Asociación Argentina de Mecánica Computacional

Mecánica Computacional Vol XXVI, pp.1724-1740
Sergio A. Elaskar, Elvio A. Pilotta, Germán A. Torres (Eds.)
Córdoba, Argentina, Octubre 2007

DISEÑO FLEXIBLE Y MODULAR DE MODELOS NUMÉRICOS BASADOS EN ELEMENTOS FINITOS

Javier Quinteros[a,b], Pablo M. Jacovkis[b,c] y Víctor A. Ramos[a]

[a]*Laboratorio de Tectónica Andina - Depto. de Geología, FCEN, Universidad Nacional de Buenos Aires, Intendente Güiraldes 2160 - Ciudad Universitaria, C1428EGA, Buenos Aires, Argentina*
[b]*Departamento de Computación, FCEN, Universidad Nacional de Buenos Aires, Intendente Güiraldes 2160 - Ciudad Universitaria, C1428EGA, Buenos Aires, Argentina*
[c]*Facultad de Ingeniería, Universidad Nacional de Buenos Aires, Paseo Colón 850, C1063ACV, Buenos Aires, Argentina*

Palabras clave: Diseño de software, elementos finitos, abstracción elemental, implementación eficiente.

Resumen. El diseño de un modelo numérico atraviesa diversas etapas hasta su puesta en funcionamiento y la obtención de resultados satisfactorios. Para los modelos basados en Elementos Finitos, está claro que es necesaria una correcta formulación del problema para asegurar que, desde un punto de vista matemático, se obtendrá una solución. Nos referimos al planteo de las ecuaciones, su discretización y la elección del elemento, entre otras.

Sin embargo, la etapa de diseño no concluye en las formulaciones matemáticas, sino que incluye también el *diseño del software*. Una buena formulación matemática puede resultar en un programa de pobre rendimiento si no se llevan a cabo las optimizaciones necesarias al momento de la implementación.

Es común que muchos investigadores codifiquen desde cero para modelar los problemas a resolver. Entre los efectos inmediatos de esta práctica notamos: pérdida de tiempo en la codificación, falta de generalidad en el planteo, repetición de errores comunes y desarrollo de código no modular, que dudosamente pueda reutilizarse para problemas similares. Asimismo, cualquier corrección a la formulación matemática será casi imposible una vez que el código se haya finalizado.

En este trabajo presentamos el diseño y la implementación de software totalmente modular y flexible para la implementación de modelos basados en Elementos Finitos. Su potencialidad está basada en una correcta modularización, donde para codificar la resolución de los problemas no es necesario conocer la formulación elemental subyacente. Tampoco es necesario conocer en el nivel de la formulación la implementación estrictamente computacional de operaciones matemáticas.

Este aislamiento en tres (o más) capas permite que se reutilicen grandes porciones de código para la resolución de problemas que pueden ser totalmente distintos, o incluso cambiar los componentes de las distintas capas sin variar el planteo del problema.

Damos un análisis detallado del diagrama de clases utilizado, su justificación y detalles de su implementación. Presentamos también un ejemplo real de un cambio de elemento utilizado y cómo es imperceptible al momento de codificar la resolución del problema, mediante el apropiado uso de interfaces definidas adecuadamente.

Copyright @ 2007 Asociación Argentina de Mecánica Computacional
http://www.amcaonline.org.ar

1. OBJETIVOS

Siempre que debe encararse la resolución mediante modelos numéricos de algún problema (clásico o no) se trata de alcanzar la solución buscada minimizando los costos. Cómo se contabiliza este costo es la pregunta para la cual no siempre existe un consenso. Varios factores deben ser ponderados, como por ejemplo: el tiempo de programación, el de validación y el tiempo necesario para correr los modelos, entre otros.

En los casos en que el problema a resolver es sencillo, no suele dedicársele el tiempo suficiente a la etapa de diseño. Los errores aparecen cuando el problema simple constituye el primer paso hacia la resolución de un problema mucho más complejo del que quizás no existan soluciones conocidas para poder validar el modelo.

En estos casos, el contar con módulos correctamente validados que puedan ser reutilizados permite que el código sea extensible con cierto margen de confianza a problemas sin soluciones analíticas conocidas. Si hubiera que volver a codificar formulaciones ya validadas, se corre el riesgo de introducir errores que difícilmente puedan ser detectados en etapas posteriores.

Si bien en un principio nos orientamos a la resolución de ciertos problemas puntuales, la realidad es que existe un alto porcentaje de pasos que se repiten en muchos de los problemas.

Debido a esto, resolvimos llevar adelante un diseño modular, flexible, altamente optimizado en cuanto al uso de recursos y que permita la reutilización de código para resolver otro tipo de problemas.

Existen antecedentes de valiosos trabajos durante los últimos años en torno a brindar herramientas y/o entornos de trabajo para que no sea necesaria una recodificación total cada vez que se quiere resolver otro problema. Podemos mencionar, entre ellos, la propuesta de Sonzogni et al. (2002), donde se brindan detalles de la implementación de diversos problemas de la mecánica del continuo en clusters Beowulf, o Cardona et al. (1994), que permite flexibilidad en la configuración del programa y los métodos de resolución mediante un poderoso intérprete de comandos, y también la propuesta de Dari et al. (1999). Este último propone un marco genérico de resolución a diversos problemas de la mecánica computacional, dejando al usuario la programación de los métodos de resolución sin preocuparse por la paralelización y otros detalles técnicos.

Nuestro trabajo intenta profundizar esta última idea definiendo interfaces entre las distintas componentes del modelo y permitiendo el reemplazo tanto de la formulación elemental, los métodos de resolución del sistema, o incluso del problema a resolver siempre y cuando se respete la arquitectura propuesta. La idea es no sólo proveer herramientas para la resolución de problemas de mecánica del continuo, sino un marco de trabajo que permita reutilizar las clases desarroladas mediante abstracciones apropiadas y permitir el continuo crecimiento de las opciones en cada nivel entre otras.

2. ESQUEMA GENERAL DEL DISEÑO PARA RESOLUCIÓN MEDIANTE FEM

2.1. Identificación de etapas y entidades

El método de los elementos finitos (FEM) es ampliamente utilizado para la resolución de ecuaciones diferenciales en derivadas parciales. Se sabe que, si bien existen varios tipos de elementos con características propias, la teoría sobre la que subyacen las resoluciones mediante FEM no depende de un elemento en particular. La técnica en sí es un marco teórico dentro del cual pueden variar múltiples aspectos particulares de la formulación.

