UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Departamento de Ingeniería Mecánica y de Materiales (DIMM)

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA MECÁNICA

MEJORA DE LA EFICIENCIA DE LA APLICACIÓN FEAVOX PARA EL CÁLCULO DE ESFUERZOS MECÁNICOS ENTRE CUERPOS EN CONTACTO

Por Adrián Arenas Martínez

Tutores:
Enrique Nadal Soriano
José Manuel Navarro Jiménez

Septiembre de 2019

Índice

Índice de figuras

Agradecimientos

A mi familia, por su incansable ayuda y sacrificio cada día hasta hoy para que tuviese todas las oportunidades que fuesen posibles. Jamás habría llegado a este punto sin su ánimo e insistencia, y ayudándome a salir siempre de la procrastinación; especialmente con este trabajo en su empeño por que no me cerrase ninguna puerta.

A Enrique y José Manuel, mis tutores, por su eterna paciencia con este trabajo así como su enorme ayuda para sacarlo adelante. Agradecer la oportunidad de hacer un proyecto de este tipo con ellos, sus conocimientos y cada una de las tutorías y correos.

A mis amigos, aquellos que me han soportado tanto tiempo y han transformado de forma increíble cada momento. En especial, a aquellos con los que comparto esa incansable compañía y he podido compartir tanto, incluido este y más proyectos.

Muchísimas gracias a todos por tanto.

Resumen

En este proyecto se va a desarrollar una solución capaz de mejorar la eficiencia de la aplicación FEAVOX, permitiendo mejorar el cálculo de esfuerzos para sólidos en contacto.

Como punto de partida se expondrá la motivación que anima al desarrollo de este trabajo, puesto que en el mundo de la ingeniería conseguir cálculos de la mejor forma es una obligación. Para conseguir estos datos se recurre a simulaciones realizadas gracias al Método de los Elementos Finitos (MEF), y es por esto que se hablará del Estado del Arte del método y por qué usar una forma diferente de aplicarlo en FEAVOX: el "Cartesian grid Finite Element Method", cgFEM.

Tras una breve descripción de la historia del MEF, se pasa a describir paso por paso los procesos a los que se somete un sólido para poder analizarlo acompañado de ejemplos visuales de esto. Una vez presentado el MEF, se expone uno de sus problemas más costosos: el problema de contacto. No exclusivamente, pero sí en gran medida, esta será la razón por la que se hará hincapié en la necesidad de usar cgFEM y mejorar el cálculo del problema para esta variante.

Por lo tanto, a continuación se describe en qué consiste cgFEM, seguido de un ejemplo a estudiar y alrededor del cual girará la comprobación del objetivo de este proyecto: la mejora en la eficiencia de la búsqueda de puntos en contacto entre sólidos.

El siguiente bloque del trabajo corresponderá a la implementación, donde se detalla cada uno de los aspectos y "Retos" para la mejora. En el apartado de retos se expondrán los condicionantes del resultado final. No obstante, se propone una solución general a continuación, analizando los puntos a favor para mejorar la velocidad o la forma en que se obtienen los cálculos necesarios.

Este apartado con la solución se verá seguido de una descripción de una serie de problemas y cómo se han solucionado para satisfacer el apartado de "Retos". Aquí se describe cómo es necesario mejorar la velocidad en el programa de MATLAB y algunas de las opciones, así como soluciones para operaciones repetidas muchas veces a través del uso de archivos MEX lanzados desde MATLAB.

A continuación, se añade una descripción del programa, línea a línea y una descripción de cómo funcionan. Cada una de las partes de la explicación va acompañada de una descripción del resto de funciones que intervienen.

Finalmente, en el último bloque se muestra un ejemplo de aplicación del trabajo desarrollado. Este es una simulación de contacto usando cgFEM y donde se ha aplicado la mejora, viéndose su función y dónde actúa para conseguir un análisis correcto.

Resum

En aquest projecte es va a desenvolupar una solució capaç de millorar la eficiència de la aplicació FEAVOX, permitent millorar el càlcul d'esforços per a sòlids en contacte.

Com a punt de partida es va a expondre la motivació que anima al desenvolupament d'aquest treball, ja que al món de la ingenieria aconseguir càlculs de la millor forma es una obligació. Per a aconseguir aquestes dates es recorre a simulacions realitzades gràcies al Mètode dels Elements Finits (MEF), i es per això que es parlarà del Estat de l'Art del mètode i per què fer us d'una forma diferent d'aplicar-ho a FEAVOX: el "Cartesian grid Finite Element Method", cgFEM.

Tras una breu descripció de la història del MEF, es passa a descriure pas a pas els processos als que es sometix un sòlid per a poder analitzar-lo acompanyat d'exemples visuals d'açò. Una vegada presentat el MEF, s'expossa un dels problemes més costosos: el problema de contacte. No exclusivament, però si en gran mesura, esta serà la raó per la qual es possarà l'accent en la necessitat d'usar cgFEM i millorar el càlcul del problema amb aquesta variant.

Per tant, a continuació es descriu en què consistix cgFEM, seguit d'un exemple a estudiar i al voltant del qual girarà la comprobació de l' objectiu d'aquest projecte: la millora en la eficiència de la búsqueda de punts en contacte entre sòlids.

El següent bloc de treball correspondrà a la implementació, a on es detalla cadascún dels aspectes i reptes per a la millora. A l'apartat de "Reptes" s'expondràn els condicionants del resultat final. No obstant això, es proposa una solució general a continuació, analitzant els punts a favor per a millorar la velocitat o la forma en que s'obtenen els càlculs necessaris.

Aquest apartat amb la solució es verà seguit d'una descripció d'una serie de problemes i com s'han solucionat per a satisfer l'apartat de "Reptes". Ací es descriu com es necessari millorar la velocitat en el programa de MATLAB i algunes de les opcions, així com solucions per a operacions repetides moltes vegades a través de l'ús d'arxius MEX llançats des de MATLAB.

A continuació, s'afegix una descripció del programa, línia a línia i una descripció de com funcionen. Cadascuna de les parts de l'explicació va acompanyada d'una descripció de la resta de funcions que intervenen.

Finalment, en l'últim bloc es mostrarà un exemple d'aplicació del treball desenvolupat. Aquest es una simulació de contacte usant cgFEM i on s'ha aplicat la millora, veient-se la seua funció i on actúa per a conseguir un anàlisi correcte.

Summary

A solution capable of improving the efficiency of the application FEAVOX is being developed in this project, allowing to improve the effort calculation for solids in contact.

As a start point the motivation that encourages this project development will be explained, as in the engineering world it is mandatory to achieve optimal calculations. In order to get this data simulations, they are carried on with the Finite Element Method (FEM), and it's because of this that the State of the Art will be explained and why is it necessary to use a diferent way to apply it to FEAVOX: cgFEM.

After a brief description about the FEM's history, the different processes followed in the method are described step by step together with visual examples of it. Once the FEM is presented, one of the most 'expensive' problems will be exposed: the contact problem. Not exclusively but mostly, this will be the reason of highlighting the necessity to use cgFEM and to improve the problem calculations for this variant.

Next, the cgFEM methodology will be presented, followed by an example to study, which will be used as well to test the objective of this project, this is, the efficiency improvement in the contact point search between solids.

The next section of this work will correspond to the implementation, where every feature and challenge will be described. In the challenges section the conditions of the final result will be exposed. Nevertheless, a general solution is proposed then, analysing the advantages in improving the speed or the way the needed calculations are obtained.

This section with the solution will be followed by a description of a series of issues and how they have been solved to accomplish the solution. Here it is described how it is necessary to improve the speed in MATLAB and some of the options, as well as solutions for the operations repeated a lot of times through the use of MEX archives launched in MATLAB.

Next, a description of the program will be presented with a description of how it works. Every one of those parts of the explanation will be followed by the description of the underlying subfunctions that take part.

Finally, the last section will be the example described at the beginning but as an application example. This is a contact simulation using cgFEM where the improvement has been used, showing its performance and where it acts to achieve a correct analysis.

1. Introducción

Para los ingenieros, el análisis es una necesidad imprescindible en muchos ámbitos de su profesión: estructural, térmico, de fluidos, o de vibraciones, por citar varios. A lo largo de cualquier proceso, comprobar hipótesis y realizar procesos de cálculo son actividades más que frecuentes. Por ello y desde su postulación, el Análisis por Elementos Finitos ha contribuido en gran medida a llevar la ingeniería a un nuevo nivel. Los métodos de Análisis por Elementos Finitos han demostrado desde su implementación su gran utilidad, así como una mejora sustancial en los flujos de trabajo y herramientas disponibles tanto dentro de la ingeniería como en muchos otros campos. Su gran utilidad radica en la capacidad de simular con gran precisión el comportamiento de gran cantidad de componentes, usados en una gran variedad de ámbitos.

En los últimos años se han producido avances tanto en los programas de análisis como en los sistemas de computación, y esto ha dado pie a nuevas soluciones a la forma en la que llevar a cabo y enfocar este cálculo. Una de estas nuevas alternatias es el "cgFEM: Cartesian Grid Finite Element Method" [4] que es el método escogido para los desarrollos presentados en este trabajo. El método cgFEM no solo optimiza ciertos aspectos del ya establecido MEF, sino que permite una representación de mayor precisión de las geometrías a estudiar. Las innovaciones presentes serán descritas junto a sus homólogos más convencionales a lo largo de los distintos apartados de este trabajo.

El campo de aplicación de esta tecnología para este trabajo será el de la ingeniería mecánica, un ámbito que siempre se encuentra evolucionando y adapatándose a los nuevos tiempos. Y como no podría ser de otra forma, se analizará un sistema en contacto, poniendo de manifiesto la necesidad de tener en cuenta la geometría analítica en este tipo de problemas.

Poder conseguir resultados fiables dependerá en gran medida de la calidad del cálculo. Un resultado será fiable cuando la experiencia real se asemeje en gran medida a su simulación. Analizar el contacto entre sólidos es un campo de aplicación que depende completamente de esta fiabilidad.

1.1. Motivación

Los problemas de contacto son una realidad cotidiana en la industria y algo habitual en la ingeniería mecánica. Son los responsables del desgaste, de la transmisión de una acción en concreto o simplemente de la interacción entre dos objetos en un momento puntual.

El fenómeno de contacto aparece múltiples veces en cualquier máquina estudiada en el ámbito de la ingeniería mecánica. Por ejemplo, en un coche existen interacciones de contacto: dentro del motor, en la dirección, en la transmisión,...

A continuación se describen algunos de estos contactos:

- En el motor:

  En partes como el árbol de levas, estas entran en contacto con las válvulas de cada cilindro como se puede ver en la Figura 1. Resultado de esta relación de contacto, se deslizan y permiten la admisión y el escape durante el funcionamiento.

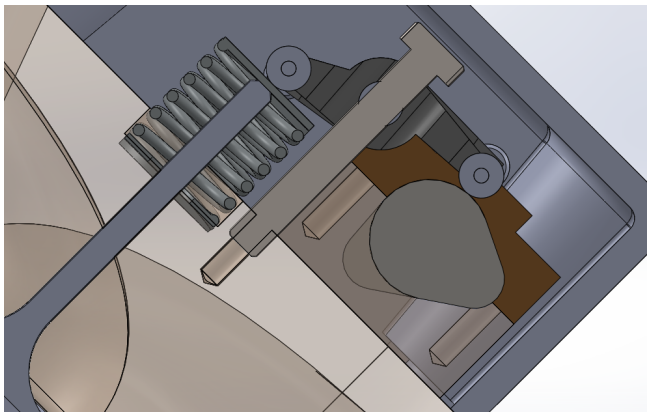


Figura 1: Mecanismo similar al encontrado en un árbol de levas.

En el cigüeñal, cuya sección se muestra en la Figura 2, el eje motor fricciona y mantiene contacto constante con los cojinetes que lo sostienen.

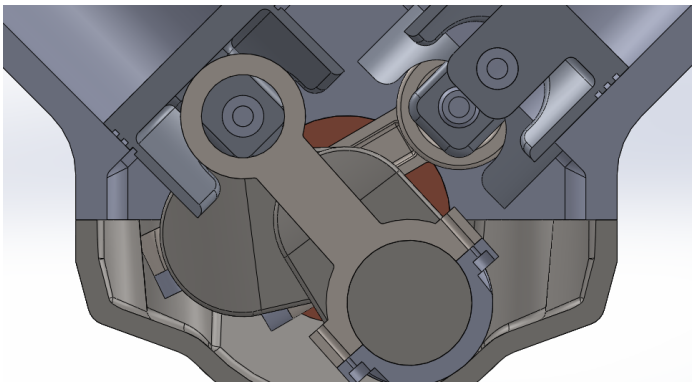


Figura 2: Articulaciones en el juego biela-manivela-cigüeñal.

Para la correa de distribución como la mostrada en la Figura 3 existe contacto con las distintas poleas por las que pasa.

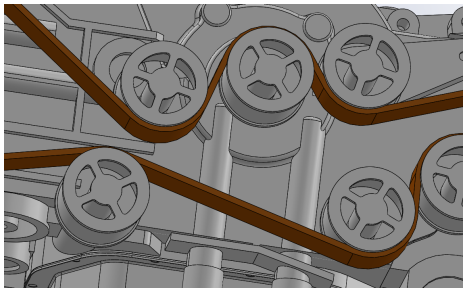


Figura 3: Poleas de la distribución en un tipo de motor Subaru.

Se puede observar en la Figura 4 el contacto entre los segmentos de los pistones y las paredes del cilindro, que necesitan lubricación constante a través de aceite por la fricción que existe.

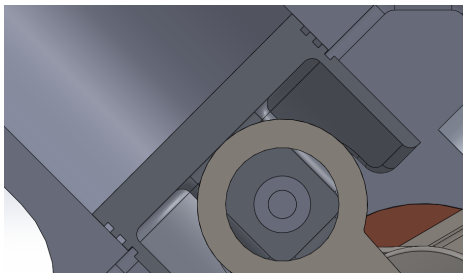


Figura 4: Segmentos del pistón en contacto con la pared del cilindro.

- En la dirección:

  La dirección del vehículo en muchos modelos funciona a través de una cremallera. El volante según gira, desplaza esta cremallera a un lado y a otro accionando el mecanismo de la dirección. Esto se ilustra en la Figura 5.

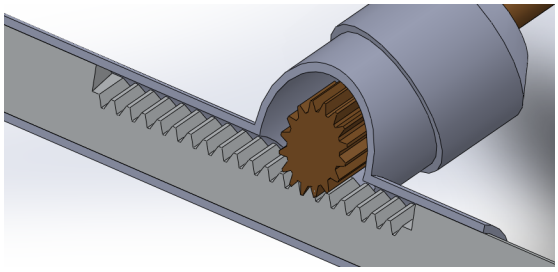


Figura 5: Cremallera de la dirección.

- En la transmisión:

  Como se puede ver en la Figura 6, la transmisión en un coche está constituida por un eje que generalmente une el motor con el diferencial a través de juntas cardán.

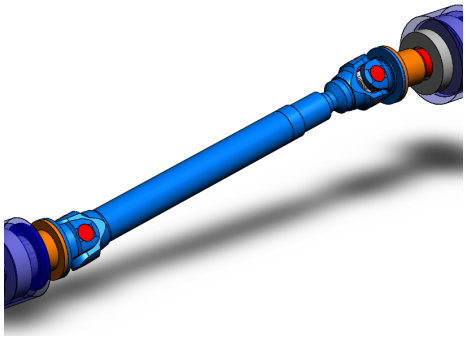


Figura 6: Juntas cardán del diferencial.

- En el diferencial: Es una de las partes de la transmisión aunque se trata más de un aporte de características a esta, a través de unos engranajes (como se muestra en la Figura 7). El diferencial se encargará de que el movimiento del motor pase correctamente a cada rueda, permitiendo mantener el giro incluso cuando alguna rueda se encuentre bloqueada, evitando esfuerzos innecesarios.

