MÁSTER EN MATEMÁTICA INDUSTRIAL

PROGRAMACIÓN EN C++

Métodos de Elementos Finitos 2D en C++

ALBERTO CUADRA LARA

Proyecto presentado en cumplimiento de los requisitos de la asignatura Programación en C++ curso 2017-2018 para el Máster en Matemática Industrial

15 de abril de 2018

Índice general

Boletín 4

Método de elementos finitos 2D en C++

4.1. Introducción

En el presente proyecto se ha realizado un programa en C++ que resuelve la ecuación de Poisson bidimensional con condiciones de contorno tipo Dirichlet mediante el método de elementos finitos, concretamente, elementos Lagrange P_1, a partir de un mallado y del esqueleto del código dados, facilitado por los profesores de la asignatura.

En primer lugar, se describe el problema a resolver seguido de una explicación sobre la resolución mediante el método de elementos finitos. Posteriormente, se explica el código elaborado y las características de éste, finalizando con la exposición de posibles trabajos futuros del proyecto realizado.

La elaboración del código ha sido llevada a cabo mediante el programa KDevelop 4 en Linux. Para el postprocesado de los datos, así como, la representación del dominio de cálculo se ha utilizado el programa MATLAB R2018a. En el código adjunto se detallan más comentarios sobre la realización del código. Para su resolución se han tomado como referencia los apuntes de la asignatura [5], así como el libro *Programación orientada a objetos con C++* [2].

De forma paralela al proyecto se incluye un archivo comprimido que incluye:

- Programa FEM2D realizado en C++.
- Archivo *FEM2D.m* utilizado para el posprocesado.
- *matriz_A.txt*, *vector_b.txt* y *output.txt*, correspondiendo con la matriz global, término independiente y solución del problema, respectivamente.
- Archivo de registro *log.txt*.

4.2. Descripción del problema

Sea Ω abierto acotado de $\Re^2$ de frontera Γ de clase C^1 a trozos, se desea aproximar $u(x, y)$ en el dominio Ω (ver figura 4.1). El problema a resolver es

$$\begin{cases} -\Delta u(x,y) = f(x,y) & (x,y) \in \Omega, \\ u = 0 & (x,y) \in \partial\Omega, \end{cases} \tag{4.1}$$

donde

$$f(x,y) = -2x^4 + x^2\left(\frac{33}{2} - 24y^2\right) - 2y^4 + \frac{33}{2}y^2 - 5. \tag{4.2}$$

El problema en cuestión tiene la siguiente solución analítica

$$u(x,y) = (x^2-1)(y^2-1)\left(x^2+y^2-\frac{1}{4}\right), \tag{4.3}$$

expresión con la que se comprobará la precisión de los resultados obtenidos.

El sistema de ecuaciones 4.1 corresponde con la formulación fuerte del problema que con las condiciones de contorno queda cerrado y puede resolverse mediante métodos numéricos, en este caso, mediante el método de elementos finitos.

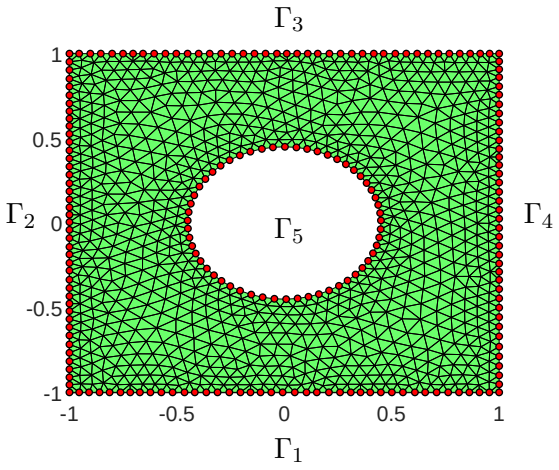

FIGURA 4.1: Dominio de cálculo.