A grandes rasgos pueden identificarse distintas etapas como:

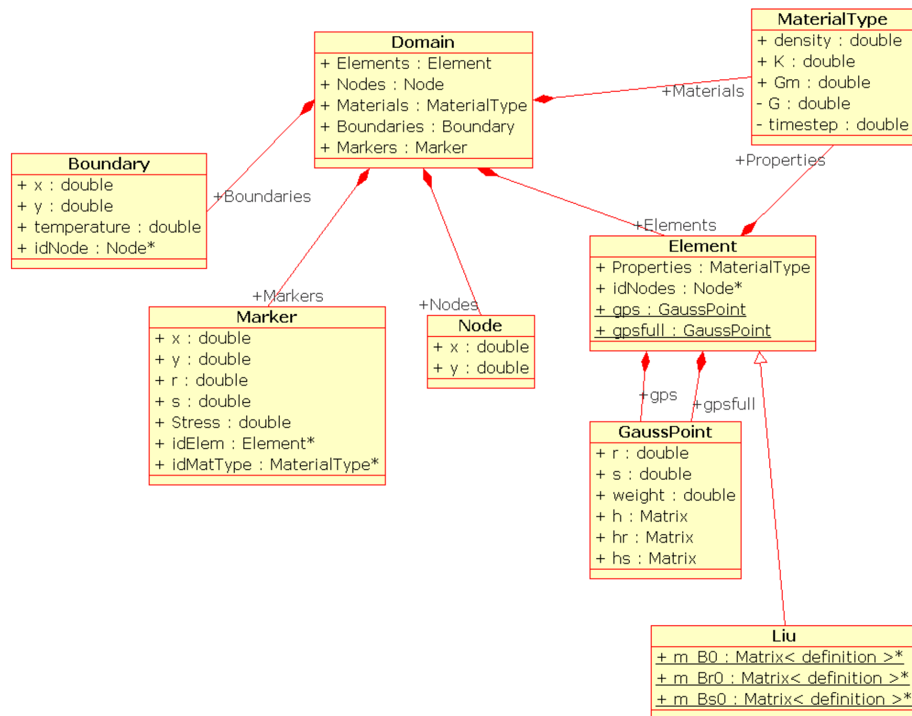


Figura 1: Diagrama de clases del modelo

- descripción del dominio,
- discretización del mismo,
- cálculo de matrices elementales,
- integración numérica en puntos de Gauss,
- ensamblado de matrices globales,
- imposición de condiciones de borde y
- resolución del sistema de ecuaciones, entre otras.

Cada una de estas etapas está definida independientemente de las otras. En la mayoría de los casos, las variaciones en la formulación a utilizar pueden ser parametrizadas, aislando de esta manera cada uno de los procesos.

Así como en el orden *secuencial* de operaciones relacionadas con FEM pueden encontrarse etapas factibles de ser pensadas de forma genérica, tal como se detalló en la sección anterior, también existe una separación en niveles desde el planteo mismo de la discretización del problema.

Los conceptos espaciales asociados a la discretización como: *dominio*, *elemento*, *condición de borde*, *nodo* y *punto de Gauss* introducen un nuevo nivel de abstracción y modularización en un sentido, no ya secuencial, sino de *tipo de datos*.

A partir de estos conceptos, se propuso el diagrama de clases que puede verse en la Fig. 1. Por simplicidad se muestran sólo los atributos más importantes de cada clase.

La clase `Domain` es la que contiene la información completa del dominio a modelar mediante un conjunto de elementos y nodos almacenados en los atributos `Elements` y `Nodes`.

A través de estos dos conjuntos se tiene toda la información espacial del dominio a modelar. Una vez que el dominio es conocido, cada una de las condiciones de borde que determinan el problema se representa en `Boundary`. El conjunto de condiciones de contorno es almacenado en otro atributo de `Domain` (`Boundaries`).

En los problemas en que la malla se deforma pueden utilizarse diversas técnicas para trasladar el estado del sistema de la malla deformada a una nueva sin deformación. Una de estas técnicas es la de *seguimiento de partículas* (Harlow and Welch, 1965; Moresi et al., 2003), por la cual se guarda el estado de varias partículas (o `Markers`) dentro de cada elemento a cada paso de tiempo. Para cada partícula se almacena su posición, tanto en coordenadas reales (`x`, `y`) como naturales (`r` y `s`), el elemento de pertenencia (`idElem`) y las variables que describen su estado (`Stress` u otras). Al momento de remallar, el nuevo estado para cada elemento es creado a partir del estado de las partículas que lo componen.

Dentro de la clase `MaterialType` se guardan todos los atributos del material a modelar. La clase `Domain` contiene el conjunto de materiales que se corresponden con la totalidad de los materiales involucrados en el problema. Por otro lado, cada una de las partículas hace referencia a un solo tipo de material a través del atributo `idMatType`. Sin embargo, puede darse el caso de que, debido a la discretización, las partículas contenidas en un elemento sean de distintos tipos de material. Es por esto que el elemento contiene un tipo de material propio que es calculado como una combinación de los materiales de las partículas que lo componen (Quinteros et al., 2006).

Es importante remarcar que, si bien los nodos determinan un elemento, no pertenecen exclusivamente a él, ya que en general son compartidos por dos o más elementos. De esta manera, en `Element` sólo se tiene una serie de referencias a los nodos que lo determinan (`idNodes`).

Por último, al momento de calcular las matrices de rigidez elementales, se debe hacer una integración numérica en los puntos de Gauss que correspondan según el orden de la interpolación usada en el elemento. Como la posición de los puntos de Gauss es siempre la misma para todos los elementos, debido a que se integra sobre el elemento maestro, esta información es almacenada como variable de tipo `static` dentro de la clase `Element` en el atributo `gps`, lo que permite que no sea replicada en cada instancia del elemento sino que es única y pertenece a todas las instancias de la clase. En problemas con una gran cantidad de elementos, eso favorece no sólo un mejor uso de la memoria sino el rendimiento en el acceso mediante la reutilización del *caché*.

Esta forma de conservar los puntos de Gauss es muy flexible en caso de implementar algún tipo de integración reducida (Hughes, 1980) (ver sección 5.1) ya que sólo basta con no incluir los `GaussPoint` en que no se va a integrar y modificar los pesos de los puntos en los que se integra. Aparte del atributo `gps` donde se guardan los puntos de integración efectivos, se cuenta con otro atributo (también de tipo `static`) en el que se almacenan la totalidad de los puntos de Gauss que corresponderían en caso de hacer una integración completa. Este atributo (`gpsfull`) es utilizado para ciertos procedimientos que no admiten integración reducida.

Como la evaluación de las funciones de forma (`h`) y sus derivadas por ambas coordenadas elementales (`hr` y `hs`) se calculan también en el elemento maestro, son precalculadas y almacenadas en `GaussPoint`. Luego, al momento de operar sobre el elemento real sólo resta transformarla, en caso que sea necesario, mediante la inversa del jacobiano.

2.2. Interfaz con *solvers* y bibliotecas matemáticas

Una vez resuelto el diseño de las clases de más alto nivel del problema, es decir, las estructuras de datos que lo representan, se debe pasar a la parte operativa. Sabemos por ejemplo que

```
Matrix<double> m_A( 2, 4, "A");
Matrix<double> m_B( 4, 2, "B");
Matrix<double> m_2x2( 2, 2, "2x2");
Matrix<double> m_4x4( 4, 4, "4x4");
...
m_A(0, 2) = 1.2;
m_A(0, 3) = 3.5;
m_B(0, 1) = 8.7;
m_B(3, 0) = 0.1;

m_2x2  = m_A * m_B;
m_4x4  = m_A.trans() * m_B.trans();
m_A   += m_B.trans();
```

Tabla 1: Ejemplo de sintaxis en el uso de la clase Matrix

la matriz de rigidez asociada a cada instancia de la clase `Element` surge de la multiplicación de matrices que describen características del elemento. Existiendo bibliotecas de álgebra lineal optimizadas y testeadas intensamente, se decidió desarrollar una clase `Matrix` que funcionara como interfaz entre el modelo y la biblioteca Lapack (Anderson et al., 1999), que es la utilizada en este momento. De esta manera, la biblioteca podrá ser reemplazada muy fácilmente, ya que no se la utiliza nunca directamente sino a través de la interfaz definida en la clase `Matrix`.