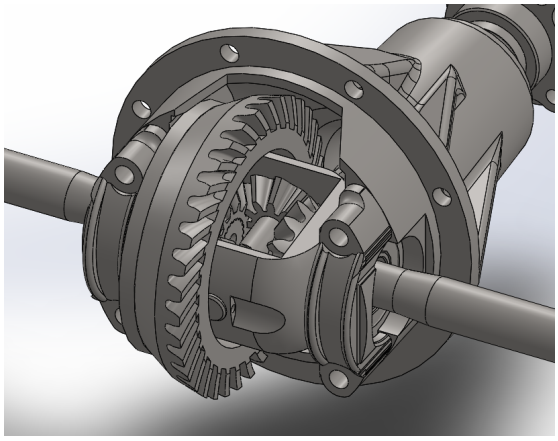


Figura 7: Engranajes del diferencial.

Todas estas situaciones ocurren de forma simultánea un gran número de veces por segundo (un coche solo al ralentí suele estar alrededor de las 900rpm, unas 15 revoluciones por segundo). Y como se puede imaginar, ensayar un sistema de estas características podría ser catastrófico sin una serie de pruebas previas.

Los vehículos pasan por un gran número de pruebas para asegurar su puesta en marcha. Y muchas de estas pruebas ocurren durante el proceso de diseño, corroborando que uno u otro componente que se vaya a poner en servicio cumple este grado de confianza cuando se ensamble completamente.

Antes de la existencia del Método de Elementos Finitos, todas estas pruebas se debían realizar de forma empírica. Por supuesto existían cálculos y unos valores esperados, pero nunca una precisión suficiente como para poder analizar el conjunto completo bajo esos cálculos y obtener una aproximación del resultado.

Actualmente, se entiende la forma de llevar esto a cabo y se cuenta con la tecnología suficiente para hacerlo. Es por esto que se usa regularmente en la industria, permitiendo modificar a tiempo diseños, corregir condiciones o entender los comportamientos de componentes.

Se puede entender que las aplicaciones basadas en el Método de Elementos Finitos tendrán un alto coste computacional, y en concreto, el problema de contacto se encontrará entre los cálculos más costosos. Y aunque ya se han optimizado en gran medida estos procesos de cálculo, es una necesidad continuar en la búsqueda de unos resultados mejores y más rápidos.

Añadido a esto, hablábamos de cgFEM como una versión del Método de Elementos Finitos y una ventaja, representando los objetos de forma más fiel como se verá a lo largo de los siguientes apartados. Usando esta característica se consigue solucionar uno de los principales inconvenientes del propio problema: superfícies en contacto poco precisas en los métodos convencionales. Problema cuyas razones y resolución se irá describiendo a lo largo de los siguientes apartados.

Y es que a pesar de todas las ventajas que presenta cgFEM y aunque este ya tiene el problema de contacto implementado, es susceptible de mejoras aprovechando su propia morfología y propiedades. Por lo tanto, se estará mejorando la resolución de los problemas de contacto en esta aplicación basada en cgFEM con MATLAB, consiguiendo unos resultados precisos en todos los sentidos.

1.2. Estado del arte

En la actualidad, existe un gran número de aplicaciones que hacen uso del Método de los Elementos Finitos para llevar a cabo análisis. Algunos de ellos específicos, mientras que otros implementan servicios de este tipo en programas de Diseño Asistido por Ordenador (CAD). A pesar de la mayor versatilidad o no de estos, todos tienen una base sólida que gira en torno a unos principios: la forma de discretizar el sólido en elementos, las funciones de interpolación o la transformación de coordenadas.

Discretizar el sólido será de hecho el gran pilar de estos análisis y de donde

tomará su nombre "Elementos Finitos", pudiendo dividir el sólido en geometrías más pequeñas llamadas "elementos". Los elementos son la unidad básica para el análisis mediante el Método de los Elementos Finitos y son el pilar entorno al cual se construye el método.

En todos los cálculos usando el Método de los Elementos Finitos, el procedimiento de cálculo es común: discretización del sólido, obtención de las funciones de interpolación, aplicación de las condiciones de contorno y finalmente, resolución, que se explicará en las secciones siguientes. También se explicará hasta qué punto se puede actuar sobre este procedimiento, describiendo el método más general para terminar con cgFEM y así entender el avance que supone y por qué se ha elegido.

1.2.1. Historia del Método de los Elementos Finitos

Para analizar el origen del Método de los Elementos Finitos se debe retroceder hasta los años 40, cuando una serie de matemáticos, ingenieros y físicos empezaron a postular las bases de éste.

El ingeniero Hreinkoff propone en 1941 que, en condiciones de carga, una placa continua se comporta de forma similar a vanos unidos por puntos discretos. Una idea posteriormente recuperada y precisada por McHenry y Newmark, y que significaba poder resolver este tipo de problemas a partir de métodos usados en cálculo estructural.

Courant, propone en 1943 ideas que aparecen de forma similar durante los años 50 de mano de otros matemáticos. En 1946 se publica la teoría de los splines, dando comienzo a la utilización de polinomios para interpolar geometrías.

En este tiempo una pareja de físicos, Prager y Synge, dan pie al desarrollo de métodos para interpretar geométricamente principios de la teoría de elasticidad clásica. Synge definirá funciones lineales sobre regiones trianguladas usando un método de Ritz.

Durante los años 50 en el campo matemático Greenstadt introduce la división de un dominio en áreas más pequeñas poseedoras de una función propia. Este procedimiento se llevaría a cabo de nuevo por parte de White y Friedrichs, que usan elementos triangulares y aplican principios variacionales como su predecesor. Más tarde, McMahon (físico) resolverá un problema de electrostática usando elementos tetraédricos y funciones lineales.

Con el nacimiento de la computación digital, se adapta el cálculo estructural para conseguir eficiencia: se utilizan matrices como inputs.

A mediados de los 50, se diseñan estructuras de aviones usando elementos

triangulares. Turner, Clough, Martin y Top pronosticarán el amplio futuro que le esperaba al método que usaron.

En los siguientes 10 años se consolida tanto el método como todos los conceptos básicos que lo rodean y sus aplicaciones, apareciendo como "Elementos Finitos" hacia los años 60.

Durante esta década se muestra que el Método de Elementos Finitos se puede aplicar a muchos más campos que el del cálculo estructural, diversificándose su uso en campos muy diversos. El método toma un carácter más matemático, teniéndose en cuenta conceptos nuevos que extienden el propio método. En los años de evolución se publican miles de artículos y se refinan tanto los procedimientos como los resultados.

Actualmente, el Método de Elementos Finitos está completamente integrado en la industria. Todos los años de desarrollo lo han llevado a ser capaz de resolver problemas de diferente naturaleza como uno solo y ser la herramienta preferida para problemas complejos. Las mejoras en el cálculo del error también han sido clave ya que como se sabe, la solución de un problema de Elementos Finitos es aproximada.

1.2.2. FEM: Finite Element Method

El Método de Elementos Finitos es un método numérico que permite obtener soluciones muy próximas a la realidad bajo unas condiciones dadas. Habiendo presentado ya en los apartados anteriores su historia, la extensión de las disciplinas que hacen uso del MEF y la importancia que tienen, se expone a continuación cómo consiguen su objetivo.

Como ya se ha visto y a pesar de ser un método reciente en comparación a otros métodos de cálculo ingenieriles, su base matemática es conocida desde hace tiempo aunque no aplicada de la forma en que se hace desde su invención.

1.2.2.1. Ámbito de aplicación

El Método de los Elementos Finitos se usa habitualmente en todo aquello que necesita simularse para conocer su comportamiento, antes de experimentarlo. También se puede usar para intentar aproximar las condiciones a que está sometido un componente o un ensamblaje de algún tipo.

Partiendo de esta premisa, se encuentran en ingeniería mecánica análisis estructurales y análisis de dinámica de fluidos (CFD), análisis térmicos, análisis de vibraciones o caracterización de materiales heterogéneos.

1.2.2.2. Discretización

Tanto en la Introducción del trabajo como en la de este propio apartado del "Estado del Arte" se habla de unos elementos, eje central del éxito del método. Estos elementos son divisiones de un tamaño reducido que buscan aproximar de una manera fiel el sólido del que se pretende conocer su comportamiento al mismo tiempo que se simplifica su definición.

Hay que destacar también que se habla de sólidos porque es el objetivo final de este trabajo. Estos elementos van desde unidimensionales a tridimensionales, aplicándose según se de el caso usando habitualmente las formas que se muestran en la Figura 8. Como en este trabajo se busca resolver un cuerpo tridimensional, los elementos serán tridimensionales.

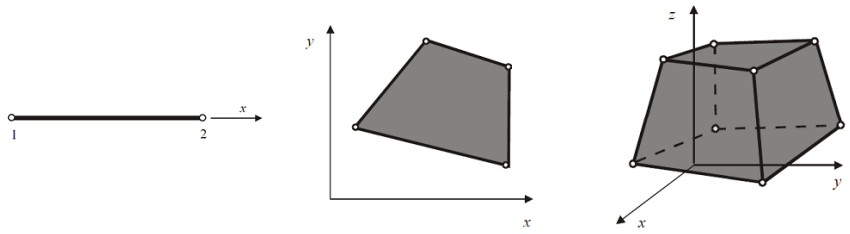


Figura 8: Elementos 1D, 2D y 3D; respectivamente.

Tomando el sólido como un medio continuo, es decir, uniforme y sin ningún tipo de interrupción en las superfícies de su geometría, discretizar este sólido significa reducir a un número de grados de libertad finito este medio continuo (un medio continuo poseería hipotéticamente infinitos grados de libertad) y que solo mediante este mapeado o como se llamará, "mallado", podrá ser resuelto. Con esto se consiguen definir unos puntos de referencia (nodos), con los que se contabilizarán sus grados de libertad (Figura 9) y que permitirán realizar el cálculo.

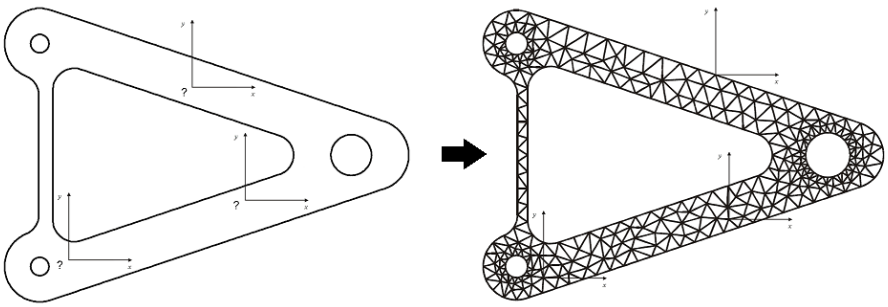


Figura 9: Sólido de medio continuo, discretizado mediante elementos triangulares.

Además, en el método convencional los elementos suelen presentar 2 topologías en 3D, de manera habitual: hexaedros y tetraedros (Figura 10). Los más habituales son los lineales, ya que los problemas no suelen requerir el uso de cuadráticos. Además, los hexaedros son la mejor opción puesto que sus resultados son más precisos. No obstante es más difícil obtener una malla de calidad.

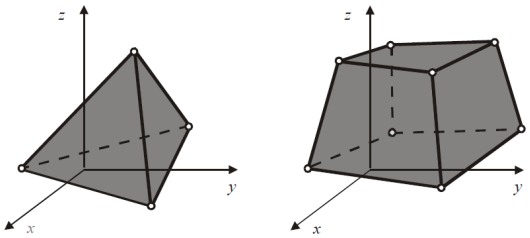


Figura 10: Elementos tetraédricos y hexaédricos, ambos lineales.

1.2.2.3. Interpolación

Una vez se ha discretizado el sólido, se debe poder representar la solución sobre él. Dado que son divisiones del sólido original se debe poder interpretar cada parte del sólido en conjunto. Para ello, dentro de cada elemento la solución vendrá definida por la solución en los nodos del elemento y será interpolada según las funciones de forma. Dichas funciones de forma son polinomios de interpolación de cierto orden, ya que la aproximación de la solución dentro de cada elemento es generalmente polinómica, y que da las características a los elementos lineales o cuadráticos (polinomios de grado lineal o cuadrático) habituales en problemas estructurales.

Estas funciones de forma aportarán a cada posición dentro del elemento un "peso" concreto de cada nodo del mismo. Las funciones de forma, permiten saber la influencia que tiene una variación en esos grados de libertad sobre el elemento en su conjunto.

Las funciones de forma que se definen en cada elemento deben cumplir condiciones de continuidad entre elementos. Ello obligaría a definir específicamente para cada elemento una serie de funciones de forma distintas, lo que sería complejo y muy costoso. Para ello se utilizan elementos isoparamétricos [3, Tema 3: pág. 23]. Estos elementos se definen en coordenas locales una única vez y mediante una transformación de coordenadas se obtiene la geometría real de cada elemento.

Como se ha adelantado, las funciones de forma se obtendrán entonces en coordenadas locales; así como la transformación de coordenadas para obtener sus coordenadas globales definidas para cada elemento, en función de la posición de sus nodos, tal y como se ilustra en la Figura 11. Y por ejemplo, para elementos

lineales hexaédricos 3D sus funciones de forma serán como se muestra en la ecuación (1):

$$N_i = \frac{1}{8}(1+\xi_0)(1+\eta_0)(1+\tau_0)$$
$$\xi_0 = \xi\xi_i; \quad \eta_0 = \eta\eta_i; \quad \tau_0 = \tau\tau_i \tag{1}$$

Donde las variables con subíndice cero van a poder tomar valores tanto positivos como negativos, siendo la variable con subíndice "i" la correspondiente a las coordenadas locales del nodo a analizar y la variable sin subíndice la propia variable. N_i representa por tanto las funciones de forma locales y las variables (ξ, η, τ) las coordenadas de los nodos en el sistema de referencia global.

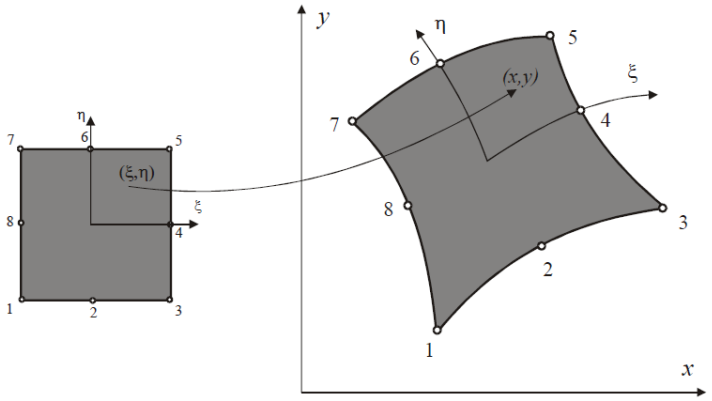


Figura 11: Transformación de coordenadas.

Esta es la forma más común de realizar una transformación de coordenadas, pudiéndose interpolar cualquier punto del elemento.

Por lo tanto, en resumen, las funciones de forma son función de las coordenadas locales que posea el elemento, de forma que siga existiendo una continuidad en las uniones entre elementos. Pudiendo conservar así las propiedades que como ya se comentaba, se pretendían obtener a partir de las funciones de interpolación. Estas propiedades incluyen que la interpolación sea similar en todas las direcciones y que dependa solamente de las funciones de interpolación en los nodos (de los límites) para cada elemento.

La transformación de coordenadas tiene el siguiente aspecto:

$$x = \sum N_i \cdot x_i \tag{2}$$

Esta transformación de coordenadas lleva asociada la matriz Jacobiana de la transformación, que se expone a continuación:

$$J(x,y) = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \tau} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \tau} \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \tau} \end{bmatrix} \tag{3}$$

Conociendo que cada elemento anterior es igual al siguiente sumatorio:

$$\frac{\partial x}{\partial \xi} = \sum_{i=1}^{i} \frac{\partial N_i}{\partial \xi} \cdot x_i \tag{4}$$

Donde la i será el número de nodos, y se obtendrá el producto entre la derivada parcial respecto de la coordenada local, de la función de forma correspondiente a un nodo; y la coordenada global correspondiente para ese mismo nodo.

1.2.2.4. Condiciones de contorno

Como ya se ha indicado, los problemas se resuelven bajo unas condiciones. Estas condiciones se llamarán "de contorno", ya que afectarán precisamente al contorno del sólido; restringiendo este de diferentes formas.