4.3. Formulación variacional

Para poder resolver el problema mediante elementos finitos es necesario primero obtener su formulación variacional, lo que corresponde con la formulación débil del problema. Para ello, se define una función test $v(x,y) \in \Omega$, tal que

$$\Omega = \left\{v(x,y) \in H^1(\Omega)/v(x,y) = 0, \quad (x,y) \in \Gamma_i, \text{con } i = 1,\ldots,5\right\}, \tag{4.4}$$

donde H^1 es el espacio de energía asociado al problema de contorno [8], que corresponde con el espacio de Sobolev de orden 1 y se define como el espacio de las funciones de cuadrado integrable $H^1 \in L^2(\Omega)$[1], con derivadas generalizadas de primer orden de cuadrado integrable y dotado del producto escalar[2][7]:

$$(u,v)_{1,\Omega} = \sum_{|\alpha|\leq 1} (\partial^\alpha u, \partial^\alpha v)_{0,\Omega} = \sum_{|\alpha|\leq 1} \int_\Omega \partial^\alpha u \cdot \partial^\alpha v \, dx, \tag{4.5}$$

[1]Denotamos por $L^2(\Omega)$ el conjunto de funciones medibles $u : \Omega \to \Re$ tales que: $\int_\Omega |u(x)|^2 \, dx < \infty$ (integral de Lebesque), dotado de la seminorma $\left(\int_\Omega |u(x)|^2 \, dx\right)^2$[7].

[2]∂^α indica derivada en sentido de distribuciones.

que induce la norma:

$$||u||_{1,\Omega} = (u,u)^{1/2}_{1,\Omega}, \tag{4.6}$$

donde u representa la solución del problema.

Una vez definido el espacio de funciones admisibles, se multiplica ambos miembros de la ecuación 4.1 por la función test $v(x,y)$ e integramos en el dominio

$$-\int_\Omega \Delta u(x,y)v(x,y)\,dxdy = \int_\Omega f(x,y)v(x,y)\,dxdy. \tag{4.7}$$

La 1ª fórmula de Green (integración por partes) en el espacio definido dicta que:

$$\int_\Omega u\frac{\partial v}{\partial x_i}\,dx = \int_\Gamma u|_\Gamma v|_\Gamma \nu_i\,d\gamma - \int_\Omega \frac{\partial u}{\partial x_i}u\,dx, \tag{4.8}$$

donde ν_i es la componente i-ésima de $\vec{\nu}$ que representa la normal exterior de Γ.

Aplicando la fórmula anterior al primer término se tiene

$$\begin{aligned}\int_\Omega \Delta u(x,y)v(x,y)\,dxdy \;&= \int_\Omega \nabla u(x,y)\nabla v(x,y)\,dxdy \\ &\quad - \int_{\partial\Omega} \nabla u(x,y)v(x,y)\vec{\nu}\,d\gamma.\end{aligned} \tag{4.9}$$

Dado que todas las condiciones de contorno son condiciones tipo Dirichlet, la integral en la frontera es nula por la propia definición de la función test, obteniendo

$$\underbrace{\int_\Omega \nabla u(x,y)\nabla v(x,y)\,dxdy}_{a(u,v)} = \underbrace{\int_\Omega f(x,y)v(x,y)\,dxdy}_{l(v)}, \tag{4.10}$$

que corresponde con la formulación variacional del problema, también conocido como igualdad integral. El problema queda establecido en encontrar $u \in \Omega$ tal que

$$a(u,v) = l(v), \quad \forall v \in \Omega. \tag{4.11}$$

4.4. Discretización del problema, método de Galerkin

Se reemplaza Ω de dimensión infinita por un subespacio de dimensión finita Ω_h y se trata de encontrar la solución aproximada $u_h \in \Omega_h$ tal que

$$a(u_h,v_h) = l(v_h), \quad \forall v_h \in \Omega_h. \tag{4.12}$$

Sea $\{\varphi_1,\varphi_2,...,\varphi_{N_h}\}$ una base de Ω_h, siendo N_h son el número total de funciones de base (una por cada nodo), entonces

$$u_h = \sum_{i=1}^{N_h} u_i\varphi_i, \tag{4.13}$$

Así, tenemos que encontrar los valores $\{u_i\}_{i=1}^{N_h}$ tales que

$$\sum_{i=1}^{N_h} a(\varphi_i, \varphi_j) = l(\varphi_j), \quad \forall v_h \in \Omega_h. \tag{4.14}$$

Empleando la siguiente notación:

$$A_{ij} = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j \, dxdy, \tag{4.15}$$

$$b_j = \int_\Omega f\varphi_j \, dxdy. \tag{4.16}$$

Se trata de encontrar el vector $u_h = [u_1, u_2, ..., u_{N_h}]^T$ solución del siguiente sistema de ecuaciones:

$$Au_h = b. \tag{4.17}$$

4.4.1. Observación

- Nótese, que en la Ec. 4.15 hay que recorrer todos los elementos del mallado, y aproximar la integral de ese producto para añadir en el lugar (i, j) de la matriz global. Análogamente, para la Ec. 4.16, pero añadiéndolo en el lugar (j).