Esta clase provee un manejo simple y eficaz de manipulación de matrices densas. Su interfaz pública incluye todas las operaciones que se creyeron convenientes para la resolución de problemas mediante FEM. Se hizo un uso intensivo de la sobrecarga de operadores y el polimorfismo, de manera que la notación al momento de programar sea intuitiva, obviando las llamadas a funciones siempre que fuera posible (tabla 1), en especial para la asignación de valores a posiciones particulares de la matriz.

Las matrices elementales suelen ser densas, ya que todos los nodos del elemento tienen relación entre sí. En estos casos, las operaciones suelen ser eficientes porque no se precisa mucha memoria ni poder de cómputo para operar sobre matrices cuyo lado es del orden de la cantidad de nodos por elemento. Sin embargo, en el caso de la matriz de rigidez global asociada al problema no ocurre lo mismo.

Para problemas discretizados con una cantidad considerable de elementos, la matriz será rala y su tamaño del orden de la cantidad total de nodos. Para este tipo de matrices se diseñó otra clase denominada `SparseMatrix`, que cumple la misma función que la clase `Matrix` al implementar una interfaz con una biblioteca especial para matrices ralas. La funcionalidad de la clase `SparseMatrix` es menor, ya que sólo suele utilizarse para la resolución de sistema lineales de gran tamaño. Para esto, se implementó un formato de almacenamiento llamado *Harwell-Boeing* (Duff et al., 1992), también conocido como *almacenamiento por columnas comprimido*, que es soportado por varias bibliotecas que resuelven sistemas lineales con matrices ralas, en particular por SuperLU (Demmel et al., 1999). La estructura asociada al formato de almacenamiento dentro de la clase es la siguiente:

```
template<class T>
class SparseMatrix{
  public:
```

```
    T *values;   /* valores distintos de cero almacenados por
                 columna */
    int *rowind; /* fila en que está almacenado cada valor */
    int *colind; /* colind[j] almacena el offset dentro de
                 values y rowind en el que comienzan los datos
                 de la columna j. Tiene cols+1 entradas y
                 colind[cols] = realnnz */
    int rows;    /* número de filas de la matriz */
    int cols;    /* número de columnas de la matriz */
    int realnnz; /* número de entradas distintas de cero */
    ......
}
```

El acceso a los elementos de la matriz y las operaciones matriciales básicas tienen exactamente la misma definición que la clase `Matrix`, por lo que tampoco es necesario modificar el código en caso de realizar en una primera instancia una implementación de prueba con matrices densas.

Todas las clases contienen el método `Save`, que permite grabar la instancia a un *stream* de salida. Asimismo, todas esas clases cuentan con un constructor que recibe como parámetro un *stream* de entrada, a partir del cual se crea nuevamente el objeto. Esto permite grabar el estado de la simulación, en caso de involucrar varios pasos de tiempo, y reiniciar la ejecución a partir del último paso de tiempo calculado.

La idea primordial subyacente a las interfaces implementadas es poder brindar variantes a las librerías que se utilizan. Por ejemplo, poder reemplazar la implementación de Lapack que se usa o cambiar la biblioteca SuperLU por otra (i.e. SPOOLES) que se adapte mejor a las características de cierto tipo de problemas.

3. ABSTRACCIÓN A NIVEL ELEMENTAL

Hasta este punto no existe ninguna mención al tipo de elemento implementado. Esto se debe a que el esquema general de resolución no necesita conocer el tipo de elemento con el que se resolverá el problema. Mediante este diseño, se puede tener una libertad casi total de la implementación subyacente, siempre y cuando la interfaz diseñada pueda satisfacer los pasos necesarios para la resolución del problema.

Existen varias operaciones a nivel elemental cuya definición se da de forma genérica sin necesidad de contar con información de la implementación específica del elemento. Por ejemplo, el *push forward* de la formulación actualizada de Lagrange o el gradiente uniforme en un elemento están definidos de forma tal que la cantidad de nodos sólo es necesaria para el cálculo de operadores de más bajo nivel.

En el diseño propuesto, la clase `Element` está definida como una clase base que brinda un marco común a todos los tipos de elementos. En principio, brinda la interfaz para la resolución de problemas genéricos. Sin embargo, incluye también toda la implementación que puede brindar más allá de la elección puntual del elemento a utilizar.

En la Fig. 2 se puede ver el esquema general de secuencia de mensajes entre clases al momento de crear un dominio y los objetos asociados.

Para la implementación de un elemento, todo lo que no pueda ser definido de forma genérica tiene que ser implementado en una clase que derive de `Element`. La clase `Liu` que aparece en la Fig. 2 es un ejemplo de implementación de un elemento particular que hereda toda la

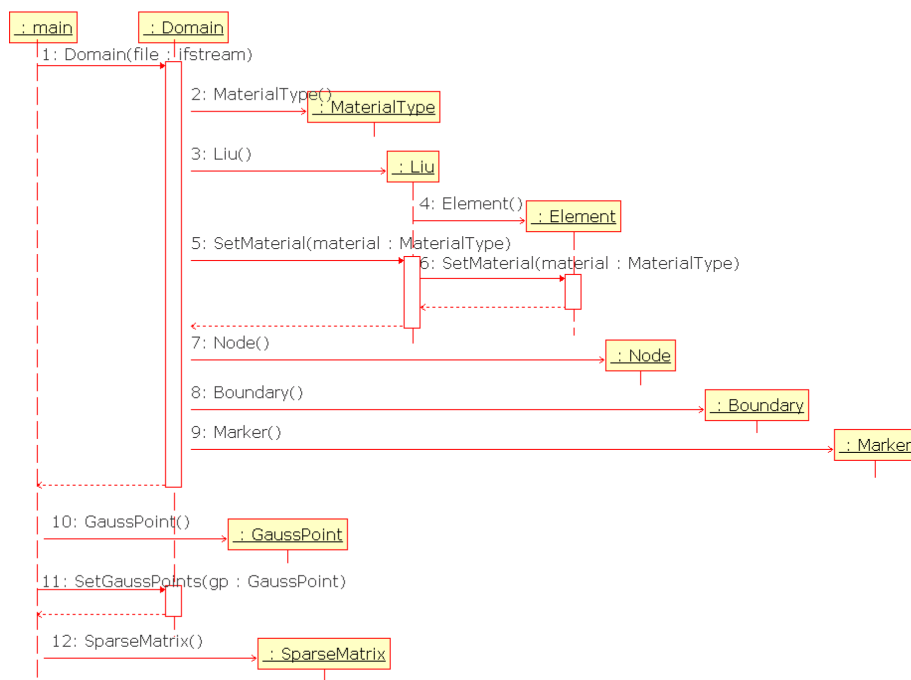


Figura 2: Diagrama esquemático de secuencia de mensajes para la creación del dominio y objetos relacionados

funcionalidad de `Element`. Por ejemplo, puede verse que la llamada a `SetMaterialType` es la heredada de `Element`.

En su forma más sencilla, la resolución de ecuaciones en derivadas parciales mediante FEM implica, para cada elemento, la evaluación en distintos puntos de Gauss de un producto de matrices (dependiente de la ecuación), que éstas sean ponderadas por el peso asociado al punto de Gauss, que sean sumadas para obtener la matriz de rigidez elemental y que eventualmente se le aplique a ésta alguna operación adicional relacionada con el problema.

