{\sigma}}^T + Tr(\bar{\bar{\sigma}}\nabla\partial\bar{a}) \tag{2.7}$$

$$Tr(\bar{\bar{\sigma}}^T\nabla\partial\bar{a}) = \bar{\bar{\sigma}}^T : \nabla\partial\bar{a} \tag{2.8}$$

$$\bar{\bar{\sigma}}^T = \bar{\bar{\sigma}} \tag{2.9}$$

Que queda expresada así:

$$\int_V \boldsymbol{\nabla} \cdot \bar{\bar{\sigma}} \cdot \partial\bar{a}\ dv - \int_V \bar{\bar{\sigma}} : \nabla\partial\bar{a}\ dv + \int_V \bar{f} \cdot \partial\bar{a}\ dv = 0 \tag{2.10}$$

Finalmente, el teorema de la divergencia y la ley de Cauchy ($\bar{t} = \bar{\bar{\sigma}}\bar{n}$) permiten escribir el PTV de la siguiente forma:

$$\int_S \bar{t} \cdot \partial\bar{a}\ ds + \int_V \bar{f} \cdot \partial\bar{a}\ dv = \int_V \bar{\bar{\sigma}} : \nabla\partial\bar{a}\ dv \tag{2.11}$$

2.2.3. PTV en función de la deformación unitaria:

Para escribir el PTV en función de $\bar{\bar{\epsilon}}$, se descompone $\nabla\partial\bar{u}$ en su parte simétrica y su parte antisimétrica:

$$\nabla\partial\bar{a} = \partial\bar{\bar{\epsilon}}(simétrica)+\partial\bar{\bar{\omega}}(antisimétrica) = \frac{1}{2}(\nabla\partial\bar{a}+\nabla\partial\bar{a}^T)+\frac{1}{2}(\nabla\partial\bar{a}-\nabla\partial\bar{a}^T) \tag{2.12}$$

Observando que al contraer un tensor simétrico (σ) con uno antisimétrico ($\partial\omega$) el resultado es igual a 0 obtenemos:

$$\bar{\bar{\sigma}} : \nabla\partial\bar{a} = \bar{\bar{\sigma}} : \partial\bar{\bar{\epsilon}} \tag{2.13}$$

Finalmente:

$$\int_S \bar{t}\cdot\partial\bar{a}\ ds + \int_V \bar{f}\cdot\partial\bar{a}\ dv = \int_V \bar{\bar{\sigma}} : \partial\bar{\bar{\epsilon}}\ dv \tag{2.14}$$

2.2.4. Ecuaciones constitutivas:

En la ecuación del PTV no queda expresado explícitamente uno de los resultados que se pretende obtener, el campo de deformaciones $\bar{u}$. Para introducirlo en la ecuación es necesario el uso de una relación constitutiva, para el caso de la elasticidad lineal esta relación es la ley de Hooke (ecn. 2.2), que relaciona tensión con la deformación unitaria. También es necesaria una forma de expresar la deformación unitaria en función de la deformación, para deformaciones infinitesimal se usa la ecn 2.3.

En notación ingenieril los tensores $\bar{\bar{\sigma}}$ y $\bar{\bar{\epsilon}}$ se reescriben como vectores, esto es posible debido a que ambos son tensores simétricos.

$$\bar{\sigma} = \begin{bmatrix} \sigma x \\ \sigma y \\ \sigma z \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{xz} \end{bmatrix} \quad \bar{\epsilon} = \begin{bmatrix} \epsilon x \\ \epsilon y \\ \epsilon z \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{xz} \end{bmatrix} \tag{2.15}$$

En el caso 2D estos vectores son vectores 3x3, ya que las variables no dependen de la coordenada z.

Escritos de forma tensorial, la relación entre $\bar{\bar{\sigma}}$ y $\bar{\bar{\epsilon}}$ es un tensor de cuarto orden. Sin embargo, en notación ingenieril, se relacionan mediante una matriz, la matriz de rigidez del material (C).

Para problemas en tres dimensiones y para un material isótropo:

$$C = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{(1-2\nu)}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{(1-2\nu)}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{(1-2\nu)}{2} \end{bmatrix} \tag{2.16}$$

Para problemas de tensión plana y para un material isótropo:

$$C = \frac{E}{(1-\nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{(1-\nu)}{2} \end{bmatrix} \tag{2.17}$$

Para problemas de deformación plana y para un material isótropo:

$$C = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{(1-\nu)}{2} \end{bmatrix} \tag{2.18}$$

2.2.5. Discretización del PTV según el MEF:

Una vez la ecuación de gobierno del problema está escrita en su formulación débil (PTV) se puede resolver mediante el MEF.

Para ello, se discretiza el dominio a estudiar. Las particiones que resultan de ello se denominan elementos. Cada elemento tiene un número fijo de nodos. La complejidad del elemento aumenta con el número de nodos, así como su capacidad de aproximar mejor la solución del problema.

El tipo de elemento usado depende del problema a estudiar, para problemas 2D se pueden usar elementos triangulares, cuadriláteros, etc. Para problemas 3D, elementos hexaédricos, tetraédricos, etc.

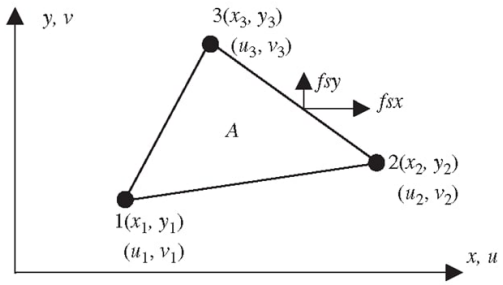


Figura 2.1: Ejemplo: Elemento triangular lineal. De [2].

De esta forma el campo de deformaciones se evalúa solo en los nodos, y se interpola dentro del elemento mediante lo que se denomina una función base o función de forma. Las funciones de forma pueden ser lineales, como las que se usan en este trabajo, o pueden ser de mayor orden para obtener más exactitud en los resultados.

El uso de las funciones de forma permite resolver el PTV sobre cada uno de los elementos, de tal forma que las únicas incógnitas sean los valores de los desplazamientos en los nodos. Finalmente, se pueden ensamblar el resultado obtenido de cada elemento para obtener las ecuaciones sobre el dominio completo del problema a estudiar.

El proceso anteriormente explicado se detalla a continuación:

Se parte del PTV, se ha añadido la contribución de las posibles fuerzas puntuales ($\bar{P}$) aplicadas sobre nodos. También se ha reescrito el principio usando la forma vectorial de $\bar{\bar{\sigma}}$ y $\bar{\bar{\epsilon}}$.

$$\int_S \partial\bar{a}^T \bar{t}\, ds + \int_V \partial\bar{a}^T \bar{f}\, dv + \sum \partial\bar{a}^T \bar{P} = \int_V \partial\bar{\epsilon}^T \bar{\sigma}\, dv \tag{2.19}$$

Para facilitar la integración de las funciones de forma, estas se escriben en función de un sistema de coordenadas natural (r,s,t), es decir uno que tiene su punto (0,0,0) en el centroide del elemento usado.

Usando las funciones de forma (N_i) el campo de desplazamientos queda definido por su valor en los nodos del elemento de la siguiente forma:

$$u = \sum_{i=1}^{n} N(r,s,t)_i u_i; \quad v = \sum_{i=1}^{n} N(r,s,t)_i v_i; \quad w = \sum_{i=1}^{n} N(r,s,t)_i w_i \tag{2.20}$$

El campo de desplazamientos virtuales se representa de la misma manera:

$$\partial u = \sum_{i=1}^{n} N(r,s,t)_i \partial u_i; \quad \partial v = \sum_{i=1}^{n} N(r,s,t)_i \partial v_i; \quad \partial w = \sum_{i=1}^{n} N(r,s,t)_i \partial w_i \tag{2.21}$$

Donde n es el número de nodos en el elemento tratado. El valor de las funciones de forma (N_i) depende del tipo de elemento y se tratará más adelante.

Sus principales propiedades son las siguientes:

$$\sum_{i=1}^{n} N(r,s,t)_i = 1; \quad N(r,s,t)_i = \begin{cases} 1 & \text{En el nodo i.} \\ 0 & \text{En cualquier otro nodo.} \end{cases} \tag{2.22}$$

A continuación, para obtener $\partial\bar{\epsilon}$ se usa la definición 2.3. Se escribe $\partial\epsilon$ en función de $\partial\bar{a}$ (desplazamiento virtual en los nodos del elemento) mediante lo que se denomina la matriz de deformación (B). La matriz B depende del elemento y las funciones de forma usadas, su descripción exacta se hará más adelante en este trabajo.

$$\partial\bar{\epsilon} = \bar{\bar{B}}\bar{\partial a} \tag{2.23}$$

Se escribe $\bar{\epsilon}$ de la misma forma:

$$\bar{\epsilon} = \bar{\bar{B}}\bar{a} \tag{2.24}$$

Para calcular esta matriz B es necesario derivar las funciones de forma respecto a las coordenadas generales del problema (x,y,z), pero estas están escritas en función de las coordenadas naturales (r,s,t).

Por tanto, para poder derivar N(x,y,z) se usa la regla de la cadena:

$$\frac{d}{dx} = \frac{d}{dr}\frac{dr}{x} + \frac{d}{ds}\frac{ds}{dx} + \frac{d}{dt}\frac{dt}{dx} \tag{2.25}$$

Esto se puede escribir de forma más compacta y para todas las derivadas de la siguiente forma:

$$\frac{\bar{d}}{dx} = \bar{\bar{J}}^{-1}\frac{\bar{d}}{dr} \tag{2.26}$$

Donde J es el operador Jacobiano, definido de la siguiente forma:

$$\bar{\bar{J}} = \begin{bmatrix} \frac{dx}{dr} & \frac{dy}{dr} & \frac{dz}{dr} \\ \frac{dx}{ds} & \frac{dy}{ds} & \frac{dz}{ds} \\ \frac{dx}{dt} & \frac{dy}{ds} & \frac{dz}{dt} \end{bmatrix} \tag{2.27}$$

Para evaluar J es necesario interpolar la geometría de manera similar a como se ha hecho con los desplazamientos. Se ponen las coordenadas dentro del elemento en función del valor de estas en los nodos.

En el caso tratado, se usan las mismas funciones de forma que se han usado para interpolar los desplazamientos. A los elementos tratados de esta forma se les llama elementos isoparamétricos.

$$x = \sum_{i=1}^{n} N(r,s,t)_i x_i; \quad y = \sum_{i=1}^{n} N(r,s,t)_i y_i; \quad z = \sum_{i=1}^{n} N(r,s,t)_i z_i \tag{2.28}$$

Una vez introducido el uso de las funciones de forma, se puede evaluar el PTV en el dominio de un elemento usando las identidades 2.2, 2.19, 2.23, 2.24. La ecuación resultante toma forma de sistema de ecuaciones algebraicas, usualmente este se escribe en forma matricial.

$$\partial \bar{a}^T \left(\int_S N^T \bar{t}\, ds + \int_V N^T \bar{f}\, dv + \sum N^T \bar{P} \right) = \partial \bar{a}^T \left(\int_V B^T C B\, dv \right) \bar{a} \tag{2.29}$$

Debido a que las funciones de forma están en términos de las coordenadas naturales, también es necesario el uso de la siguiente propiedad del operador Jacobiano para poder evaluar las integrales:

$$\frac{dV}{dv} = detJ \tag{2.30}$$

$$\frac{dS}{ds} = detJ_s \tag{2.31}$$

Donde dv es el volumen infinitesimal en el espacio de las coordenadas generales (x,y,z) y dV es el volumen infinitesimal en el espacio de las coordenadas naturales (r,s,t). Para $\frac{dS}{ds}$ existe una relación similar, donde J_s es un Jacobiano 2D que se construye con las coordenadas locales de la superficie tratada.

$$\partial\bar{a}^T(\int_S N^T\bar{t}detJ\ dS+\int_V N^T\bar{f}detJ\ dV+\sum N^T\bar{P}) = \partial\bar{a}^T(\int_V B^TCBdetJ\ dV)\ \bar{a} \tag{2.32}$$

Finalmente, debido a que $\partial\bar{a}^T$ puede tomar cualquier valor sujeto a las condiciones de borde:

$$\int_S N^T\bar{t}detJ\ dS + \int_V N^T\bar{f}detJ\ dV + \sum N^T\bar{P} = (\int_V B^TCBdetJ\ dV\ )\bar{a} \tag{2.33}$$

O en forma matricial:

$$\bar{f}_e = \bar{\bar{K}}_e\ \bar{a} \tag{2.34}$$

Donde $\bar{\bar{K}}_e$ es la matriz de rigidez del elemento, $\bar{a}$ es el vector de desplazamientos nodales y $\bar{f}_e$ es el vector de fuerzas que actúa sobre el elemento.

Para resolver el problema completo, es necesario ensamblar cada una de las matrices de rigidez elementales en una matriz de rigidez global. Este proceso de ensamblaje se ilustra a continuación mediante un ejemplo.

Ejemplo: Construcción de la matriz de rigidez global de un problema 1D con 3 elementos tipo barra.

Con este ejemplo se trata de mostrar como mediante la ecuación de equilibrio de un elemento se puede construir la ecuación de equilibrio de toda la estructura.

Para mayor simplicidad se usa un problema de cálculo matricial de estructuras (que usa la formulación de la resistencia de materiales), que sigue el mismo procedimiento de ensamblaje que un problema MEF.

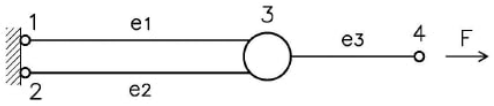


Figura 2.2: Ejemplo 1: Problema con tres elementos 1D tipo barra. De [3].

En la siguiente Figura se muestra la notación utilizada para el desplazamiento de los nodos (u_i) y la fuerza aplicada en los mismos (n_i):

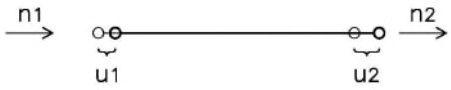


Figura 2.3: Elemento 1D, barra entre rótulas. De [4].

La relación entre fuerzas y desplazamientos de los nudos de cada uno de los elementos se puede escribir mediante el siguiente sistema de ecuaciones.:

$$\begin{bmatrix} n_1 \\ n_2 \end{bmatrix} = \begin{bmatrix} \frac{EA}{L} & -\frac{EA}{L} \\ -\frac{EA}{L} & \frac{EA}{L} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{2.35}$$

Para simplificar se reescribe:

$$\begin{bmatrix} n_1 \\ n_2 \end{bmatrix} = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{2.36}$$

A continuación, se ensambla cada una de las matrices de rigidez elementales en la matriz de rigidez global. Para ello se substituye la numeración local por la global.

En el elemento 1: El nodo 1 (elemental) corresponde al nodo 1 (global) y el nodo 2 (elemental) corresponde al nodo 3 (global).

Su matriz de rigidez es:

$$\begin{bmatrix} n_1 \\ n_3 \end{bmatrix} = \begin{bmatrix} K(1)_{11} & K(1)_{13} \\ K(1)_{31} & K(1)_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_3 \end{bmatrix} \tag{2.37}$$

Donde K(e) es la rigidez del elemento e.

Finalmente, la matriz de rigidez global queda escrita de la siguiente forma:

$$\begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ n_4 \end{bmatrix} = \begin{bmatrix} \sum_{e=1}^{3} K(e)_{11} & \sum_{e=1}^{3} K(e)_{13} & \sum_{e=1}^{3} K(e)_{13} & \sum_{e=1}^{3} K(e)_{13} \\ \sum_{e=1}^{3} K(e)_{21} & \sum_{e=1}^{3} K(e)_{22} & \sum_{e=1}^{3} K(e)_{23} & \sum_{e=1}^{3} K(e)_{24} \\ \sum_{e=1}^{3} K(e)_{31} & \sum_{e=1}^{3} K(e)_{32} & \sum_{e=1}^{3} K(e)_{33} & \sum_{e=1}^{3} K(e)_{34} \\ \sum_{e=1}^{3} K(e)_{41} & \sum_{e=1}^{3} K(e)_{42} & \sum_{e=1}^{3} K(e)_{43} & \sum_{e=1}^{3} K(e)_{44} \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ n_4 \end{bmatrix} \tag{2.38}$$

Se puede deducir que la matriz de rigidez será siempre una matriz cuadrada de columnas y filas con tamaño nº dimensiones por nº nodos.

Una vez el sistema queda escrito de esta forma, el problema se puede resolver. Para ello, antes es necesaria la introducción de las condiciones de borde, como en cualquier problema gobernado por una ecuación diferencial. Tras obtener el vector desplazamientos $\bar{a}$ se obtienen los desplazamientos con las ecuaciones 2.24 y 2.2.

Capítulo 3

Descripción general del código:

3.1. Programación de un código MEF.

Como ya se ha comentado, el MEF es un método idóneo para ser programado en un ordenador. Programar el método en un algoritmo permite resolver gran cantidad de problemas complejos que, sin el uso de un ordenador, tendrían demasiadas operaciones.

3.2. Pre-proceso y Post-proceso.

Sin Pre-proceso y Post-proceso la resolución de problemas también se complica. Esto es debido a que introducir los datos iniciales e interpretar los resultados obtenidos puede ser difícil sin este módulo.

Para la creación de este módulo se usa GiD, un programa desarrollado por el Centro Internacional de Métodos Numéricos en la Ingeniería (CIMNE). Este programa proporciona una base sobre la que crear los módulos de proceso, para ello es necesario personalizar GiD creando lo que se denomina un ”Problemtype”.

Pre-proceso:

El Pre-proceso se realiza previamente a la resolución del problema. Permite plantear el problema de forma gráfica (dibujando la geometría) e introducir fácilmente las condiciones de borde (cargas y zonas fijadas) y los datos (constantes del material). Otra funcionalidad muy importante del preproceso es la generación de la malla (subdivisión del dominio en elementos), generar la malla manualmente puede ser ineficiente para algunos problemas y supone un coste de tiempo importante, es por ello que poder mallar el dominio fácilmente es una gran ventaja.

Para hacer de puente entre el Pre-proceso y el Solver es necesaria la creación de un archivo de cálculo. En este archivo, el Pre-proceso escribe los datos del problema, la malla, las condiciones de borde, las propiedades del material y otros datos necesarios. Este archivo ha de poder ser interpretado por el Solver.

En este trabajo el preproceso se realiza mediante GiD, la personalización se usa en este caso para establecer la introducción de los datos y para la generación del archivo de cálculo.

Para personalizar GiD es necesario modificar la carpeta Problemtypes que se encuentra en los archivos del programa. Dentro de esta carpeta se crea el tipo de problema personalizado (FEMproject_problemtype) que para funcionar debe contener los siguientes archivos:

- Archivo.bas: Controla el modo en que se genera el archivo de cálculo.
- Archivo.cnd: Controla el tipo de condiciones de borde y cargas que se pueden aplicar al problema.
- Archivo.dat: Permite crear una base de datos de materiales posibles, en este caso solo se ha creado un material genérico en el que se pueden modificar sus propiedades.
- Archivo.prb: Permite introducir datos relevantes sobre el tipo de problema a solucionar (tipo de análisis y tipo de elemento) para que el Solver pueda ejecutarse.

En el anejo 3 se adjuntan los archivos programados para generar el tipo de problema FEMproject_problemtype.

Post-proceso:

El Post-proceso es el módulo que permite visualizar los resultados obtenidos. Sin este módulo sería muy difícil interpretar algunos datos, por ejemplo el campo de desplazamientos o tensiones sobre un dominio 3D.

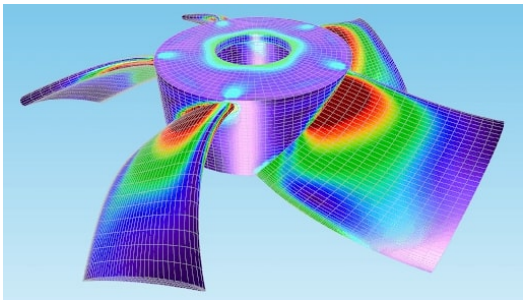


Figura 3.1: Campo de tensiones sobre una hélice. De [5].

En la figura 6 podemos ver un ejemplo del uso de este módulo para visualizar uno de los resultados del análisis MEF, el campo de tensiones se representa como un gradiente de color sobre la geometría. Es notable que intentar interpretar el mismo resultado leyendo el archivo de salida sería prácticamente imposible.

Para usar este módulo es necesario que el Solver escriba los datos de salida en un fichero que pueda ser interpretado por el Post-proceso. Los datos de salida se escriben de tal forma que GiD pueda interpretarlos.

La forma en que se deben escribir los datos de salida se encuentra en la ayuda de personalización de GiD, se programa Matlab (concretamente la función ”write”) para que genere los archivos adecuados:

- Archivo.msh: Este archivo contiene los datos de la malla: Las coordenadas de sus nodos.
- Archivo.res: Este archivo contiene los resultados obtenidos con el Solver (tensión, deformación, deformación unitaria y fuerzas de reacción).