Se distinguen tanto esfuerzos como relaciones de posición, así como restricciones. Serán todas aquellas cualidades que afecten a la resolución. Por ejemplo, se puede imaginar una viga empotrada con un peso colgando de su extremo de la forma que se muestra en la Figura 12.

Figura 12: Esquema del problema.

Sabemos que al ser empotrada, existirán unas restricciones en el extremo izquierdo, manteniendo la viga inmóvil. Mientras que en el extremo derecho, actuará una fuerza vertical. En ANSYS, con su aplicación Workbench, se puede representar como se ve en la Figura 13.

Figura 13: Condiciones de contorno para el problema en ANSYS Workbench. La etiqueta A señala la cara fija, y la B la cara con la fuerza vertical.

Estas dos características son representativas de las dos condiciones principales en MEF: condiciones de desplazamiento (habitualmente denominadas condiciones de Dirichlet) y de esfuerzos (o condiciones de Neumann). Sin embargo, presentan una diferencia, las condiciones de desplazamiento se aplican directamente en los nodos mientras que las de esfuerzos se integran para obtener una fuerza nodal equivalente.

Al poderse extrapolar a gran variedad de situaciones, muchos casos son resueltos siguiendo esta lógica. Obviamente hablando de un entorno ingenieril, no todas las soluciones serán tan sencillas y por ello el tipo de condiciones aplicadas se podría ampliar en gran medida.

A la hora de aplicar estas condiciones al sólido, se definirán sobre cualquiera de sus superfícies, aristas o incluso vértices. Estas partes del sólido que como ya se ha visto han sido discretizadas, recibirán estos esfuerzos o restricciones sobre aquellos elementos que correspondan. Se puede conocer la extensión de esto gracias a la posterior interpolación como ya se describía, completando la información necesaria para resolver el problema.

1.2.2.5. Resolución

A la hora de resolver un problema por el Método de los Elementos Finitos se busca obtener una solución que minimice la energía potencial del sistema variando los desplazamientos nodales. Este objetivo se puede definir como el siguiente problema elástico:

$$min\left\{\Pi_p(u) = \sum_{i=1,2}\left(\Pi_e^{(i)}(u) - \int_{\Gamma_N^{(i)}} u\cdot\hat{t}d\Gamma\right)\right\} \tag{5}$$

Donde $\Pi_p(u)$ es la energía potencial en función de los desplazamientos nodales, con la energía de deformación almacenada en el sólido (ya deformado) definida como Π_e. La integral definida en Γ_N simboliza el contorno en el cual ocurren las cargas que se hayan definido (llamado contorno de Neumann), que se encuentran en el interior de la integral: $u\cdot\hat{t}$ serán esos desplazamientos nodales multiplicados a los esfuerzos de tracción.

Tomando como ejemplo el de la viga anterior, el resultado visual sería similar al que se puede ver en la Figura 14. Una vez obtenido, se deben comprobar los valores en las zonas críticas del sólido según el análisis. A pesar de representarse en escala de colores, esta escala no representa realmente lo crítico que es un resultado y se trata únicamente de una ayuda para identificar resultados.

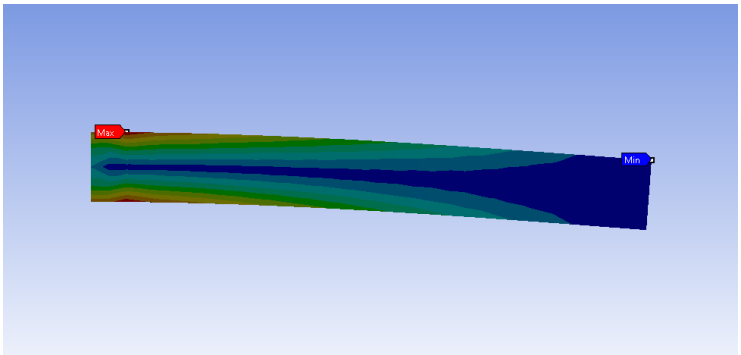


Figura 14: Representación en escala de colores de la solución para el problema anterior.

1.2.3. FEM Contacto

Anteriormente se ha hablado de numerosos ejemplos de problemas de contacto. A la complejidad del contacto se le puede sumar además la consideración de grandes deformaciones elásticas, por lo que las cargas variarán según se deformen los cuerpos analizados. En esta sección se va a presentar el problema de contacto, donde una situación de contacto tiene el aspecto que muestra la Figura 15:

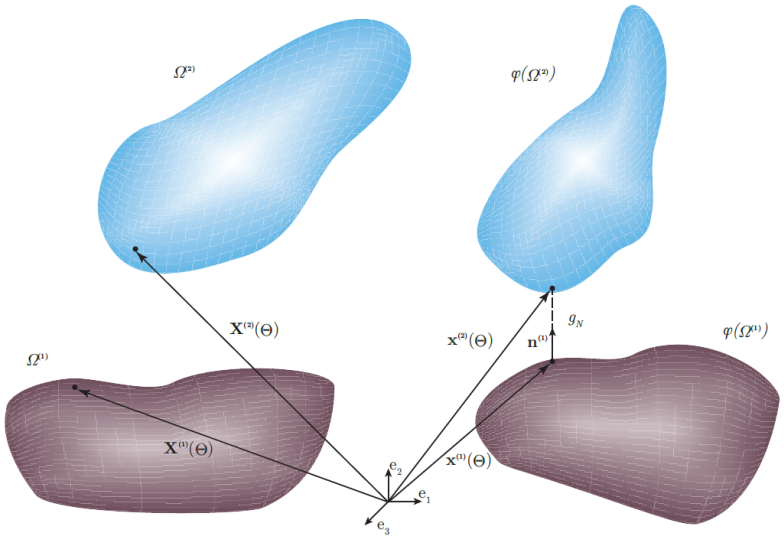


Figura 15: Esquema de la situación de contacto.

En la figura 15 se muestra el esquema de dos cuerpos elásticos que entran en contacto. El contorno de estos cuerpos se divide en tres áreas que no se solapan entre ellas, y son el contorno de Dirichlet (Γ_D), el de Neumann (Γ_N) y la zona donde el contacto entre cuerpos puede ocurrir (Γ_C). Las condiciones de contorno de Dirichlet y Neumann han sido presentadas anteriormente y quedan por definir las condiciones de contorno asociadas al contacto, que se pueden formular como las condiciones de Hertz-Signorini-Moreau:

$$g_n \geq 0,\ p_n \leq 0,\ p_n g_n = 0\ en\ \Gamma_C \tag{6}$$

Donde se han incluido las variables g_n y p_n, que representan la distancia entre cuerpos y la presión normal a la superficie, respectivamente. Estas restricciones fuerzan tanto la impenetrabilidad entre sólidos como que únicamente son posibles tensiones de compresión en la zona de contacto (una tracción implicaría el despegue entre cuerpos).

Para resolver el problema de contacto mediante el MEF se ha utilizado la formulación propuesta en [1] y [2], que impone las restricciones de contacto mediante la introducción de multiplicadores de Lagrange. De esta forma, el problema a resolver se puede escribir como:

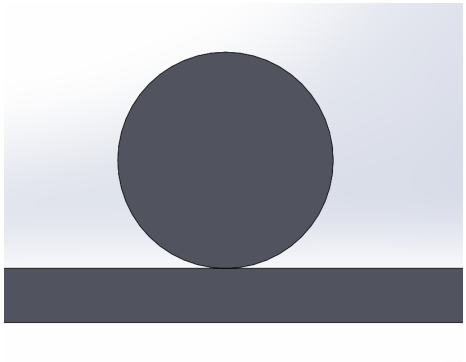
$$opt\left\{\Pi_p(u) + \int_{\Gamma_C^{(1)}} \lambda_N g_N d\Gamma\right\} \tag{7}$$

Sujeto a las siguientes restricciones:

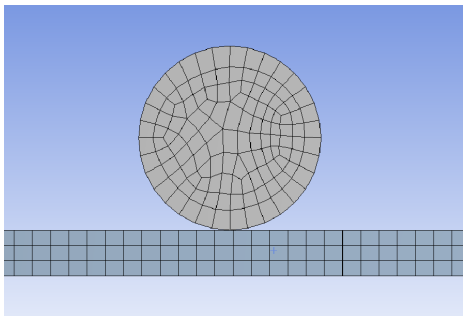
$$g_n \geq 0,\ \lambda_n \leq 0,\ \lambda_n g_n = 0\ en\ \Gamma_C \tag{8}$$

A diferencia de los contornos de Neumann y Dirichlet, la superficie final de contacto no es conocida a priori, es decir, también es una incógnita del problema. Es por ello que el problema de contacto requiere de un proceso iterativo para obtener su solución.

De la forma que se ha descrito, las geometrías son aproximadas a partir de poliedros basados en elementos geométricos. No obstante, como es de esperar, existirán vértices. Estos vértices contribuyen a crear cambios bruscos en la superficie del sólido discretizado, permitiendo que existan situaciones irreales con cierta frecuencia. Un ejemplo muy claro de esto se puede ver en la Figura 16:



(a) Modelo CAD



(b) Sólido discretizado

Figura 16: Modelo CAD, y modelo discretizado de dos sólidos en contacto.

Al hacer zoom sobre la zona de contacto, en la Figura 17 se puede ver que el contacto no es real, y está aproximado. Aún tras un refinamiento de la malla, no se alcanzará ese contacto tangente que se busca.

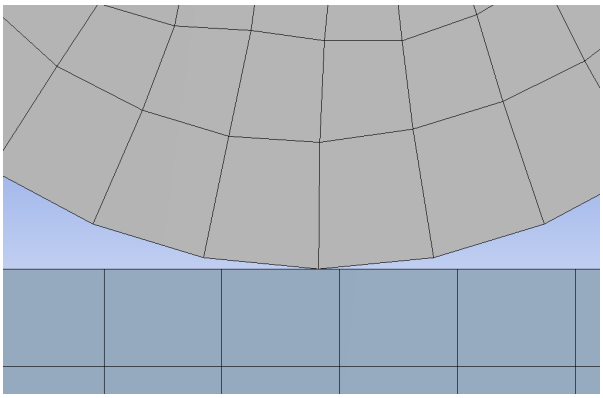


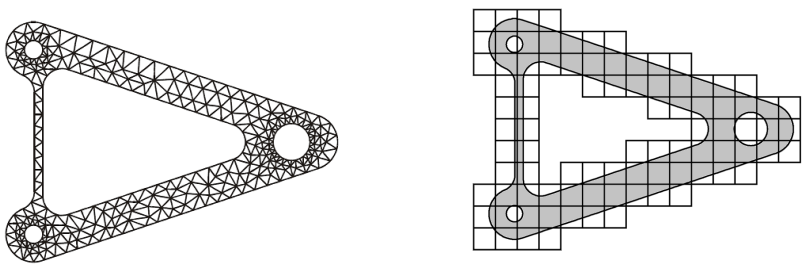
Figura 17: Dramatización del modelo discretizado en contacto con una superficie, en un MEF.

1.2.4. cgFEM: Cartesian Grid Finite Element Method

La alternativa propuesta a los métodos convencionales sería un análisis utilizando cgFEM, el cual da un enfoque diferente a esta práctica. A diferencia de una malla que trata de replicar aproximadamente una superficie a partir de elementos polinómicos, en este método la malla se genera de manera independiente a la geometría original.

Conservar estas caracterísiticas favorece una mayor optimización a la hora del análisis y ha obtenido una gran notoriedad. Sus aplicaciones extienden las que ya se le atribuían a los análisis por Elementos Finitos y hacen uso de NURBS (Non-Uniform Rational B-Splines) aplicados a esta técnica, que son las entidades utilizadas por los programas CAD para definir las geometrías de cálculo.

La resolución por cgFEM hará uso de una discretización regular en toda la malla, permitiendo obtener una distribución uniforme considerando además la geometría CAD del problema. En el siguiente ejemplo (Figura 18) se pueden distinguir las diferencias entre FEM y cgFEM:



(a) FEM (b) cgFEM

Figura 18: Diferencia entre la forma de discretizar para FEM y cgFEM.

Organizar de esta forma los elementos obtenidos para un análisis en concreto proporciona un enorme margen de maniobra en el remallado, no teniendo que reorganizar una compleja malla en un cuerpo tridimensional al conocerse la relación entre los elementos, que es fija. Independientemente del tamaño del elemento, se organizaría en hexaedros que rodean la forma objetivo y que rapidamente permiten distinguir dónde se encuentran los límites del mismo. Usando por ejemplo el Toroide de la Figura 19:

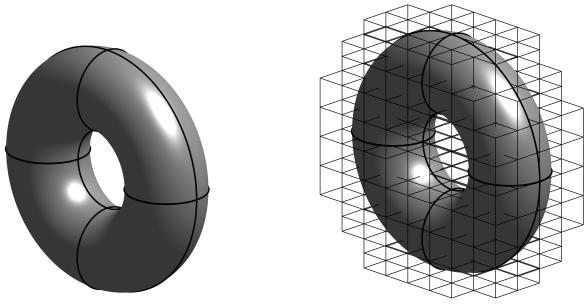


Figura 19: Ejemplo del mallado de discretización aplicado a un Toroide.

Esta malla que rodea el cuerpo estudiado se denomina "Malla de discretización". La malla de discretización embeberá la llamada "Malla de integración", formada por los elementos que rodean los elementos inscritos en el sólido y que se usará de forma auxiliar (Figura 20), contando como ya se viene diciendo, con la geometría CAD.

Las superfícies del sólido que se encuentren dentro de estos elementos exteriores serán integrados para conocer los límites de la malla final. Además, a pesar de este volumen parcial y con una forma diferente a un elemento completo, nuestro Jacobiano se mantendrá constante ya que si bien su distorsión no es la misma que en la malla de discretización sus funciones de interpolación están relacionadas.

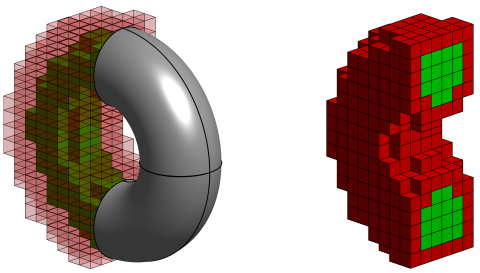


Figura 20: Discretización del Toroide usando cgFEM.

La combinación de estas características permiten resolver el ya también introducido, análisis de contacto. Donde la ausencia de imprecisiones en la geometría

permite evitar distancias entre elementos y puntos de contacto, con superfícies tangentes y evita llevar a resultados erróneos. Al final, cuando se trata de obtener deformaciones y cálculos de esfuerzos un resultado aproximado posee un error capaz de mostrar una situación nada real y es importante que sea preciso.

En la siguiente imagen (Figura 21) se puede ver cómo sería la discretización del mismo caso descrito en el apartado 1.2.3 pero esta vez usando cgFEM. De la misma forma que para el toroide, los hexaedros pertenecientes por completo al sólido se encuentran en verde. Aquellos en los que no está por completo, estarán indicados en rojo. Para aquellos en rojo, se usará la superficie como zona de contacto para conseguir esa tangencia entre el cilindro y la base:

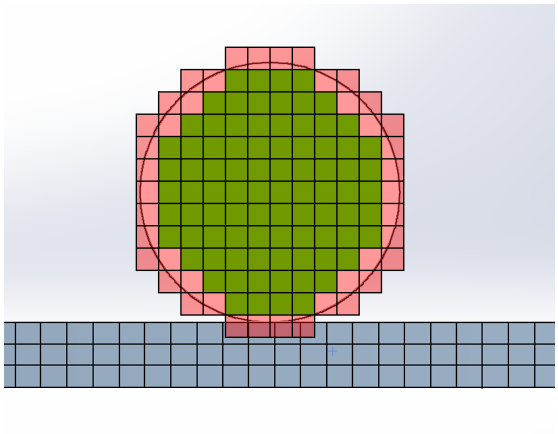


Figura 21: Detalle de la discretización del caso de contacto del apartado 1.2.3 usando cgFEM.

1.3. Objeto de estudio

A lo largo del documento se ha hecho especial hincapié en los casos de contacto como campo de estudio de los Elementos Finitos. Bien destacada su importancia, se puede presentar el objeto de estudio alrededor del cual gira este Trabajo de Fin de Grado: el método de Elementos Finitos con problemas de contacto a resolver.