De todas estas tareas, sólo la evaluación en un punto de Gauss particular necesita, en principio, contar con información detallada del elemento utilizado. Es por eso que `Element` contará con un método `calc_equation` donde estarán implementados todos los pasos necesarios para la obtención de la matriz elemental asociado con la ecuación a resolver (`equation`). La excepción será la evaluación puntual, que se encontrará en el método `eval_equation` implementado en la clase derivada (en este caso `Liu`) y que recibirá como parámetro el punto de Gauss en el cual se debe integrar.

En la Fig. 3 puede verse el esquema general de resolución de una ecuación. En este caso se trata de una ecuación de Poisson, aunque muchas se pueden ajustar a esta operatoria.

Nótese que la clase que efectivamente implementa las particularidades del elemento (`Liu`) deriva los mensajes a la clase `Element`, en donde fueron realment implementados, con excepción de la integración en un punto de Gauss (`eval_poisson`).

4. ESQUEMA GENERAL DE COMPONENTES DEL MODELO

El modelo presentado se relaciona con otros componentes de software para realizar tareas específicas como la discretización del dominio y la resolución del sistema de ecuaciones. En la fig. 4 puede verse cómo es la conexión entre estos componentes.

Al diseñar el modelo de manera general, es decir, no sujeto al estudio de un dominio en particular, la descripción del dominio y su discretización están más relacionados con parámetros

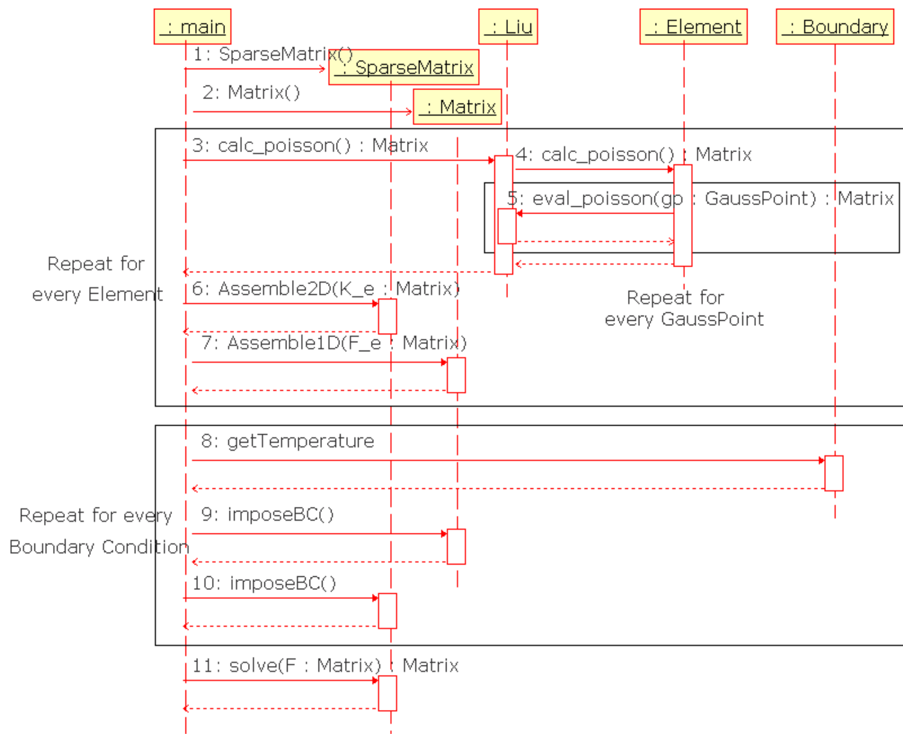


Figura 3: Diagrama esquemático de la resolución de la ecuación de Poisson.

de entrada que con la constitución del modelo mismo. Existen múltiples opciones de interfaces gráficas desarrolladas que pueden realizar esta tarea, como por ejemplo *GID*, desarrollada por el CIMNE (International Center for Numerical Methods in Engineering). Dentro de nuestro modelo se implementó una interfaz dentro de la clase `Domain` para leer desde este sistema y se crearon estructuras de datos apropiadas para su manipulación que son *independientes* del sistema en que fueron generadas. En el caso de querer generar las especificaciones del dominio mediante otra interfaz gráfica, el sistema no debe modificarse sino que la clase `Domain` deberá enriquecer su interfaz.

Las operaciones sobre matrices densas se realizan mediante la biblioteca `Lapack` (Anderson et al., 1999). Sin embargo, el único contacto con el sistema es a través de la interfaz implementada en la clase `Matrix`. Por lo tanto, en el modelo sólo se utilizan funciones de esta clase

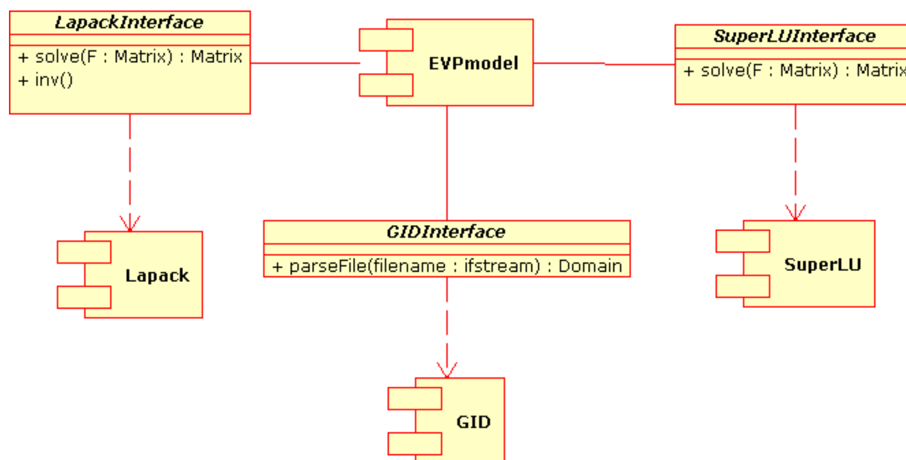


Figura 4: Diagrama esquemático de los componentes del modelo y las interfaces que permiten su conexión.

abstrayéndose por completo de la forma en la que se llevan a cabo.

En el caso de la matriz de rigidez global, debido al tamaño y a la gran cantidad de valores que posee se implementa mediante otra clase (`SparseMatrix`) que brinda métodos orientados a la representación de sistemas de ecuaciones de gran tamaño más que a matrices de propósito general. Esta clase brinda una interfaz hacia la biblioteca `SuperLU` (Demmel et al., 1999), que permite resolver sistemas lineales con matrices ralas. Los parámetros de bajo nivel de SuperLU permiten obtener muy buenos tiempos de resolución en caso de simetría e incluso almacenar el resultado de la etapa de preprocesamiento para no volver a realizarlo en caso que se quiera resolver otro sistema de condiciones muy similares. Esto suele acelerar de forma considerable los tiempos de resolución en sistemas transitorios y/o no lineales.

Asimismo, para optimizar el tiempo necesario al ensamblar la matriz de rigidez del sistema se replica la estructura *Harwell-Boeing* (Duff et al., 1992) en un buffer cuyo tamaño es parámetro del constructor. De esta manera, se reducen considerablemente la cantidad de movimientos de memoria necesarios para construir la matriz.

La misma clase da la opción de almacenar sólo la mitad de la matriz en caso de ser simétrica, lo que suele ocurrir en casi todos los problemas resueltos mediante el Método de Elementos Finitos.