3.3. Solver.

El Solver es el módulo que ejecuta las operaciones necesarias para ensamblar y resolver el problema, este trabajo se centra en el desarrollo de este módulo.

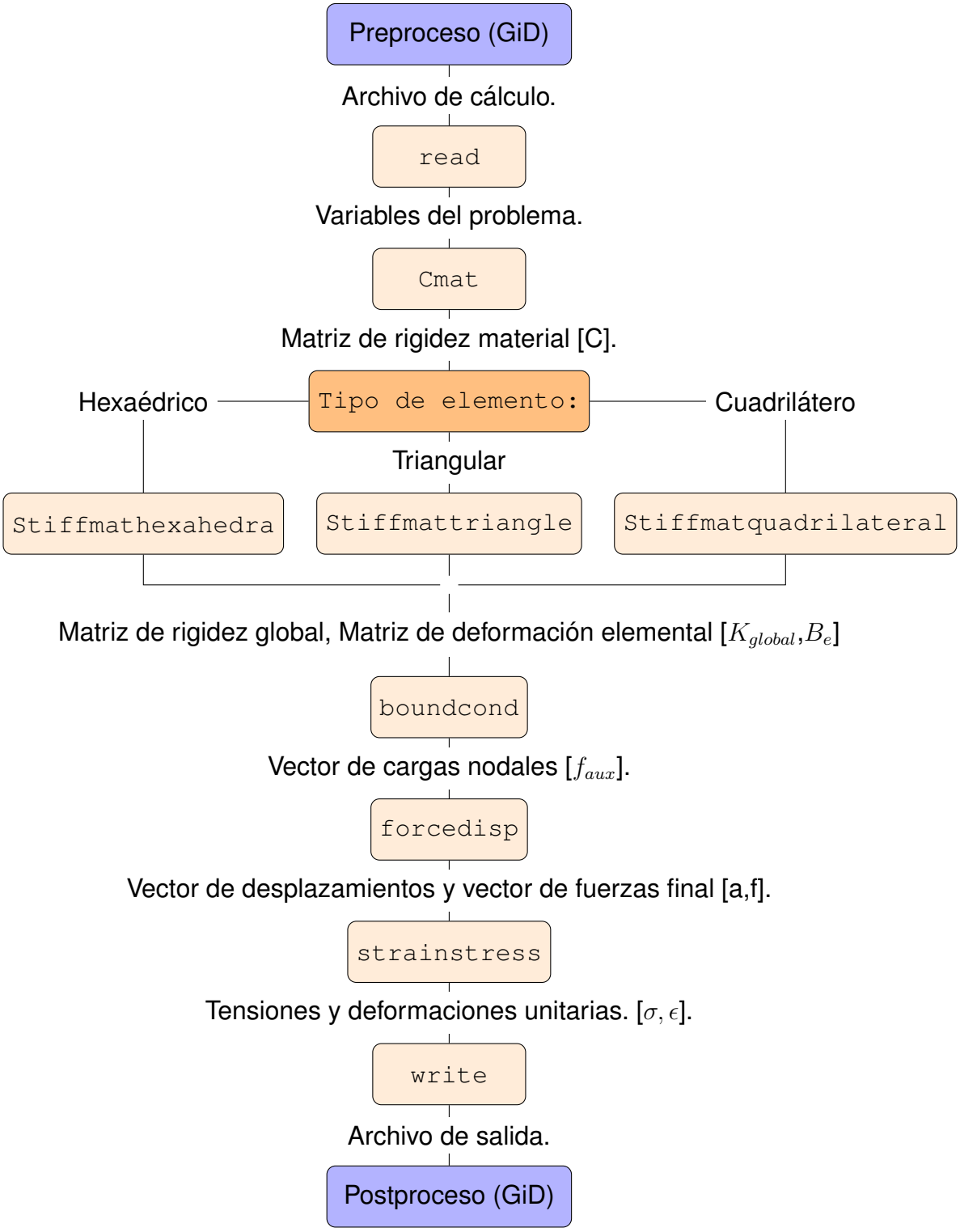
La estructura general del Solver para la resolución de problemas estructurales creado es la siguiente:

- Lectura del fichero de cálculo.
- Cálculo de las ecuaciones constitutivas del problema (ley de Hooke para problemas 2D o 3D): C
- Cálculo de las matrices de rigidez elementales y ensamblaje de la matriz de rigidez global: $K_{elemental}, K_{global}$
- Creación del vector de fuerzas nodales: f
- Creación de la matriz de rigidez auxiliar K_{aux} con las condiciones de borde incluidas (nodos fijos).
- Cálculo de los campos desplazamientos y fuerzas: u, f
- Cálculo de los campos de tensiones y deformaciones unitarias: σ, ϵ
- Escritura de los resultados en el fichero de salida.

En este trabajo el Solver se implementará en Matlab, que permite manipular grandes matrices eficientemente. Cada uno de los módulos se programará en forma de función para mejorar la eficiencia del código.

3.3.1. Diagrama de flujo.

A continuación, se muestra un diagrama de flujo de la función principal del código escrito en Matlab. En este se indican tanto las sub-funciones como los datos de entrada/salida de cada una de ellas.


Preproceso (GiD)
Archivo de cálculo.
read
Variables del problema.
Cmat
Matriz de rigidez material [C].
Hexaédrico
Tipo de elemento:
Cuadrilátero
Triangular
Stiffmathexahedra
Stiffmattriangle
Stiffmatquadrilateral
Matriz de rigidez global, Matriz de deformación elemental [K_{global},B_e]
boundcond
Vector de cargas nodales [f_{aux}].
forcedisp
Vector de desplazamientos y vector de fuerzas final [a,f].
strainstress
Tensiones y deformaciones unitarias. [σ, ϵ].
write
Archivo de salida.
Postproceso (GiD)

3.3.2. Descripción básica de los elementos del programa.

Se describe a continuación el funcionamiento de cada una de las funciones del Solver.

Read: Lectura del fichero de cálculo.

La función que contiene este módulo dentro del código se llama ”read”.

En este trabajo, el fichero que se extrae del Pre-proceso es un archivo ”.dat”. Para poder leerlo en Matlab se usa el comando ”Readmatrix”, este comando permite leer los datos de un archivo ”.dat” seleccionando el rango a leer en el archivo. En el anejo 1 se adjunta a modo de ejemplo uno de los archivos de cálculo creados con GiD que la función es capaz de leer.

Para conocer el rango de datos a leer es necesario tener cierta información previa, por ejemplo, para leer la información de cada nodo es necesario saber cuantos nodos tiene el problema. Por esta razón, esta función lee primero la información característica del problema y a continuación obtiene los datos que se van a usar para la resolución.

Se listan a continuación los datos obtenidos mediante esta función y la forma en que se almacenan en Matlab.

Variables usadas a modo de información inicial para la lectura de datos:

- Número de elementos. **Elem**.
- Número de nodos. **Nod**.
- Número de materiales. **Numbmat**.
- Número de nodos fijos. **Nfixnod**.
- Número de lineas fijas. **Nfixline**.
- Número de superfícies fijas. **Nfixsurf**.
- Número de nodos con carga. **Nloadnod**.

- Número de lineas cargadas. **Nloadline**.
- Número de superfícies cargadas. **Nloadsurf**.
- Número de volúmenes cargados. **Nloadvolume**.
- Tipo de analysis. **Analysistype**: 1=tensión plana, 2=deformación plana, 3=3D.
- Tipo de elemento. **Elemtype**: 1=triangular, 2=cuadrilátero, 3=hexaédrico.

Variables que contienen datos necesarios para la resolución del problema. Se especifica también el orden en que se almacena la información en el vector:

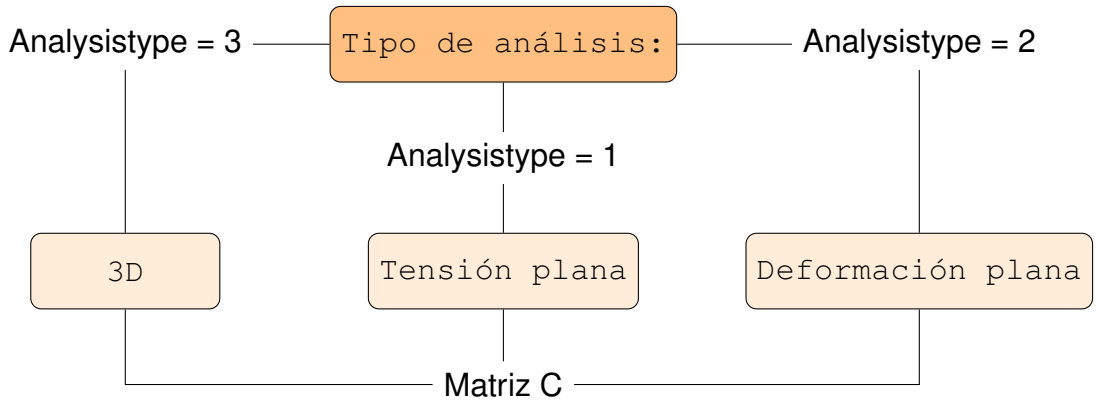
- Información sobre los nodos. **nodcord**:
  [número de nodo, (Coordenadas)]
- Información sobre los elementos. **Elemcon**:
  [(conexiones nodales), número de material, número de elemento]
- Información de material. **Mat**:
  [número de material, módulo de Poison, módulo cortante, módulo de Young, espesor(si el problema es 2D)]
- Información sobre nodos fijos. **FixNod**:
  [Número de nodo, (Direcciones fijas)]
- Información sobre lineas fijas. **fixline**:
  [Número de nodo, (Coordenadas fijas)]
- Información sobre superficies fijas. **fixsurf**:
  [Número de nodo, (Coordenadas fijas)]
- Información sobre nodos con carga. **LoadNod**:
  [Número de nodo, (cargas según coordenadas)]
- Información sobre lineas cargadas. **Loadline**:
  [número de elemento, Nodos de la linea, (cargas según coordenadas)]
- Información sobre las superficies cargadas. **Loadsurf**:
  [número de e[lemento, Nodos de la superficie, (cargas según coordenadas)]
- Información sobre los volúmenes cargados. **Loadvolume**:
  [número de elemento, (cargas según coordenadas)]

Cmat: Cálculo de las ecuaciones constitutivas del problema.

Esta función se llama ”Cmat” en el código, calcula la matriz de rigidez del material en función del tipo de problema (Tensión plana, Deformación plana, 3D). Para poder solucionar problemas con diferentes materiales, ”Cmat” calcula tantas matrices de rigidez C cómo materiales hay asignados.

Más adelante en el código se usará la matriz C correspondiente para calcular la matriz de rigidez elemental (K_e).

Diagrama de flujo de la función:



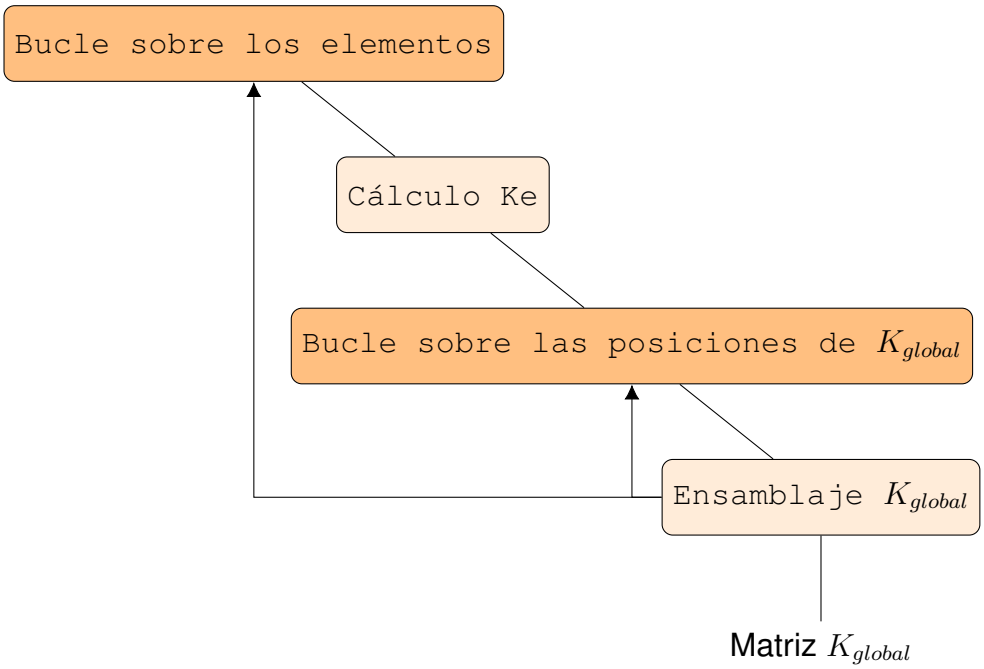
Stiffmat: Cálculo de las matrices de rigidez elementales y ensamblaje de la matriz de rigidez.

Este módulo se encarga de generar la matriz de rigidez global. El nombre de la función que lo ejecuta es ”Stiffmattriangle” para elementos triangulares, ”Stiffmatquadrilateral” para elementos cuadriláteros y ”Stiffmathexahedra” para elementos hexaedros. El programa elige la función a ejecutar dependiendo del tipo de elemento. La separación del código en función del tipo de elemento se hace para simplificarlo y evitar que Matlab lea funciones demasiado grandes.

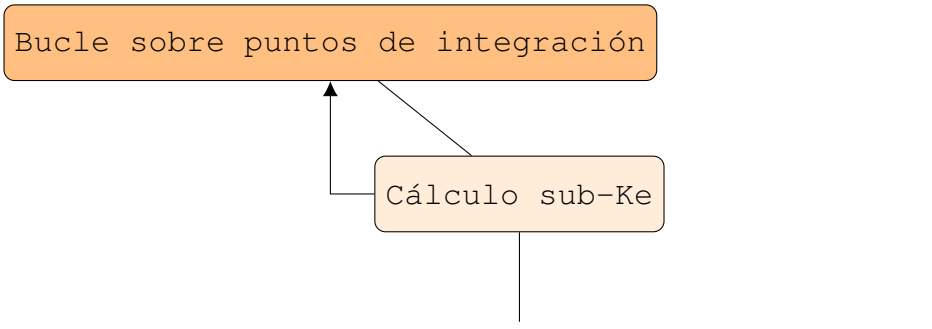
Las funciones ”Stiffmat” se pueden subdividir a su vez en dos partes, primero el cálculo de las matrices de rigidez elementales y a continuación el ensamblaje de estas en la matriz de rigidez global, este algoritmo se repite para cada elemento.

En los elementos cuadriláteros y los hexaedros, el cálculo de las matrices de rigidez elementales se hace por integración numérica, para ello se crean dos sub-funciones, ”Jacobianmat” y ”Deformationmat”, que calculan el Jacobiano y la matriz de deformaciones correspondiente.

A excepción de las diferencias citadas, el diagrama de flujo de las funciones ”Stiffmat” es el siguiente:



Para los elementos triangulares, el cálculo de K_e es una operación matricial mientras que para el resto de elementos este cálculo requiere bucles como se muestra en el siguiente diagrama de flujo:



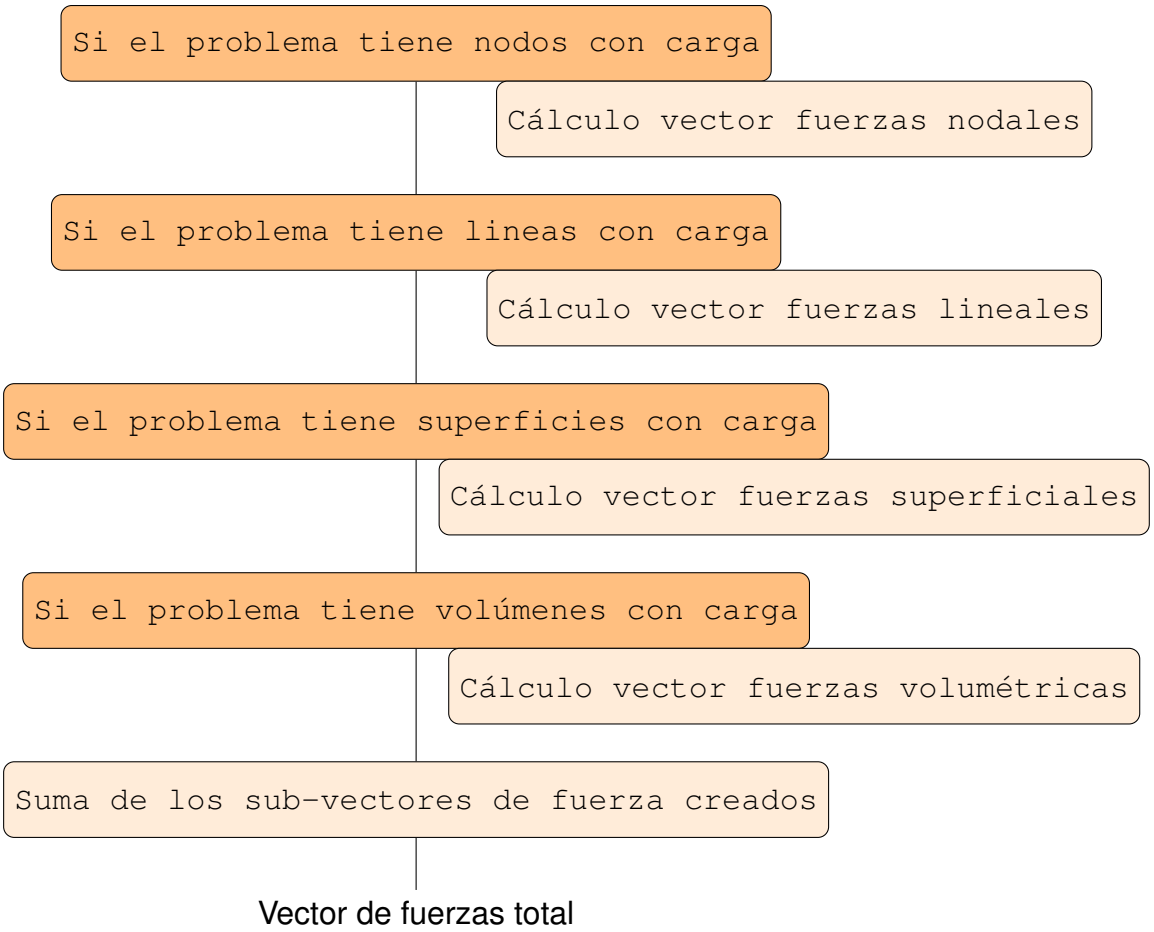
Boundcond: Creación del vector de fuerzas nodales.

En este módulo se ensambla el vector $\bar{f}$, en el código Matlab a la función que lo ejecuta se la llama ”boundcond”.

Para crear $\bar{f}$ es necesario conocer el valor de la fuerza aplicada y el nodo en que esta se aplica. Debido a que en el código se aplican varios tipos de cargas (puntuales, repartidas sobre superficie y repartidas sobre un volumen) esta función también se subdivide en bloques, uno para cada tipo de carga.

Cabe destacar que para calcular el vector de fuerzas nodales se ha evitado la integración numérica, que complica innecesariamente el código y lo hace ineficiente. El proceso de cálculo de $\bar{f}$ en función de la carga se detallará más adelante.

El diagrama de flujo general de esta función es el siguiente:



Forcedisp: Creación de la matriz de rigidez auxiliar K_{aux} con las condiciones de borde incluidas (nodos fijos).

Debido a que las condiciones de contorno actúan sobre las fuerzas y los desplazamientos, ambos vectores tienen incógnitas. Para resolver el problema este se transforma en un sistema equivalente en el que únicamente hay incógnitas en el vector de desplazamientos. La creación de este problema equivalente se hace de la siguiente forma:

Se modifica la matriz de rigidez del problema de tal manera que toda la fila correspondiente al desplazamiento que se quiere fijar tenga valor 0, excepto en la diagonal, donde se escribe un valor muy grande (10^{99}). De esta forma se crea una matriz singular que al ser invertida fuerza que el desplazamiento fijado sea muy pequeño, tan pequeño que Matlab lo considera 0.