1.3.1. Contacto entre mallas

A lo largo del programa se hará uso de dos sólidos en contacto para probar la validez de este método. Se trata de un ortoedro regular con dos tipos de malla, una regular de mayor tamaño y otra también regular pero con un mallado más fino. Teniendo estos modelos la ventaja de poseer dos tamaños de malla diferentes, permitiendo probar la precisión con la que una solución general aplicada a este problema podría resolverlo.

Sobre estos ortoedros actuará una esfera entrando en contacto, con una malla definida. Este caso será por tanto como se muestra en la Figura 22:

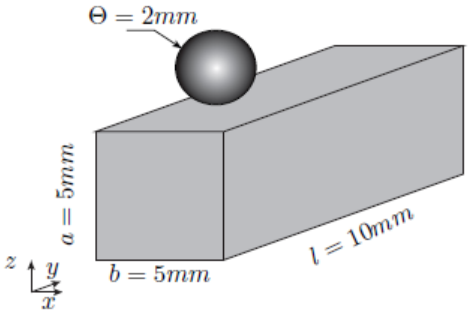


Figura 22: Caso de aplicación.

Las medidas para el ortoedro serán de **5x10x5mm**, y para la esfera un radio **r = 1mm**.

Además, cada uno de estos sólidos tendrá unas propiedades mecánicas que actuarán sobre los resultados obtenidos de su análisis. No solo eso, sino que la malla de los ortoedros se podrá presentar deformada para la solución implementada que se mostrará más adelante.

Es decir, se tendrá un caso de contacto con diversos tipos de mallas y del cual se extraerán una serie de desplazamientos. No obstante, esos desplazamientos ya registrados se han usado también en la implementación para conseguir comprobar que se estaba realizando correctamente.

Este caso ya discretizado quedará como se ve en la Figura 23:

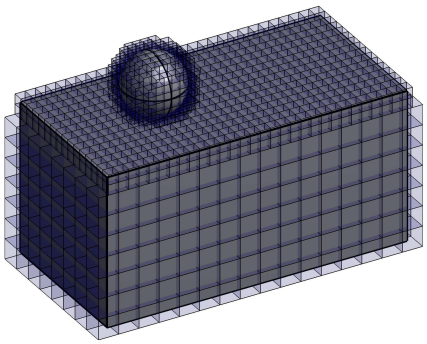


Figura 23: Caso de aplicación discretizado.

En el apartado 3.1 se describirán las condiciones de contorno para este caso y el proceso realizado sobre él, así como las propiedades mecánicas que ya se adelantaban. De esta forma se podrá llevar a cabo un análisis detallado del caso para corroborar las aplicaciones que lo aportado en este trabajo puede tener.

2. Implementación

En esta sección se presenta la implementación realizada en MATLAB para conseguir aplicar toda la teoría y prevista práctica descrita en los apartados anteriores. Se va a describir el proceso tanto de diseño, como iterativo y de programación que se ha llevado a cabo para lograr tener a punto este programa.

2.1. Retos

El objetivo principal de este proyecto es la mejora de la metodología cgFEM en la resolución de problemas de contacto.

Como ya se han introducido y explicado ampliamente, los conceptos de Elementos Finitos alrededor de los cuales vamos a trabajar, ayudan a entender mejor la dificultad que tiene conocer una solución óptima para un problema de contacto.

Para resolver el problema de contacto primero es necesario establecer relaciones entre ambos cuerpos, es decir, definir parejas de puntos de contacto entre ambos cuerpos. Entonces, las fuerzas debidas al contacto se aplicarán entre estas parejas previamente definidas. La implementación existente en cgFEM se divide en dos procesos diferenciados: dado un punto en uno de los cuerpos del análisis (p. ej. el cuerpo 1), se realiza primero un filtrado de los puntos en el cuerpo correspondiente (cuerpo 2) candidatos a ser pareja de contacto. A continuación, se resuelve un problema no lineal mediante el método de Newton-Raphson para seleccionar el punto de contacto de entre esos candidatos. El primer proceso (filtrado de puntos) es muy importante en términos de coste computacional, ya que reduce directamente el número de problemas no lineales que se han de resolver para encontrar las parejas de puntos de contacto. No obstante, este proceso ha de ser robusto, ya que en caso contrario podríamos eliminar el punto real de contacto durante el filtrado. En la implementación original se llevaba a cabo un filtrado utilizando las distancias entre los puntos en contacto (Figura 24) y los centros de los elementos de la malla correspondiente, escogiendo todos los puntos que estuvieran dentro de un radio determinado.

Esta decisión no está para nada desencaminada, ya que es una opción sencilla y que aporta un resultado aproximado y rápido. El problema radica en que la selección de todos los puntos de contacto en ese radio puede llevar a error en casos de mallas muy deformadas e incluso en mallas con contornos difíciles. El riesgo de que un punto del cuerpo 1 en un elemento que no contiene un punto de contacto, se encuentre muy próximo al elemento que realmente contiene al punto de contacto del cuerpo 2, puede inducir a error detectándose como correcto cuando no lo es.

Por lo tanto, el filtrado de puntos en contacto es una opción que debe venir complementada con algún proceso de descarte para garantizar un buen resultado.

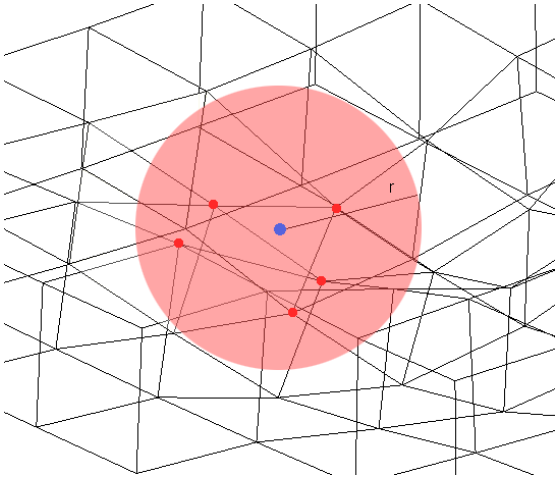


Figura 24: Muestra de la medición de distancias punto-nodo.

En otro orden de cosas, otro reto más es reducir el coste computacional. Y es que este análisis de contacto es uno de los procesos con más coste computacional en lo que a Elementos Finitos respecta. Mejorar la eficiencia computacional a la vez que se incrementa la precisión de este proceso se convierte en una necesidad. En cualquier situación de ingeniería, es habitual encontrar modelos analizados que contienen miles de nodos en sus mallas. Es esto lo que hace verdaderamente complicado optimizar de una forma sencilla. Un tiempo que puede parecer mínimo (del orden de milisegundos) cobra una increíble importancia cuando se enfrenta a un volumen tan grande, haciendo muy importante "arañar" cada fracción de segundo que se pueda ahorrar.

2.2. Solución general

Se ha hablado de retos, características de los Elementos Finitos y de en qué consiste el Método de Elementos Finitos. Todo esto lleva a la explicación de la solución más óptima para llevar a cabo el análisis.

En este apartado se describirán las soluciones que llevan a la consecución de esta optimización y cómo se consigue mejorar el desempeño del programa.

En el apartado anterior se ha presentado un método de búsqueda de puntos en contacto, el cual usa un radio para encontrar los elementos candidatos que podrían tener puntos en contacto en su interior. El problema con este método como ya se decía y por las razones que se describían, es que no es preciso y en casos particulares (pero muy comunes) no funcionaría.

Viendo la situación que se plantea, se propone una solución similar pero con otro enfoque. Si bien conseguir un radio que circunscriba el elemento y usarlo

de esta forma ha demostrado los problemas descritos, debe existir una distancia límite a la que poder comprobar la posición de los nodos.

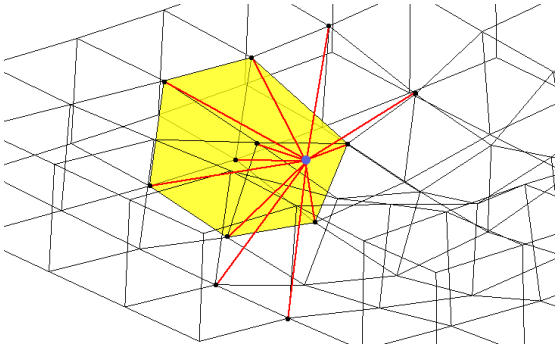


Figura 25: Muestra de la medición de distancias punto-nodo.

A modo de rango, se propone usar esta idea, pero por si la malla no fuese regular, tomar la medida mínima de distancia que exista entre los nodos en la malla y el punto de contacto (tal y como se muestra en la Figura 25). Así se consigue disminuir de manera eficiente el espacio de búsqueda, ya que al utilizar los nodos de los elementos y no su centro (como se hacía anteriormente) tiene en cuenta la distorsión de los elementos en la búsqueda, incrementando la robustez del método para encontrar los elementos candidatos.

Así se da con una forma de encontrar rápidamente los elementos candidatos al tener el punto de contacto dentro de ellos. Esos puntos comprobarán todos los nodos que se encuentran a su alcance a partir de la distancia mínima que se haya calculado, de la forma que se indica en la Figura 26.

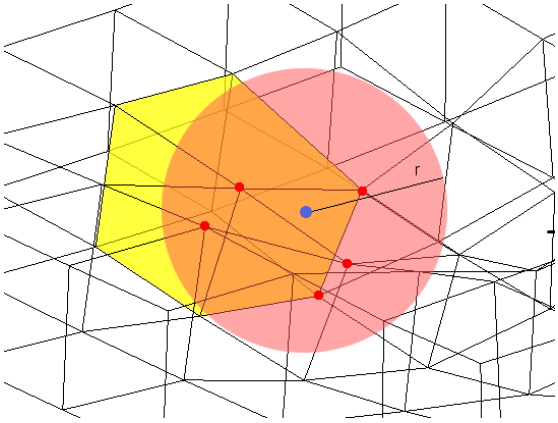


Figura 26: Método de aplicación del radio mínimo para la malla.

Una vez detectados los nodos que recaen dentro de un radio desde el punto de contacto, el siguiente paso es hallar aquellos elementos que recaen en esa zona.

Para ello, se seleccionan todos los elementos asociados a dichos nodos. De esta lista de elementos, se vuelven a seleccionar únicamente aquellos que tienen todos sus nodos dentro de la zona de búsqueda. Para ello, se utiliza un código binario (0/1), asignando un 1 a aquellos nodos que recaen en la zona de búsqueda y 0 al resto. Con ello, únicamente se seleccionarán los elementos que tengan nodos marcados como 1, tal y como se observa en la Figura 27.

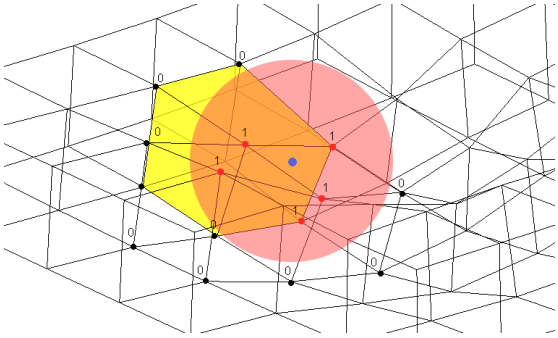


Figura 27: Clasificación de los nodos como miembros de potenciales candidatos.

Existe no obstante un paso más, ya que aunque se haya definido una lista de elementos, de momento son solo candidatos. Aquí difiere este cálculo del planteado originalmente. Se realiza un tercer paso para averiguar dentro de qué elemento de la lista se halla el punto de contacto.

Se toman todos los elementos candidatos para cada uno de los puntos de contacto (Figura 28). Esta será la última comprobación que se realizará, ya que parece obvio que aunque un solo punto de contacto puede pertenecer hasta a 8 elementos a la vez en el caso de que el punto de contacto coincida con un nodo de la malla, la mayoría de puntos de contacto pertenecerá únicamente a un elemento.

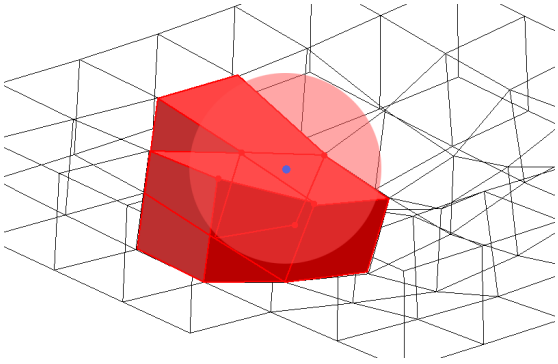
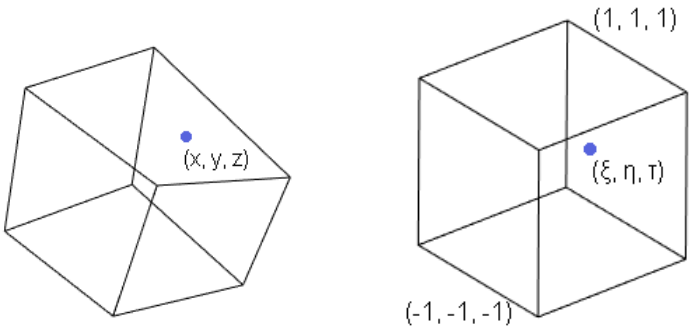


Figura 28: Elementos candidatos resultantes.

El procedimiento planteado trata de ver que el punto de contacto se encuentra dentro de un elemento en concreto. Para ello, se elige un elemento candidato y se hallan las coordenadas locales del punto candidato en el elemento. Como las coordenadas locales de un elemento de este tipo toman valores entre -1 y 1, si las coordenadas locales del punto de contacto se encuentran dentro del rango válido, entonces el punto de contacto estará dentro del elemento. Si alguna de ellas superan dichos valores límite, se continuaría buscando con otros elementos de la lista.

Se usará entonces la transformación de coordenadas ya introducida con este propósito en el apartado 1.2.1, buscando obtener las coordenadas locales de esos puntos de contacto respecto de cada elemento candidato y comprobar su pertenencia.

Por lo tanto, y siguiendo las pautas marcadas en la explicación del apartado mencionado, debe ser posible transformar las coordenadas "globales" del punto de contacto, a las coordenadas "locales" dentro del elemento (esto se ilustra con un caso típico en la Figura 29). El problema aquí es que se desconocen completamente las coordenadas locales, y mientras que transformar las coordenadas de locales a globales resulta relativamente sencillo, el caso inverso resulta más complejo ya que no existe una relación lineal (al desconocer sus coordenadas locales, desconocemos su jacobiano) dando lugar a un proceso iterativo.



(a) Coordenadas globales (b) Coordenadas locales

Figura 29: Las dos formas de un mismo punto dentro de un elemento.

Sabiendo sus coordenadas "globales" y siendo sencillo pasar de locales a globales, la iteración consistirá en suponer que el nodo se encuentra en un punto arbitrario dentro del elemento e iterar mediante el método de Newton-Raphson hasta conocer las coordendas locales del punto. La expresión para el método de Newton-Raphson aplicado a este trabajo será la siguiente:

$$\boldsymbol{\xi}_{i+1} = \boldsymbol{\xi}_i + \boldsymbol{J}^{-1} R(\boldsymbol{\xi}_i)$$
$$\text{donde } R(\boldsymbol{\xi}_i) = ||\boldsymbol{x}(\boldsymbol{\xi}_i) - \boldsymbol{x}_0|| \tag{9}$$

2.3. Problemas y soluciones

A lo largo de este apartado se enumeran los problemas surgidos en el camino de conseguir lo descrito en el apartado de "Retos". Esta sección sirve de precedente al apartado de "Iteraciones" donde se describirán las modificaciones hechas una vez logradas las soluciones a los problemas surgidos y los resultados obtenidos.

Estos problemas no son de conceptos relacionados con el objetivo final del programa en cuanto a teoría se refiere como ocurría en la "Solución general", se trata de problemas a la hora de alcanzar la mejor eficiencia o la mejor solución basándose en un desarrollo correcto de la "Solución general".

En el desarrollo del proyecto se han sucedido las siguientes ideas:

2.3.1. Problemas de velocidad

MATLAB es una gran herramienta de cálculo, sencilla de utilizar y muy popular en ingeniería. Sin embargo, presenta algunos inconvenientes relacionados con alguna serie de procesos que no es capaz de realizar todo lo rápido que a veces es necesario.