El lenguaje utilizado para la codificación del modelo propiamente dicho fue `C++` estandar (GCC). El desarrollo se hizo bajo una conocida distribución del sistema operativo Linux (*Gentoo*) y también fue compilado en otras distribuciones sin inconvenientes. Todo el desarrollo ha sido realizado con el objetivo último de correr el modelo en un cluster de tipo *Beowulf*. Ese fue uno de los factores más importantes al optar por las bibliotecas de software mencionadas, ya que tanto `Lapack` como `SuperLU` cuentan con versiones para este tipo de configuraciones.

5. FORMULACIONES ELEMENTALES

Como ejemplo de la flexibilidad del modelo desarrollado, se implementaron dos tipos de elementos cuadriláteros con características muy diversas. En las siguientes secciones se detallan ambas formulaciones elementales, las cuales son implementadas en las clases `Liu` y `EightNodes` (ver secciones 5.1 y 5.2 respectivamente).

5.1. Cuatro nodos con integración reducida/selectiva

Uno de los elementos implementados fue un elemento cuadrilátero bidimensional de 4 nodos ($\mathrm{NN} = 4$) propuesto por Liu et al. (1994). Como características fundamentales de la formulación elemental se puede señalar que utiliza la técnica de integración reducida/selectiva para evitar el bloqueo volumétrico y de corte, así como también para disminuir el costo computacional. A su vez, proponen una aproximación para el control de los modos de *hourglass* de manera tal que el operador de estabilización se obtiene simplemente tomando derivadas parciales del vector de tasa de deformación generalizado con respecto a las coordenadas naturales.

Las funciones de forma (h) son las usuales para un elemento de 4 nodos. Las coordenadas espaciales (x), así como las velocidades (v), son aproximadas por las combinaciones lineales

$$x_i = \sum_{a=1}^{\mathrm{NN}} h_a(r,s) x_{ia} \tag{1}$$

y

$$v_i = \sum_{a=1}^{\mathrm{NN}} h_a(r,s) v_{ia} \quad , \tag{2}$$

donde los subíndices i y a representan la dimensión y el número de elemento, respectivamente.

La tasa de deformación se aproxima usualmente por

$$\dot{\epsilon}(r,s) = \sum_{a=1}^{\mathrm{NN}} B_a(r,s) v_a \quad , \tag{3}$$

donde B_a es la matriz gradiente que contiene las derivadas de las funciones de forma.

Si expandimos $\dot{\epsilon}$ en una serie de Taylor centrada en las coordenadas naturales (0,0) del elemento tendremos

$$\dot{\epsilon}(r,s) = \dot{\epsilon}(0,0) + \dot{\epsilon}_{,r}(0,0) r + \dot{\epsilon}_{,s}(0,0) s \quad , \tag{4}$$

por lo que se puede aproximar como

$$\dot{\epsilon}(r,s) = \sum_{a=1}^{\mathrm{NN}} \bar{B}_a(r,s) v_a \quad , \tag{5}$$

donde

$$\bar{B}_a(r,s) = B_a(0,0) + B_{a,r}(0,0) r + B_{a,s}(0,0) s \quad . \tag{6}$$

Para aliviar el bloqueo volumétrico se utiliza integración reducida/selectiva (Hughes, 1980). $\bar{B}_a(r,s)$ se separa en sus partes desviadora y volumétrica

$$\bar{B}_a(r,s) = \bar{B}_a^{vol}(0,0) + \bar{B}_a^{dev}(r,s) \tag{7}$$

La parte volumétrica de la matriz se evalúa solamente en el punto (0,0) para evitar el bloqueo volumétrico. Si se expande $\bar{B}_a^{dev}$ tal como en la ecuación 6, puede reescribirse como

$$\bar{B}_a(r,s) = B_a(0,0) + B_{a,r}^{dev}(0,0) r + B_{a,s}^{dev}(0,0) s \quad , \tag{8}$$

donde $B_a(0,0)$ es la matriz gradiente evaluada en el (0,0) con su parte volumétrica y desviadora.

Más allá de la formulación elemental, la cual asegura el alivio del bloqueo volumétrico aunque se integre utilizando un punto de Gauss, esto suele no ser suficiente en el caso en que se quieran detectar con precisión frentes de deformación plástica en problemas elasto-plásticos. Es por eso que para integrar se utilizan dos puntos de Gauss (Ec. 9):

$$\text{Punto 1} : \left(+\frac{1}{\sqrt{3}}, +\frac{1}{\sqrt{3}}\right) \qquad \text{Punto 2} : \left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right) \quad . \tag{9}$$

5.2. Elemento clásico de ocho nodos

Se sabe que los elementos cuadriláteros de 4 nodos, entre otros, no cumplen la condición de Brezzi y Babuschka (Brezzi and Fortin, 1991), necesaria para asegurar la convergencia a la solución y/o evitar efectos no deseados como la distribución *checkerboard* al calcular la presión.

El segundo elemento implementado fue un elemento cuadrilátero bidimensional de 8 nodos ($\mathrm{NN} = 8$) clásico. No se consideró necesario incluir ningún tipo de operador de estabilización ni control de los modos de *hourglass* ya que, debido al mayor orden en la interpolación, la convergencia está asegurada.

Las funciones de forma (h) se definen también de forma normal para un elemento de 8 nodos. Tanto las coordenadas espaciales (x) como las velocidades (v) se aproximan igual que en el elemento detallado en la sección anterior (Ec. 1 y 2).

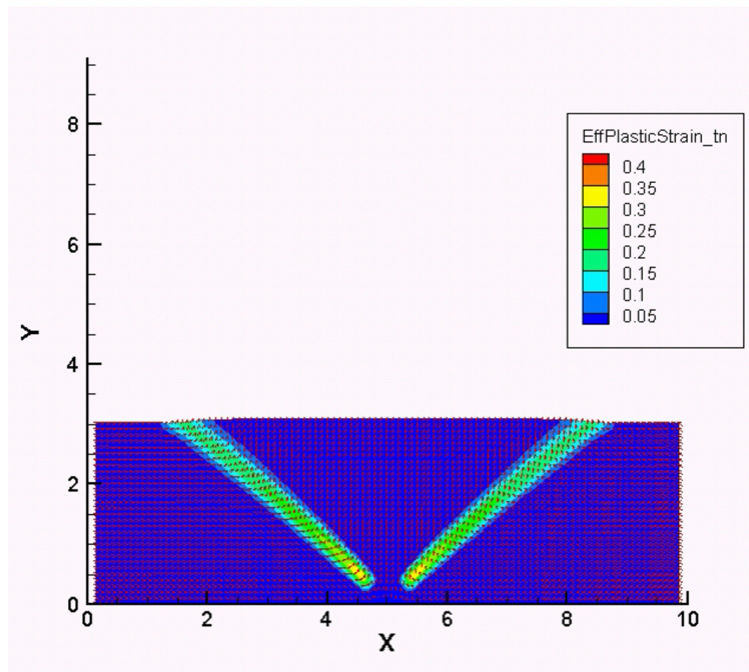


Figura 5: Concentración de la deformación plástica provocada por la compresión en presencia de un punto de debilidad.

La tasa de deformación se aproxima de la forma estandar por la ecuación 3, donde B_a es la matriz gradiente que contiene las derivadas de las funciones de forma. A diferencia del elemento anterior, no se hace ningún desarrollo de Taylor y está evaluada de forma total en los puntos de Gauss usados para integrar.

Los puntos de Gauss en los que se integra y sus pesos son los indicados teóricamente (Bathe, 1996) para funciones de interpolación de segundo orden.

6. RESULTADOS

Una vez que el modelo fue concluido se corrieron los casos de prueba que se detallan a continuación para validar los distintos tipos de comportamiento. En principio todos los tests fueron corridos con elementos de 8 nodos, debido a sus propiedades de convergencia.