Esto permite calcular el vector de desplazamientos y el vector de fuerzas nodales mediante las siguientes ecuaciones:

$$\bar{a} = \bar{\bar{K}}_{aux}^{-1} \bar{f} \tag{3.1}$$

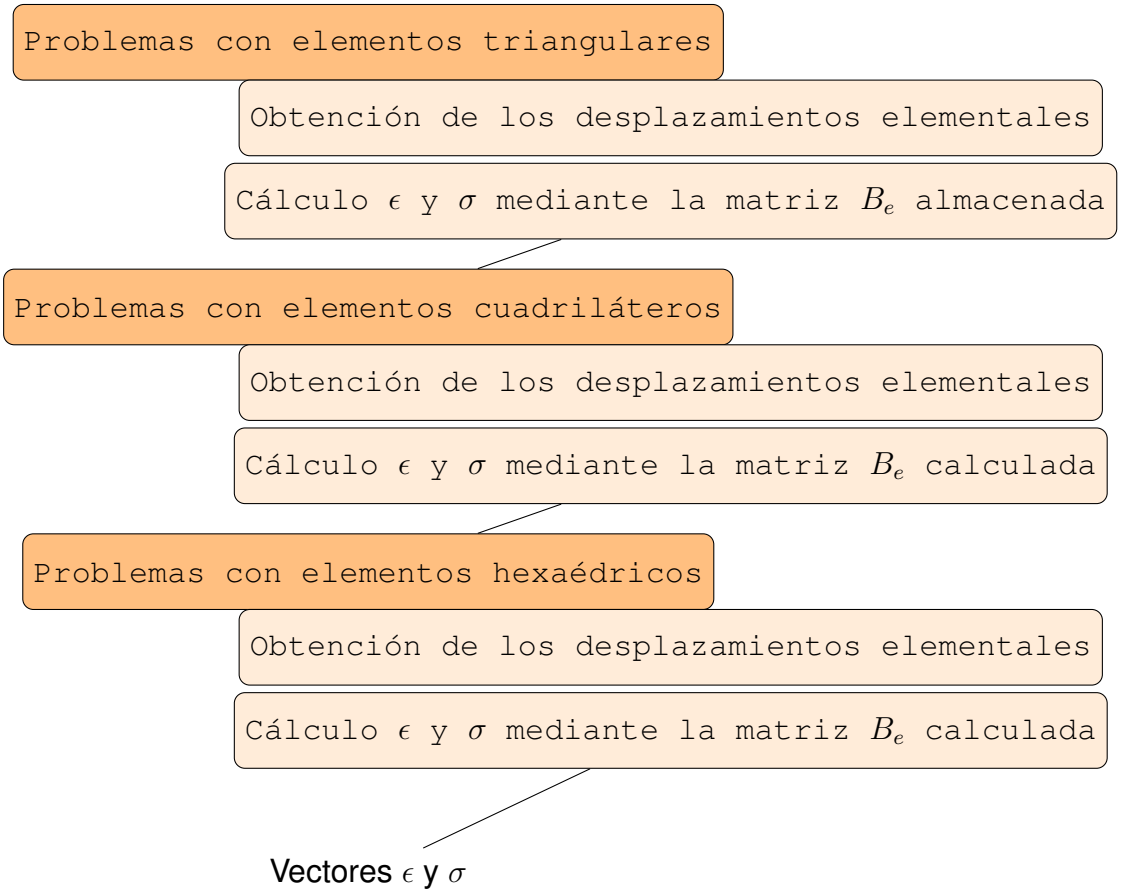
$$\bar{f} = \bar{\bar{K}} \bar{a} \tag{3.2}$$

Strainstress: Cálculo del campo de tensiones y deformaciones unitarias.

En función del tipo de elemento esta función se ejecuta de una forma u otra. En elementos triangulares, para calcular las deformaciones unitarias se usa la matriz de deformación del elemento, que previamente se ha almacenado durante la rutina ”stiffmattiangle”.

En los elementos cuadriláteros y los hexaédricos, se calcula la matriz de deformación de nuevo mediante integración numérica. Esto se debe a que para evaluar las deformaciones unitarias en los puntos de integración sería necesario almacenar una cantidad demasiado grande de información (una matriz de rigidez por punto de integración).

Para todos los elementos se calcula el campo de tensiones mediante la matriz de rigidez del material (C) y las deformaciones unitarias previamente calculadas.



Write: Escritura de los resultados en el fichero de salida.

La función que escribe los resultados en el fichero de salida se llama ”write”. Para la escritura de datos se usa la función de Matlab ”fprintf”. Para que el archivo generado pueda ser leído por GiD, se sigue el formato especificado por este programa en su ayuda de personalización.

3.3.3. Uso del código.

En esta sección se explica el procedimiento a seguir para realizar un cálculo usando el código desarrollado.

En primer lugar, es necesario descargar tanto GiD como Matlab. Adicionalmente, se debe tener en una carpeta todas las funciones Matlab del Solver y en la carpeta ”problemtypes” de GiD todos los archivos necesarios para la personalización del proceso (FEMproject_problemtype).

El proceso de resolución de un problema es el siguiente:

1. Se abre el Problemtype creado en GiD (FEMproject_problemtype) para tener acceso a las opciones personalizadas.
2. Se dibuja la geometría del problema de interés, se indican los materiales de construcción, las condiciones de borde y las cargas aplicadas. Se crea la malla y se guarda el archivo, generando una carpeta del problema.
3. Se genera el archivo de cálculo, este se extrae de la carpeta del problema y se copia en la carpeta con las funciones del Solver.
4. Se abre la función principal del Solver en Matlab (FEMproject) y se escribe el nombre del archivo de cálculo. Se ejecuta el código.
5. Tras finalizar la ejecución del código, se generan dos archivos en la carpeta del Solver: ”.res” (archivo de resultados), ”.msh” (archivo con datos de geometría y malla). Estos archivos se deben trasladar de nuevo a la carpeta del problema GiD.
6. Finalmente al volver a abrir la carpeta del problema se puede acceder al módulo de Post-proceso y ver los resultados obtenidos de forma gráfica.

Capítulo 4

Descripción detallada de las características más relevantes del Solver.

4.1. Características del código.

En este capítulo, se detalla el funcionamiento de las funcionalidades del Solver más importantes. Estas permiten particularizar el código para que se ajuste a diferentes tipos de problemas, haciéndolo más versátil.

Algunas de estas funcionalidades son: Uso de varios tipos de elementos finitos (triangulares, cuadriláteros, hexaédricos), uso de diferentes espacios de resolución del problema (tensión plana, deformación plana, 3D), uso de diferentes cargas aplicadas.

4.2. Elección del tipo de elemento.

Para este capítulo se han tomado como referencia los siguientes documentos: [12],[10],[13],[14]

Poder elegir el tipo de elemento con el que mallar es una funcionalidad necesaria para resolver un amplio abanico de problemas. En dos dimensiones, elegir entre elementos triangulares y cuadriláteros facilita el mallado del dominio. Para resolver problemas en tres dimensiones es necesario que el mallado se haga con elementos tridimensionales, en este caso los elementos usados son los hexaédricos.

La diferencia principal en el código para el tratamiento de los diferentes elementos radica en la creación y ensamblaje de la matriz de rigidez, las funciones ”stiffmat”.

Según el elemento la evaluación de la integral de rigidez del elemento (ecn.4.1) se hace de forma diferente.

$$Ke = \int_V B^T C B det J \; dV \tag{4.1}$$

Se observa que para calcular estas matrices de rigidez elementales es necesaria la computación de la matriz de rigidez del material (C), la matriz de deformación (B) y el Jacobiano (J), que dependen del tipo de problema y del tipo de elemento respectivamente. Cabe destacar que la matriz C se calcula previamente, durante la rutina ”Cmat”.

A continuación se detallan las particularidades del código según el tipo de elemento:

4.2.1. Elementos triangulares:

Este tipo de elementos tiene una particularidad que hace que su programación sea relativamente sencilla. Debido a que las funciones de forma son lineales, todas las computaciones que dependan de las derivadas de estas (Matriz B, Jacobiano) quedan constantes, esto permite evaluar la integral muy fácilmente sin tener que recurrir a la integración numérica.

La funciones de forma de este tipo de elementos son las siguientes:

$$N_1 = r; \quad N_2 = s; \quad N_3 = 1 - r - s \tag{4.2}$$

Donde r y s son las coordenadas naturales del elemento. Se puede comprobar que las funciones de forma tienen las propiedades mostradas en el apartado 2.1.5.

Siguiendo la ecuación 2.26 se calcula el Jacobiano de la siguiente forma:

$$\bar{\bar{J}} = \begin{bmatrix} \frac{dx}{dr} & \frac{dy}{dr} \\ \frac{dx}{ds} & \frac{dy}{ds} \end{bmatrix} \tag{4.3}$$

Dónde:

$$x = \sum_{i=1}^{n} N(r,s)_i x_i; \quad y = \sum_{i=1}^{n} N(r,s)_i y_i \tag{4.4}$$

Por tanto las derivadas quedan de la siguiente forma:

$$\frac{dx}{d()} = \sum_{i=1}^{n} \frac{dN(r,s)_i}{d()} x_i; \quad \frac{dy}{d()} = \sum_{i=1}^{n} \frac{dN(r,s)_i}{d()} y_i \tag{4.5}$$

Evaluando el Jacobiano se obtiene:

$$\bar{\bar{J}} = \begin{bmatrix} (x_1 - x_3) & (y_1 - y_3) \\ (x_2 - x_3) & (y_2 - y_3) \end{bmatrix} \tag{4.6}$$

Se observa que:

$$det(J) = (x_1 - x_3)(y_2 - y_3) - (x_2 - x_3)(y_1 - y_3) = 2Ae \tag{4.7}$$

Dónde Ae es el área del elemento.

Calculando la inversa del Jacobiano se puede escribir la ecuación 2.26 de la siguiente forma:

$$\frac{\bar{d}}{dx} = \frac{1}{det(J)} \begin{bmatrix} (x_2 - x_3) & -(y_1 - y_3) \\ -(x_2 - x_3) & (y_1 - y_3) \end{bmatrix} \frac{\bar{d}}{dr} \tag{4.8}$$

Finalmente, se calcula la matriz de deformación usando las expresiones 2.3 y 2.24:

$$\epsilon_x = \frac{du}{dx} = \sum_{i=1}^{n} \frac{dN_i}{dx} u_i = \frac{1}{2Ae} \sum_{i=1}^{n} b_i u_i \tag{4.9}$$

$$\epsilon_y = \frac{dv}{dy} = \sum_{i=1}^{n} \frac{dN_i}{dy} v_i = \frac{1}{2Ae} \sum_{i=1}^{n} c_i v_i \tag{4.10}$$

$$\epsilon_{xy} = \frac{du}{dy} + \frac{dv}{dx} = \sum_{i=1}^{n} \frac{dN_i}{dy} u_i + \sum_{i=1}^{n} \frac{dN_i}{dx} v_i = \frac{1}{2Ae} (\sum_{i=1}^{n} c_i u_i + \sum_{i=1}^{n} b_i v_i) \tag{4.11}$$

Donde:

$$b_i = y_j - y_k \tag{4.12}$$

$$c_i = x_k - x_j \tag{4.13}$$

Escribiendo en forma matricial la expresión anterior, se obtiene el equivalente a la ecuación 2.24 para elementos triangulares, donde la matriz B es la siguiente:

$$\bar{\bar{B}} = \begin{bmatrix} \frac{dN_1}{dx} & 0 & \frac{dN_2}{dx} & 0 & \frac{dN_3}{dx} & 0 \\ 0 & \frac{dN_1}{dy} & 0 & \frac{dN_2}{dy} & 0 & \frac{dN_3}{dy} \\ \frac{dN_1}{dy} & \frac{dN_1}{dx} & \frac{dN_2}{dy} & \frac{dN_2}{dx} & \frac{dN_3}{dy} & \frac{dN_3}{dx} \end{bmatrix} = \frac{1}{2Ae} \begin{bmatrix} b_1 & 0 & b_2 & 0 & b_3 & 0 \\ 0 & c_1 & 0 & c_2 & 0 & c_3 \\ c_1 & b_1 & c_2 & b_2 & c_3 & b_3 \end{bmatrix} \tag{4.14}$$

Debido a que todos los elementos necesarios para calcular la integral 4.1 son constantes:

$$Ke = B^T C B det J \int_V dV \tag{4.15}$$

La integral de volumen se puede calcular fácilmente sabiendo que cada elemento tiene un espesor constante (t):

$$\int_V dV \ = t \int_S dS \ = tAe \tag{4.16}$$

Finalmente, Ke se expresa de la siguiente forma:

$$Ke = \frac{1}{4Ae}(B^T C B) \tag{4.17}$$

Se puede observar que para realizar este cálculo solamente es necesario tener la matriz C, las coordenadas de los nodos, el área del elemento y su espesor. De este grupo de datos el único que se calcula dentro de la función "Stiffmattriangle" es el área del elemento.

También cabe destacar que la función "Stiffmattriangle" almacena la matriz de deformación B para el cálculo posterior del campo de tensiones.

4.2.2. Elementos cuadriláteros:

Para este tipo de elementos la integral 4.1 se evalúa numéricamente mediante una cuadratura de Gauss.

La cuadratura de Gauss es un método de integración numérica. Consiste en realizar una suma ponderada de la función a integrar, evaluándola sobre algunos de los puntos del dominio. Los puntos en los que se evalúa la función se denominan puntos de integración.

Mediante este método una integral de superficie se evalúa de la siguiente forma:

$$\int_S F \ dS \ = \sum_{i=1}^{2} \sum_{j=1}^{2} w_{ij} F_{ij} \tag{4.18}$$

De la misma forma, una integral de volumen:

$$\int_V F\ dV\ = \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} w_{ijk}F_{ijk} \tag{4.19}$$

Donde w_{ij} y w_{ijk} son los pesos correspondientes a cada punto de integración.

Se muestran a continuación los puntos de integración usados en este trabajo para los elementos cuadriláteros.

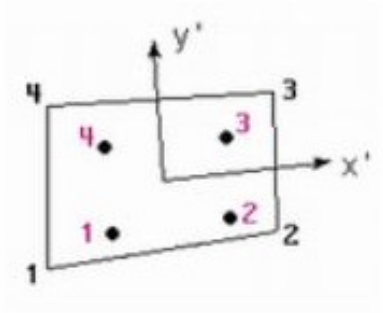


Figura 4.1: Puntos de integración de un elemento cuadrilátero. De [6].

En la siguiente tabla se muestran las coordenadas de cada punto de integración y su peso. Cabe destacar que las coordenadas se escriben según los ejes naturales del elemento.

punto.	**Coordenadas**	**Peso**
1	$(-\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}})$	1
2	$(\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}})$	1
3	$(-\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}})$	1
4	$(\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}})$	1

Tabla 4.1: Valores para integración numérica en elementos cuadriláteros.

Para evaluar la integral es necesario escribir B y J en función de las coordenadas naturales, esto se consigue fácilmente usando las funciones de forma escritas también en estas coordenadas.

Estas funciones de forma son las siguientes:

$$N_i = \frac{(1+rr_i)(1+ss_i)}{4}; \quad i = 1,2,3,4 \tag{4.20}$$

Dónde r_i y s_i son las coordenadas naturales del nodo i.

Tras interpolar la geometría y evaluar las derivadas se escribe el Jacobiano:

$$\bar{\bar{J}} = \begin{bmatrix} \frac{dx}{dr} & \frac{dy}{dr} \\ \frac{dx}{ds} & \frac{dy}{ds} \end{bmatrix} \tag{4.21}$$

Dónde:

$$\frac{dx}{dr} = \frac{1}{4}[(1+s)x_1 - (1+s)x_2 - (1-s)x_3 + (1-s)x_4] \tag{4.22}$$

$$\frac{dx}{ds} = \frac{1}{4}[(1+r)x_1 + (1-r)x_2 - (1-r)x_3 - (1+r)x_4] \tag{4.23}$$

$$\frac{dy}{dr} = \frac{1}{4}[(1+s)y_1 - (1+s)y_2 - (1-s)y_3 + (1-s)y_4] \tag{4.24}$$

$$\frac{dy}{ds} = \frac{1}{4}[(1+r)y_1 + (1-r)y_2 - (1-r)y_3 - (1+r)y_4] \tag{4.25}$$

Dónde $x_i \;\; y_i \;\; i = 1,2,3,4$ són las coordenadas globales (x,y) del nodo i.

La evaluación del Jacobiano se hace en la función ”Jacobianmat”.

A continuación, se evalúa la matriz de deformación. Para ello se usa la inversa del Jacobiano como se muestra a continuación.

$$\bar{\bar{B}} = \begin{bmatrix} \frac{du}{dx}(1) & \frac{du}{dx}(2) & \frac{du}{dx}(3) & \frac{du}{dx}(4) & \frac{du}{dx}(5) & \frac{du}{dx}(6) & \frac{du}{dx}(7) & \frac{du}{dx}(8) \\ \frac{dv}{dy}(1) & \frac{dv}{dy}(2) & \frac{dv}{dy}(3) & \frac{dv}{dy}(4) & \frac{dv}{dy}(5) & \frac{dv}{dy}(6) & \frac{dv}{dy}(7) & \frac{dv}{dy}(8) \\ \frac{du}{dy}(1) & \frac{dv}{dx}(2) & \frac{du}{dy}(3) & \frac{dv}{dx}(4) & \frac{du}{dy}(5) & \frac{dv}{dx}(6) & \frac{du}{dy}(7) & \frac{dv}{dx}(8) \end{bmatrix} \tag{4.26}$$

Dónde:

$$\begin{bmatrix} \frac{du}{dx} \\ \frac{du}{dy} \end{bmatrix} = \frac{1}{4} J^{-1} \begin{bmatrix} 1+s & 0 & -(1+s) & 0 & -(1-s) & 0 & 1-s & 0 \\ 1+r & 0 & 1-r & 0 & -(1-r) & 0 & -(1+r) & 0 \end{bmatrix} \tag{4.27}$$

$$\begin{bmatrix} \frac{dv}{dx} \\ \frac{dv}{dy} \end{bmatrix} = \frac{1}{4} J^{-1} \begin{bmatrix} 0 & 1+s & 0 & -(1+s) & 0 & -(1-s) & 0 & 1-s \\ 0 & 1+r & 0 & 1-r & 0 & -(1-r) & 0 & -(1+r) \end{bmatrix} \tag{4.28}$$

El algoritmo que calcula la matriz de rigidez elemental dentro de ”Stiffmat” sigue el proceso que se explica a continuación:

Para cada uno de los puntos de integración se evalúa tanto la matriz de deformación cómo el Jacobiano, finalmente se integra por Gauss para obtener el valor final de Ke.

$$Ke = \int_V B^T C B det J \; dV \; = \sum_{i=1}^{2} \sum_{j=1}^{2} w_{ij} (B^T C B det J)_{ij} \tag{4.29}$$

4.2.3. Elementos hexaédricos:

El procedimiento para este tipo de elementos es similar al seguido con los elementos cuadriláteros, ya que la integral 4.1 también se evalúa de forma numérica en este caso.

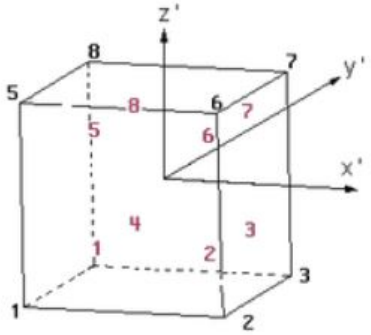


Figura 4.2: Puntos de integración de un elemento hexaédrico. De [6].

Los pesos y coordenadas para la integración numérica se muestran en la siguiente tabla:

punto.	**Coordenadas**	**Peso**
1	$(-\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}})$	1
2	$(\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}})$	1
3	$(\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}})$	1
4	$(-\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}})$	1
5	$(-\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}})$	1
6	$(\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}})$	1
7	$(\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}})$	1
8	$(-\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}},\frac{1}{\sqrt{3}})$	1

Tabla 4.2: Valores para integración numérica en elementos hexaédricos.

Las funciones de forma de este tipo de elemento son las siguientes:

$$N_i = \frac{(1+rr_i)(1+ss_i)(1+tt_i)}{8}; \quad i = 1, 2, ..., 8 \tag{4.30}$$

Dónde r_i, s_i, t_i son las coordenadas naturales del nodo i.