4.4.2. Elementos Lagrange P_1

Para resolver el problema se va a realizar una discretización por elementos Lagrange P_1, que son elementos triangulares lineales. La malla utilizada en la discretización está compuesta por 990 nodos ($N_h = 990$) y 1764 elementos ($E = 1765$). En la figura 4.1, se observa la discretización utilizada. Se denota a un elemento contenido en Ω como E.

Sea Ω la unión de un conjunto $\Omega_h = E$, de triángulos no superpuestos E tales que ningún vértice de algún triángulo se ubique sobre el lado de otro triángulo, se define el espacio[1]:

$$P_1(E) = \{v(x, y) = a_0 + a_1 x + a_2 y \quad \forall (x, y) \in E, a_i \in \Re\}. \tag{4.18}$$

Teniendo en cuenta lo anterior, para calcular las integrales 4.15 y 4.16 se puede emplear el elemento de referencia o bien calcular la aproximación directa, dado que los gradientes son constantes

$$A_{ij} = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j \, dxdy \approx |T| \nabla\varphi_i|_T \cdot \nabla\varphi_j|_T, \tag{4.19}$$

donde $|T|$ es el área del triángulo.

El área se ha obtenido mediante la siguiente expresión

$$|T| = \sqrt{s(s - d_1)(s - d_2)(s - d_3)}, \tag{4.20}$$

$$s = \frac{d_1 + d_2 + d_3}{2}, \tag{4.21}$$

donde d_i es la distancia del lado i-ésimo con $i = 1, 2, 3$ y s es el semiperímetro.

Las funciones base son

$$\varphi_i(x, y) = \frac{1}{2|T|} det \begin{bmatrix} 1 & x & y \\ 1 & x_{i+1} & y_{i+1} \\ 1 & x_{i+2} & y_{i+2} \end{bmatrix}, \tag{4.22}$$

y el gradiente

$$\begin{aligned} \nabla\varphi_i &= \frac{1}{2|T|} \left(- \begin{vmatrix} 1 & y_{i+1} \\ 1 & y_{i+2} \end{vmatrix}, \begin{vmatrix} 1 & x_{i+1} \\ 1 & x_{i+2} \end{vmatrix} \right) \\ &= \frac{1}{2|T|} \left(-(y_{i+2} - y_{i+1}), (x_{i+2} - x_{i+1}) \right). \end{aligned} \tag{4.23}$$

En el caso del término independiente b, su expresión puede verse simplificada empleando la siguiente regla de cuadratura

$$b_j = \int_\Omega f\varphi_j \, dxdy \approx \frac{1}{3}|T| \sum_{l=1}^{3} f(x_l, y_l)\varphi(x_l, y_l), \tag{4.24}$$

donde x_l e y_l con $l = 1, 2, 3$ indican la coordenadas del triángulo $T \in \Omega_h$. Esta regla de cuadratura integra exactamente la función lineal y tiene una precisión de primer orden [3].

4.4.3. Bloqueo de las condiciones de contorno

Bloqueando las condiciones de contorno se tiene un sistema 990×990, donde los valores de la diagonal de la matriz de coeficientes no se calculan, sino que se imponen. Mismo procedimiento para el elemento del vector independiente correspondiente, salvo que éste se multiplica por la condición tipo Dirichlet asociada (condiciones esenciales) consiguiendo así que la incógnita tome dicho valor [4]. De este modo, se preserva la estructura y propiedades de la matriz. En este caso, correspondería con forzar los valores siguientes

$$u_i = 0, \tag{4.25}$$

siendo i el índice de los respectivos nodos frontera.