Por ejemplo, MATLAB es "muy lento" llevando a cabo bucles que se repiten muchas veces. Bucles típicos en programación como "for", "while", ... son costosos y reducirán la eficiencia del código. Esto se debe a que MATLAB utiliza un lenguaje de programación interpretado, es decir, "lee" línea por línea durante su ejecución sin previo conocimiento de lo que está "leyendo". Desconoce el número de iteraciones del bucle y debe procesarlo cada vez.

Además, aunque presenta una serie de funciones muy útiles para la resolución de problemas generales como puede ser "A\b", se trata de operaciones que al realizarse muchas veces dejan entrever una velocidad inferior a cualquier otra solución en un lenguaje de programación compilado (como C, por ejemplo).

Para resolver algunos de los problemas de velocidad de resolución encontrados en el código se propone por parte de los tutores de este trabajo utilizar código C, interpretado en MATLAB a través de un archivo MEX previamente compilado.

Los archivos MEX tienen un abanico de lenguajes de programación que son capaces de adaptar a MATLAB, tales como C, C++ o FORTRAN. Esto es logrado a partir de la función MEX, una serie de instrucciones que adaptan los inputs y outputs de los procesos dentro de un archivo de estos lenguajes a soluciones capaces de ser procesadas y llamadas de forma corriente por MATLAB.

Tras investigar una serie de ejemplos, esta solución pareció más que válida para todos aquellos procesos que se podían optimizar en gran medida consiguiendo

así un código mucho más veloz y con un resultado igual al del proceso original dentro de MATLAB.

2.3.2. Operaciones repetitivas

Si antes se hablaba de problemas de velocidad relacionados con la incapacidad de MATLAB para conocer procesos a priori, en este caso se describen herramientas del propio MATLAB para lidiar con operaciones repetitivas.

En primer lugar está la propia capacidad de MATLAB para hacer esto. Está optimizado y basado para usarse en operaciones con matrices y vectores, haciendo muy sencillo organizar datos de forma vectorial o matricial y recorrerlos de forma más rápida. Esto se llama vectorización y se utilizará a lo largo del código.

Otra solución será usar bucles "parfor", una variante del bucle "for" clásico. Se llama así por ser un bucle "for" en paralelo, es decir ejecutando varias iteraciones del mismo bucle al mismo tiempo como una forma de computación paralela. Para un volumen tan grande de iteraciones esta solución es definitivamente interesante.

También existen una serie de procesos que se comportan igual que, por ejemplo, varios bucles. De la misma forma que con el bucle "parfor", cuentan con la ventaja de estar ya implementados en MATLAB. Esto ahorra la necesidad de programar más archivos MEX o sacrificar coste computacional. Estas operaciones repetitivas se encuentran en las partes del código donde se lleva a cabo la diferencia entre cada uno de los nodos de una malla con todos los puntos de contacto. La función encargada de llevarlo a cabo será "bsxfun".

La función "bsxfun" aplica una condición de operación a todos los elementos de dos matrices indicadas en la declaración. Las condiciones van precedidas de una arroba y tienen sus propios nombres. Por ejemplo, para la diferencia se ha usado "@minus", pero existen otras como "@power", "@plus" u operaciones lógicas como "@and" u "@or".

La declaración será de la forma:

```
1 C = bsxfun(fun, A, B);
```

Donde "fun" será esa operación precedida de arroba. Y "A" y "B" las dos funciones sobre las que se actúa.

2.4. Código: ”Where am I?’

Finalmente, y tras validar las soluciones planteadas, se han implementado en la aplicación FEAVOX. El código de esta mejora estará compuesto por las funciones previamente comentadas entre otras, las cuales se recorrerán a lo largo de esta sección.

A continuación se desgrana por orden cada paso que se da en cada una de las funciones junto con todas sus respectivas mejoras, y cómo dándoles uso, el programa es capaz de reconocer los contactos.

Este es el archivo de MATLAB que condensa el grueso de cálculos que se realizarán. A lo largo de la funcion ”whereami” se recorren el resto de funciones extrayendo los datos necesarios y operándolos. La selección de los elementos correctos de entre los candidatos también se realiza en su interior.

La función únicamente requiere tres inputs: los puntos de contacto, la selección de la malla base y la selección del cuerpo de la malla a mostrar:

```
1  function E = whereami(XYZ, iMesh, iBody)
2  % Se define la función ''whereami''.
```

Siendo ”XYZ” los puntos de contacto que ya introducíamos, procedentes del archivo de los puntos que entran en contacto con la malla.

Al ser la malla colisionada una malla deformada, la forma de almacenar su información será tomando por separado los nodos sin deformar y los desplazamientos que se han encontrado tras una simulación de esfuerzos. Por ello, se deben obtener las coordenadas ya deformadas:

```
9  XYZ_N = Mesh(iMesh, iBody).Node.XYZ + Results(iMesh, iBody).Node(:,
       1:3)';
10 % Se suman las coordenadas de la malla a las deformaciones.
```

Algo esencial es que el programa entienda el orden de magnitud en que trabaja. Para esto se hará uso de la función ”Radius.m”, la cual analiza todos los nodos y obtiene la menor distancia existente entre ellos, convirtiéndose en la máxima distancia a la que un punto de contacto puede estar alejado, perteneciendo a un elemento.

```
9  radius = Radius(XYZ_N);
10 % Se calcula la distancia mínima en la malla.
```

”Radius.m” lo realiza de la siguiente forma:

- **”Radius.m”**

```
6  function radius = Radius(xyzN)
7  % Se cuenta el número de nodos
8      Node_Dim = size(xyzN, 2);
9
```

```
10      parfor i = 1:Node_Dim
11  % Primero, cada nodo en la malla es restado respecto del resto
        de nodos en la malla.
12          NodeDiff = vecnorm(bsxfun(@minus, (xyzN(:, i)),...
13                      (xyzN(:, 1:end ~= i))));
14  % La mínima distancia es extraída como la mínima distancia
        entre nodos, es decir, como ya se decía, la máxima
        distancia a la que un nodo puede estar de un elemento.
15          radius = min(NodeDiff);
16      end
17
18  end
```

De vuelta en "whereami.m" y para el la representación final, se separan las coordenadas "XYZ" en una variable separada para cada una:

```
13  Xp = XYZ(1,:);
14  Yp = XYZ(2,:);
15  Zp = XYZ(3,:);
16  % Se guardan los valores de los puntos.
```

Como últimos preparativos, se inicializa la variable "E" con un vector de ceros del tamaño correspondiente a la cantidad de puntos:

```
19  E = zeros(1, size(XYZ, 2));
20  % Se crea el vector de elementos resultado.
```

Además se inicia un contador y la utilidad "profile" con el objetivo de medir el funcionamiento del programa, así como el coste computacional. Ambas utilidades, darán por finalizada su medición al final del programa, sin tener en cuenta el gráfico final:

```
21  TStart = tic;
22  profile on
23  % Se empieza a contar el tiempo de ejecución.

    ⋮

61  Time = toc(TStart);
62  profile off
63  profile viewer
64  % Se termina con esta línea de contar y se visualizan los
        resultados.
```

Estos son los inputs que obtiene la función principal para lograr calcular el problema. No obstante, y complementando a la solución propuesta, se exponen los pasos tomados a continuación:

Como primer paso del cálculo dentro de "whereami" se recurre a la función "NodeRad.m", responsable de encontrar los elementos candidatos. Esta función encontrará todos los elementos del objeto que probablemente contengan el punto

de contacto, y que se encuentran a la distancia "Radius" de cada uno del resto de los puntos en la malla base:

```
19 Elm = NodeRad(XYZ,XYZ_N,Mesh,radius,iMesh,iBody);
20 % Se guardan los resultados de ''NodeRad''.
```

La función "NodeRad.m" es similar a "Radius.m" en cierto modo aunque sobre sus resultados se realizan más operaciones, tomando una serie de medidas para asegurar que un elemento es un potencial candidato para un punto de contacto en concreto. En principio, "NodeRad" tenía el siguiente aspecto:

- **"NodeRad.m"**

```
6  function Candidates = NodeRad(XYZ,XYZ_N,Mesh,radius,iMesh,
       iBody)
7      XYZ_Dim = size(XYZ,2);
8
9  % Se inicia el bucle de cálculos:
10     for i = 1:XYZ_Dim
11
12         NodeDiff = vecnorm(bsxfun(@minus, XYZ(:,i),...
13             (Mesh(iMesh,iBody).Node.XYZ)));
14 % Primero, cada nodo en la malla en contacto se resta a cada
       nodo de la malla base.
15 % Después, el módulo del vector resultante se obtiene para
       tener un valor de distancia.
16         NormDiff = NodeDiff <= radius;
17 % Cada distancia nodo-nodo se compara con el mínimo radio.
18         NodeCell = find(NormDiff ~= 0);
19 % Entonces, se almacena el número de la columna con valores
       diferentes de 0.
20         NodesCheck = ismember(Mesh(iMesh,iBody).Element.
               Topology,NodeCell);
21 % Cada elemento se comprueba buscando un nodo válido en su
       interior, tomando '1' cuando lo hay, y '0' cuando no.
22         FilteredNodes = vecnorm(double(NodesCheck));
23 % Se calcula el módulo del vector de cada columna, asegurando
       así que no hay ningún valor nulo. Obteniendo solo
       elementos con el nodo en su interior y no algunos que solo
        tengan algunos nodos en común.
24         Candidates{i} = find(FilteredNodes ~= 0);
25 % Estos candidatos se almacenan en la variable vector ''
       Candidates''.
26     end
27 end
```

Se trata además de una función ya compacta en su origen, donde se realiza una operación de resta para todos los nodos usando "bsxfun" con su argumento "@minus" para indicar esta misma operación. A lo largo del código se van almacenando las matrices resultantes en las llamadas celdas (donde una celda es una matriz de matrices), creando una matriz de celdas (Ecuación 10) albergando

otras matrices de diferentes tamaños a su vez. Esta vez unidimensionales, es decir, vectores:

$$\left\{ \begin{matrix} [x_1\ x_2\ x_3\ x_4] \\ [y_1\ y_2] \\ [z_1\ z_2\ z_3] \\ [n_1] \\ \vdots \end{matrix} \right\} \tag{10}$$

No obstante, y dado que se trabaja con una variable por cada línea, se buscaba hacer más eficiente y fácil de trabajar, llevando a la versión definitiva. En esta última versión se concatenan las funciones que "daban" los valores resultantes a esas variables:

```
6  function Candidates = NodeRad(XYZ, XYZ_N, Mesh, radius, iMesh,
       iBody)
7      XYZ_Dim = size(XYZ, 2);
8
9      parfor i = 1:XYZ_Dim
10         NodeCell = find((vecnorm(bsxfun(@minus, XYZ(:,i), ...
11             (XYZ_N))) <= radius) ~= 0);
12
13         Candidates{i} = find(vecnorm(double(ismember(Mesh(iMesh,
               ...
14             iBody).Element.Topology, NodeCell))) ~= 0);
15 % Como se puede ver, se han introducido algunas funciones dentro de
       otras.
16 % Se puede advertir el alcance de este cambio viendo las variables
       que han quedado.
17     end
18 end
```

Se puede ver cómo se usa por primera vez el bucle "parfor" en esta función, y se continuará para el propio "whereami". La influencia de este cambio se puede comprobar en la Figura 30, donde se compara el programa completo antes y después de usar "parfor" en todos las iteraciones posibles. Además se concatenan los "outputs", que simplemente se añaden como una condición más, siempre y cuando la función lo permita. Así se consigue resolver este bucle con solo dos variables.

Además, se puede ver que cumple con el otro objetivo, que era conseguir un menor coste computacional. Al final esta es una función que se repite muchas veces, teniendo que recorrer y realizar estos cálculos sobre la malla para cada punto de contacto. El código "pierde" menos tiempo en otras líneas y se concentra el coste computacional en unas líneas más concretas. Reduciendo la función de esta forma se conseguía mejorar el tiempo ligeramente, comprobándose que esta manera de organizarla era la más óptima a pesar de obtener una mejora ínfima como se muestra en la Figura 31.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
whereami	1	12.954 s	1.961 s	
EvalLocCoords3D	144312	8.536 s	6.778 s	
NodeRad	1	2.398 s	1.891 s	
ShapeFunctions (MEX-file)	385822	0.845 s	0.845 s	
Jacobian (MEX-file)	192910	0.547 s	0.547 s	
ismember	6700	0.507 s	0.046 s	
ismember>ismemberR2012a	6700	0.461 s	0.066 s	
ismember>ismemberBuiltinTypes	6700	0.395 s	0.395 s	
Inverse (MEX-file)	192910	0.366 s	0.366 s	
cell2mat	6700	0.059 s	0.059 s	
deal	1	0.000 s	0.000 s	

(a) "whereami" sin usar "parfor"

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
whereami	1	12.594 s	0.007 s	
parallel_function	2	12.581 s	3.733 s	
EvalLocCoords3D	144312	8.308 s	6.588 s	
NodeRad	1	2.277 s	0.002 s	
ShapeFunctions (MEX-file)	385822	0.830 s	0.830 s	
Jacobian (MEX-file)	192910	0.533 s	0.533 s	
ismember	6700	0.483 s	0.046 s	
ismember>ismemberR2012a	6700	0.437 s	0.052 s	
ismember>ismemberBuiltinTypes	6700	0.385 s	0.385 s	
Inverse (MEX-file)	192910	0.355 s	0.355 s	
cell2mat	6700	0.054 s	0.054 s	
parfor_sliced_fcnhdl_check	2	0.002 s	0.002 s	
onCleanup>onCleanup.delete	2	0.001 s	0.001 s	
parallel_function>results	2	0.001 s	0.001 s	
incrementParallelFunctionDepth	4	0.001 s	0.001 s	
parfor_endpoint_check	4	0.001 s	0.001 s	
deal	1	0.001 s	0.001 s	
...al.incrementParallelFunctionDepth(-1)	2	0.001 s	0.000 s	
parallel_function>PCTInstalled	2	0.000 s	0.000 s	
onCleanup>onCleanup.onCleanup	2	0.000 s	0.000 s	

(b) "whereami" usando "parfor"

Figura 30: Comparación del coste computacional para "whereami" con y sin el bucle "parfor".

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
13	XYZ_N));	6700	0.654 s	46.4%	
17	FilteredNodes = vecnorm(double...	6700	0.350 s	24.8%	
16	NodesCheck = ismember(Mesh(iMe...	6700	0.303 s	21.5%	
18	Candidates{i} = find(FilteredN...	6700	0.038 s	2.7%	
15	NodeCell = find(NormDiff ~= 0)...	6700	0.037 s	2.6%	
All other lines			0.030 s	2.1%	
Totals			1.412 s	100%	

(a) NodeRad V1

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
27	(XYZ_N))) <= radius) ~= 0);	6700	0.687 s	49.8%	
45	iBody).Element.Topology,NodeCe...	6700	0.677 s	49.1%	
26	NodeCell = find((vecnorm(bsxfu...	6700	0.015 s	1.1%	
46	end	6700	0.001 s	0.0%	
44	Candidates{i} = find(vecnorm(d...	6700	0.000 s	0.0%	
All other lines			0.000 s	0.0%	
Totals			1.380 s	100%	

(b) NodeRad V2

Figura 31: Tiempo usado por "NodeRad" en sus diferentes versiones.

Una vez obtenidos los elementos candidatos, se prepara el proceso para obtener los elementos finales donde se encuentran los puntos. Por lo tanto, se debe obtener la matriz de funciones de forma (matriz N) y su primera derivada parcial respecto de cada coordenada local (matriz dN) para el origen de las coordenadas locales de cada elemento.