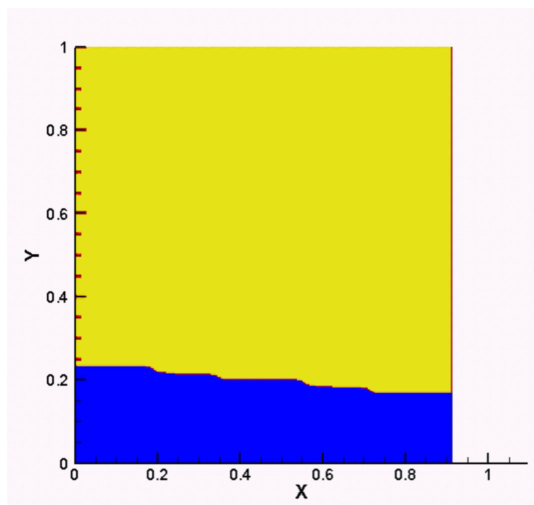
6.1. Concentración de la deformación plástica

Se modela un sólido elasto-plástico al que se le aplican condiciones de borde de desplazamiento con el borde superior libre de tensiones. El proceso de concentración de la deformación es inducido por la inclusión de material *débil* con un módulo de corte dos órdenes de magnitud menor en la parte central del borde inferior.

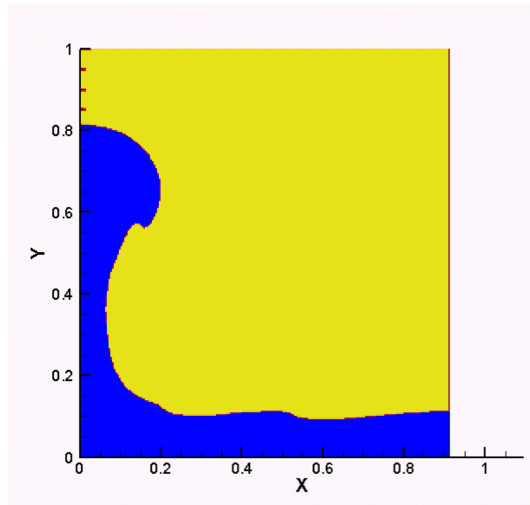
Puede verse en la figura 5 que la deformación plástica efectiva se concentra en dos fallas con un ancho del orden del tamaño del elemento, resultando en la separación en tres bloques rígidos. Esto se logra incluso ante una aproximación estándar de Von Mises y sin utilizar mecanismos de *strain softening*. Las características de los materiales se muestran en la tabla 2.

6.2. Inestabilidad de Rayleigh-Taylor

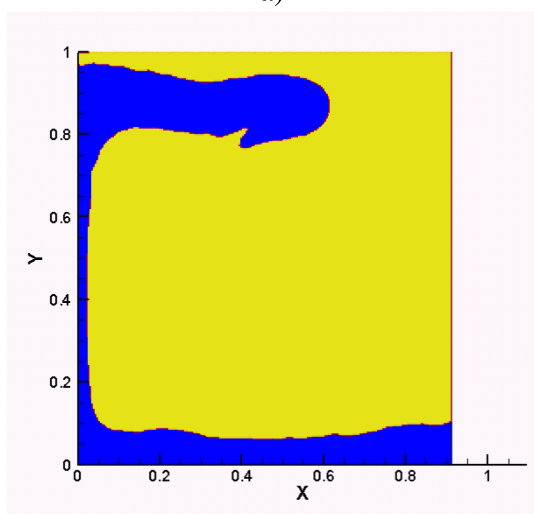
Consiste en dos fluidos viscosos con diferentes densidades ubicados en un recipiente cerrado de altura igual a la unidad y con un ancho de 0.9142. El estado inicial es inestable debido a que



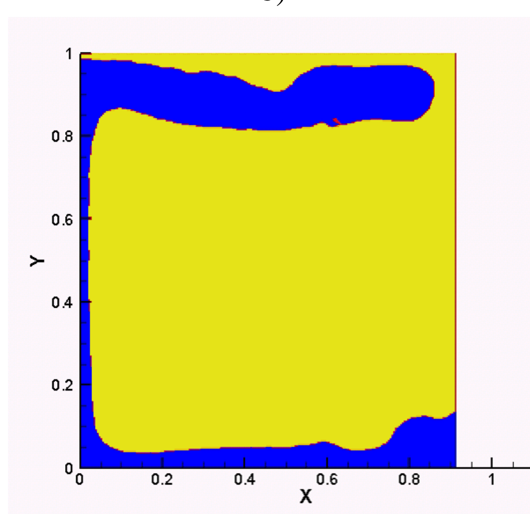
a)



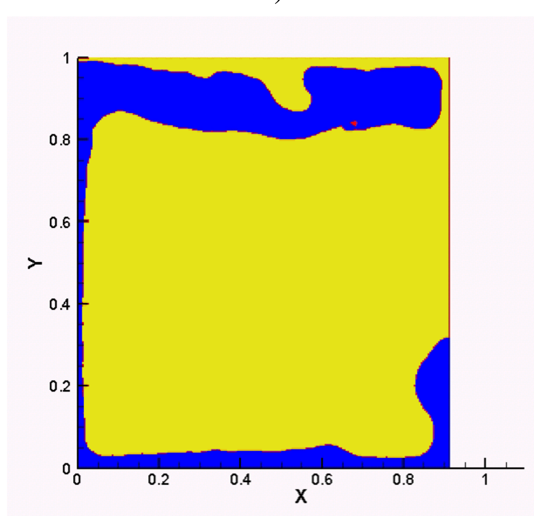
b)



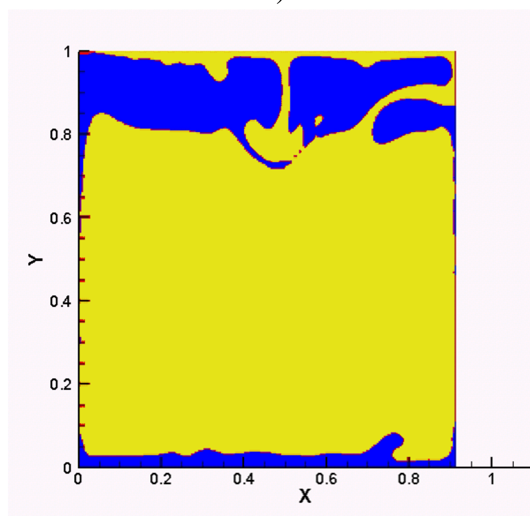
c)



d)



e)



f)

Figura 6: Validación de comportamiento viscoso en el test de Rayleigh-Taylor según los parámetros propuestos por van Keken et al. (1997)

Característica	Material 1	Material 2
Densidad	1	1
Viscosidad	∞	∞
Elastic Bulk modulus	1000	1000
Elastic Shear modulus	90	0.9
σ_0	1.3	1.3
Gravedad	1	1

Tabla 2: Características de los dos materiales del caso de prueba de comportamiento plástico.

el fluido que se encuentra en la parte superior es el más denso, lo que provoca el comienzo de la convección. Las características de ambos fluidos pueden verse en la tabla 3.

La frontera entre ambos tipos de fluido se mantiene bien definida a lo largo de toda la simulación y no ocurre ningún tipo de difusión numérica. La evolución completa puede observarse en la figura 6.