A continuación se puede evaluar el Jacobiano con la geometría interpolada:

$$\bar{\bar{J}} = \begin{bmatrix} \frac{dx}{dr} & \frac{dy}{dr} & \frac{dz}{dr} \\ \frac{dx}{ds} & \frac{dy}{ds} & \frac{dz}{ds} \\ \frac{dx}{dt} & \frac{dy}{ds} & \frac{dz}{dt} \end{bmatrix} \tag{4.31}$$

Dónde:

$$\frac{dx}{dr} = \frac{1}{8}[-(1-s)(1-t)x_1 + (1-s)(1-t)x_2 + (1+s)(1-t)x_3 - (1+s)(1-t)x_4 \\ -(1-s)(1+t)x_5 + (1-s)(1+t)x_6 + (1+s)(1+t)x_7 - (1+s)(1+t)x_8]$$

$$\frac{dx}{ds} = \frac{1}{8}[-(1-r)(1-t)x_1 - (1+r)(1-t)x_2 + (1+r)(1-t)x_3 + (1-r)(1-t)x_4 \\ -(1-r)(1+t)x_5 - (1+r)(1+t)x_6 + (1+r)(1+t)x_7 + (1-r)(1+t)x_8]$$

$$\frac{dx}{dt} = \frac{1}{8}[-(1-r)(1-s)x_1 - (1+r)(1-s)x_2 - (1+r)(1+s)x_3 - (1-r)(1+s)x_4 \\ +(1-r)(1-s)x_5 + (1+r)(1-s)x_6 + (1+r)(1+s)x_7 + (1-r)(1+s)x_8]$$

$$\frac{dy}{dr} = \frac{1}{8}[-(1-s)(1-t)y_1 + (1-s)(1-t)y_2 + (1+s)(1-t)y_3 - (1+s)(1-t)y_4 \\ -(1-s)(1+t)y_5 + (1-s)(1+t)y_6 + (1+s)(1+t)y_7 - (1+s)(1+t)y_8]$$

$$\frac{dy}{ds} = \frac{1}{8}[-(1-r)(1-t)y_1 - (1+r)(1-t)y_2 + (1+r)(1-t)y_3 + (1-r)(1-t)y_4 \\ -(1-r)(1+t)y_5 - (1+r)(1+t)y_6 + (1+r)(1+t)y_7 + (1-r)(1+t)y_8]$$

$$\frac{dy}{dt} = \frac{1}{8}[-(1-r)(1-s)y_1 - (1+r)(1-s)y_2 - (1+r)(1+s)y_3 - (1-r)(1+s)y_4 \\ +(1-r)(1-s)y_5 + (1+r)(1-s)y_6 + (1+r)(1+s)y_7 + (1-r)(1+s)y_8]$$

$$\frac{dz}{dr} = \frac{1}{8}[-(1-s)(1-t)z_1 + (1-s)(1-t)z_2 + (1+s)(1-t)z_3 - (1+s)(1-t)z_4 \\ -(1-s)(1+t)z_5 + (1-s)(1+t)z_6 + (1+s)(1+t)z_7 - (1+s)(1+t)z_8]$$

$$\frac{dz}{ds} = \frac{1}{8}[-(1-r)(1-t)y_1 - (1+r)(1-t)y_2 + (1+r)(1-t)y_3 + (1-r)(1-t)y_4 \\ -(1-r)(1+t)y_5 - (1+r)(1+t)y_6 + (1+r)(1+t)y_7 + (1-r)(1+t)y_8]$$

$$\frac{dz}{dt} = \frac{1}{8}[-(1-r)(1-s)z_1 - (1+r)(1-s)z_2 - (1+r)(1+s)z_3 - (1-r)(1+s)z_4 \\ +(1-r)(1-s)z_5 + (1+r)(1-s)z_6 + (1+r)(1+s)z_7 + (1-r)(1+s)z_8]$$

Una vez obtenido el Jacobiano se puede encontrar la matriz de deformación de la siguiente forma:

$$\bar{\bar{B}} = \begin{bmatrix} \frac{du}{dx}(1) & \frac{du}{dx}(2) & \frac{du}{dx}(3) & \frac{du}{dx}(4) & \frac{du}{dx}(5) & \frac{du}{dx}(6) & .... & \frac{du}{dx}(24) \\ \frac{dv}{dy}(1) & \frac{dv}{dy}(2) & \frac{dv}{dy}(3) & \frac{dv}{dy}(4) & \frac{dv}{dy}(5) & \frac{dv}{dy}(6) & .... & \frac{dv}{dy}(24) \\ \frac{dw}{dz}(1) & \frac{dw}{dz}(2) & \frac{dw}{dz}(3) & \frac{dw}{dz}(4) & \frac{dw}{dz}(5) & \frac{dw}{dz}(6) & .... & \frac{dw}{dz}(24) \\ \frac{du}{dy}(1) & \frac{dv}{dx}(2) & 0 & \frac{dv}{dx}(4) & \frac{du}{dy}(5) & 0 & .... & 0 \\ \frac{du}{dz}(1) & 0 & \frac{dw}{dx}(2) & \frac{du}{dz}(4) & 0 & \frac{dw}{dx}(5) & .... & \frac{dw}{dx}(24) \\ 0 & \frac{dv}{dz}(2) & \frac{dw}{dy}(1) & 0 & \frac{dv}{dz}(5) & \frac{dw}{dy}(4) & .... & \frac{dw}{dy}(24) \end{bmatrix} \quad (4.32)$$

Dónde las componentes de B se muestran a continuación:

$$\begin{bmatrix} \frac{du}{dx} \\ \frac{du}{dy} \\ \frac{du}{dz} \end{bmatrix} = \frac{1}{8} J^{-1} \begin{bmatrix} -(1-s)(1-t) & 0 & 0 & (1-s)(1-t) & 0 & 0 & .... \\ -(1-r)(1-t) & 0 & 0 & -(1+r)(1-t) & 0 & 0 & .... \\ -(1-r)(1-s) & 0 & 0 & -(1+r)(1-s) & 0 & 0 & .... \end{bmatrix} \tag{4.33}$$

$$\begin{bmatrix} \frac{dv}{dx} \\ \frac{dv}{dy} \\ \frac{dv}{dz} \end{bmatrix} = \frac{1}{8} J^{-1} \begin{bmatrix} 0 & -(1-s)(1-t) & 0 & 0 & (1-s)(1-t) & 0 & .... \\ 0 & -(1-r)(1-t) & 0 & 0 & -(1+r)(1-t) & 0 & .... \\ 0 & -(1-r)(1-s) & 0 & 0 & -(1+r)(1-s) & 0 & .... \end{bmatrix} \tag{4.34}$$

$$\begin{bmatrix} \frac{dz}{dx} \\ \frac{dz}{dy} \\ \frac{dz}{dz} \end{bmatrix} = \frac{1}{8} J^{-1} \begin{bmatrix} 0 & 0 & -(1-s)(1-t) & 0 & 0 & (1-s)(1-t) & .... \\ 0 & 0 & -(1-r)(1-t) & 0 & 0 & -(1+r)(1-t) & .... \\ 0 & 0 & -(1-r)(1-s) & 0 & 0 & -(1+r)(1-s) & .... \end{bmatrix} \tag{4.35}$$

Una vez se tiene todo lo necesario se evalúa la integral 4.1 numéricamente:

$$Ke = \int_V B^T C B det J \, dV = \sum_{i=1}^{2} \sum_{j=1}^{2} \sum_{k=1}^{2} w_{ijk} (B^T C B det J)_{ijk} \tag{4.36}$$

4.2.4. Diferencias en el módulo de ensamblaje:

Adicionalmente, cómo se ha deducido en el ejemplo 1, las dimensiones de la matriz de rigidez varían en función de la dimensión del problema, esto hace necesario el uso de una rutina de ensamblaje diferente dependiendo de si el problema es 2D o 3D.

La diferencia principal en la rutina de ensamblaje es que para problemas en tres dimensiones también se ensamblan en la matriz de rigidez global las componentes de la matriz de rigidez elemental que corresponden a la coordenada z.

4.3. Elección del tipo de carga aplicada.

Para resolver un abanico amplio de problemas, el código debe poder calcular el vector $\bar{f}$ generado por diferentes tipos de cargas. La función encargada de este módulo es ”boundcond”.

Cabe destacar que para mejorar el tiempo de computación del programa (sobre todo en elementos cuadriláteros y hexaédricos) se ha buscado un enfoque que evita la integración numérica para realizar esta tarea:

Se lee en el archivo de cálculo el valor de la fuerza aplicada por unidad de distancia/volumen/área. A continuación, se puede obtener el valor total de la carga calculando la magnitud de el dominio sobre el que se aplica. Finalmente, la fuerza se divide entre el número de nodos a los que afecta y se ensambla en el vector de cargas nodales. Esta simplificación es posible debido a que en los problemas tratados las cargas siempre son constantes sobre el dominio en que se aplican.

Aunque el procedimiento general se ha explicado previamente, se detalla como se tratan cada una de las posibles cargas a aplicar en la función ”boundcond”:

4.3.1. Cargas nodales.

No requieren un tratamiento especial, el archivo de cálculo tiene información sobre el valor de la carga aplicada en cada nodo. La función asigna este valor a la posición correspondiente del vector de fuerzas nodales.

4.3.2. Cargas repartidas sobre longitud.

Este tipo de carga solo se ha implementado para la resolución de problemas en dos dimensiones.

El algoritmo es el siguiente:

1. Se obtienen el número de cargas lineales diferentes en el problema.
2. Se calcula por cada una de ellas la longitud sobre la que se aplican y el valor total de la fuerza aplicada.
3. Se identifican los nodos cargados y la fuerza se reparte de forma equitativa sobre estos.
4. Se ensambla el vector de fuerzas nodales.

Es importante destacar que al evaluar cargas lineales en 2D en realidad se están evaluando cargas de superficie, esto es debido a que se considera que la carga se aplica sobre la superficie lateral del dominio. Es por esta razón que en el cálculo de la longitud se tiene también en cuenta el espesor del dominio estudiado, que es constante.

4.3.3. Cargas repartidas sobre superficie.

Esta carga se puede aplicar tanto en problemas 2D cómo 3D. Sin embargo, de forma similar al caso anterior, al aplicar este tipo de carga en problemas 2D en realidad se esta aplicando una carga volumétrica.

El algoritmo es muy parecido al usado para cargas lineales. La diferencia principal es que en vez de calcular la longitud de aplicación de la carga se calcula el área de aplicación. En 2D el área calculada se multiplica por el espesor del dominio para obtener el volumen.

4.3.4. Cargas volumétricas.

Este tipo de carga solo se puede aplicar en problemas de tres dimensiones por razones obvias.

El procedimiento es similar al aplicado en las otras cargas pero se calcula el volumen de aplicación para obtener el valor total de la fuerza. Otra diferencia es que generalmente al aplicar una carga volumétrica se aplicará sobre todo el dominio del problema, por lo que normalmente solo habrá una carga volumétrica por problema.

4.3.5. Vector de fuerzas final.

El vector final de fuerzas nodales se obtiene sumando las contribuciones de todos los vectores anteriormente mencionados.

4.4. Elección de las condiciones de borde.

De forma similar a las cargas, es necesario tener cierta variedad en las condiciones de borde aplicables para resolver varios tipos de problema. Todas las condiciones de borde usadas fijan el desplazamiento de los nodos en que se aplican.

A continuación se enumeran las diferentes condiciones de borde programadas en el algoritmo ”forcedisp”, su funcionamiento se ha explicado en la sección 3.2.1.

Las condiciones de borde que el código permite usar son los nodos fijados, las lineas fijadas y las superficies fijadas.Estas condiciones se pueden aplicar tanto en 2D como en 3D.

El algoritmo a seguir es el siguiente:

1. Identificación del nodo a fijar en su numeración global.
2. Realización de los cambios adecuados sobre la matriz de rigidez en la fila correspondiente al nodo fijo, como se ha explicado en la sección 3.2.1.

4.5. Elección del espacio de resolución del problema.

La ecuación constitutiva implementada en el código es la de la elasticidad, a partir de la ley de Hooke, que define una relación lineal entre las tensiones y las deformaciones del material. En función de la dimensión del problema, 2D o 3D, la matriz de rigidez que define la relación entre tensiones y deformaciones, tendrá una expresión u otra.

Las matrices de rigidez material particulares de cada tipo de problema se han mostrado ya en el apartado 2.1.4. Por tanto, el algoritmo de este módulo es sencillo, según el tipo de problema se calculará C de la forma correspondiente.

La función que contiene este algoritmo es ”Cmat”, que se ejecuta antes de cualquiera de las funciones ”Stiffmat”. Cabe destacar que ”Cmat” permite tener diferentes tipos de materiales para un mismo problema.

4.6. Mejora del tiempo de computación.

Para esta sección se han tomado como referencia los siguientes documentos:[15]

Esta quizás no es una funcionalidad del programa propiamente dicha, sin embargo debido a la magnitud de los problemas a tratar, conseguir un código capaz de solucionarlos de forma eficiente supone una parte importante del desarrollo.

El obstáculo principal de este tipo de programas en cuanto a la eficiencia es la manipulación de grandes matrices. A modo de ejemplo, un problema en 3D con 100000 nodos tendría una matriz de rigidez de 300000x300000. Esto no solo supone un problema de eficiencia, la mayoría de ordenadores no pueden llegar a almacenar matrices tan grandes.

Para solventar esta problemática se recurre a otra de las propiedades de este tipo de problemas: Las matrices de rigidez son matrices simétricas y muy dispersas, es decir, contienen una gran cantidad de ceros. Matlab tiene una funcionalidad que permite tratar eficientemente este tipo de matrices, la función ”sparse”.

El funcionamiento de esta función es el siguiente:

En vez de almacenar la matriz al completo, esta función solo almacena los elementos de esta que son diferentes de cero. Esto se consigue con la creación de tres vectores, dos con las coordenadas i y j del valor no-cero en la matriz y un tercero con dicho valor.

Otra forma de mejora del tiempo de computación en Matlab es la vectorización del código. Cómo Matlab esta optimizado para tratar con matrices, es mucho mas eficiente evitar bucles ”for” siempre que se pueda, substituyéndolo por una ecuación vectorial.

Finalmente, se ha realizado un estudio de los tiempos de cálculo del programa.

4.6.1. Estudio de tiempos de cálculo.

Para obtener información relevante sobre la eficiencia del código, se ha estudiado la duración de cada una de las funciones en varios estados del desarrollo. Usando esta información se han mejorado aquellas funciones de mayor coste para obtener un código con tiempos de computación asumibles.

En el estudio que se presenta a continuación se muestra la situación actual del código en cuanto a su eficiencia. Para realizar este estudio se calculan dos problemas diferentes, una viga de sección cuadrada cargada axialmente y esa misma viga cargada a flexión, con una carga puntual en el centro. Cada uno de los problemas se calcula con todos los tipos de elementos y se comparan los resultados obtenidos. Para los elementos triangulares y cuadriláteros el problema se ha simplificado a un problema 2D.

En este apartado se muestra un resumen de estos datos: La evolución del tiempo de computación (segundos) en función del número de elementos usados y el porcentaje de tiempo medio consumido por función.

Viga con carga axial.

Triángulos

Elementos	t. computación (s)
5000	6,1
10000	7,5
20000	10,9
40000	24,9
80000	94,2
Media	35,9

Cuadriláteros

Elementos	t. computación (s)
5000	5,9
10000	12,5
20000	32.5
40000	109,0
80000	427,7
Media	146,9

Hexaedros

Elementos	t. computación (s)
10000	24,4
20000	227,7
40000	972,9
80000	3547,9
Media	1193,5

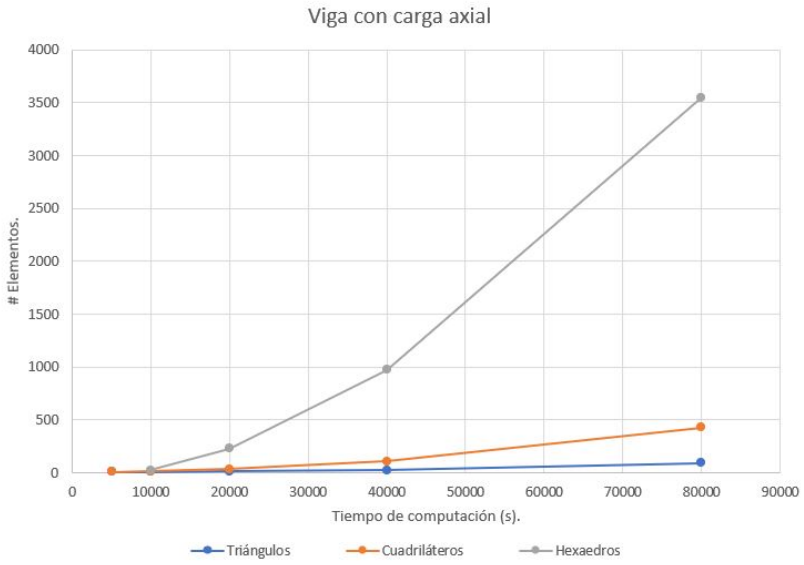


Figura 4.3: Gráfica de tiempo de computación.

Se muestra a continuación el porcentaje de tiempo que usa de media cada función:

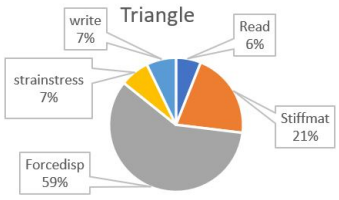


Figura 4.4: Porcentajes de tiempo de cada función.

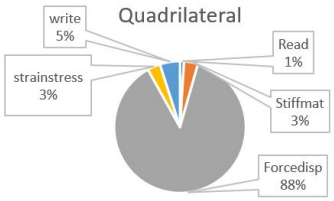


Figura 4.5: Porcentajes de tiempo de cada función.

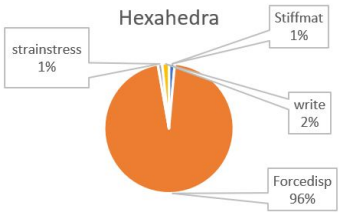


Figura 4.6: Porcentajes de tiempo de cada función.

Interpretación de los resultados:

El tiempo necesario para resolver este tipo de problemas es asumible incluso para los elementos más complejos y el mayor número de elementos.

Como era esperable, los elementos hexaédricos consumen la mayor cantidad de recursos. Observando los porcentajes de cada función se puede observar que la función más crítica es la función ”forcedisp”, esto es debido a que en esta función se resuelve el sistema $\bar{a} = \bar{\bar{K}}_{aux}^{-1}\bar{f}$ y se requiere el cálculo de la matriz de rigidez inversa, que es muy costoso.

4.6.2. Viga cargada a flexión.

Triángulos

Elementos	t. computación (s)
5000	1,9
10000	3,2
20000	9,2
40000	11,6
80000	34,1
Media	15,0

Cuadriláteros

Elementos	t. computación (s)
5000	3,2
10000	6,9
20000	19,1
40000	17,6
80000	40,9
Media	21,9

Hexaedros

Elementos	t. computación (s)
10000	18,1
20000	33,1
40000	140,1
80000	244,3
Media	108,9

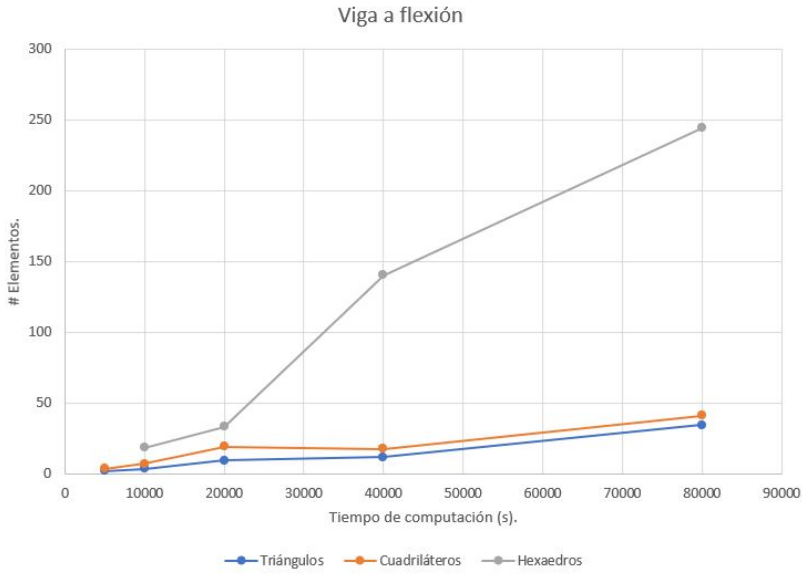


Figura 4.7: Gráfica de tiempo de computación.

Se muestra a continuación el porcentaje de tiempo que usa de media cada función:

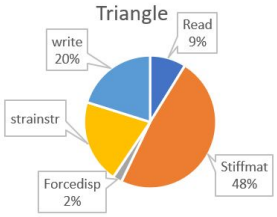


Figura 4.8: Porcentajes de tiempo de cada función.

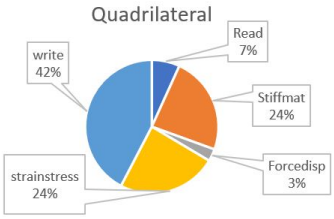


Figura 4.9: Porcentajes de tiempo de cada función.

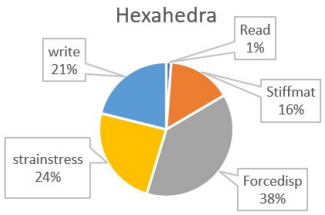


Figura 4.10: Porcentajes de tiempo de cada función.