Este proceso se describe en los siguientes pasos dentro del código. Para preparar las funciones que darán solución, se hará uso de la función "ShapeFunctions", el archivo MEX compilado a partir de un archivo en C: "ShapeFunctions.c' (y sustituyendo al script "CubeShapeFunctions.m")'. Se usará un archivo MEX porque como ya se ha descrito, un archivo C es mucho más rápido que un script de MATLAB. Se trata de la implementación de algunas funciones de forma (ya descritas brevemente y que siguen la forma de la Ecuación 1). La utilización de esta función se llevará a cabo de la siguiente forma:

```
26 [dN] = ShapeFunctions(0, 0, 0, 1, 1);
27 [N] = ShapeFunctions(0, 0, 0, 1, 0)';
28 % Donde los tres primeros dígitos serán las coordenadas locales, el
       siguiente el grado del elemento (''1'', es decir, lineal) y el
       último indica si se trata de las derivadas o no de forma
     binaria (''0'' o ''1'').\\
```

Se organizan entonces cada una de las filas de la matriz dN (matriz N derivada parcialmente respecto de cada una de las coordenadas locales), dividiéndose en dNdPsi, dNdEta y dNdTau respectivamente.

```
28 [dNdPsi, dNdEta, dNdTau] = deal(dN(1,:), dN(2,:), dN(3,:));
```

El paso siguiente es realizar el corte y conseguir conocer los elementos correctos. Esto será un bucle que irá desde el primero al último nodo de la malla en contacto.

La primera operación realizada en el bucle va a ser convertir cada celda de "Elm" que se obtenía de la función "NodeRad" a vectores individuales. Como se adelantaba en "Retos" y posteriormente en "Soluciones", se elegían las celdas como almacenamiento por el tamaño cambiante del vector resultado en "NodeRad".

A pesar de tratarse de un bloque de código para un bucle parfor, se va a recorrer línea por línea ya que en cada una hay un proceso diferente:

```
31 parfor i = 1:size(XYZ,2)
32 % Un nuevo bucle recorre los nodos.
33     Candidate = cell2mat(Elm(i));
34 % Es realizada la conversión de celdas a vectores (se usa cell2mat
       donde al final la matriz resultado será unidimensional, el
       vector de candidatos).

   ⋮

45 end
```

Siguiente a esto, necesitamos otro bucle para recorrer cada uno de los vectores de candidatos. Es entonces donde se comprueba de forma más precisa si nuestros candidatos son en realidad candidatos válidos.

```
   ⋮

35     for j = 1:size(Candidate,2)
36 % Se inicia un nuevo bucle que recorre los candidatos.

   ⋮

43     end

   ⋮
```

Dentro de este bucle concatenado al anterior, se analizan finalmente las conclusiones obtenidas hasta ahora sobre los candidatos. Y para esto, se hará uso de una función más: "EvalLocCoords3D.m" la cual se pasa a explicar a continuación:

```
⋮
36          XYZElm = XYZ_N(:, Mesh(iMesh, iBody).Element.Topology(:,
                Candidate(j)));
37          PosL = EvalLocCoords3D(XYZElm', XYZ(:, i), N, dNdPsi,
                dNdEta, dNdTau);
⋮
```

En la primera parte, se escribirá en XYZElm el vector correspondiente a la columna donde se encuentra el elemento candidato a analizar en ese momento. Se debe recordar que j era la variable que recorría los vectores de candidatos. Accede a la estructura de la malla "Mesh" con nuestro "iMesh" y "iBody" correspondiente, dentro de "Topology" en "Element". Y una vez dentro, selecciona el número de nodo correspondiente a cada uno de los nodos de ese elemento en cuestión.

Una vez definido "XYZElm" se le puede pasar a la función "EvalLocCoords3D". Cuyo objetivo es obtener "PosL" una variable vector que almacenará las coordenadas locales de un punto en coordenadas globales. De esta forma podemos analizar su pertenencia sin vernos afectados por la posible distorsión del elemento en coordenadas globales.

El código de EvalLocCoords3D estará enteramente enfocado a este objetivo y se detalla a continuación, línea por línea:

- **"EvalLocCoords.m"**

```
6   function [PsiEtaTau] = EvalLocCoords3D(XYZElm, XYZ, N, dNdPsi,
        dNdEta, dNdTau)
7   % La variable FEDegree tomará valor 1 ya que los elementos son
        lineales.
8       FEDegree = 1;
9   % PsiEtaTau se inicializa al origen de coordenadas locales,
        recordemos que estas coordenadas tomarán valores de -1 a
        1.
10      PsiEtaTau = [0; 0; 0];
11  % Se define una tolerancia que será el máximo error que se
        pretende tener en el cálculo.
12      Tol = 10e-2;
13
14      NIter = 0;
15  % Las coordenadas globales iniciales se obtienen multiplicando
         las funciones de forma por las coordenadas del elemento.
16      TrialXYZ = (N * XYZElm)';
17  % Un residuo inicial es obtenido haciendo la diferencia entre
        las coordenadas globales objetivo y las coordenadas
        globales obtenidas inicialmente.
18      Res = XYZ - TrialXYZ;
```

```
19  % Se recorre un bucle iterativo mientras el valor del residuo
        sea superior a la tolerancia.
20      while norm(Res) > Tol
21          NIter = NIter + 1;
22
23  % Se obtiene el Jacobiano del elemento.
24          J = Jacobian(dNdPsi', dNdEta', dNdTau', XYZElm(:, 1),
                XYZElm(:, 2), XYZElm(:, 3));
25
26  % Se obtiene la inversa del Jacobiano.
27          I = (Inverse(J(1,:), J(2,:), J(3,:)))';
28
29  % El valor de las coordenadas locales para cada iteración se
        obtiene mediante el sumatorio de los valores de estas
        coordenadas locales sumados al producto de la inversa del
        jacobiano y el residuo.
30  % Este mismo producto se encuentra precedido por la función ''
        round'' ya que debido a la gran cantidad de decimales
        genera conflicto con MATLAB en puntos avanzados del cá
        lculo.
31  % Así se consigue ir modificando el valor de coordenadas
        locales hasta dar con las que corresponden al nodo dentro
        del elemento en cuestión.
32          PsiEtaTau = PsiEtaTau + round(I*Res, 14);
33
34  % Se obtienen las funciones de forma y sus derivadas.
35          [dN] = ShapeFunctions(PsiEtaTau(1, :)', PsiEtaTau(2,
                :)', PsiEtaTau(3, :)', FEDegree, 1);
36          [N] = ShapeFunctions(PsiEtaTau(1, :)', PsiEtaTau(2, :)
                ', PsiEtaTau(3, :)', FEDegree, 0)';
37
38          dNdPsi = dN(1, :, :);
39          dNdEta = dN(2, :, :);
40          dNdTau = dN(3, :, :);
41
42  % Tras cada iteración se obtiene un valor de coordenadas
        globales nuevo de la misma forma que antes, iteración que
        continúa hasta que se obtienen las coordenadas globales
        que se conocen.
43          TrialXYZ = (N * XYZElm)';
44  % Se refresca el residuo.
45          Res = XYZ - TrialXYZ;
46
47      end
48  end
```

Vuelve a aparecer en esta parte del código la función "ShapeFunctions", archivo MEX que se compila previamente para usarlo en "EvalLocCoords3D". Es un archivo muy importante para la mejora del coste en esta parte de la búsqueda de elementos candidatos, reduciendo en gran medida el tiempo usado para calcular

las funciones de forma de cada elemento.

En la Figura 32 se puede ver cómo usando esta función en lugar de su antecesora (CubeShapeFunctions.m, el script hecho en MATLAB), se reduce en un 54 % el tiempo de cálculo de la propia función "EvalLocCoords3D":

EvalLocCoords3D	144312	6.188 s	2.713 s	
CubeShapeFunctions	387078	3.159 s	3.159 s	

(a) EvalLocCoords3D con CubeShapeFunctions.m

EvalLocCoords3D	144312	3.346 s	2.780 s	
NodeRad	1	1.440 s	1.157 s	
ismember	6700	0.283 s	0.022 s	
ShapeFunctions (MEX-file)	387078	0.276 s	0.276 s	

(b) EvalLocCoords3D con ShapeFunctions.c

Figura 32: Comparación del coste computacional para el script en MATLAB y su versión MEX.

Cuando se analiza el tiempo para cada línea, se observa que la función consume mucho menos que usada en su forma de MATLAB. Mientras que en la figura 33 donde "CubeShapeFunctions" constituía el 56,7 % del tiempo de "EvalLocCoords3D", se ve que en su forma MEX es poco más del 20 % (20,4 %).

Esta función condensa muchas de las mejoras realizadas al código y no deben pasar desapercibidas las funciones "Jacobian" e "Inverse", que también son archivos MEX. Al analizar el rendimiento de "EvalLocCoords3D", los procesos que más consumen son el ya resuelto "CubeShapeFunctions", el cálculo del Jacobiano y el cálculo de la iteración de Newton-Raphson.

El cálculo del Jacobiano estaba organizado en una línea para cada posición de la matriz, de la forma:

```
15      J(1,1) = dNdPsi * XYZElm(:,1);
16      J(1,2) = dNdEta * XYZElm(:,1);
17      J(1,3) = dNdTau * XYZElm(:,1);
18      J(2,1) = dNdPsi * XYZElm(:,2);
19      J(2,2) = dNdEta * XYZElm(:,2);
20      J(2,3) = dNdTau * XYZElm(:,2);
21      J(3,1) = dNdPsi * XYZElm(:,3);
22      J(3,2) = dNdEta * XYZElm(:,3);
23      J(3,3) = dNdTau * XYZElm(:,3);
```

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
56	[dN] = CubeShapeFunctions(PsiE...	193538	3.043 s	49.2%	
54	PsiEtaTau = PsiEtaTau + round(...	193538	0.653 s	10.5%	
19	J = (Jacobian(dNdPsi, dNdEta, ...	193538	0.630 s	10.2%	
57	[N] = CubeShapeFunctions(PsiEt...	193538	0.465 s	7.5%	
30	I = (Inverse(J(1,:), J(2,:), J...	193538	0.454 s	7.3%	
All other lines			0.943 s	15.2%	
Totals			6.188 s	100%	

(a) CubeShapeFunctions.m

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
54	PsiEtaTau = PsiEtaTau + round(...	193538	0.640 s	19.1%	
19	J = (Jacobian(dNdPsi, dNdEta, ...	193538	0.592 s	17.7%	
30	I = (Inverse(J(1,:), J(2,:), J...	193538	0.455 s	13.6%	
56	[dN] = ShapeFunctions(PsiEtaTa...	193538	0.387 s	11.6%	
57	[N] = ShapeFunctions(PsiEtaTau...	193538	0.294 s	8.8%	
All other lines			0.978 s	29.2%	
Totals			3.346 s	100%	

(b) ShapeFunctions.c (MEX)

Figura 33: Comparación del coste computacional para el script en MATLAB y su versión MEX.

De esta forma se consigue elaborar el Jacobiano pero empeora mucho el rendimiento del código, sobretodo cuando se tiene en cuenta el gran número de iteraciones al que va a estar sujeto. Se crea entonces "Jacobian.c", compilado como MEX para usarlo en sustitución de esta función, obteniendo un resultado mucho mejor que en la anterior propuesta como se ve en la Figura 34. El tiempo de ejecución se reduce de 1,81 segundos, a 0,63 segundos.

```
0.38   213554   20    J(1,1) = dNdPsi * XYZElm(:,1);
0.28   213554   21    J(1,2) = dNdEta * XYZElm(:,1);
0.15   213554   22    J(1,3) = dNdTau * XYZElm(:,1);
0.42   213554   23    J(2,1) = dNdPsi * XYZElm(:,2);
0.12   213554   24    J(2,2) = dNdEta * XYZElm(:,2);
0.11   213554   25    J(2,3) = dNdTau * XYZElm(:,2);
0.14   213554   26    J(3,1) = dNdPsi * XYZElm(:,3);
0.11   213554   27    J(3,2) = dNdEta * XYZElm(:,3);
0.10   213554   28    J(3,3) = dNdTau * XYZElm(:,3);
```

(a) Primera propuesta para el cálculo del Jacobiano

```
0.63   193538   19    J = (Jacobian(dNdPsi, dNdEta, dNc
```

(b) Jacobian.c (MEX)

Figura 34: Comparación para el Jacobiano en MATLAB, y su MEX.

Para terminar con la descripción se debe presentar la última función MEX: "Inverse.c". Se crea para conseguir completar la mejora del tiempo en el cálculo del Newton-Raphson (Ecuación 9) en la línea 32 del código antes mostrado para "EvalLocCoords3D". La operación realizada es J\Res, que se desprende de la utilidad A\b característica de MATLAB (donde el resultado es la solución del sistema de ecuaciones Ax = b). Esta línea será, con esta operación, una de las más costosas del código por la cantidad de iteraciones que se realizarán. El coste computacional en toda la función será del 24,7 % y se muestra en la Figura 35:

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
55	`PsiEtaTau = PsiEtaTau + J\Res;`	204615	0.768 s	24.7%	
19	`J = (Jacobian(dNdPsi, dNdEta, ...`	204615	0.607 s	19.5%	
56	`[dN] = ShapeFunctions(PsiEtaTa...`	204615	0.424 s	13.6%	
57	`[N] = ShapeFunctions(PsiEtaTau...`	204615	0.306 s	9.8%	
77	`end`	204615	0.179 s	5.8%	
All other lines			0.824 s	26.5%	
Totals			3.108 s	100%	

Figura 35: Coste computacional de la operación J\Res en "EvalLocCoords3D".

Como el proceso A\b es como resolver la operación "$A \cdot x = b$", se puede resolver obteniendo "x" de la forma "$x = A^{-1} \cdot b$". Por lo tanto, J\Res se puede cambiar por "$J^{-1} \cdot Res$" y en lugar de utilizar "inv(J)" que ya está implementado en MATLAB (y se demuestra en la Figura 36 que es mucho menos eficiente y se indentifica "$J^{-1} \cdot Res$" como "$I \cdot Res$"), se elabora un archivo "Inverse.c" con el que hacer la inversa del Jacobiano y multiplicarla al residuo "Res".

```
-----
Tiempo A\b: 9.7004e-05 s

Tiempo inv(J)Res: 7.0849e-05 s

Tiempo I*Res: 2.0857e-05 s
-----
```

Figura 36: Coste computacional para cada forma del proceso en un cálculo individual.

Haciendo esto, se consigue mejorar el tiempo de ejecución de la iteración por Newton-Raphson como se muestra en la Figura 37. Manteniendo el cálculo de esta iteración en la primera posición, pero con un porcentaje respecto de toda la función del 19,1 %. Se puede ver que el tiempo total se mantiene similar, pero también se debe puntualizar que en ambos el cálculo del Jacobiano (que ya se ha demostrado muy costoso) está realizado con el archivo MEX. Por lo tanto, y aunque recaiga en gran medida en el Jacobiano, el conjunto de cambios consiguen optimizar el coste de la función "EvalLocCoords3D".

Así termina "EvalLocCoords3D.m" dando como "output" el valor del vector "PsiEtaTau" con la información de las coordenadas locales de los puntos de contacto, para cada elemento estudiado. Esta información será utilizada para conocer cuál de los elementos candidatos elegidos previamente es el correcto, ya que solo existirá un elemento candidato que reúna en todas sus coordenadas locales valores menores de 1 y mayores de -1.

```
⋮
41          if ~any(abs(PosL) > 1)
42              E(i) = Candidate(j);
43              break
44          end
⋮
```

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
54	PsiEtaTau = PsiEtaTau + round(...	193538	0.633 s	19.1%	
19	J = (Jacobian(dNdPsi, dNdEta, ...	193538	0.582 s	17.5%	
30	I = (Inverse(J(1,:), J(2,:), J...	193538	0.452 s	13.6%	
56	[dN] = ShapeFunctions(PsiEtaTa...	193538	0.381 s	11.5%	
57	[N] = ShapeFunctions(PsiEtaTau...	193538	0.294 s	8.9%	
All other lines			0.976 s	29.4%	
Totals			3.317 s	100%	

Figura 37: Coste final de la función "EvalLocCoords3D" con todos los cambios.