El sistema final a resolver sería

$$A_{ij}u_{h,i} = b_i, \quad i = j = 1, ..., 990. \tag{4.26}$$

Ejemplo del bloqueo

Sea $a_{ij}u_i = b_j$, se tiene

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,30} \\ a_{21} & a_{22} & \cdots & a_{2,30} \\ \vdots & & \ddots & \vdots \\ a_{30,1} & \cdots & & a_{30,30} \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{990} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{990} \end{bmatrix}, \tag{4.27}$$

para imponer la condición en el caso de u_{990} sería

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,990} \\ a_{21} & a_{22} & \cdots & a_{2,990} \\ \vdots & & \ddots & \vdots \\ a_{990,1} & \cdots & & 10^{20} \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{990} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ 10^{20} \cdot \alpha \end{bmatrix}, \tag{4.28}$$

donde α, es la condición de contorno. Debido al valor desorbitado en comparación con el resto de contribuciones de a_{ij} y b_j, prácticamente, $u_{990} = \alpha$. Se procede igual en el resto de casos.

4.5. Programación en C++

El programa, al que denominaremos en adelante como FEM2D, está organizado por los siguientes directorios (véase 4.2):

- *bin*: incluye el archivo ejecutable del programa con extensión *.exe*.
- *build*: incluye los archivos objetos *.o* generados de los archivos *.cpp* mediante el *makefile*.
- *Eigen*: librería para operaciones con álgebra lineal: matrices, vectores, métodos numéricos, y algoritmos relacionados [6].
- *include*: incluye los archivos de cabecera de los archivos *.cpp*. Los archivos *data.h* y *mesh.h* no se han separado con sus respectivos *.cpp*, por lo que no se generan archivos objetos de estos.
- *mesh*: incluye los datos del mallado con extensión *.dat*.
- *sources*: incluye los archivos *.cpp* donde se definen los métodos indicados en sus respectivas cabeceras.

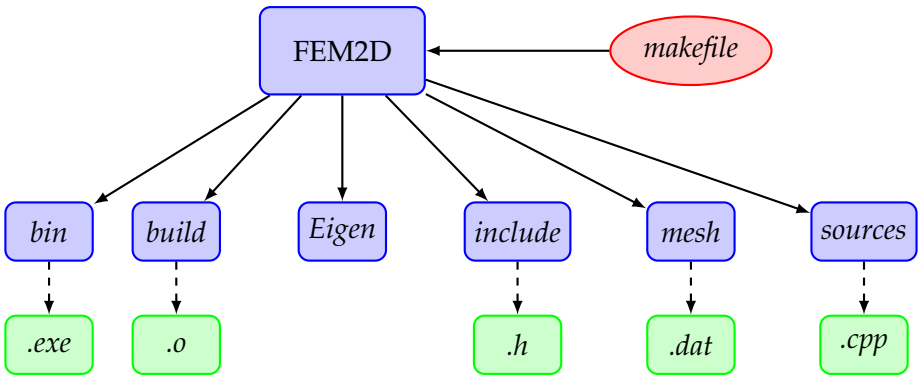

FIGURA 4.2: Organigrama de FEM2D.

A continuación se detalla el contenido del código.

4.5.1. Archivo *makefile* y *header*

El archivo *makefile* se encarga de la correcta compilación de los archivos *.cpp* generando el correspondiente archivo objeto *.o* para el ensamblaje final en un archivo ejecutable *.exe*. Todo ello, se hace por medio de declaraciones. Para su ejecución es necesario ubicarse en el directorio de la carpeta raíz y teclear en el terminal `make`. Para limpiar los archivos generados utilizar el comando `make clean`.

El archivo *header.h* incluye las cabeceras comunes necesarias para la ejecución del código.

4.5.2. Archivo *main*

El archivo *main.cpp* se encarga de la llamada a las funciones principales necesarias para la resolución del código. Estas son la lectura del mallado y llamada a las funciones correspondientes al tipo de malla seleccionado. Adicionalmente, se ha añadido una función que genera un archivo de registro y el cálculo del tiempo de ejecución.

4.5.3. Lectura de datos

El archivo *data.h* se encarga de la lectura de los datos del mallado a partir de un archivo *datos.dat* que indica el nombre de tres archivos derivados:

- `mesh/mesh_node.dat`: contiene las coordenadas de cada nodo del mallado.
- `mesh/mesh_tnode.dat`: contiene los índices de los vértices de cada elemento P_1.
- `mesh/mesh_bnd_node.dat`: contiene los índices de los nodos en la frontera.

Adicionalmente, incluye dos funciones relevantes:

- `void save(T &m)`: método que almacena una variable tipo `T`[3] (en nuestro caso `MatrixXf` o `VectorXf`) y la exporta en un fichero *output.txt*.
- `double f(T &x, T &y)`: función que devuelve el valor de la función $f(x, y)$, ver Ec. 4.2 en función a dos parámetros de entrada tipo `T`.