Interpretación de los resultados:

Los tiempos de computación son asumibles e incluso se obtienen mejores resultados que en el anterior problema. El problema que consume un mayor número de recursos es obviamente el que usa elementos hexaédricos. Se observa en la gráfica tiempo de computación/nº elementos que la evolución del tiempo de computación es más lenta, esto se debe al tipo de problema.

En cuanto al porcentaje usado por cada función, se puede observar que en los elementos hexaédricos el cálculo de la inversa tiene ocupa gran parte del coste de computación. Como se comenta más adelante, se podría hacer mejoras al cálculo del sistema $\bar{a} = \bar{\bar{K}}_{aux}^{-1} \bar{f}$.

Capítulo 5

Verificación del código.

5.1. Proceso de verificación:

Para evaluar el funcionamiento del código éste se verifica mediante problemas con soluciones analíticas sencillas, los problemas usados en este caso son los mismos que para el estudio de computación.

El proceso de verificación es el siguiente:

- Cálculo analítico del problema. Obtención de la tensión y la deformación máxima.
- Modelado del problema mediante GiD y resolución de este usando el código desarrollado.
- Interpretación de los resultados obtenidos y comparación con los resultados analíticos.

Para la verificación se aprovecha que se ha realizado el cálculo con un numero muy elevado de elementos (80000) en el anterior apartado: En la mayoría de casos se toman los datos obtenidos con dicho número de elementos. Esto asegura que la malla es suficientemente fina para dar buenos resultados sin tener que realizar un estudio de convergencia de malla. Por contrapartida, al usar elementos más pequeños es más posible que aparezcan concentraciones de tensión elevadas.

Para la representación de resultados se mostrará la información obtenida en el post-proceso.

El campo de deformaciones se representa gráficamente mediante el dibujo de la geometría deformada tras aplicar la carga. Cabe destacar que el dibujo de la geometría deformada es una ampliación (x10) del comportamiento real de la estructura y por tanto da información cualitativa y no cuantitativa sobre la deformación de esta.

El campo de tensiones se representa mediante un gradiente de color sobre dicha geometría deformada, donde cada gradación de color representa un valor determinado de tensión.

5.2. Viga bi-apoyada con carga puntual en el centro:

5.2.1. Resultado analítico:

El problema que se resuelve en este apartado es una viga bi-apoyada de sección cuadrada de área 1 m2 y longitud 10 m.

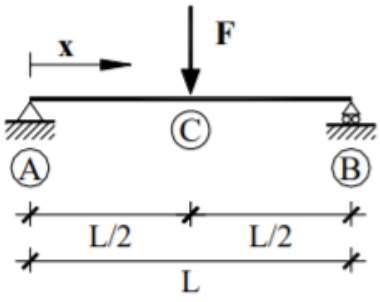


Figura 5.1: Esquema del problema. De [7].

$$L = 10m; \quad F = 10000N; \quad I = \frac{h^4}{12} = 1/12m^3;$$

$$W = \frac{h^3}{6} = 1/6m^3; \quad E = 210E9N/m^2;$$

La flecha máxima es:

$$Y_{max} = \frac{FL^3}{48EI} = -11,9 * 10^-6m;$$

El momento máximo es:

$$M_{max} = \frac{FL}{4} = 25000Nm;$$

La tensión máxima es:

$$\sigma = \frac{M_max}{W} = 150000N/m^2;$$

5.2.2. 2D - Elementos triangulares:

Calculado con 80000 elementos.

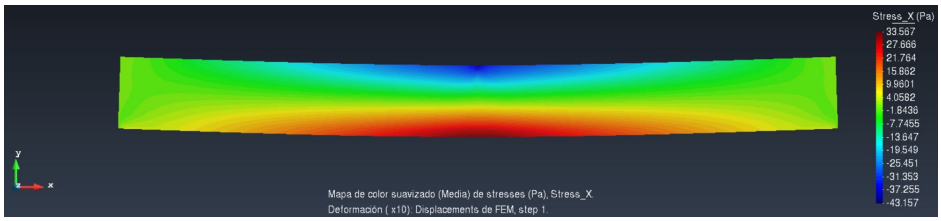


Figura 5.2: Campo de tensiones obtenido mediante el código.

Las tensiones máximas son:

$$\sigma 1 = -43,157N/m^2; \quad \sigma 2 = 33,567N/m^2;$$

La flecha máxima es:

$$Y_{max} = -12,129 * 10^-6m;$$

En los elementos triangulares las tensiones y las deformaciones unitarias calculadas con el programa no son correctas, aunque cualitativamente los campos concuerdan con lo esperado. Por el contrario, la flecha máxima da el resultado esperado. Se incluye una reflexión sobre esto en las conclusiones.

5.2.3. 2D - Elementos cuadriláteros:

Calculado con 80000 elementos.

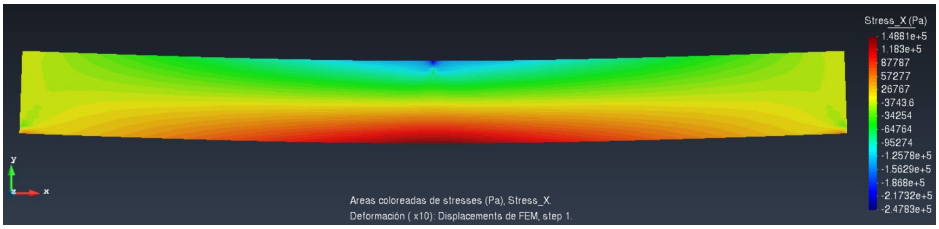


Figura 5.3: Campo de tensiones obtenido mediante el código.

Las tensiones máximas son:

$$\sigma 1 = -247830 N/m^2; \quad \sigma 2 = 148810 N/m^2;$$

La flecha máxima es:

$$Y_{max} = -12,429 * 10^{-}6m;$$

Todos los resultados obtenidos concuerdan con los calculados analíticamente, a excepción de las tensiones en los puntos fijados y cargados donde hay concentración de tensiones ($\sigma 1$).

5.2.4. 3D - Elementos Hexaédricos:

Calculado con 50000 elementos.

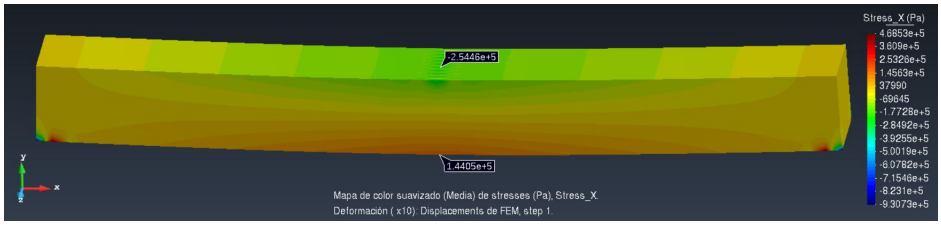


Figura 5.4: Campo de tensiones obtenido mediante el código.

La tensión máxima es:

$$\sigma 1 = -254460 N/m^2; \quad \sigma 2 = 144050 N/m^2;$$

La flecha máxima es:

$$Y_{max} = -12,998 * 10^-6 m;$$

Los valores obtenidos concuerdan con los analíticos, a excepción de $\sigma 1$ como se espera. La desviación en la flecha máxima se debe al desplazamiento en los puntos fijados, que se magnifica debido a la concentración de tensiones.

5.2.5. Comparación de resultados:

	$\sigma 1 (N/m^2)$	$\sigma 2 (N/m^2)$	$\mathbf{Y}_{max}(m)$
Resultados analíticos:	-150000	150000	$-11,9 * 10^-6$
2D - Elementos triangulares:	-43,157	33,567	$-12,129 * 10^-6$
2D - Elementos cuadriláteros:	-247830	148810	$-12,429 * 10^-6$
3D - Elementos hexaédricos	-254460	144050	$-12,998 * 10^-6$

Se observa que todos los valores de deformación máxima se aproximan razonablemente al valor analítico. Sin embargo, la tensión solo se calculan correctamente en los elementos cuadriláteros y hexaédricos, (los resultados de $\sigma 1$ no se tienen en cuenta ya que están sometidos a concentración de tensiones). Los elementos que consiguen una mejor aproximación son los cuadriláteros, mientras que los triangulares fallan al calcular la tensión.

5.3. Viga con carga axial:

5.3.1. Resultado analítico:

El problema a resolver es el de una viga de sección cuadrada de área 1 m2 y longitud 10 m cargada de forma axial.

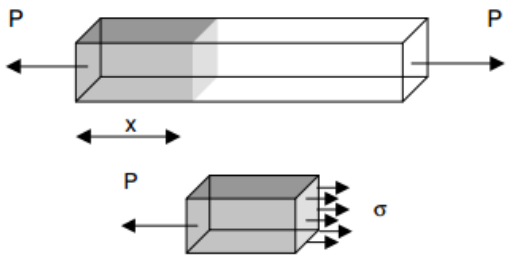


Figura 5.5: Esquema del problema. De [8].

$$L = 10m; \quad P = 10000N; \quad E = 210E9N/m^2;$$

La tensión es:

$$\sigma = \frac{F}{A} = 10000N/m^2;$$

La deformación unitaria es:

$$\varepsilon = \frac{\sigma}{E} = 47,62 * 10^-9;$$

La deformación es:

$$L = \epsilon L = 0,476 * 10^-6m;$$

5.3.2. 2D - Elementos triangulares:

Calculado con 80000 elementos.

Figura 5.6: Campo de tensiones obtenido mediante el código.

La tensión es:

$$\sigma = 2,5N/m^2;$$

La deformación unitaria es:

$$\varepsilon = 0,011905 * 10^{-}9;$$

La deformación medida es:

$$L = 0,477 * 10^{-}6m;$$

La tensión y la deformación unitaria en este tipo de elementos no concuerdan con los resultados analíticos. La deformación se calcula bien.

5.3.3. 2D - Elementos cuadriláteros:

Calculado con 80000 elementos.

Figura 5.7: Campo de tensiones obtenido mediante el código.

La tensión es:

$$\sigma = 10000 N/m^2;$$

La deformación unitaria es:

$$\varepsilon = 47,619 * 10^{-}9;$$

La deformación medida es:

$$L = 0,488 * 10^{-}6m;$$

Los resultados con este tipo de elementos concuerdan con los analíticos.

5.3.4. 3D - Elementos Hexaedros:

Calculado con 80000 elementos.

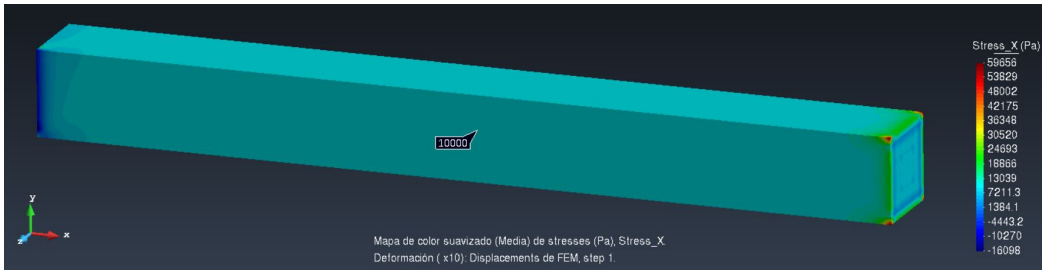


Figura 5.8: Campo de tensiones obtenido mediante el código.

La tensión es:

$$\sigma = 10000 N/m^2;$$

La deformación unitaria es:

$$\varepsilon = 47,619 * 10^{-}9;$$

La deformación medida es:

$$L = 0,480 * 10^{-}6m;$$

Los resultados con este tipo de elementos concuerdan con los analíticos.

5.3.5. Comparación de resultados:

	$\sigma(N/m^2)$	ϵ	$\mathrm{L}_f(m)$
Resultados analíticos:	10000	$47,62 * 10^{-}9$	$0,476 * 10^{-}6$
2D - Elementos triangulares:	2,5	$0,011905 * 10^{-}9$	$0,477 * 10^{-}6$
2D - Elementos cuadriláteros:	10000	$47,619 * 10^{-}9$	$0,488 * 10^{-}6$
3D - Elementos hexaédricos	10000	$47,619 * 10^{-}9$	$0,480 * 10^{-}6$

Comparando los resultados se extraen soluciones similares a las obtenidas con el otro problema. Los elementos triangulares no calculan correctamente ni la tensión ni la deformación unitaria. En los elementos cuadriláteros y hexaédricos los valores de σ y ϵ se calculan sin error, esto es debido a la simplicidad del problema calculado.

Capítulo 6

Conclusiones.

Este trabajo tiene como objetivo principal la programación de un código MEF para el análisis estructural que sea versátil. En este apartado se trata de evaluar si el objetivo se ha cumplido satisfactoriamente, también se recogen los principales aprendizajes adquiridos al realizar el proyecto y los posibles campos de mejora.

6.1. Evaluación de resultados:

Juzgando el objetivo final obtenido (código Matlab + Módulo de procesado GiD) se puede concluir que el objetivo principal del trabajo se ha cumplido en gran medida: El código es capaz de solucionar problemas estructurales tanto 2D cómo 3D con elementos cuadriláteros y hexaedros y permitiendo cierta flexibilidad en las condiciones de borde aplicables. Adicionalmente, el código es eficiente y no supone tiempos de computación inasumibles.

Sin embargo, cómo se puede observar en el apartado de resultados, los elementos triangulares no se han podido implementar satisfactoriamente. Esto se debe a que no se ha encontrado el error que impide el funcionamiento correcto de estos, algunas de las posibles fuentes de error se listan a continuación:

- Existe algún error en el código que realiza el calculo de tensiones y deformaciones.
- Existe un error en el cálculo de las deformaciones a partir de los desplazamientos.d

El módulo para elementos triangulares se ha mantenido en el código a pesar de su mal funcionamiento para poder repararlo en un futuro.

Además, se ha creído interesante mostrar esta implementación porque parte del aprendizaje de este proyecto es que la implementación numérica de una formulación no es una tarea trivial. Si bien la formulación que rige los elementos triangulares y cuadrados es la misma, la implementación de los primeros ha sido problemática y no se ha podido localizar el error en el código.

6.2. Posibles nuevas implementaciones al código.

Como se ha comentado anteriormente, el código cumple sus funciones. Sin embargo, existen funcionalidades que inicialmente estaban fuera del alcance del proyecto pero que en este punto se han estudiado para poderlas implementar en un futuro.

Algunas de estas funcionalidades se describen a continuación:

Cálculo dinámico (vibraciones): Mediante algunas modificaciones el código se podría preparar para calcular problemas dinámicos. Esto sería sencillo usando el método de Newmark: discretizando también el dominio temporal del problema usando el método implícito de Euler para evaluar la aceleración en cada instante de tiempo.

Elementos de 2do orden: A priori, sería relativamente sencillo implementar cualquier otro elemento. La misma sencillez existe para los elementos de 2do orden, ya que solamente se tendrían que usar funciones de forma de 2do orden. Adicionalmente sería necesario adaptar los módulos de post-proceso y Pre-proceso para implementar esta funcionalidad. Este tipo de elementos ofrecen una mejor aproximación al problema ya que usan un polinomio cuadrático (en lugar de uno lineal) para aproximar la solución.

Métodos de solución de un sistema lineal de ecuaciones: El método usado en este trabajo para solucionar esta ecuación consiste en calcular la inversa y evaluar $b = A^{-}1c$, este método de solución no es el más óptimo. Sin embargo, Matlab esta muy optimizado para solucionar este tipo de problema, por esta razón el cálculo de la inversa no supone un gran incremento del tiempo de computación (En los problemas tratados).

Para profundizar en el método de solución se podría implementar una función que decida el método de solución a aplicar según el tipo de matriz A que tiene el problema.

Ejecutable: Otra posible mejora del código es la creación de un ejecutable que permita realizar todo el proceso de resolución de problemas dentro del entorno de GiD. Para ello es necesario convertir el archivo Matlab en un ".exe" mediante Matlab compiler y a continuación usar la personalización de GiD para implementar el ".exe" entre los módulos de proceso.

Base de datos materiales: El código creado tiene información de un material base (acero estructural). Aunque se puede modificar los datos de este material una posibilidad de mejora del Pre-proceso sería crear una base de datos de materiales modificando el archivo ".mat" del problemtype.

6.3. Aprendizaje adquirido.

Este apartado se centra en recoger una lista de aprendizajes adquiridos durante el transcurso de este trabajo que facilitan la realización de cualquier otro proyecto similar.

Tiempo de computación y programación:

Gran parte del trabajo de programación se ha centrado en hacer el código eficiente y rápido, a continuación se muestran algunos de los aprendizajes adquiridos durante el proceso:

Al empezar a escribir el código en Matlab, es usual empezar a escribir todo el código en un mismo script. Sin embargo, tras familiarizarse un poco más con la forma de funcionar del lenguaje, se ha optado por escribir el código mediante funciones, cada una de ellas encargada de ejecutar una parte del algoritmo. Este cambio hace que el código sea más rápido y organizado, también ayuda a hacer el código más modular y fácil de modificar.

Uno de los desafíos principales de un código FEM es el manejo de grandes matrices, almacenando al completo las matrices del problema consume demasiada memoria. Una solución a esta problemática es usar las propiedades de las matrices a manejar para almacenarlas. En el caso del FEM una de las propiedades de las matrices de rigidez es que son matrices dispersas, con la función ”sparse” Matlab manejar este tipo de matrices tiene un coste mucho más bajo.

Esta función no es muy intuitiva y requiere saber usarla para evitar incrementar mucho el tiempo de computación; en cualquier caso es necesaria para un código MEF en Matlab.

Como se ha comentado, Matlab es muy eficiente al realizar operaciones matriciales y pierde eficiencia en los ”loops”. Para ello Matlab recomienda vectorizar el código, es decir reescribir estos ”loops” en forma de operaciones vectoriales. Vectorizar el código es complicado y no siempre posible. Sin embargo, si se plantea el código desde esta perspectiva desde un inicio, en vez de tener que reescribir el código, es posible conseguir un código muy compacto y eficiente en Matlab.

Una de las grandes preguntas al empezar a programar un código MEF es que lenguaje es el mejor para ello. La pregunta obvia al finalizarlo es si Matlab es una buena opción. A continuación se describen algunas de las ventajas y desventajas que ofrece el lenguaje:

- Funciones dentro del programa: tratar de escribir un código de este tipo en C++ sería mucho más complicado, ya que funciones como ”sparse”no existen y se deben crear.

- Vectorización del código: Implementar FEM es difícil de forma vectorizada, por esta razón quizá lenguajes como Pithon permitan programar el algoritmo de forma más sencilla.

Método:

Otra de las partes importantes de este trabajo es la relacionada con la propia aplicación del método. Es decir, la adaptación de las funciones de forma, la implementación de estas en el código, etc. En este apartado se destacan los aprendizajes más importantes adquiridos en relación al método.

Como se muestra en apartados anteriores, uno de los procesos necesarios para implementar cada tipo de elemento finito es la obtención de sus funciones de forma y su derivación. Este paso no es complicado ya que las funciones de forma y sus derivadas se suelen encontrar fácilmente en varios manuales, el proceso difícil es el de escribir estas derivadas en forma matricial de tal forma que al multiplicar esta matriz por la inversa del Jacobiano se obtenga la matriz de deformación del elemento (B).

Una de las decisiones tomadas durante el proceso de programación es si obtener las matrices de rigidez elementales fragmentadas o no. Obtenerlas fragmentadas significa evaluarlas de esta forma $Ke_{ij} = B_{ij}C_{ij}B_{ij}$, donde ij es la numeración nodal. Calculándolo de esta forma se evita hacer la operación con las matrices completas, pero es necesario usar un mayor número de bucles ”for”. En este trabajo se ha optado por calcular Ke entera en vez de por partes, esto aumenta la eficiencia del código aunque requiere obtener la expresión completa de B en su forma matricial.

Por ultimo se discute el uso de integración numérica. Para el tipo de elementos usados en este trabajo (1r orden) muchas de las integrales se pueden calcular analíticamente, aunque supone integrar grandes matrices. Se ha elegido usar la integración numérica de Gauss para evitar trabajo innecesario y para obtener un código simplificado y general, aun aumentando algo el tiempo de computación. Cabe destacar que con el código de integración usado al integrar polinomios (cómo las funciones de forma) el resultado es exacto.

6.4. Desglose de horas invertidas.

	Horas
Estudio de la formulación.	90
Programación del codigo.	120
Programación procesado	60
Validación	30

Bibliografía

[1] Omar M.Z. Typical fea procedures by commercial software. https://www.researchgate.net/figure/Typical-FEA-procedures-by-commercial-software_fig1_259115423, 2013.

[2] Bhatt S. A constant strain triangle element. http://homepages.rpi.edu/~bhatts8/plate_hole/Report.pdf.

[3] Martinez X. Apuntes de clase: Three bars between hinges. https://web.cimne.upc.edu/users/xmartinez/MAE656/01-Intro/MAE%20656%20-%2001-D03.pdf, 2012.