6.3. Viga empotrada

Para validar el comportamiento elástico se corrieron varios casos de prueba de vigas empotradas sometidas a distintas condiciones. Sólo mostramos uno de los resultados en la figura 7, donde aparece la solución, tanto analítica como numérica.

6.4. Mediciones de eficiencia

Consideramos en este trabajo dos formas de medir la eficiencia del modelo presentado. En primer lugar presentamos algunas mediciones para ver qué porcentaje del tiempo total está invirtiéndose en cada etapa y qué tiempo toma la resolución para distintas cantidades de grados de libertad.

Sin embargo, consideramos muy importante otro indicador de eficiencia que está más relacionado con el buen diseño del modelo en todos los aspectos descriptos. Decidimos contabilizar el costo extra (en líneas de código) de agregar la resolución de las distintas ecuaciones al modelo o de cambiar por completo el elemento que se utiliza para resolver las ecuaciones.

6.4.1. Mediciones de tiempos

Para la medición de tiempos de ejecución se corrió el test de viscosidad de Rayleigh-Taylor con el elemento cuadrilátero propuesto por Liu et al. (1994) y explicado en la sección 5.1.

Las simulaciones fueron ejecutadas en una laptop con procesador Intel Pentium Dual Core

Característica	Fluido 1	Fluido 2
Densidad	1.3	1.0
Viscosidad	1	1
Elastic Bulk modulus	1000	1000
Elastic Shear modulus	1000	1000
σ_0	∞	∞
Gravedad	1	1

Tabla 3: Características de los dos fluidos de la simulación para comportamiento viscoso.

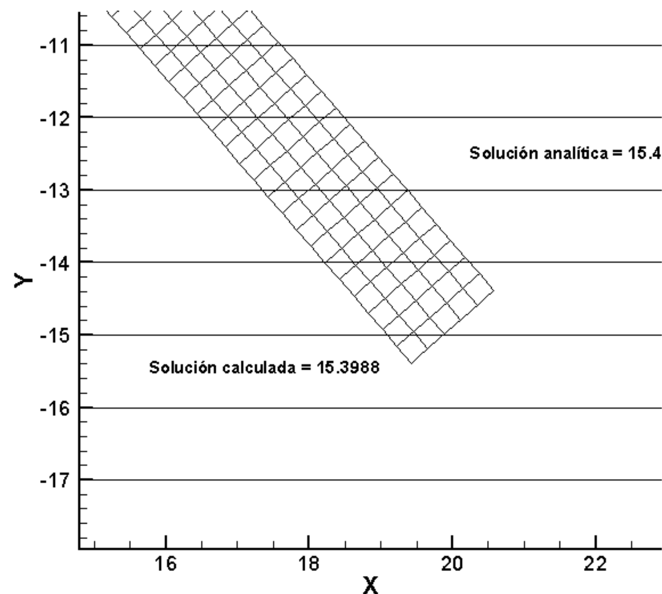


Figura 7: Ejemplo de uno de los casos de test que valida el comportamiento elástico.

de 1.6 GHz y 800 MB de memoria RAM. El sistema operativo es una versión de Gentoo y al momento de la ejecución de las simulaciones no se ejecutaban otras tareas que interfirieran. Los tiempos de corrida según el tamaño del problema pueden verse en la tabla 4, mientras que en la tabla 5 se pueden observar los mismos tiempos pero expresados como porcentaje del total consumido por una iteración.

6.4.2. Medidas de escalabilidad de funcionalidad

La flexibilidad, modularización y generalidad en el diseño del sistema han permitido que varios tipos de problemas puedan resolverse reutilizando casi la totalidad del sistema y codificando sólo las particularidades del nuevo problema a resolver.

En la tabla 6 pueden verse los problemas implementados hasta el momento, el tipo de ecuaciones en el que están basados y la cantidad de líneas de código extra[1] que se necesitaron para implementar la resolución. Está claro que la implementación en el caso de un nuevo problema

[1] Se calcula como el porcentaje de incremento del total de líneas de código del modelo básico presentado.

Grados de libertad	Ensamble	Condiciones de Borde	Resolución	Total Iteración
2000	0,14	0,08	0,03	0,25
20000	1,84	2,63	1,21	5,68
30000	2,84	4,88	2,04	9,76
40000	3,69	7,68	3,64	15,01
50000	4,71	10,59	5,01	20,31
60000	5,66	14,51	7,15	27,32
100000	9,29	31,73	15,77	56,79

Tabla 4: Tiempo en segundos que insume cada etapa para la ejecución de una iteración

Grados de libertad	Ensamble	Condiciones de Borde	Resolución
2000	56,00 %	32,00 %	12,00 %
20000	32,39 %	46,30 %	21,30 %
30000	29,10 %	50,00 %	20,90 %
40000	24,58 %	51,17 %	24,25 %
50000	23,19 %	52,14 %	24,67 %
60000	20,72 %	53,11 %	26,17 %
100000	16,36 %	55,87 %	27,77 %

Tabla 5: Porcentaje del tiempo total que insume cada etapa.

implica un costo muy bajo debido a la generalidad y modularización del diseño que favorece la reutilización de código.

Tal como se mencionó en la sección anterior, son bien conocidas las propiedades en cuanto a convergencia de los elementos cuadriláteros de 8 ó 9 nodos (Brezzi and Fortin, 1991). Lamentablemente, los cálculos con elementos de tantos nodos tienen su contraparte en cuanto al costo computacional de la resolución. Para problemas de tamaño acotado pueden utilizarse. Sin embargo, podrían ser inviables al momento de refinar la solución encontrada.

En el caso en que una nueva formulación a un problema quiera ser testeada puede pasar que un experimento fallido se deba al elemento utilizado y no a la formulación misma. Por ejemplo, si por cuestiones de costo computacional se decide utilizar un elemento de 4 nodos, la convergencia ante ciertas condiciones no está asegurada.

En nuestro caso particular, se deseaba validar una formulación para problemas de deformación geológica a escala continental que incluía comportamiento elástico, plástico y viscoso tomando como base los trabajos previos de Babeyko et al. (2002); Sobolev and Babeyko (2005).

Sin embargo, una vez que la formulación fue validada, era claro que la resolución de problemas con una gran cantidad de elementos demandaría mucho tiempo de cálculo. Para resolver esto, se implementó un elemento de 4 nodos con integración reducida/selectiva y gradiente uniforme (ver sección 5.1) propuesto por Liu et al. (1994). Las validaciones pudieron ser corridas nuevamente simplemente cambiando el tipo de elemento con el que se compilaba el código. Ninguna modificación se llevó a cabo en el código principal, ya que la clase `Element` funciona como interfaz hacia las implementaciones elementales.

Los resultados finales fueron idénticos. La gran ventaja de utilizar 4 nodos fue la aceleración del tiempo utilizado. La única desventaja es que se debió remallar más veces para solucionar los pocos efectos de *hourglass* que aparecieron en el interior del dominio. Si bien, el elemento de Liu et al. (1994) alivia la aparición de estos modos, en problemas transitorios estos efectos

Problema	Ecuación	Líneas de código extra en función del número total
Térmico	Poisson	1.36 %
Remallado	Laplace	1.13 %
Deformación de sólidos elasto-visco-plásticos	(ver Quinteros et al. (2006))	4.91 %
Cambio completo de elemento para las tres ecuaciones		6.94 %

Tabla 6: Problemas resueltos hasta el momento y líneas de código necesarias para implementar la resolución.