Así volviendo a "whereami.m" se cierra el bucle encargado de moverse entre los candidatos. Esta parte del código tendrá en conjunto el siguiente aspecto:

```
⋮
31 parfor i = 1:size(XYZ,2)
32     Candidate = cell2mat(Elm(i));
33
34     for j = 1:size(Candidate, 2)
35         XYZElm = XYZ_N(:, Mesh(iMesh, iBody).Element.Topology(:,
               Candidate(j)));
36         PosL = EvalLocCoords3D(XYZElm', XYZ(:, i), N, dNdPsi,
               dNdEta, dNdTau);
37
38         if ~any(abs(PosL) > 1)
39             E(i) = Candidate(j);
40             break
41         end
42     end
43 end
```

Lo único que queda es mostrar el tiempo y estos resultados, lo cual se convierte en una tarea sencilla comparado a los procesos anteriores:

```
⋮
Time = toc(TStart);
profile off
% Muestra el rendimiento del código.
profile viewer
% El tiempo de ejecución es mostrado.
disp(['Tiempo: ' num2str(Time) 's']);

Plot_Mesh(1,1,'Active',1,0,'b',0) % ''Dibuja'' los elementos que se
     están analizando.
hold on % Mantiene el ''Dibujo'' 3D.
Plot_Mesh(1,1,E(E~=0),0,0.5,'y',0)
scatter3(Xp,Yp,Zp,'filled') % ''Dibuja'' los puntos que se han
    probado.

hold off
end
```

Para ”dibujar” la malla utilizada y llevar a cabo la visualización de los elementos es necesario hacer uso de ”Plot_Mesh”, la función encargada de mostrar esa información.

En el apartado de ensayos, con los ejemplos, será de gran utilidad ya que los ”inputs” que recibe permiten modificar la forma de visualización así como hacer uso del sólido que se vaya a analizar en el momento. Por ejemplo, este caso hará un ”dibujo” de los elementos en contacto en amarillo añadiendo la condicion $\mathbf{E(E = 0)}$ como ”input” de los elementos a mostrar.

Gracias a lo obtenido en este último paso (”profile off” y el ”dibujo”) se podrán ver los resultados obtenidos del programa. Estos datos que se mostrarán son los que se van a analizar en el siguiente apartado ”Ejemplos de aplicación”.

3. Ejemplos de aplicación

Ahora que ya se ha presentado el programa, su funcionamiento en relación con la resolución de un problema de Elementos Finitos y las ventajas que conlleva; es hora de mostrar la resolución de estos mismos problemas. Se debería explicar por tanto en qué va a consistir este ejemplo de aplicación:

- **Ensayo de contacto**

  Basándose en las mallas de ejemplo usadas en el desarrollo del programa, se usarán como se ha hecho hasta ahora en las pruebas de contacto, solo que también dentro de un análisis de elementos finitos.

  Se trata de una esfera en contacto con un bloque prismático, en los que a ambos se les ha realizado un mallado siguiendo el procedimiento del cgFEM. Estos se encuentran claramente en contacto e incluso existe un desplazamiento de la esfera con respecto al bloque.

3.1. Ensayo de contacto

Buscando aunar lo ya introducido en el apartado 1.3.1 describiendo las características del análisis, aquí se pondrán de manifiesto los resultados obtenidos, así como los pasos seguidos para obtener las soluciones.

Como ya se sabe, los puntos de contacto del ensayo a llevar a cabo se encuentran cuantificados así como los elementos del análisis por cgFEM tal y como se mostraba. Por tanto, son muy sencillos de volcar en MATLAB como se veía en el apartado anterior a partir del análisis de las líneas de código en "whereami.m".

Para comenzar con el ensayo se deben cargar los puntos de contacto correspondientes a la esfera que entrará en contacto con el prisma. Los puntos de contacto se organizarán en una matriz, pudiendo navegar entre ellos de la forma que se ha hecho en la explicación anterior.

Esta matriz será uno de los "inputs" de la función "whereami", junto con los valores para "iMesh" e "iBody" (ambos "1"). A su vez, estos mismos nodos son los que se usarán como referencia para obtener esos cálculos de coordenadas locales dentro de los elementos. Por lo que son imprescindibles estos valores.

Una vez lanzado el comando **E = whereami(XYZPoints', 1, 1);** (donde XYZPoints será simplemente el nombre que tomará la matriz de puntos de contacto) MATLAB empezará a calcular. El comando se organiza de esta forma para poder llamar a la función "whereami", guardando el valor que se obtenga (los elementos en contacto) en una variable "E". El resultado será el mostrado en la Figura 38.

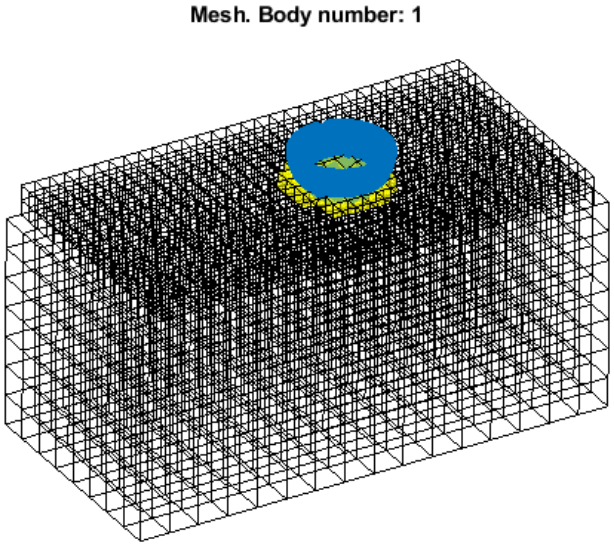


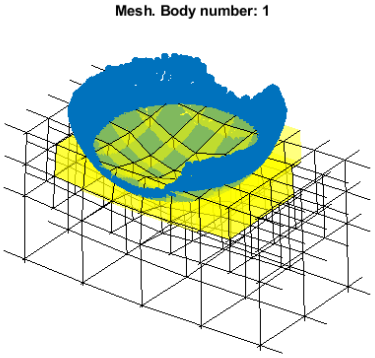
Figura 38: Resultados de lanzar la función "whereami" para la malla y los puntos de contacto.

Destacan los elementos en amarillo como elementos en contacto teniendo en cuenta los datos que se recibían de la malla del prisma y de los puntos de la esfera. También se puede observar la ligera deformación en la malla del prisma, ese "XYZ_N" que incluía las deformaciones y que se nombraban al analizar "whereami.m".

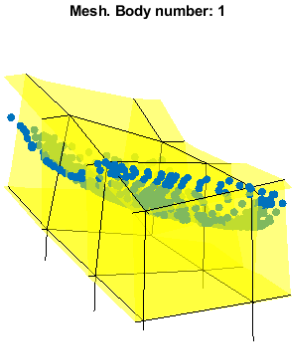
Al hacer zoom sobre los resultados (Figura 39) solo se corroboran más los resultados obtenidos como válidos, pudiéndose observar también el gran volumen de puntos que la esfera tiene, difíciles de distinguir a simple vista y que confirman los datos que se mostraban al leer el "profiler" y ver las pasadas que se hacían sobre cada elemento.

Una vez pasada esta información y realizado el Análisis de Elementos Finitos, el resultado es satisfactorio. En la Figura 40, se muestra una sección de la esfera en contacto con el prisma; junto a los valores para el "gap" (introducido en el apartado 1.2.3). El valor es negativo para la zona también obtenida en la búsqueda de candidatos en "whereami.m", confirmando la utilidad de la mejora, al obtener un valor idéntico en un caso de contacto pero con un proceso optimizado.

Además se comprueba cómo existe una mejora en la eficiencia computacional en la Figura 41, donde se compara el tiempo de ejecución del programa antes y después de las mejoras llevadas a cabo. Se trata de una sustancial mejora, obteniendo un modelo capaz de calcular este resultado 8,41 segundos más rápido.



(a) La zona de contacto.



(b) Detalle de algunos elementos con nodos de la esfera en su interior.

Figura 39: Detalles de la zona de contacto entre la esfera y el prisma.

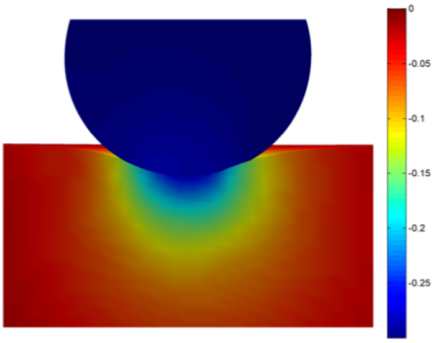


Figura 40: Resultado del Análisis de Elementos Finitos para el caso de contacto.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
whereami	1	12.594 s	0.007 s	
parallel_function	2	12.581 s	3.733 s	
EvalLocCoords3D	144312	8.308 s	6.588 s	
NodeRad	1	2.277 s	0.002 s	
ShapeFunctions (MEX-file)	385822	0.830 s	0.830 s	
Jacobian (MEX-file)	192910	0.533 s	0.533 s	
ismember	6700	0.483 s	0.046 s	
ismember>ismemberR2012a	6700	0.437 s	0.052 s	
ismember>ismemberBuiltinTypes	6700	0.385 s	0.385 s	
Inverse (MEX-file)	192910	0.355 s	0.355 s	
cell2mat	6700	0.054 s	0.054 s	
parfor_sliced_fcnhdl_check	2	0.002 s	0.002 s	
onCleanup>onCleanup.delete	2	0.001 s	0.001 s	
parallel_function>results	2	0.001 s	0.001 s	
incrementParallelFunctionDepth	4	0.001 s	0.001 s	
parfor_endpoint_check	4	0.001 s	0.001 s	
deal	1	0.001 s	0.001 s	
...al.incrementParallelFunctionDepth(-1)	2	0.001 s	0.000 s	
parallel_function>PCTInstalled	2	0.000 s	0.000 s	
onCleanup>onCleanup.onCleanup	2	0.000 s	0.000 s	

(a) "whereami" optimizado con las medidas descritas

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
whereami	1	21.364 s	2.083 s	
EvalLocCoords3D	144312	16.767 s	9.424 s	
CubeShapeFunctions	447314	7.348 s	7.348 s	
NodeRad	1	2.443 s	1.903 s	
ismember	6700	0.539 s	0.047 s	
ismember>ismemberR2012a	6700	0.492 s	0.055 s	
ismember>ismemberBuiltinTypes	6700	0.437 s	0.437 s	
cell2mat	6700	0.066 s	0.066 s	
deal	1	0.000 s	0.000 s	

(b) Propuesta inicial de "whereami"

Figura 41: Comparación del coste computacional para "whereami" antes y después de las mejoras.

4. Conclusión

A lo largo del proyecto se han descrito los detalles que influyen en las decisiones a tomar, así como en las características que tendrá la solución y el proceso para obtenerla. Primero, se ha presentado una descripción de la necesidad de usar el Método de Elementos Finitos en ingeniería. Además se han enumerado ciertas aplicaciones del método haciendo hincapié en el problema de contacto ya que es el objetivo de este trabajo.

Esto ha sido necesario para llegar a la explicación del método usado por la aplicación FEAVOX para el análisis: cgFEM. Un enfoque diferente para el MEF y que por sus propiedades se aporta como solución al increíble coste computacional en esta clase de problemas donde hay contacto. Es entonces cuando se ha podido introducir un ejemplo que pueda someterse a análisis y que contiene una esfera en contacto con la superfície de un poliedro.

Se han descrito los inconvenientes del MEF en problemas de contacto y se ha propuesto el uso del método cgFEM que palia parte de dichos problemas. No obstante existen ciertas partes de la implementación susceptibles a un incremento en su eficiencia computacional.

Haciendo uso de los beneficios de cgFEM, se implementa un método robusto capaz de obtener los resultados de contacto. Tras realizar una búsqueda de estos puntos, se obtienen de forma precisa los elementos con los que se encuentran en contacto, a partir de sus coordenadas locales.

El coste de la implementación puede mejorarse usando las propias herramientas que el programa en que está creado, MATLAB, ofrece. Y donde si se analiza el antes y el después del código con las mejoras, se puede observar una mejora general del desempeño del programa. El tiempo gastado en realizar las operaciones es menor gracias a la implementación de la computación paralela para las iteraciones, de la vectorización de los cálculos (datos organizados en vectores) y el uso de MEX para funciones de un coste demasiado alto.

Una vez aplicadas las mejoras al código se ha comprobado la validez de la búsqueda de puntos de contacto para poder llevarlo a un caso particular. Al someter el ejemplo de aplicación propuesto a análisis mediante cgFEM y junto con las mejoras propuestas, se obtiene un resultado válido y de las características esperadas, acompañado de un resultado positivo en el gasto computacional respecto de las versiones anteriores.

En este sentido, en el trabajo se propone una mejora en la eficiencia computacional a través de la mejora de los algoritmos de búsqueda de puntos de contacto. Los resultados han mostrado una gran robustez y además un incremento de la eficiencia computacional de un 169 %.

5. Bibliografía

Referencias

[1] Navarro-Jiménez, José Manuel; Tur, Manuel; Albelda, José; Ródenas, Juan José (2017) *Large deformation frictional contact analysis with immersed boundary method.* Valencia, España.
https://doi.org/10.1007/s00466-017-1533-x

[2] Navarro-Jiménez, José Manuel (Junio, 2019) *Contact problem modelling with the Cartesian grid Finite Element Method.* Valencia, España.

[3] Fuenmayor Fernández, Javier; Ródenas García, Juan José; Tarancón Caro, José Enrique; Tur Valiente, Manuel; Vercher Martínez, Ana (2017) *Apuntes de la asignatura de "Tecnologías computacionales para la Ingeniería Mecánica".* Valencia, España.

[4] Nadal, Enrique (2014) *Cartesian grid FEM (cgFEM): High performance h adaptive FE analysis with efficient error control. Application to structural shape optimization.* Valencia, España.

6. Presupuesto

A continuación se describirá el coste económico que este trabajo ocasionaría, enumerando los distintos recursos empleados y el precio de cada uno.

Entre los recursos que entran dentro del desarrollo de este trabajo se encuentran desde herramientas informáticas como MATLAB, hasta gastos para contratar un ingeniero que lleve a cabo la programación y el correspondiente análisis de los datos.

Al tratarse de un trabajo compuesto de una gran cantidad de recursos de licencia libre, aunque existan una serie de recursos estos serán de coste cero.

Además, hay que tener en cuenta el tiempo empleado para la realización de este Trabajo de Fin de Grado. Se estima una cantidad de 300 horas, cantidad correspondiente al desarrollo del programa y los análisis, así como el desarrollo de este documento.

Se describen a continuación los recursos que se han empleado en la realización de este trabajo:

- **Ordenador de sobremesa**
- **MATLAB 2018a**
- **ANSYS Workbench**
- **Dev-C++**
- **SOLIDWORKS 2018**
- **LaTeX**
- **Servicio de reprografía**

Teniendo en cuenta que las licencias se consideran de forma anual, así como el equipo de sobremesa, se ha calculado tomando el coste dividiéndolo entre las horas de un año laboral (1760 horas). Ello multiplicado posteriormente por las horas dedicadas al proyecto.

El coste de reprografía se considera fijo, basándose en el coste de la impresión de este documento y entrega de los elementos típicos que se deben preparar para la defensa.

A continuación se describen los costes ocasionados por los recursos utilizados:

Desglose				
Nombre	**Precio anual**	**Precio/hora**	**Nº de horas**	**Precio total**
Ordenador de sobremesa	600€	0,34€/h	300	102€
MATLAB 2018a	800€	0,45€/h	137	61,65€
ANSYS 2019 R2	8300€	4,72€/h	5	23,60€
Dev-C++	0€	0€/h	29	0€
SOLIDWORKS 2018	8100€	4,60€/h	6	27,60€
LaTeX	0€	0€/h	123	0€
Servicio de reprografía	-	-	-	20€
			Total	**234,85€**

Cuadro 1: Desglose del importe de los recursos.

Como se adelantaba, esta tarea debe ser llevada a cabo por un Ingeniero Mecánico. Se deben añadir por tanto los honorarios del ingeniero a cargo del proyecto.

En cuanto al ingeniero, se tomará un valor de 25€/hora. El importe total será:

Desglose			
Nombre	**Precio/hora**	Nº de horas	Precio total
Ingeniero Mecánico	25€/h	270	6750€
José Manuel Navarro		20	500€
Enrique Nadal		10	250€
		Total	7500€

Cuadro 2: Desglose del importe de los costes en personal.