[4] Martinez X. Apuntes de clase: Single bar between hinges. https://web.cimne.upc.edu/users/xmartinez/MAE656/01-Intro/MAE%20656%20-%2001-D03.pdf, 2012.

[5] ANSYS Academics brochure. https://www.fluidcodes.bg/img/ANSYS%20Academics%20Brochure.pdf.

[6] CIMNE, Manual de ayuda GiD.

[7] Prontuario basico de estructuras simples. Viga simple apoyada. https://www.slideshare.net/LuisGuzman276/prontuario-basico-de-estructuras-simples.

[8] Nx-siemens implementation and piloting. Beam subjected to tension stress. https://www.theseus.fi/bitstream/handle/10024/131919/Altes_Francesc.pdf?sequence=1&isAllowed=y.

[9] Piaras Kelly. Stress and balance principles. http://homepages.engineering.auckland.ac.nz/~pkel015/SolidMechanicsBooks/Part_III/Chapter_3_Stress_

Mass_Momentum/Stress_Balance_Principles_Complete.pdf.

[10] Klaus-Jürgen Bathe. *Finite element procedures.* Prentice hall, 1996.

[11] R.L. Taylor O.C Zienkiewikz. *The finite element method.* McGraw-Hill, 1967.

[12] Joseph E. Flaherty. Numerical integration. http://www.cs.rpi.edu/~flaherje/pdf/fea6.pdf.

[13] Robert Greenlee. 2d triangular elements. http://www.unm.edu/~bgreen/ME360/2D%20Triangular%20Elements.pdf.

[14] Singiresu S. Rao. *The finite element method in engineering.* Butterworth-Heinemann, 2018.

[15] Pascal Getreuer. Writing fast matlab code. http://www.csc.kth.se/utbildning/kth/kurser/DN2255/ndiff13/matopt.pdf.

Anejos.

Anejo 1: Ejemplo de fichero de cálculo.

El archivo de cálculo mostrado es el correspondiente a un problema de carga axial sobre una viga rectangular, el problema se ha discretizado con 4 elementos para que la extensión de la imagen sea manejable.

```
4   % nº Elementos.
9   % nº Nodos.
1   % nº Materiales.
0   % nº Nodos fijos.
5   % nº Lineas fijas.
0   % nº Superfícies fijas.
0   % nº Nodos con carga.
2   % nº Lineas con carga.
0   % nº Superfícies con carga.
0   % nº Volúmenes con carga.
1   % nº Tipo de analisis.
2   % nº Tipo de elemento.
Nodcord     % Numeración y coordenadas de los nodos.
1,          10,           0,           0
2,          10,         0.5,           0
3,          10,           1,           0
4,           5,           0,           0
5,           5,         0.5,           0
6,           5,           1,           0
7,           0,           0,           0
8,           0,         0.5,           0
9,           0,           1,           0
Elemtype     % Información sobre los elementos.
4,5,8,7,1,1
1,2,5,4,1,2
5,6,9,8,1,3
2,3,6,5,1,4
Mat     % Información sobre los materiales.
1,0.3,80E9,210E9,1.0
Fixed_nodes
Fixed_lines
1,0,1,0
4,0,1,0
7,0,1,0
8,1,0,0
9,1,0,0
Fixed_surfaces
Loads_over_nodes
Loads_over_lines
2,1,2,10000.0,0.0,0.0
4,2,3,10000.0,0.0,0.0
Loads_over_surfaces
```

Figura 1: Ejemplo de archivo de cálculo.

Cada uno de los vectores del archivo de cálculo debe contener la información en el orden descrito en el trabajo, se deben usar comas a modo de separador.

Anejo 2: Código Matlab.

A continuación se muestra el código escrito desglosado por funciones. Algunas de las lineas (Creación de vectores, operaciones con matrices) son demasiado largas para caber en el documento, se adjunta el código de todas formas para proporcionar una idea de su funcionamiento.

FEMproject.

```
clc,clear var,close all

%-------------------------------------------------------------------------
%set Filename to the name of the file to read.
FILE = 'hexahedra_flex';
[Filename,meshfile,resfile] = filenaming(FILE);

%-------------------------------------------------------------------------
%reads Filename into matlab variables.
read;

%-------------------------------------------------------------------------
%computes C matrix (compliance matrix) of the materials.
C = Cmat(Numbmat,Mat,Analysistype);

%-------------------------------------------------------------------------
%computes element stiffness matrix and ensambles global stiffness matrix.
if Elemtype == 1
[KGf,Be] = Stiffmattriangle(Elem,nodcord,Elemcon,Nod,C,Mat);
end

if Elemtype == 2
[KGf,Be] = Stiffmatquadrilateral(Elem,nodcord,Elemcon,C,Mat,Elemtype);
end

if Elemtype == 3
[KGf,Be] = Stiffmathexahedra(Elem,nodcord,Elemcon,C,Elemtype);
end

%-------------------------------------------------------------------------
%ensambles nodal load vector.
faux = boundcond(LoadNod,Nod,Nloadnod,nodcord,Nloadline,Loadline,Mat,Nloadsurf,Loadsurf,Nloadvolu

%-------------------------------------------------------------------------
%ensambles auxiliar stiffness matrix and solves for displacements and forces.
[f,a] = forcedisp(KGf,Nfixnod,FixNod,faux,Nfixline,fixline,Nfixsurf,fixsurf,Analysistype);

%-------------------------------------------------------------------------
%computes strain and stress of each element.
[Strain,Stress] = StrainStress(Elem,Nod,a,Elemcon,C,Elemtype,Be,nodcord);

%-------------------------------------------------------------------------
%writes results on text file.
fileID = write(a,f,Stress,Strain,Elem,Nod,nodcord,Elemcon,meshfile,resfile,Elemtype);
```

read.

```
Elem = readmatrix(Filename,'range',[1 1 1 1],'delimiter',','); %Sets Elem to the number of elemen
Nod = readmatrix(Filename,'range',[2 1 2 1],'delimiter',','); %Sets Nod to the number of nodes in
Numbmat= readmatrix(Filename,'range',[3 1 3 1],'delimiter',','); %Sets Numbmat to the number of m
%------------------------------------------------------------------------
Nfixnod= readmatrix(Filename,'range',[4 1 4 1],'delimiter',','); %Sets Nfixnod to the number of f
Nfixline = readmatrix(Filename,'Range',[5, 1, 5, 1],'delimiter',','); %Sets Nfixline to the numbe
Nfixsurf = readmatrix(Filename,'Range',[6, 1, 6, 1],'delimiter',','); %Sets Nfixsurf to the numbe
Nloadnod= readmatrix(Filename,'range',[7 1 7 1],'delimiter',','); %Sets Nloadnod to the number of
Nloadline = readmatrix(Filename,'Range',[8 1 8 1],'delimiter',','); %Sets Nloadline to the number
Nloadsurf = readmatrix(Filename,'Range',[9 1 9 1],'delimiter',','); %Sets Nloadsurface to the num
Nloadvolume = readmatrix(Filename,'Range',[10, 1, 10, 1],'delimiter',','); %Sets Nloadvolume to t
Ndisp = readmatrix(Filename,'Range',[11, 1, 11, 1],'delimiter',','); %Sets Ndisp to the number of
Analysistype = readmatrix(Filename,'Range',[12, 1, 12, 1],'delimiter',','); %Analysistype; 1=plane
Elemtype = readmatrix(Filename,'Range',[13, 1, 13, 1],'delimiter',','); %Elemtype; 1=triangle, 2=q
%------------------------------------------------------------------------

%sets dim to the number of dimensions of the problem.
if Analysistype==1||Analysistype==2
dim=2;
end

if Analysistype==3
dim=3;
end
%

%------------------------------------------------------------------------

nodcord=readmatrix(Filename,'range',[15 1 14+Nod 1+dim],'delimiter',','); %Uses Nod to set nodcor

if Elemtype==1
Elemcon=readmatrix(Filename,'range',[16+Nod 1 15+Nod+Elem 5],'delimiter',','); %Uses Elem to set
end

if Elemtype==2
Elemcon=readmatrix(Filename,'range',[16+Nod 1 15+Nod+Elem 6],'delimiter',','); %Uses Elem to set
end

if Elemtype==3
Elemcon=readmatrix(Filename,'range',[16+Nod 1 15+Nod+Elem 10],'delimiter',','); %Uses Elem to set
end
```

```
Mat=readmatrix(Filename,'Range',[(17+Nod+Elem) 1 (16+Nod+Elem+Numbmat) 5],'delimiter',','); %Uses

%-------------------------------------------------------------------------

if Nfixnod==0 %executes when there's no nodes fixed in the problem.
FixNod=0;
else
FixNod=readmatrix(Filename,'range',[(18+Nod+Elem+Numbmat), 1, (17+Nod+Elem+Numbmat+Nfixnod) 1+dir
end


if Nfixline==0 %executes when there's no lines fixed in the problem.
fixline=0;
else
fixline=readmatrix(Filename,'range',[(19+Nod+Elem+Numbmat+Nfixnod), 1, (18+Nod+Elem+Numbmat+Nfixn
end


if Nfixsurf==0 %executes when there's no lines fixed in the problem.
fixsurf=0;
else
fixsurf=readmatrix(Filename,'range',[(20+Nod+Elem+Numbmat+Nfixnod+Nfixline), 1, (19+Nod+Elem+Numb
end


if Nloadnod==0 %executes when there's no loaded nodes in the problem.
LoadNod=0;
else
LoadNod=readmatrix(Filename,'Range',[(21+Nod+Elem+Numbmat+Nfixnod+Nfixline+Nfixsurf), 1, (20+Nod+
end


if Nloadline==0 %executes when there's no loaded lines in the problem.
Loadline=0;
else
Loadline=readmatrix(Filename,'Range',[(22+Nod+Elem+Numbmat+Nfixnod+Nfixline+Nfixsurf+Nloadnod), 1
end


if Nloadsurf==0 %executes when there's no loaded surfaces in the problem.
Loadsurf=0;
else

if Elemtype==1
Loadsurf=readmatrix(Filename,'Range',[(23+Nod+Elem+Numbmat+Nfixnod+Nfixline+Nfixsurf+Nloadnod+Nlo
end

if Elemtype==2
Loadsurf=readmatrix(Filename,'Range',[(23+Nod+Elem+Numbmat+Nfixnod+Nfixline+Nfixsurf+Nloadnod+Nlo
end

if Elemtype==3
Loadsurf=readmatrix(Filename,'Range',[(23+Nod+Elem+Numbmat+Nfixnod+Nfixline+Nfixsurf+Nloadnod+Nlo
end

end
```

```
if Nloadvolume==0 %executes when there's no loaded surfaces in the problem.
Loadvolume=0;
else
Loadvolume=readmatrix(Filename,'Range',[(24+Nod+Elem+Numbmat+Nfixnod+Nfixline+Nfixsurf+Nloadnod+N
end


if Ndisp==0 %executes when there's no imposed strains in the problem.
disp=0;
else
disp=readmatrix(Filename,'Range',[(25+Nod+Elem+Numbmat+Nfixnod+Nfixline+Nfixsurf+Nloadnod+Nloadli
end
```

Cmat.

```
function C = Cmat(Numbmat,Mat,Analysistype)

i = 1:Numbmat;  %repeats loop for every material.


if Analysistype==1  %compliance matrix for plane stress problems.

C = zeros(3,3,Numbmat);

rowmat = find(Mat(:,1) == i);  %locates the material i in the matrix Mat.

C(:,:,i) = (1/(1-Mat(rowmat,2)*Mat(rowmat,2)).*[Mat(rowmat,4) (Mat(rowmat,4)*Mat(rowmat,2)) 0;(Ma
0 ((1-(Mat(rowmat,2)*Mat(rowmat,2)))*Mat(rowmat,3))]);%computes C matrix. stores as C(:,:i)

end

%-------------------------------------------------------------------------

if Analysistype==2  %Compliance matrix for plane strain problems.

C = zeros(3,3,Numbmat);

rowmat = find(Mat(:,1) == i);  %locates the material i in the matrix Mat.

a=1-Mat(rowmat,2)^2;

b=Mat(rowmat,2)+Mat(rowmat,2)^2;

C(:,:,i) = (1/(a^2-b^2)).*[a*Mat(rowmat,4) b*(Mat(rowmat,4)*Mat(rowmat,2)) 0;b*(Mat(rowmat,4)*Mat
0 (a^2-b^2)*Mat(rowmat,3)];%computes C matrix. stores as C(:,:i)

end

%-------------------------------------------------------------------------

if Analysistype==3  %Compliance matrix for 3D problems.

C = zeros(6,6,Numbmat);

rowmat = find(Mat(:,1) == i);  %locates the material i in the matrix Mat.

C(:,:,i) = ((Mat(rowmat,4)*(1-Mat(rowmat,2)))/((1+Mat(rowmat,2))*(1-2*Mat(rowmat,2)))).*[1 Mat(ro
```

```
end

end
```

Stiffmattriangle.

```
function [KGf,Be] = Stiffmattriangle(Elem,nodcord,Elemcon,Nod,C,Mat)

%variable prealocation
Be = zeros(3,6,Nod);
a = zeros(Elem*36,1);
b = zeros(Elem*36,1);
KG = zeros(Elem*36,1);
%
%-------------------------------------------------------------------------

iter = 1;

for e=1:Elem %repeats loop for every element.

%-------------------------------------------------------------------------------------------------

Ke = zeros(6);

row = find(Elemcon(:,5) == e); %locates e in the Elemcon matrix.
row1 = find(nodcord(:,1) == Elemcon(row,1));%locates node 1 of the element in the Nodcord matrix.
row2 = find(nodcord(:,1) == Elemcon(row,2));%locates node 2 of the element in the Nodcord matrix.
row3 = find(nodcord(:,1) == Elemcon(row,3));%locates node 3 of the element in the Nodcord matrix.
rowt = Mat(:,1) == Elemcon(row,4);%locates material of the element in the Mat matrix.

v1 = [nodcord(row1,2), nodcord(row1,3), 0];

v2 = [nodcord(row2,2), nodcord(row2,3), 0];

v3 = [nodcord(row3,2), nodcord(row3,3), 0];

A = 1/2*norm(cross(v2-v1,v3-v1));%computes the area of the element

Be(:,:,row) = [ v2(2)-v3(2) 0 v3(2)-v1(2) 0 v1(2)-v2(2) 0; 0 v3(1)-v2(1) 0 v1(1)-v3(1) 0 v2(1)-v1

Ke(:,:) = (Mat(rowt,5)/(4*A)).*(transpose(Be(:,:,e))*(C(:,:,Elemcon(row,4))*Be(:,:,e)));%computes

%-------------------------------------------------------------------------------------------------

El = Elemcon(row,1:3);

for k = 1:3%loop for every position in KG.
for m = 1:3%""

j = El(k);
i = El(m);

a(iter,1) = 2*i-1;
b(iter,1) = 2*j-1;
KG(iter,1) = Ke(2*m-1,2*k-1);%stores corresponding value of the Ke in the Kglobal.
```

```
iter = iter + 1;

a(iter,1) = 2*i-1;
b(iter,1) = 2*j;
KG(iter,1) = Ke(2*m-1,2*k);%stores corresponding value of the Ke in the Kglobal.

iter = iter + 1;

a(iter,1) = 2*i;
b(iter,1) = 2*j-1;
KG(iter,1) = Ke(2*m,2*k-1);%stores corresponding value of the Ke in the Kglobal.

iter = iter + 1;

a(iter,1) = 2*i;
b(iter,1) = 2*j;
KG(iter,1) = Ke(2*m,2*k);%stores corresponding value of the Ke in the Kglobal.

iter = iter + 1;

end
end
%-------------------------------------------------------------------------------------------------
end
KGf = sparse(a,b,KG);
end
```

Stiffmatquadrilateral.

```
function [KGf,Be] = Stiffmatquadrilateral(Elem,nodcord,Elemcon,C,Mat,Elemtype)

%variable prealocation
Be = [];
a = zeros(Elem*64,1);
b = zeros(Elem*64,1);
KG = zeros(Elem*64,1);
%
%-------------------------------------------------------------------------

iter = 1;

for e=1:Elem %repeats loop for every element.

%-------------------------------------------------------------------------------------------------

Ke = zeros(8);

row = find(Elemcon(:,6) == e); %locates e in the Elemcon matrix.

rowt = Mat(:,1) == Elemcon(row,5);%locates material of the element in the Mat matrix.

for p=1:2 %loop over integration points.
for q=1:2 %"

r = ((-1)^q)*0.5773502692; %computes x natural cordinate of the point pq.
```

```
s = ((-1)^p)*0.5773502692;  %computes y natural cordinate of the point pq.

t = 0;


J = Jacobianmat(r,s,t,nodcord,Elemcon,Elemtype,e);

B = Deformationmat(r,s,t,J,Elemtype);

F = (transpose(B)*C(:,:,Elemcon(row,5))*B).*det(J);  %computes value of the function to integrate


Ke = Ke + Mat(rowt,5).*F;  %Computes numerical integration for Ke.

end
end

%-------------------------------------------------------------------------------------------------

El = Elemcon(row,1:4);

for k=1:4 %loop for every node of the element.
for m=1:4 %""

j = El(k);
i = El(m);

a(iter,1) = 2*i-1;
b(iter,1) = 2*j-1;
KG(iter,1)= Ke(2*m-1,2*k-1);%stores corresponding value of the Ke in the Kglobal.

iter = iter + 1;

a(iter,1) = 2*i-1;
b(iter,1) = 2*j;
KG(iter,1)= Ke(2*m-1,2*k);%stores corresponding value of the Ke in the Kglobal.

iter = iter + 1;

a(iter,1) = 2*i;
b(iter,1) = 2*j-1;
KG(iter,1)= Ke(2*m,2*k-1);%stores corresponding value of the Ke in the Kglobal.

iter = iter + 1;

a(iter,1) = 2*i;
b(iter,1) = 2*j;
KG(iter,1)= Ke(2*m,2*k);%stores corresponding value of the Ke in the Kglobal.

iter = iter + 1;

end
end
%-------------------------------------------------------------------------------------------------

end
KGf = sparse(a,b,KG);
end
```

Stiffmathexahedra.

```
function [KGf,Be] = Stiffmathexahedra(Elem,nodcord,Elemcon,C,Elemtype)

%variable prealocation
Be = [];
a = zeros(Elem*576,1);
b = zeros(Elem*576,1);
KG = zeros(Elem*576,1);
%

%-------------------------------------------------------------------------

iter = 1;

for e=1:Elem %repeats loop for every element.

%-------------------------------------------------------------------------------------------------------

Ke = zeros(24);

row = find(Elemcon(:,10) == e); %locates e in the Elemcon matrix.

for p=1:2 %loop over integration points.
for q=1:2 %"
for d=1:2 %"

r = ((-1)^p)*0.577350269189626; %computes x natural cordinate of the point pqd.

s = ((-1)^q)*0.577350269189626; %computes y natural cordinate of the point pqd.

t = ((-1)^d)*0.577350269189626; %computes z natural cordinate of the point pqd.


J = Jacobianmat(r,s,t,nodcord,Elemcon,Elemtype,e);

B = Deformationmat(r,s,t,J,Elemtype);

F = (transpose(B)*C(:,:,Elemcon(row,9))*B).*det(J);


Ke = Ke + F; %Computes numerical integration for Ke.

end
end
end

%-------------------------------------------------------------------------------------------------------

El = Elemcon(row,1:8);


for k=1:8 %loop for every node of the element.
for m=1:8 %""

j = El(k);
i = El(m);
```

```
a(iter,1) = (3*i-2);%stores corresponding value of the Ke in the Kglobal.
b(iter,1) = (3*j-2);
KG(iter,1) = Ke(3*m-2,3*k-2);

iter = iter + 1;

a(iter,1) = (3*i-2);%stores corresponding value of the Ke in the Kglobal.
b(iter,1) = (3*j-1);
KG(iter,1) = Ke(3*m-2,3*k-1);

iter = iter + 1;

a(iter,1) = (3*i-2);%stores corresponding value of the Ke in the Kglobal.
b(iter,1) = (3*j);
KG(iter,1) = Ke(3*m-2,3*k);

iter = iter + 1;

a(iter,1) = (3*i-1);%stores corresponding value of the Ke in the Kglobal.
b(iter,1) = (3*j-2);
KG(iter,1) = Ke(3*m-1,3*k-2);

iter = iter + 1;

a(iter,1) = 3*i-1;%stores corresponding value of the Ke in the Kglobal.
b(iter,1) = 3*j-1;
KG(iter,1) = Ke(3*m-1,3*k-1);

iter = iter + 1;

a(iter,1) = 3*i-1;%stores corresponding value of the Ke in the Kglobal.
b(iter,1) = 3*j;
KG(iter,1) = Ke(3*m-1,3*k);

iter = iter + 1;

a(iter,1) = 3*i;%stores corresponding value of the Ke in the Kglobal.
b(iter,1) = 3*j-2;
KG(iter,1) = Ke(3*m,3*k-2);

iter = iter + 1;

a(iter,1) = 3*i;%stores corresponding value of the Ke in the Kglobal.
b(iter,1) = 3*j-1;
KG(iter,1) = Ke(3*m,3*k-1);

iter = iter + 1;

a(iter,1) = 3*i;%stores corresponding value of the Ke in the Kglobal.
b(iter,1) = 3*j;
KG(iter,1) = Ke(3*m,3*k);

iter = iter + 1;
end
end
%-------------------------------------------------------------------------------------------------

end
KGf = sparse(a,b,KG);
```

```
end
```