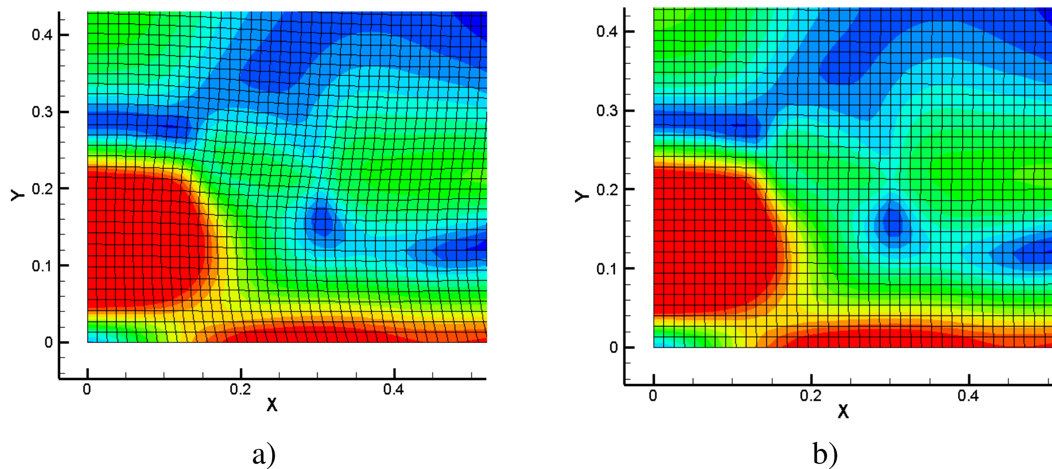


Figura 8: Estado de la componente desviadora de tensiones y de la malla en dos pasos de tiempo sucesivos, (a) previo y (b) posterior al remallado

pueden acumularse y crecer a lo largo de los pasos de tiempo. Sin embargo, la aplicación de técnicas apropiadas de remallado elimina la acumulación de errores de estos modos.

Las distintas técnicas de remallado implementadas (Carey and Oden, 1984) y la utilización de partículas marcadoras o *Markers* (Harlow and Welch, 1965) para transmitir el estado de la malla distorsionada a la nueva malla fueron indirectamente validadas mediante el caso de prueba de comportamiento viscoso de Rayleigh-Taylor. En la figura 8 puede observarse el resultado del remallado de un paso de tiempo al siguiente para este test según las condiciones propuestas por van Keken et al. (1997).

7. CONCLUSIONES Y TRABAJOS EN PROGRESO

Se han presentado los resultados del diseño e implementación de un modelo de Elementos Finitos que brinda un marco general para la resolución de distintos tipos de problemas. Su diseño modular y el uso de abstracciones apropiadas dan una amplia flexibilidad en cuanto al tipo de elemento utilizado para la resolución y permite reutilizar grandes porciones de código que fueron validadas, brindando así mayor confiabilidad en la resolución de problemas cuya solución analítica no es conocida.

En la actualidad, se está enriqueciendo el sistema mediante la implementación de elementos espectrales (Patera, 1984; Karniadakis et al., 1985) y elementos que permitan la *condensación* de sus nodos internos (ver por ejemplo (Henderson and Karniadakis, 1995)).

En cuanto a los problemas resueltos mediante este *framework*, se está avanzando en la implementación del método conocido como *Kinematic Laplacian Equation* desarrollado por Ponta (2005) para problemas de dinámica de fluidos. Algunos resultados de este método puede verse en Otero and Ponta (2006).

REFERENCIAS

Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., and Sorensen D. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

Babeyko A.Y., Sobolev S.V., Trumbull R.B., Oncken O., and Lavier L.L. Numerical models of crustal scale convection and partial melting beneath the Altiplano - Puna plateau. *Earth and*

Planetary Science Letters, 199:373–388, 2002.
Bathe K.J. *Finite Element Procedures*. Prentice Hall, New Jersey, 1996.
Brezzi F. and Fortin M. *Mixed and Hybrid Finite Elements Methods*. Springer-Verlag, 1991.
Cardona A., Klapka I., and Geradin M. Design of a new finite element programming environment. *Engineering Computation*, 11(4):365–381, 1994.
Carey G.F. and Oden J.T. *Finite Elements - Computational Aspects - Volume III*. Prentice-Hall, Inc., New Jersey, first edition, 1984.
Dari E.A., Buscaglia G.C., and Lew A. A parallel general purpose finite element system. In *IX SIAM Conference on Parallel Processing for Scientific Computing*. San Antonio, Texas, USA, 1999.
Demmel J.W., Eisenstat S.C., Gilbert J.R., Li X.S., and Liu J.W.H. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.
Duff I., Grimes R., and Lewis J. Users' guide for the Harwell-Boeing sparse matrix collection (release 1). Technical Report RAL-92-086, Rutherford Appleton Laboratory, 1992.
Harlow F.H. and Welch J.E. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *The Physics of Fluids*, 8(12):2182–2189, 1965.
Henderson R.D. and Karniadakis G.E. Unstructured spectral element methods for simulation of turbulent flows. *Journal of Computational Physics*, 122:191–217, 1995.
Hughes T.J.R. Generalization of selective integration procedures to anisotropic and nonlinear media. *International Journal of Numerical Methods for Engineering*, 15:1413–1418, 1980.
Karniadakis G.E., Bullister E.T., and Patera A.T. A spectral element method for solution of two- and three-dimensional time-dependent incompressible Navier-Stokes equations. In *Finite Element Methods for Nonlinear Problems*, page 803. Springer-Verlag, 1985.
Liu W.K., Hu Y.K., and Belytschko T. Multiple Quadrature underintegrated finite elements. *International Journal for Numerical Methods in Engineering*, 37:3263–3289, 1994.
Moresi L.N., Dufour F., and Muhlhaus H.B. A Lagrangian integration point finite element method for large deformation modeling of viscoelastic geomaterials. *Journal of Computational Physics*, 184:476–497, 2003.
Otero A.D. and Ponta F.L. Spectral-Elemental of the KLE method: A (ω,V) formulation of the Navier-Stokes equations. In A. Cardona, N. Nigro, V. Sonzogni, and M. Storti, editors, *Mecánica Computacional*, volume XXV, pages 2649–2668. Asociación Argentina de Mecánica Computacional, Santa Fe, Argentina, 2006.
Patera A.T. A spectral element method for fluid dynamics: laminar flow in a channel expansion. *Journal of Computational Physics*, 54:468–488, 1984.
Ponta F.L. The kinematic Laplacian equation method. *Journal of Computational Physics*, 207:405–426, 2005.
Quinteros J., Jacovkis P.M., and Ramos V.A. Formación de cordilleras y delaminación litosférica. Un modelo elasto-visco-plástico mediante elementos finitos. In A. Cardona, N. Nigro, V. Sonzogni, and M. Storti, editors, *Mecánica Computacional*, volume XXV, pages 2669–2686. Asociación Argentina de Mecánica Computacional, Santa Fe, Argentina, 2006.
Sobolev S.V. and Babeyko A.Y. What drives orogeny in the Andes? *Geology*, 33:617–620, 2005.
Sonzogni V.E., Yommi A.M., Nigro N.M., and Storti M.A. A parallel finite element program on a beowulf cluster. *Advances in Engineering Software*, 33:427–443, 2002.
van Keken P.E., King S.D., Schmeling H., Christensen U.R., Neumeister D., and Doin M.P. A comparison of methods for the modeling of thermochemical convection. *Journal of Geophysical Research*, 102:22477–22495, 1997.