Se puede concluir por tanto con una tabla resumen de todos los costes. Aquí se añadirá el IVA del 21 %. Esta quedará:

Resumen	
Nombre	**Precio total**
Costes recursos	234.85€
Costes personal	7500€
Subtotal	**7734,85€**
IVA 21 %	1624,32€
Total	**9359,17€**

Cuadro 3: Resumen costes.

Se puede por tanto concluir que el coste total del proyecto sería la suma de:

NUEVE MIL TRESCIENTOS CINCUENTA Y NUEVE EUROS CON DIECISIETE CÉNTIMOS

7. Pliego de condiciones

Este proyecto, así como cualquiera que aspire a ser considerado de forma oficial lleva consigo una normativa.

En lo que respecta al estudio del Métodos de Elementos Finitos no parece existir una normativa aplicable a este trabajo.

Sin embargo, se puede considerar la Ley de Propiedad Intelectual como legislación a aplicar. Se desprende de esta que el contenido del trabajo debe ser original y de libre uso.

La Ley de Propiedad Intelectual vigente se puede encontrar en el BOE (Boletín Oficial del Estado) con la referencia "BOE-A-1996-8930", aprobada por Real Decreto Legislativo el 12 de abril de 1996. En caso de que la información tuviera Derechos de Autor, se describe en el Artículo 32 la excepción de Citas y reseñas usadas para casos de ilustración de la educación o investigaciones científicas.

En cuanto a la normativa propia del proyecto se puede enumerar la que rodea al propio Trabajo de Fin de Grado, que regula las reglas de redacción, presentación y posterior calificación de este proyecto.

Esta normativa se encuentra descrita en el documento titulado "NORMATIVA MARCO DE TRABAJO FIN DE GRADO Y FIN DE MÁSTER DE LA UNIVERSITAT POLITÈCNICA DE VALÈNCIA"del Boletín oficial de la Universidad Politécnica de Valencia, accesible desde la propia página de la ETSID. Documento con las especificaciones que harán válido un trabajo de estas características y cuyo documento fue aprobado el 7 de marzo de 2013, con su última modificación el 13 de marzo de 2018.

8. Anexos

8.1. whereami.m

```
% Authors:
%
% * Adrián Arenas Martínez (<adarmar4@etsid.upv.es>)
%
% Date: June 2019
function E = whereami(XYZ,iMesh,iBody)

load Mesh2.mat; %loads the global variable ''Mesh''
move = [0.03, 0, 0];
XYZ_N = Mesh(iMesh,iBody).Node.XYZ + Results(iMesh,iBody).Node
    (:,1:3)';
%The ''Mesh'' global variable contains all the elements' info
radius = Radius(XYZ\_N); % Dependerá del tamano máximo del elemento
     de la malla. Distancia de la diagonal máxima de la malla.

Xp = XYZ(1,:); %X coordinates of the point/s we specify
Yp = XYZ(2,:); %Y coordinates of the point/s we specify
Zp = XYZ(3,:); %Z coordinates of the point/s we specify

%DISTANCE TO EVERY ELEMENT
%
    ------------------------------------------------------------------------

E = zeros(1, size(XYZ,2));

TStart = tic;
profile on
% Chooses candidate elements for each point
Elm = NodeRad(XYZ,XYZ_N,Mesh,radius,iMesh,iBody);

[dN] = ShapeFunctions(0, 0, 0, 1, 1);
[N] = ShapeFunctions(0, 0, 0, 1, 0)';
[dNdPsi, dNdEta, dNdTau] = deal(dN(1,:), dN(2,:), dN(3,:));

parfor i = 1:size(XYZ,2)
    Candidate = cell2mat(Elm(i));

    for j = 1:size(Candidate,2)
        XYZElm = XYZ_N(:,Mesh(iMesh,iBody).Element.Topology(:,
            Candidate(j)));
        PosL = EvalLocCoords3D(XYZElm',XYZ(:,i),N,dNdPsi,dNdEta,
            dNdTau);

        if ~any(abs(PosL) > 1)
            E(i) = Candidate(j);
            break
```

```
        end
    end
end

Time = toc(TStart);
profile off
profile viewer
disp(['Tiempo: ' num2str(Time) 's']);

Plot_Mesh(1,1,'Active',1,0,'b',0) %plots the elements we're
    analyzing
hold on %holds the 3D plot
Plot_Mesh(1,1,E(E~=0),0,0.5,'y',0)
scatter3(Xp,Yp,Zp,'filled') %plots the points we're testing

%Limits of Mesh
Limits = meshdim(Mesh,iMesh,iBody);
scatter3(Limits(1,1),Limits(2,1),Limits(3,1));
scatter3(Limits(1,2),Limits(2,2),Limits(3,2));

hold off
end
```

8.2. Radius.m

```
% Authors:
%
% * Adrián Arenas Martínez (<adarmar4@etsid.upv.es>)
%
% Date: July 2018
function radius = Radius(xyzN)
    %Counts the number of nodes.
    Node_Dim = size(xyzN,2);

    for i = 1:Node_Dim
        %Firstly, every node in the mesh is substracted to every
            other node in
        %the mesh but itself.
        %Secondly, the vector modulus is obtained in order to get a
        %node-to-node distance.
        NodeDiff = vecnorm(bsxfun(@minus,(xyzN(:,i)),...
                    (xyzN(:,1:end ~= i))));
        %The minimum distance is extracted as the minimum node-to-
            node
        %radius that can be achieved inside the mesh, hence the
            maximum
        %distance a point can be away from a node but inside an
            element.
        radius = min(NodeDiff);
    end
end
```

8.3. NodeRad.m

```
%Authors:
%
% * Adrián Arenas Martínez (<adarmar4@etsid.upv.es>)
%
%Date: August 2018
function Candidates = NodeRad(XYZ,XYZ_N,Mesh,radius,iMesh,iBody)
    XYZ_Dim = size(XYZ,2); %Counts the number of samples

    parfor i = 1:XYZ_Dim
        %Firstly, every node in the mesh is substracted to every
            samples's
        %coordinates.
        %Secondly, the vector modulus of the result is calculated
            in order
        %to get a distance value.
        %Every point-to-node distance is compared to the minimum
            radius
        %value (minimum node-to-node value) in the mesh.
        %Then, stores every column number with a data different
            from 0
        %(node inside radius).

        NodeCell = find((vecnorm(bsxfun(@minus, XYZ(:,i),...
            (XYZ_N))) <= radius) ~= 0);

        %Every element is checked looking for a suitable node
            inside them,
        %making it a 1 where a node is found, and 0 where it is not
            .
        %The vector modulus of the columns with the nodes checked
            is
        %obtained. This helps avoiding not suitable elements to be
            chosen
        %as candidates, only taking elements with a non-zero result
            .
        %Elements matching the requirements (any node inside == non
            -zero
        %result) are stored as potential candidates.

        Candidates{i} = find(vecnorm(double(ismember(Mesh(iMesh,...
            iBody).Element.Topology,NodeCell))) ~= 0);
    end
end
```

8.4. EvalLocCoords3D.m

```
%Authors:
%
% * Adrián Arenas Martínez (<adarmar4@etsid.upv.es>)
%
%Date: June 2019
function [PsiEtaTau] = EvalLocCoords3D(XYZElm,XYZ,N,dNdPsi,dNdEta,
    dNdTau)

FEDegree = 1;

PsiEtaTau = [0;0;0];
Tol = 10e-2;
NIter = 0;
TrialXYZ = (N * XYZElm)';
Res = XYZ - TrialXYZ;

while norm(Res) > Tol
    NIter = NIter + 1;
    J = (Jacobian(dNdPsi, dNdEta, dNdTau, XYZElm(:,1), XYZElm(:,2),
         XYZElm(:,3)))';
    I = (Inverse(J(1,:), J(2,:), J(3,:)))';

    PsiEtaTau = PsiEtaTau + round(I*Res, 14);

    [dN] = ShapeFunctions(PsiEtaTau(1,:)',PsiEtaTau(2,:)',PsiEtaTau
        (3,:)',FEDegree,1);
    [N] = ShapeFunctions(PsiEtaTau(1,:)',PsiEtaTau(2,:)',PsiEtaTau
        (3,:)',FEDegree,0)';

    dNdPsi = dN(1,:,:);
    dNdEta = dN(2,:,:);
    dNdTau = dN(3,:,:);

    TrialXYZ = (N * XYZElm)';
    Res = XYZ - TrialXYZ;
end
```

8.5. Jacobian.c

```
#include mex.h

void Jacobian(double dNdPsi[], double dNdEta[], double dNdTau[],
    double XYZElmpsi[], double XYZElmeta[], double XYZElmtau[],
    double J[][3]) {
        size_t rows = 8;
    int i, j, k, x;
    double B = 0, C = 0;
    double XYZElm[8][3];

        for(i=0;i<3;i++) {
        for(j=0;j<3;j++) {
            J[i][j] = 0;
            XYZElm[i][j] = 0;
        }
    }
    for(i=0;i<8;i++) {
        for(j=0;j<3;j++) {
            switch(j) {
                case 0:
                    XYZElm[i][j] = XYZElmpsi[i];
                    break;
                case 1:
                    XYZElm[i][j] = XYZElmeta[i];
                    break;
                case 2:
                    XYZElm[i][j] = XYZElmtau[i];
                    break;
            }
        }
    }

        for(k=0;k<rows;k++) {
        J[0][0] = J[0][0] + dNdPsi[k] * XYZElm[k][0];
        J[0][1] = J[0][1] + dNdEta[k] * XYZElm[k][0];
        J[0][2] = J[0][2] + dNdTau[k] * XYZElm[k][0];
        J[1][0] = J[1][0] + dNdPsi[k] * XYZElm[k][1];
        J[1][1] = J[1][1] + dNdEta[k] * XYZElm[k][1];
        J[1][2] = J[1][2] + dNdTau[k] * XYZElm[k][1];
        J[2][0] = J[2][0] + dNdPsi[k] * XYZElm[k][2];
        J[2][1] = J[2][1] + dNdEta[k] * XYZElm[k][2];
        J[2][2] = J[2][2] + dNdTau[k] * XYZElm[k][2];
    }
}
// /* la llamada en Matlab debería ser así:
//  * J = Jacobian(dNdPsi, dNdEta, dNdTau, XYZElm); */

void mexFunction( int nlhs, mxArray *plhs[],
```

```
47                  int nrhs, const mxArray *prhs[] ) {
48
49      double *dNdPsi, *dNdEta, *dNdTau, *XYZElmpsi, *XYZElmeta, *
            XYZElmtau; //Input vectors and matrix.
50      double *J;

            //Output matrix.
51      size_t mrows, ncols;
52    /* Check for proper number of arguments. */
53    if(nrhs!=6) {
54      mexErrMsgIdAndTxt( MATLAB:sum_dif:invalidNumInputs,
55              Six inputs required.);
56    } else if(nlhs!=1) {
57      mexErrMsgIdAndTxt( MATLAB:sum_dif:maxlhs,
58              One output required.);
59    }
60    mrows = 3;
61    ncols = 3;
62
63     /* Assign pointers to each input and output. */
64    dNdPsi = mxGetPr(prhs[0]);
65    dNdEta = mxGetPr(prhs[1]);
66    dNdTau = mxGetPr(prhs[2]);
67
68    XYZElmpsi = mxGetPr(prhs[3]);
69    XYZElmeta = mxGetPr(prhs[4]);
70    XYZElmtau = mxGetPr(prhs[5]);
71
72     /* Create matrix for the return argument. */
73    plhs[0] = mxCreateDoubleMatrix((mwSize)mrows, (mwSize)ncols,
          mxREAL);
74
75    J = mxGetPr(plhs[0]);
76
77    Jacobian(dNdPsi, dNdEta, dNdTau, XYZElmpsi, XYZElmeta, XYZElmtau,
          J);
78  }
```

8.6. Inverse.c

```
#include mex.h

double Det(double N[3][3]) {
        double n;
    size_t rows = 3, cols = 3;

        n = N[0][0]*N[1][1]*N[2][2] + N[0][1]*N[1][2]*N[2][0] + N
            [1][0]*N[2][1]*N[0][2] - N[0][2]*N[1][1]*N[2][0] - N
            [0][1]*N[1][0]*N[2][2] - N[1][2]*N[2][1]*N[0][0];

        return n;
}

void Inverse(double Jacobian0[], double Jacobian1[], double
    Jacobian2[], double I[3][3]) {
    int i, j, a, b, s, n = 0;
    size_t rows = 3, cols = 3;
    double J[][3] = {
        {0, 0, 0},
        {0, 0, 0},
        {0, 0, 0}
    };
    double T[][3] = {
        {0, 0, 0},
        {0, 0, 0},
        {0, 0, 0}
    };
    double A[][3] = {
        {0, 0, 0},
        {0, 0, 0},
        {0, 0, 0}
    };

    double D;
    double Tx[3], Ty[3], Tz[3];

        for(i = 0; i < rows; i++) {
                for(j = 0; j < cols; j++) {
                        switch(i) {
                                case 0:
                                        J[i][j] = Jacobian0[j];
                                        break;
                                case 1:
                                        J[i][j] = Jacobian1[j];
                                        break;
                                case 2:
                                        J[i][j] = Jacobian2[j];
                                        break;
```

```
46              }
47                  }
48          }
49
50      //Determinante
51      D = Det(J);
52
53      //Adjunta
54      for(i = 0; i < rows; i++) {
55                  for(j = 0; j < cols; j++) {
56              if((n % 2) > 0) {
57                  s = -1;
58              } else {
59                  s = 1;
60              }
61                          switch(j) {
62                                  case 0:
63                                          a = j + 1;
64                                          b = j + 2;
65                                          break;
66                                  case 1:
67                                          a = j - 1;
68                                          b = j + 1;
69                                          break;
70                                  case 2:
71                                          a = j - 2;
72                                          b = j - 1;
73                                          break;
74                          }
75                          switch(i) {
76                                  case 0:
77                                          A[i][j] = s*(J[i+1][a]*J[i
                                                +2][b] - J[i+1][b]*J[i
                                                +2][a]);
78                                          break;
79                                  case 1:
80                                          A[i][j] = s*(J[i-1][a]*J[i
                                                +1][b] - J[i-1][b]*J[i
                                                +1][a]);
81                                          break;
82                                  case 2:
83                                          A[i][j] = s*(J[i-2][a]*J[i
                                                -1][b] - J[i-2][b]*J[i
                                                -1][a]);
84                                          break;
85                          }
86              n++;
87                  }
88          }
89
```

```
90      //Traspuesta
91      for(i = 0; i < rows; i++) {
92                  for(j = 0; j < cols; j++) {
93                          if(i == j) {
94                                  T[i][j] = A[i][j];
95              } else {
96                                  T[i][j] = A[j][i];
97                          }
98                  }
99          }
100
101     for(i = 0; i < rows; i++) {
102         for(j = 0; j < cols; j++) {
103             I[i][j] = (1/D)*T[i][j];
104         }
105     }
106 }
107 /* la llamada en Matlab debería ser así:
108  * I = Inverse(Jacobian0, Jacobian1, Jacobian2); */
109
110 void mexFunction( int nlhs, mxArray *plhs[],
111                   int nrhs, const mxArray *prhs[] ) {
112
113     double *Jacobian0, *Jacobian1, *Jacobian2; //Input vectors and
            matrix.
114     double *I;                                  //Output matrix.
115     size_t mrows, ncols;
116
117   /* Check for proper number of arguments. */
118   if(nrhs!=3) {
119     mexErrMsgIdAndTxt( MATLAB:sum_dif:invalidNumInputs,
120             Six inputs required.);
121   } else if(nlhs!=1) {
122     mexErrMsgIdAndTxt( MATLAB:sum_dif:maxlhs,
123             One output required.);
124   }
125   mrows = 3;
126   ncols = 3;
127
128    /* Assign pointers to each input and output. */
129   Jacobian0 = mxGetPr(prhs[0]);
130   Jacobian1 = mxGetPr(prhs[1]);
131   Jacobian2 = mxGetPr(prhs[2]);
132
133    /* Create matrix for the return argument. */
134   plhs[0] = mxCreateDoubleMatrix((mwSize)mrows, (mwSize)ncols,
          mxREAL);
135
136   I = mxGetPr(plhs[0]);
137
```

```
138   Inverse(Jacobian0, Jacobian1, Jacobian2, I);
139 }
```