Deformationmat.

```
function B = Deformationmat(r,s,t,J,Elemtype)%~ substitutes t


if Elemtype == 2

%variable prealocation
du = zeros(2,8);

dv = zeros(2,8);

B = zeros(3,8);
%

du = J\((1/4)*[1+s, 0, -(1+s), 0, -(1-s), 0, 1-s, 0; 1+r, 0, 1-r, 0, -(1-r), 0, -(1+r), 0]);

dv = J\((1/4)*[0, 1+s, 0, -(1+s), 0, -(1-s), 0, 1-s; 0, 1+r, 0, 1-r, 0, -(1-r), 0, -(1+r)]);

B = [du(1,1), du(1,2), du(1,3), du(1,4), du(1,5), du(1,6), du(1,7), du(1,8); dv(2,1), dv(2,2), dv

end



if Elemtype == 3

%variable prealocation
du = zeros(3,24);

dv = zeros(3,24);

dw = zeros(3,24);

B = zeros(6,24);
%

du = J\((1/8)*[-(1-s)*(1-t),0,0,(1-s)*(1-t),0,0,(1+s)*(1-t),0,0,-(1+s)*(1-t),0,0,-(1-s)*(1+t),0,0

dv = J\((1/8)*[0,-(1-s)*(1-t),0,0,(1-s)*(1-t),0,0,(1+s)*(1-t),0,0,-(1+s)*(1-t),0,0,-(1-s)*(1+t),0

dw = J\((1/8)*[0,0,-(1-s)*(1-t),0,0,(1-s)*(1-t),0,0,(1+s)*(1-t),0,0,-(1+s)*(1-t),0,0,-(1-s)*(1+t)

B = [du(1,1), du(1,2), du(1,3), du(1,4), du(1,5), du(1,6), du(1,7), du(1,8), du(1,9), du(1,10), d

end


end
```

Jacobianmat.

```
function J = Jacobianmat(r,s,t,nodcord,elemcon,Elemtype,e) %~ substitutes t

if Elemtype==2

 %variable prealocation
J = zeros(2);
%

dxdr = (1/4)*((1+s)*nodcord(elemcon(e,1),2) -(1+s)*nodcord(elemcon(e,2),2) -(1-s)*nodcord(elemcor

dxds = (1/4)*((1+r)*nodcord(elemcon(e,1),2) +(1-r)*nodcord(elemcon(e,2),2) -(1-r)*nodcord(elemcor

dydr = (1/4)*((1+s)*nodcord(elemcon(e,1),3) -(1+s)*nodcord(elemcon(e,2),3) -(1-s)*nodcord(elemcor

dyds = (1/4)*((1+r)*nodcord(elemcon(e,1),3) +(1-r)*nodcord(elemcon(e,2),3) -(1-r)*nodcord(elemcor

J = [dxdr, dydr; dxds, dyds];

end


if Elemtype==3

 %variable prealocation
J = zeros(3);
%

dxdr = (1/8)*(-(1-s)*(1-t)*nodcord(elemcon(e,1),2)+(1-s)*(1-t)*nodcord(elemcon(e,2),2)+(1+s)*(1-t

dydr = (1/8)*(-(1-s)*(1-t)*nodcord(elemcon(e,1),3)+(1-s)*(1-t)*nodcord(elemcon(e,2),3)+(1+s)*(1-t

dzdr = (1/8)*(-(1-s)*(1-t)*nodcord(elemcon(e,1),4)+(1-s)*(1-t)*nodcord(elemcon(e,2),4)+(1+s)*(1-t


dxds = (1/8)*(-(1-r)*(1-t)*nodcord(elemcon(e,1),2)-(1+r)*(1-t)*nodcord(elemcon(e,2),2)+(1+r)*(1-t

dyds = (1/8)*(-(1-r)*(1-t)*nodcord(elemcon(e,1),3)-(1+r)*(1-t)*nodcord(elemcon(e,2),3)+(1+r)*(1-t

dzds = (1/8)*(-(1-r)*(1-t)*nodcord(elemcon(e,1),4)-(1+r)*(1-t)*nodcord(elemcon(e,2),4)+(1+r)*(1-t


dxdt = (1/8)*(-(1-r)*(1-s)*nodcord(elemcon(e,1),2)-(1+r)*(1-s)*nodcord(elemcon(e,2),2)-(1+r)*(1+s

dydt = (1/8)*(-(1-r)*(1-s)*nodcord(elemcon(e,1),3)-(1+r)*(1-s)*nodcord(elemcon(e,2),3)-(1+r)*(1+s

dzdt = (1/8)*(-(1-r)*(1-s)*nodcord(elemcon(e,1),4)-(1+r)*(1-s)*nodcord(elemcon(e,2),4)-(1+r)*(1+s


J = [dxdr, dydr, dzdr; dxds, dyds, dzds; dxdt, dydt, dzdt];

end


end
```

boundcond.

```
function faux= boundcond(LoadNod,Nod,Nloadnod,nodcord,Nloadline,Loadline,Mat,Nloadsurf,Loadsurf,N

if Analysistype==1||Analysistype==2
    dim=2;
else
    dim=3;
end


if Elemtype==1
    Nnod=3;
else
    Nnod=4;
end


faux = zeros(dim,Nod);%create matrix, column x,y and z for each node.

%-------------------------------------------------------------------------
if Nloadnod>0%executes if loaded nodes exists.

for i=1:Nloadnod%loop for every loaded node.

faux(1,LoadNod(i,1)) = LoadNod(i,2);%Load on x direction applied on the node.
faux(2,LoadNod(i,1)) = LoadNod(i,3);%Load on y direction applied on the node.

if dim==3%if the problem is in 3D

faux(3,LoadNod(i,1)) = LoadNod(i,4);%Load on z direction applied on the node.

end

end

end

%-------------------------------------------------------------------------
if Nloadline>0%only possible in 2d (GiD),executes when loaded lines exist.

Nload = unique(Loadline(:,4:5),'rows');%computes the number of different loading conditions.

for i = 1:length(Nload(:,1))%loop for each loading condition.

Loadelem = find(ismember(Loadline(:,4:5),Nload(i,:),'rows'));%searches the elements with the load

L = 0;

for j = 1:length(Loadelem)%loop over the elements with the load applied

L = L + (norm(nodcord(Loadline(Loadelem(j),2),2:3)-nodcord(Loadline(Loadelem(j),3),2:3)));%lenght

end

loadednod = unique(Loadline(Loadelem,2:3));%finds the nodes of the loaded elements (loaded nodes)

nodloadx = Nload(i,1)*L*Mat(5)/length(loadednod);%computes x component of the load applied on eve
```

```
nodloady = Nload(i,2)*L*Mat(5)/length(loadednod); %computes y component of the load applied on eve

for k = 1:length(loadednod) %loop for all the loaded nodes

faux(1,loadednod(k)) = faux(1,loadednod(k)) + nodloadx; %force in x direction applied on node load
faux(2,loadednod(k)) = faux(2,loadednod(k)) + nodloady; %force in y direction applied on node load

end
end
end

%-------------------------------------------------------------------------
if Nloadsurf>0 %executes when loaded surfaces exist.

Nload = unique(Loadsurf(:,2+Nnod:1+dim+Nnod),'rows'); %computes the number of different loading co

for i = 1:length(Nload(:,1)) %loop for each loading condition.

Loadelem = find(ismember(Loadsurf(:,2+Nnod:1+dim+Nnod),Nload(i,:),'rows')); %searches the elements

A = 0;

for j = 1:length(Loadelem) %loop over the elements with the load applied

if dim==2
nodcord(:,4) = 0;
end

v1 = [nodcord(Loadsurf(Loadelem(j),2),2), nodcord(Loadsurf(Loadelem(j),2),3), nodcord(Loadsurf(Lo

v2 = [nodcord(Loadsurf(Loadelem(j),3),2), nodcord(Loadsurf(Loadelem(j),3),3), nodcord(Loadsurf(Lo

v3 = [nodcord(Loadsurf(Loadelem(j),4),2), nodcord(Loadsurf(Loadelem(j),4),3), nodcord(Loadsurf(Lo

%calcula area elemento


if Elemtype == 2||Elemtype == 3

A = A + norm(cross(v2-v1,v3-v1)); %area where the force is applied for quadrilateral elements.

else

A = A + 1/2*norm(cross(v2-v1,v3-v1)); %area where the force is applied for triangular or tetrahedr

end

end


loadednod = unique(Loadsurf(Loadelem,2:1+Nnod)); %finds the nodes of the loaded elements (loaded n

if dim == 2

nodloadx = Nload(i,1)*A*Mat(5)/length(loadednod); %computes x component of the load applied on eve
nodloady = Nload(i,2)*A*Mat(5)/length(loadednod); %computes y component of the load applied on eve

else
```

```
nodloadx = Nload(i,1)*A/length(loadednod);%computes x component of the load applied on every node
nodloady = Nload(i,2)*A/length(loadednod);%computes y component of the load applied on every node
nodloadz = Nload(i,3)*A/length(loadednod);%computes z component of the load applied on every node

end


for k = 1:length(loadednod)%loop for all the loaded nodes

faux(1,loadednod(k)) = faux(1,loadednod(k)) + nodloadx;%force in x direction applied on node
faux(2,loadednod(k)) = faux(2,loadednod(k)) + nodloady;%force in y direction applied on node

if dim == 3

faux(3,loadednod(k)) = faux(3,loadednod(k)) + nodloadz;%force in z direction applied on node

end

end
end
end

%--------------------------------------------------------------------------
if Nloadvolume>0%only for 3D

Nload = unique(Loadvolume(:,2:4),'rows');%computes the number of different loading conditions.

for i = 1:length(Nload(:,1))%loop for each loading condition.

Loadelem = find(ismember(Loadvolume(:,2:4),Nload(i,:),'rows'));%searches the elements with the lo

V = 0;

for j = 1:length(Loadelem)%loop over the elements with the load applied

row = find(Elemcon(:,6) == Loadelem(j,1));

v1 = [nodcord(Elemcon(row,1),2), nodcord(Elemcon(row,1),3), nodcord(Elemcon(row,1),4)];

v2 = [nodcord(Elemcon(row,2),2), nodcord(Elemcon(row,2),3), nodcord(Elemcon(row,2),4)];

v3 = [nodcord(Elemcon(row,3),2), nodcord(Elemcon(row,3),3), nodcord(Elemcon(row,3),4)];

v4 = [nodcord(Elemcon(row,4),2), nodcord(Elemcon(row,4),3), nodcord(Elemcon(row,4),4)];


V = V + norm(det([v1(1), v2(1), v3(1), v4(1); v1(2), v2(2), v3(2), v4(2); v1(3), v2(3), v3(3), v4
end

loadednod = unique(Elemcon(Loadelem,2:4));%finds the nodes of the loaded elements (loaded nodes)

nodloadx = Nload(i,1)*V/length(loadednod);%computes x component of the load applied on every node
nodloady = Nload(i,2)*V/length(loadednod);%computes y component of the load applied on every node
nodloadz = Nload(i,3)*V/length(loadednod);%computes z component of the load applied on every node


for k = 1:length(loadednod)%loop for all the loaded nodes

faux(1,loadednod(k)) = faux(1,loadednod(k)) + nodloadx;%force in x direction applied on node
```

```
faux(2,loadednod(k)) = faux(2,loadednod(k)) + nodloady; %force in y direction applied on node
faux(3,loadednod(k)) = faux(3,loadednod(k)) + nodloadz; %force in z direction applied on node

end
end
end

%-------------------------------------------------------------------------
if Ndisp>0

for i=1:Ndisp

row = find(Elemcon(:,5) == disp(i,1));

faux(1,Elemcon(row,1)) = faux(1,Elemcon(row,1)) + 0.5*Mat(5)*(Be(1,1,i)*(C(1,1)*disp(i,2)-C(1,2)*
faux(2,Elemcon(row,1)) = faux(2,Elemcon(row,1)) + 0.5*Mat(5)*(Be(2,2,i)*(C(2,1)*disp(i,2)-C(2,2)*
faux(1,Elemcon(row,2)) = faux(1,Elemcon(row,2)) + 0.5*Mat(5)*(Be(1,3,i)*(C(1,1)*disp(i,2)-C(1,2)*
faux(2,Elemcon(row,2)) = faux(2,Elemcon(row,2)) + 0.5*Mat(5)*(Be(2,4,i)*(C(2,1)*disp(i,2)-C(2,2)*
faux(1,Elemcon(row,3)) = faux(1,Elemcon(row,3)) + 0.5*Mat(5)*(Be(1,5,i)*(C(1,1)*disp(i,2)-C(1,2)*
faux(2,Elemcon(row,3)) = faux(2,Elemcon(row,3)) + 0.5*Mat(5)*(Be(2,6,i)*(C(2,1)*disp(i,2)-C(2,2)*

end
end

%-------------------------------------------------------------------------
faux = reshape(faux,[dim*Nod,1]); %reshape matrix into final nodal force vector.
end
```

forcedisp.

```
function [f,a] =  forcedisp(KGf,Nfixnod,FixNod,faux,Nfixline,fixline,Nfixsurf,fixsurf,Analysisty

Kaux = KGf;

if Analysistype==1||Analysistype==2
dim=2;
end

if Analysistype==3
dim=3;
end

%-------------------------------------------------------------------------------------------------

if Nfixnod>0

for i=1:Nfixnod

M=FixNod(i,1);

if dim == 2

if FixNod(i,2)==1
Kaux(dim*M-1,:)=0;
Kaux(dim*M-1,dim*M-1)=10e99;
end
```

```
if FixNod(i,3)==1
Kaux(dim*M,:)=0;
Kaux(dim*M,dim*M)=10e99;
end

else

if FixNod(i,2)==1
Kaux(dim*M-2,:)=0;
Kaux(dim*M-2,dim*M-2)=10e99;
end

if FixNod(i,3)==1
Kaux(dim*M-1,:)=0;
Kaux(dim*M-1,dim*M-1)=10e99;
end

if FixNod(i,4)==1
Kaux(dim*M,:)=0;
Kaux(dim*M,dim*M)=10e99;
end

end

end

end

%-----------------------------------------------------------------------------------------------------

if Nfixline>0

for i=1:Nfixline

M=fixline(i,1);


if dim == 2

if fixline(i,2)==1
Kaux(dim*M-1,:)=0;
Kaux(dim*M-1,dim*M-1)=10e99;
end

if fixline(i,3)==1
Kaux(dim*M,:)=0;
Kaux(dim*M,dim*M)=10e99;
end

else

if fixline(i,2)==1
Kaux(dim*M-2,:)=0;
Kaux(dim*M-2,dim*M-2)=10e99;
end

if fixline(i,3)==1
Kaux(dim*M-1,:)=0;
Kaux(dim*M-1,dim*M-1)=10e99;
```

```
end

if fixline(i,4)==1
Kaux(dim*M,:)=0;
Kaux(dim*M,dim*M)=10e99;
end

end


end

end
%-----------------------------------------------------------------------------------------------------------

if Nfixsurf>0

for i=1:Nfixsurf

M=fixsurf(i,1);


if dim == 2

if fixsurf(i,2)==1
Kaux(dim*M-1,:)=0;
Kaux(dim*M-1,dim*M-1)=10e99;
end

if fixsurf(i,3)==1
Kaux(dim*M,:)=0;
Kaux(dim*M,dim*M)=10e99;
end

else

if fixsurf(i,2)==1
Kaux(dim*M-2,:)=0;
Kaux(dim*M-2,dim*M-2)=10e99;
end

if fixsurf(i,3)==1
Kaux(dim*M-1,:)=0;
Kaux(dim*M-1,dim*M-1)=10e99;
end

if fixsurf(i,4)==1
Kaux(dim*M,:)=0;
Kaux(dim*M,dim*M)=10e99;
end

end


end

end
```

```
%-------------------------------------------------------------------------------------------------

%solves for displacements.
a = Kaux\faux;
%[L,U] = lu(sparse(Kaux));
%a = U\(L\faux);
%a = pcg(Kaux,faux);

%solves for nodal forces.
f = KGf*a;

end
```

StrainStress.

```
function [Strain,Stress] = StrainStress(Elem,Nod,a,Elemcon,C,Elemtype,Be,nodcord)

%-------------------------------------------------------------------------------------------------

if Elemtype==1

Strain = zeros(3,1,Elem);
Stress = zeros(3,1,Elem);

for e=1:Elem
row = find(Elemcon(:,5) == e);

u = reshape(a,[2,Nod]);

ue = [u(1,Elemcon(row,1)); u(2,Elemcon(row,1)); u(1,Elemcon(row,2)); u(2,Elemcon(row,2)); u(1,Ele

Strain(:,:,row) = Be(:,:,row)*ue;
Stress(:,:,row) = C(:,:,Elemcon(row,4))*Strain(:,:,row);
end
end

%-------------------------------------------------------------------------------------------------

if Elemtype==2

Strain = zeros(3,1,Elem,2,2);
Stress = zeros(3,1,Elem,2,2);

for e=1:Elem

row = find(Elemcon(:,6) == e);%locates e in the Elemcon matrix.

u=reshape(a,[2,Nod]);

ue=[u(1,Elemcon(row,1)) u(2,Elemcon(row,1)) u(1,Elemcon(row,2)) u(2,Elemcon(row,2)) u(1,Elemcon(r

for p=1:2 %loop over integration points.
for q=1:2 %"

r = ((-1)^q)*0.5773502692; %computes x natural cordinate of the point pq.

s = ((-1)^p)*0.5773502692; %computes y natural cordinate of the point pq.
```

```
t = 0;


J = Jacobianmat(r,s,t,nodcord,Elemcon,Elemtype,e);

B = Deformationmat(r,s,t,J,Elemtype);


Strain(:,:,row,p,q)=B*transpose(ue);

Stress(:,:,row,p,q)=C(:,:,Elemcon(row,5))*Strain(:,:,row,p,q);

end
end
end
end

%-------------------------------------------------------------------------------------------------

if Elemtype==3

Strain = zeros(6,1,Elem,2,2,2);
Stress = zeros(6,1,Elem,2,2,2);

for e=1:Elem
row = find(Elemcon(:,10) == e); %locates e in the Elemcon matrix.

u=reshape(a,[3,Nod]);

ue=[u(1,Elemcon(row,1)) u(2,Elemcon(row,1)) u(3,Elemcon(row,1)) u(1,Elemcon(row,2)) u(2,Elemcon(r

for p=1:2 %loop over integration points.
for q=1:2 %"
for d=1:2%"

r = ((-1)^p)*0.577350269189626; %computes x natural cordinate of the point pqd.

s = ((-1)^q)*0.577350269189626; %computes y natural cordinate of the point pqd.

t = ((-1)^d)*0.577350269189626; %computes z natural cordinate of the point pqd.


J = Jacobianmat(r,s,t,nodcord,Elemcon,Elemtype,e);

B = Deformationmat(r,s,t,J,Elemtype);


Strain(:,:,row,p,q,d)=B*transpose(ue);

Stress(:,:,row,p,q,d)=C(:,:,Elemcon(row,9))*Strain(:,:,row,p,q,d);

end
end
end
end
end

end
```