4.5.4. Clase *point*

La clase *point* contiene los siguientes atributos y métodos:

- Atributos:
  - `double x`: coordenada x.
  - `double y`: coordenada y.
  - `int ind`: índice del nodo,
  - `int bound`: tipo de condición de contorno, 0 ó 1, nodo interior y nodo frontera con condición tipo Dirichlet, respectivamente.
  - `double sol`: solución nodal $u(x_i, y_i)$ en ese punto.
- Constructores:

[3] Plantilla (*templates*).

- `point()`: objeto tipo *point* sin parámetros de entrada estableciendo el punto centrado en el origen, es decir, x e y igual a 0, e indicando nodo interior, es decir, `bound` = 0 (**por defecto**).
- `point(const double& x,const double& y)`: objeto tipo *point* pasándose los parámetros.
- `point(const point* other)`: objeto tipo *point* el cual recibe otro objeto de tipo *point*.
- `virtual point()`: destructor *point*. `virtual` es necesario para eliminar de forma secuencial la clase heredada y no darse un *memory leak*.

- Métodos:
  - `double getx()` y `double gety()`: permiten obtener el valor de las coordenadas x e y, respectivamente.
  - `void setx(const double& x)` y `void sety(const double& y)`: permiten asignar un valor de las coordenadas x e y, respectivamente.
  - `void setxy(const double& x, const double& y)`: permite asignar un valor de las coordenadas x e y, respectivamente.
  - `void move(const double& x,const double& y)` y `void move(const point* other)`: dado un punto $p(x, y)$ mediante `p.move(a,b)` traslada las coordenadas a $p(x + a, y + b)$. Del mismo modo, para el caso de pasar un *point* al *point*, es decir, `p.move(p2)`.
  - `print_point`: imprime por pantalla las coordenadas del objeto *point*.
  - `friend double dot_escalar(point v1,point v2)`: calcula el producto escalar de dos vectores, definidos como objetos de esta clase (es necesario para calcular el producto de gradientes). *friend* inhibe el sistema de protección.

4.5.5. Clase *pelfin*

La clase *pelfin* contiene los siguientes atributos y métodos:

- Atributos:
  - `int N`: número de vértices.
  - `double area`: área del elemento finito P_1.
  - `point** V`: puntero de puntero de tipo *point*.

- Constructores:
  - `virtual P1()`: objeto tipo *P1* sin parámetros de entrada dejando el puntero doble de tipo *point* de dimensión tres sin apuntar (**por defecto**).
  - `P1()`: destructor *P1*. Al ser heredado de *point* virtual también se hereda y por eso no es necesario incluirlo.

- Métodos:
  - `void print_finel()`: imprime por pantalla los nodos del elemento almacenados en un puntero tipo *point*. Adicionalmente, indica el área del elemento.

- `double solve_area()`: calcula el área del triángulo (elemento $P1$) a partir de las ecuaciones 4.20 y 4.21. Es un método privado.
- `void connect_area()`: asigna el área del elemento al atributo `area`. Método público: tiene acceso al entorno privado.
- `double phi(double &x, double &y)`: método que calcula la función de base a partir de la ecuación 4.22.
- `point gradbase(int &i)`: método que calcula el gradiente de la función de base a partir de la ecuación 4.23, en función del valor $i = 0, 1, 2$ (téngase en cuenta el desfase unitario). Devuelve un *point* necesario para realizar el producto escalar de la matriz local.
- `void solve_matrix_local(MatrixXf &A)`: método que calcula la matriz local (9 componentes) a partir de la ecuación 4.19 y ensambla dicha contribución en la matriz global recibida por referencia.

4.5.6. Clase *mesh*

La clase *point* contiene los siguientes atributos y métodos:

- Atributos:
  - `friend class data`: suprime el sistema de protección de la clase *data* permitiendo su acceso a través de la clase *mesh*. Necesario para el acceso a los datos del mallado.
  - `static int Nnodes`: número de nodos del mallado.
  - `static int Nfinel`: número de elementos del mallado.
  - `vector<point*>Lnode`: Lista de nodos. Es un contenedor tipo vector de un puntero de *point*.
  - `list<T>Lfinel`: lista de elementos tipo `T` (depende del tipo de elemento utilizado en la discretización).
  - `double bcDirc`: valor de la condición de contorno tipo Dirichlet (condición esencial).
  - `MatrixXf A`: llamada al constructor de la clase Eigen que declara una matriz de dimensión sin fijar y sin inicializar. Posteriormente, se indica la dimensión a través del comando `A.resize(-,-)` en el cálculo de la matriz global. Se inicializa a cero mediante `MatrixXf::Zero(-,-)`. `-` es el valor a rellenar con la dimensión deseada, es decir, el número de nodos del mallado.
  - `VectorXf b`: llamada al constructor de la clase Eigen que declara un vector de dimensión sin fijar y sin inicializar. Posteriormente, se indica la dimensión a través del comando `b.resize(-)` en el cálculo de la matriz global. Se inicializa a cero mediante `VectorXf::Zero(-)`. `-` es el valor a rellenar con la dimensión deseada, es decir, el número de nodos del mallado.
- Constructores:
  - `Mfinel()`: objeto tipo *mesh* el cual recibe un contenedor tipo vector de punteros de objeto tipo `T`, donde `T` depende del elemento finito utilizado. Esta implementación se ha llevado a cabo mediante plantillas (*templates*). Adicionalmente, inicializa el contador de nodos y de elementos tipo `T` a 0, y establece a la condición de contorno tipo Dirichlet un valor nulo.

- `Mfinel()`: destructor *Mfinel*. Al ser heredado de `<T>` virtual también se hereda y por eso no es necesario incluirlo.

- Métodos:

  - `void fill_mesh`: método que recibe una variable tipo `data` por referencia con la que se encarga de rellenar la lista de nodos y elementos, e invoca la asignación del área al elemento en cuestión. Al final imprime el área total de la malla con el fin de comprobar el correcto funcionamiento del proceso. Por cada línea de dato leída crea un elemento tipo `T` temporal (generando en cadena un vector de punteros de `point`).
  - `void print_nodes`: método que imprime los nodos de cada elemento del mallado. Es necesario declarar un iterador tipo `vector<point*>` para recorrer la lista de nodos.
  - `void print_elements`: método que imprime los elementos del mallado. Es necesario declarar un iterador tipo `typename list<T>` para recorrer la lista de elementos. El sufijo `typename` es neceario para la correcta identificación de la plantilla.
  - `void build_matrix_global`: método encargado de realizar las invocaciones necesarias para rellenar el sistema lineal de ecuaciones, A y b, mediante las expresiones aproximadas de éstas, ecuaciones 4.15 y 4.24, respectivamente. Dado que la función es externa a la anterior es necesario declarar nuevamente el iterador tipo `typename list<T>`. Este método también impone las condiciones de bloqueo para la declaración de los valores de la solución $u(x, y)$ en la frontera.
  - `void solve`: método que invoca al método `build_matrix_global` que calcula A y b, actualizando sus valores en los atributos de la clase para posteriormente resolverlos mediante el método QR a través de la clase *Eigen*. Dado que no se ha creado un atributo que almacene la solución de manera directa en un vector, éste es creado mediante `VectorXf` cuya dimensión es el número de nodos del mallado. Por último, invoca a la función `save` que guarda los resultados en un archivo de texto *output.txt*

4.5.7. Aclaración

- En el programa se incluye el archivo correspondiente a la resolución de elementos Lagrange Q_1. La implementación del código no está finalizada y aún no puede utilizarse.

4.6. Resultados

En la figura 4.3 se muestra la solución exacta y aproximada por el método de elementos finitos tipo P_1, respectivamente. Se han obtenido mediante Matlab y FEM2D, respectivamente.