write.

```
function fileID = write(a,f,Stress,Strain,Elem,Nod,nodcord,Elemcon,meshfile,resfile,Elemtype)

%result display.

%-------------------------------------------------------------------------------------------------

if Elemtype==1

N = transpose(1:Nod); %creates a column vector from 1 to #nodes.

fileID=fopen(resfile,'w'); %creates file prova.post.res

%header of the file.
fprintf(fileID,'GiD Post Results File 1.0');
fprintf(fileID,'\n');

%defines gauss points
fprintf(fileID,'GaussPoints "gauss" ElemType Triangle');
fprintf(fileID,'\n');

fprintf(fileID,'Number of Gauss Points: 1');
fprintf(fileID,'\n');

fprintf(fileID,'Natural Coordinates: internal');
fprintf(fileID,'\n');

fprintf(fileID,'end gausspoints');
fprintf(fileID,'\n');


%writes displacements on the file.
fprintf(fileID,'Result "Displacements" "FEM" 1 Vector OnNodes');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Displacement_X", "Displacement_Y"');
fprintf(fileID,'\n');

fprintf(fileID,'Unit "mm"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
fprintf(fileID,'\n');

U = 1000*transpose(reshape(a,[2,Nod])); %reshape displacements vector: (x,y) where each row is a

fprintf(fileID,'%i %f %f\n',transpose([N,U(:,1),U(:,2)]));
fprintf(fileID,'\n');

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%writes forces on the file.
fprintf(fileID,'Result "Forces_on_Nodes" "FEM" 1 Vector OnNodes');
fprintf(fileID,'\n');
```

```
fprintf(fileID,'ComponentNames "Force_X", "Force_Y"');
fprintf(fileID,'\n');

fprintf(fileID,'Unit "N"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
fprintf(fileID,'\n');

U = transpose(reshape(f,[2,Nod])); %reshape force vector: (x,y) where each row is a node.

fprintf(fileID,'%i %f %f\n',transpose([N,U(:,1),U(:,2)]));
fprintf(fileID,'\n');

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%writes stresses on file.
fprintf(fileID,'Result "stresses" "FEM" 1 Vector OnGaussPoints "gauss"');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Stress_X", "Stress_Y", "Stress_XY"');
fprintf(fileID,'\n');

fprintf(fileID,'Unit "Pa"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
fprintf(fileID,'\n');

for e=1:Elem

fprintf(fileID,'%i %f %f %f\n',transpose([e;Stress(1,:,e);Stress(2,:,e);Stress(3,:,e)]));
fprintf(fileID,'\n');

end

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%writes strains on file.
fprintf(fileID,'Result "Strains" "FEM" 1 Vector OnGaussPoints "gauss"');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Strain_X", "Strain_Y", "Strain_XY"');
fprintf(fileID,'\n');

fprintf(fileID,'Unit "Dimensionless*10^-9"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
fprintf(fileID,'\n');

for e=1:Elem

fprintf(fileID,'%i %f %f %f\n',transpose([e;Strain(1,:,e)*10^9;Strain(2,:,e)*10^9;Strain(3,:,e)*1
fprintf(fileID,'\n');
```

```
end

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%mesh display.

fileID=fopen(meshfile,'w');%creates a file named prova.post.mesh for mesh information storage.

%Mesh file header.
fprintf(fileID,'MESH "mesh" dimension 2 ElemType Triangle Nnode 3');
fprintf(fileID,'\n');

%may be useful to define mesh units.

%node cordinates
fprintf(fileID,'Coordinates');
fprintf(fileID,'\n');

fprintf(fileID,'%i %f %f\n',transpose([nodcord(:,1),nodcord(:,2),nodcord(:,3)]));
fprintf(fileID,'\n');

fprintf(fileID,'end coordinates');
fprintf(fileID,'\n');

%element cordinates
fprintf(fileID,'Elements');
fprintf(fileID,'\n');

fprintf(fileID,'%i %i %i %i\n',transpose([Elemcon(:,5),Elemcon(:,1),Elemcon(:,2),Elemcon(:,3)]));
fprintf(fileID,'\n');

fprintf(fileID,'end elements');
fprintf(fileID,'\n');
end

%-------------------------------------------------------------------------------------------------

if Elemtype==2

N = transpose(1:Nod); %creates a column vector from 1 to #nodes.

fileID=fopen(resfile,'w'); %creates file prova.post.res

fprintf(fileID,'GiD Post Results File 1.0');
fprintf(fileID,'\n');

%defines gauss points
fprintf(fileID,'GaussPoints "gauss" ElemType Quadrilateral');
fprintf(fileID,'\n');

fprintf(fileID,'Number of Gauss Points: 4');
fprintf(fileID,'\n');

fprintf(fileID,'Natural Coordinates: internal');
fprintf(fileID,'\n');
```

```
fprintf(fileID,'end gausspoints');
fprintf(fileID,'\n');


%writes displacements on the file.
fprintf(fileID,'Result "Displacements" "FEM" 1 Vector OnNodes');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Displacement_X", "Displacement_Y"');
fprintf(fileID,'\n');

fprintf(fileID,'Unit "mm"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
fprintf(fileID,'\n');

U = 1000*transpose(reshape(a,[2,Nod])); %reshape displacements vector: (x,y) where each row is a

fprintf(fileID,'%i %f %f\n',transpose([N,U(:,1),U(:,2)]));
fprintf(fileID,'\n');

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%writes forces on the file.
fprintf(fileID,'Result "Forces_on_Nodes" "FEM" 1 Vector OnNodes');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Force_X", "Force_Y"');
fprintf(fileID,'\n');

fprintf(fileID,'Unit "N"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
fprintf(fileID,'\n');

U = transpose(reshape(f,[2,Nod])); %reshape force vector: (x,y) where each row is a node.

fprintf(fileID,'%i %f %f\n',transpose([N,U(:,1),U(:,2)]));
fprintf(fileID,'\n');

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%writes stresses on file.
fprintf(fileID,'Result "stresses" "FEM" 1 Vector OnGaussPoints "gauss"');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Stress_X", "Stress_Y", "Stress_XY"');
fprintf(fileID,'\n');

fprintf(fileID,'Unit "Pa"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
```

```
fprintf(fileID,'\n');

for e=1:Elem

fprintf(fileID,'%i %f %f %f\n',transpose([e;Stress(1,:,e,2,2);Stress(2,:,e,2,2);Stress(3,:,e,2,2)

fprintf(fileID,'%f %f %f\n',transpose([Stress(1,:,e,2,1);Stress(2,:,e,2,1);Stress(3,:,e,2,1)]));

fprintf(fileID,'%f %f %f\n',transpose([Stress(1,:,e,1,1);Stress(2,:,e,1,1);Stress(3,:,e,1,1)]));

fprintf(fileID,'%f %f %f\n',transpose([Stress(1,:,e,1,2);Stress(2,:,e,1,2);Stress(3,:,e,1,2)]));

fprintf(fileID,'\n');

end

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%writes strains on file.
fprintf(fileID,'Result "Strains" "FEM" 1 Vector OnGaussPoints "gauss"');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Strain_X", "Strain_Y", "Strain_XY"');
fprintf(fileID,'\n');

fprintf(fileID,'Unit "Dimensionless*10^-9"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
fprintf(fileID,'\n');

for e=1:Elem

fprintf(fileID,'%i %f %f %f\n',transpose([e;Strain(1,:,e,2,2)*10^9;Strain(2,:,e,2,2)*10^9;Strain(

fprintf(fileID,'%f %f %f\n',transpose([Strain(1,:,e,2,1)*10^9;Strain(2,:,e,2,1)*10^9;Strain(3,:,e

fprintf(fileID,'%f %f %f\n',transpose([Strain(1,:,e,1,1)*10^9;Strain(2,:,e,1,1)*10^9;Strain(3,:,e

fprintf(fileID,'%f %f %f\n',transpose([Strain(1,:,e,1,2)*10^9;Strain(2,:,e,1,2)*10^9;Strain(3,:,e

fprintf(fileID,'\n');

end

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%mesh display.

fileID=fopen(meshfile,'w');%creates a file named prova.post.mesh for mesh information storage.

%Mesh file header.
fprintf(fileID,'MESH "mesh" dimension 2 ElemType Quadrilateral Nnode 4');
fprintf(fileID,'\n');
```

```
%may be useful to define mesh units.

%node cordinates
fprintf(fileID,'Coordinates');
fprintf(fileID,'\n');

fprintf(fileID,'%i %f %f\n',transpose([nodcord(:,1),nodcord(:,2),nodcord(:,3)]));
fprintf(fileID,'\n');

fprintf(fileID,'end coordinates');
fprintf(fileID,'\n');

%element cordinates
fprintf(fileID,'Elements');
fprintf(fileID,'\n');

fprintf(fileID,'%i %i %i %i %i %i\n',transpose([Elemcon(:,6),Elemcon(:,1),Elemcon(:,2),Elemcon(:,
fprintf(fileID,'\n');

fprintf(fileID,'end elements');
fprintf(fileID,'\n');
end

%-------------------------------------------------------------------------------------------------

if Elemtype==3

N = transpose(1:Nod); %creates a column vector from 1 to #nodes.

fileID=fopen(resfile,'w'); %creates file prova.post.res

fprintf(fileID,'GiD Post Results File 1.0');
fprintf(fileID,'\n');

%defines gauss points
fprintf(fileID,'GaussPoints "gauss" ElemType Hexahedra');
fprintf(fileID,'\n');

fprintf(fileID,'Number of Gauss Points: 8');
fprintf(fileID,'\n');

fprintf(fileID,'Natural Coordinates: internal');
fprintf(fileID,'\n');

fprintf(fileID,'end gausspoints');
fprintf(fileID,'\n');


%writes displacements on the file.
fprintf(fileID,'Result "Displacements" "FEM" 1 Vector OnNodes');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Displacement_X", "Displacement_Y", "Displacement_Z"');
fprintf(fileID,'\n');

fprintf(fileID,'Unit "mm"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
```

```
fprintf(fileID,'\n');

U = 1000*transpose(reshape(a,[3,Nod])); %reshape displacements vector: (x,y) where each row is a

fprintf(fileID,'%i %f %f %f\n',transpose([N,U(:,1),U(:,2),U(:,3)]));
fprintf(fileID,'\n');

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%writes forces on the file.
fprintf(fileID,'Result "Forces_on_Nodes" "FEM" 1 Vector OnNodes');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Force_X", "Force_Y", "Force_Z"');
fprintf(fileID,'\n');

fprintf(fileID,'Unit "N"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
fprintf(fileID,'\n');

U = transpose(reshape(f,[3,Nod])); %reshape force vector: (x,y) where each row is a node.

fprintf(fileID,'%i %f %f %f\n',transpose([N,U(:,1),U(:,2),U(:,3)]));
fprintf(fileID,'\n');

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%writes stresses on file.
fprintf(fileID,'Result "stresses" "FEM" 1 Vector OnGaussPoints "gauss"');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Stress_X", "Stress_Y", "Stress_Z", "Stress_XY", "Stress_XZ", "Str
fprintf(fileID,'\n');

fprintf(fileID,'Unit "Pa"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
fprintf(fileID,'\n');

for e=1:Elem

fprintf(fileID,'%i %f %f %f %f %f %f\n',transpose([e;Stress(1,:,e,1,1,1);Stress(2,:,e,1,1,1);Stre

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Stress(1,:,e,2,1,1);Stress(2,:,e,2,1,1);Stress(3,

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Stress(1,:,e,2,2,1);Stress(2,:,e,2,2,1);Stress(3,

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Stress(1,:,e,1,2,1);Stress(2,:,e,1,2,1);Stress(3,

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Stress(1,:,e,1,1,2);Stress(2,:,e,1,1,2);Stress(3,

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Stress(1,:,e,2,1,2);Stress(2,:,e,2,1,2);Stress(3,
```

```
fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Stress(1,:,e,2,2,2);Stress(2,:,e,2,2,2);Stress(3,

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Stress(1,:,e,1,2,2);Stress(2,:,e,1,2,2);Stress(3,

fprintf(fileID,'\n');

end

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%writes strains on file.
fprintf(fileID,'Result "Strains" "FEM" 1 Vector OnGaussPoints "gauss"');
fprintf(fileID,'\n');

fprintf(fileID,'ComponentNames "Strain_X", "Strain_Y", "Strain_Z", "Strain_XY", "Strain_XZ", "Str
fprintf(fileID,'\n');

fprintf(fileID,'Unit "Dimensionless*10^-9"');
fprintf(fileID,'\n');

fprintf(fileID,'Values');
fprintf(fileID,'\n');

for e=1:Elem

fprintf(fileID,'%i %f %f %f %f %f %f\n',transpose([e;Strain(1,:,e,1,1,1)*10^9;Strain(2,:,e,1,1,1)

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Strain(1,:,e,2,1,1)*10^9;Strain(2,:,e,2,1,1)*10^9

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Strain(1,:,e,2,2,1)*10^9;Strain(2,:,e,2,2,1)*10^9

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Strain(1,:,e,1,2,1)*10^9;Strain(2,:,e,1,2,1)*10^9

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Strain(1,:,e,1,1,2)*10^9;Strain(2,:,e,1,1,2)*10^9

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Strain(1,:,e,2,1,2)*10^9;Strain(2,:,e,2,1,2)*10^9

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Strain(1,:,e,2,2,2)*10^9;Strain(2,:,e,2,2,2)*10^9

fprintf(fileID,'%f %f %f %f %f %f\n',transpose([Strain(1,:,e,1,2,2)*10^9;Strain(2,:,e,1,2,2)*10^9

fprintf(fileID,'\n');

end

fprintf(fileID,'End Values');
fprintf(fileID,'\n');


%mesh display.

fileID=fopen(meshfile,'w');%creates a file named prova.post.mesh for mesh information storage.

%Mesh file header.
fprintf(fileID,'MESH "mesh" dimension 3 ElemType Hexahedra Nnode 8');
fprintf(fileID,'\n');
```

```
%may be useful to define mesh units.

%node cordinates
fprintf(fileID,'Coordinates');
fprintf(fileID,'\n');

fprintf(fileID,'%i %f %f %f\n',transpose([nodcord(:,1),nodcord(:,2),nodcord(:,3),nodcord(:,4)]));
fprintf(fileID,'\n');

fprintf(fileID,'end coordinates');
fprintf(fileID,'\n');

%element cordinates
fprintf(fileID,'Elements');
fprintf(fileID,'\n');

fprintf(fileID,'%i %i %i %i %i %i %i %i %i\n',transpose([Elemcon(:,10),Elemcon(:,1),Elemcon(:,2),
fprintf(fileID,'\n');

fprintf(fileID,'end elements');
fprintf(fileID,'\n');
end

end
```

Anejo 3: Archivos para la creación del tipo de problema.

Archivo.bas

```
*nelem
*npoin
*nmats
*Set Cond Fixed_nodes
*CondNumEntities
*Set Cond Fixed_lines
*CondNumEntities
*Set Cond Fixed_surfaces
*CondNumEntities
*Set Cond Loads_over_nodes
*CondNumEntities
*Set Cond Loads_over_lines
*CondNumEntities
*Set Cond Loads_over_surfaces
*CondNumEntities
*Set Cond Loads_over_volumes
*CondNumEntities
*Set Cond initial_def
*CondNumEntities
*GenData(1)
*GenData(2)
Nodcord
*loop nodes
*Nodesnum,*NodesCoord(1),*NodesCoord(2),*NodesCoord(3)
*end nodes
Elemtype
*loop elems
*if(ElemsType==2)
*elemsConec(1),*elemsConec(2),*elemsConec(3),*elemsmat,*ElemsNum
*end
*if(ElemsType==3)
*elemsConec(1),*elemsConec(2),*elemsConec(3),*elemsConec(4),*elemsmat,*ElemsNum
*end
*if(ElemsType==5)
*elemsConec(1),*elemsConec(2),*elemsConec(3),*elemsConec(4),*elemsConec(5),*elemsConec(6),*elemsConec(7),*elemsConec(8),*elemsmat,*ElemsNum
*end
*end elems
Mat
*loop materials
*matnum,*MatProp(1),*MatProp(2),*MatProp(3),*MatProp(4)
*end materials
*Set Cond Fixed_nodes
*CondName
*loop nodes *OnlyInCond
*NodesNum,*cond(1),*cond(2),*cond(3)
*end nodes
*Set Cond Fixed_lines
*CondName
*loop nodes *OnlyInCond
*NodesNum,*cond(1),*cond(2),*cond(3)
*end nodes
*Set Cond Fixed_surfaces
*CondName
*loop nodes *OnlyInCond
*NodesNum,*cond(1),*cond(2),*cond(3)
*end nodes
*Set Cond Loads_over_nodes
*CondName
*loop nodes *OnlyInCond
*NodesNum,*cond(1),*cond(2),*cond(3)
*end nodes
*Set Cond Loads_over_lines
*CondName
*loop elems *OnlyInCond
*ElemsNum,*GlobalNodes(1),*GlobalNodes(2),*cond(1),*cond(2),*cond(3)
*end elems
*Set Cond Loads_over_surfaces
*CondName
*loop elems *OnlyInCond
*if(ElemsType==2)
*ElemsNum,*ElemsConec(1),*ElemsConec(2),*ElemsConec(3),*cond(1),*cond(2),*cond(3)
*end
*if(ElemsType==3)
*ElemsNum,*ElemsConec(1),*ElemsConec(2),*ElemsConec(3),*ElemsConec(4),*cond(1),*cond(2),*cond(3)
*end
*if(ElemsType==5)
*ElemsNum,*GlobalNodes(1),*GlobalNodes(2),*GlobalNodes(3),*ElemsConec(4),*cond(1),*cond(2),*cond(3)
*end
*end elems
*Set Cond Loads_over_volumes
*CondName
*loop elems *OnlyInCond
*ElemsNum,*cond(1),*cond(2),*cond(3)
*end elems
*Set Cond initial_def
*CondName
*loop elems *OnlyInCond
*ElemsNum,*cond(1),*cond(2),*cond(3)
*end elems
```

Figura 2: Genera el archivo de cálculo.

Archivo.cnd

```
:ONDITION: Fixed_nodes
:ONDTYPE: over points
:ONDMESHTYPE: over nodes
)UESTION: Fixed_X_Displ : #CB# (0,1)
/ALUE: 0
)UESTION: Fixed_Y_Displ : #CB# (0,1)
/ALUE: 0
)UESTION: Fixed_Z_Displ : #CB# (0,1)
/ALUE: 0
:ND CONDITION

:ONDITION: Fixed_lines
:ONDTYPE: over lines
:ONDMESHTYPE: over nodes
)UESTION: Fixed_X_Displ : #CB# (0,1)
/ALUE: 0
)UESTION: Fixed_Y_Displ : #CB# (0,1)
/ALUE: 0
)UESTION: Fixed_Z_Displ : #CB# (0,1)
/ALUE: 0
:ND CONDITION

:ONDITION: Fixed_surfaces
:ONDTYPE: over surfaces
:ONDMESHTYPE: over nodes
)UESTION: Fixed_X_Displ : #CB# (0,1)
/ALUE: 0
)UESTION: Fixed_Y_Displ : #CB# (0,1)
/ALUE: 0
)UESTION: Fixed_Z_Displ : #CB# (0,1)
/ALUE: 0
:ND CONDITION

:ONDITION: Loads_over_nodes
:ONDTYPE: over points
:ONDMESHTYPE: over nodes
)UESTION: Force_X (N):
/ALUE: 0.0
)UESTION: Force_Y (N) :
/ALUE: 0.0
)UESTION: Force_Z (N) :
/ALUE: 0.0
:ND CONDITION
```

Figura 3: Define las condiciones de borde.

```
CONDITION: Loads_over_lines
CONDTYPE: over lines
CONDMESHTYPE: over face elements
QUESTION: Force_X (N/m):
VALUE: 0.0
QUESTION: Force_Y (N/m) :
VALUE: 0.0
QUESTION: Force_Z (N/m) :
VALUE: 0.0
END CONDITION

CONDITION: Loads_over_surfaces
CONDTYPE: over surfaces
CONDMESHTYPE: over face elements
QUESTION: Force_X (N/m2):
VALUE: 0.0
QUESTION: Force_Y (N/m2) :
VALUE: 0.0
QUESTION: Force_Z (N/m2) :
VALUE: 0.0
END CONDITION

CONDITION: Loads_over_volumes
CONDTYPE: over volumes
CONDMESHTYPE: over body elements
QUESTION: Force_X (N/m3):
VALUE: 0.0
QUESTION: Force_Y (N/m3) :
VALUE: 0.0
QUESTION: Force_Z (N/m3) :
VALUE: 0.0
END CONDITION

CONDITION: initial_def
CONDTYPE: over surfaces
CONDMESHTYPE: over body elements
QUESTION: Strain_X (mm):
VALUE: 0.0
QUESTION: Strain_Y (mm) :
VALUE: 0.0
QUESTION: Strain_XY (mm) :
VALUE: 0.0
END CONDITION
```

Figura 4: Define las condiciones de borde.

Archivo.dat

```
MATERIAL: material_name
QUESTION: Poisson_Modulus
VALUE: 0.3
QUESTION: Shear_Modulus (Pa)
VALUE: 80E9
QUESTION: Young_Modulus (Pa)
VALUE: 210E9
QUESTION: thickness (m)
VALUE: 10.0
END MATERIAL
```

Figura 5: Define los materiales.

Archivo.prb

```
PROBLEM DATA
QUESTION:Analysis_type:_Plane_stress=1_Plane_strain=2_3D=3
VALUE: 0
QUESTION:Element_type:_Triangle=1_Quadrilateral=2_Hexahedra=3
VALUE: 0
QUESTION:Unit_system#CB#(SI,CGS,user)
VALUE: SI
END PROBLEM DATA
```

Figura 6: Introduce datos adicionales.