El error cometido se ha obtenido mediante las siguientes expresiones

$$\epsilon = \frac{||u - u_h||_2}{||u||_2}, \tag{4.29}$$

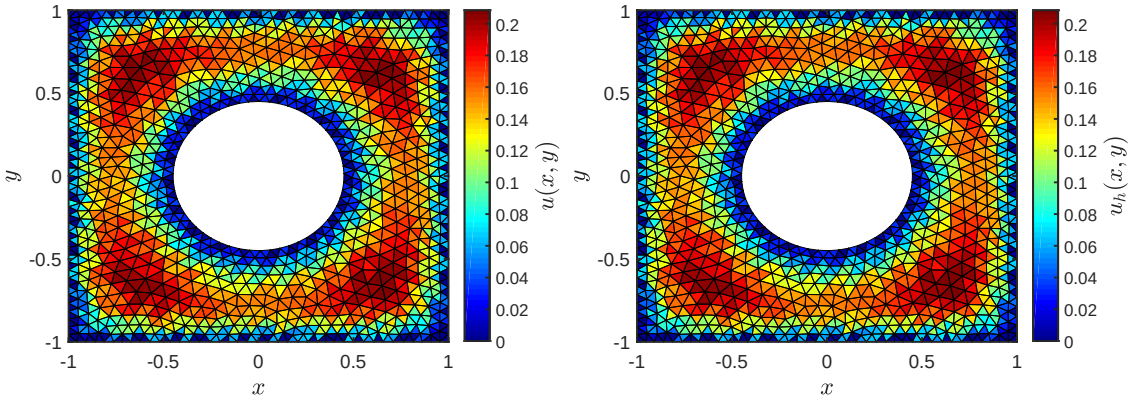

FIGURA 4.3: Solución: (a) exacta, (b) aproximada con 990 nodos y 1765 elementos Lagrange P_1.

$$\epsilon_{abs} = ||u - u_h||_2, \tag{4.30}$$

correspondiendo con el error relativo y absoluto, respectivamente. El orden del error para la malla utilizada es del orden $\epsilon \sim \mathcal{O}(10^{-1})$.

4.7. Test

Para llevar a cabo el test del programa en primer lugar se comprobó que los resultados eran acordes a la solución exacta, tal y como se ha verificado con los resultados anteriores mostrados. Posteriormente, para llevar a cabo una depuración del código se implementó una función que genera un archivo de registro (historial) *log.txt* con el que se pudieron hallar varios errores en el código[4]:

- El número de elementos `Nfinel` estaba introducido dentro del bucle de nodos del triángulo, lo que generaba el triple número de elementos finitos P_1.
- Al recorrer la asignación de conexiones del elemento: la dirección de memoria del nodo i del elemento j, con $i = 1, 2, 3$ y $j = 1, \ldots, 1765$ estaba asociada a un nodo erróneo ya que no se tuvo en cuenta el desfase unitario entre la lista y el número de éstos en el bucle.

El resto del código se verificó que cumplía su función de manera satisfactoria, por ejemplo, la correcta liberación de memoria.

[4]Obviamente, estos errores han sido corregidos.

4.8. Conclusiones y trabajos futuros

En la elaboración de este proyecto se ha podido visualizar de primera mano la versatilidad que proporciona la propia generación de un código encargado de resolver un problema numérico mediante el método de elementos finitos, recibiendo los datos de un mallado no estructurado. Son muchos los campos de mejora del código, destacando los siguientes:

- Incorporación de otro tipo de elementos finitos, por ejemplo, elementos Lagrange Q_1.
- Aumentar la precisión del método mediante otras reglas de cuadratura más precisas (implicando mayor complejidad).
- Estudiar la respuesta del código ante mallados más complejos.

Bibliografía

[1] A. Cardona y V. Fachinotti. *Introdución al Método de los Elementos Finitos*. Cimec (UNL/Conicet), Santa Fe, Argentina, 2014.

[2] Fco. J. Ceballos. *Programación orientada a objetos con C++*. RA-MA Editorial, 2007.

[3] V. Dolejsí. *Finite element methods: implementations*. "[Online; accessed 14-Apr-2018]". 2011. URL: "http://www.karlin.mff.cuni.cz/~dolejsi/Vyuka/FEM-implement.pdf".

[4] M. Generosa Fernández y G. García Lomba. *Métodos Numéricos para Ecuaciones en Derivadas Parciales*. Máster en Matemática Industrial, 2017.

[5] A. M. Ferreiro Ferreiro y J. A. García Rodrigez. *Programación en C++*. 2018.

[6] Gaël Guennebaud, Benoît Jacob y col. *Eigen v3*. "[Online; accessed 14-Apr-2018]". 2010. URL: "http://www.eigen.tuxfamily.org".

[7] B. C. Iglesias, J. D. Castrillo y F. V. Mérida. *Ecuaciones en Derivadas Parciales*. Universidad de Vigo y Universidad Politécnica de Madrid, 2017.

[8] M. G. Mañes. *Problemas de Ecuaciones en derivadas parciales. Métodos numéricos de resolución analíticos y numéricos*. E.T.S. de Ingenieros de Caminos, Madrid, 1993.