

UN MÉTODO DE ELEMENTOS FINITOS
PRECONDICIONADO PARA LA ECUACIÓN DE
STOKES 2D

LUIS FERNANDO MEJÍA RODRÍGUEZ



UNIVERSIDAD DEL VALLE
FACULTAD DE CIENCIAS
PROGRAMA ACADÉMICO DE MAESTRÍA EN CIENCIAS MATEMÁTICAS
SANTIAGO DE CALI
2014

UN MÉTODO DE ELEMENTOS FINITOS
PRECONDICIONADO PARA LA ECUACIÓN DE
STOKES 2D

LUIS FERNANDO MEJÍA RODRÍGUEZ

Trabajo de grado presentado como requisito para optar al título de Magíster en
Ciencias Matemáticas

Director
JAIRO DUQUE, Dr. rer. nat.

UNIVERSIDAD DEL VALLE
FACULTAD DE CIENCIAS
PROGRAMA ACADÉMICO DE MAESTRÍA EN CIENCIAS MATEMÁTICAS
SANTIAGO DE CALI

2014

UNIVERSIDAD DEL VALLE
FACULTAD DE CIENCIAS
PROGRAMA ACADÉMICO DE MAESTRÍA EN CIENCIAS MATEMÁTICAS

LUIS FERNANDO MEJÍA RODRÍGUEZ, 1982

UN MÉTODO DE ELEMENTOS FINITOS PRECONDICIONADO PARA LA
ECUACIÓN DE STOKES 2D

PALABRAS CLAVES: EL PROBLEMA DE STOKES, MÉTODO DE ELEMENTOS
FINITOS MIXTOS,
FORMULACIÓN DÉBIL, PRECONDICIONADOR.

SANTIAGO DE CALI

2014

Nota de Aceptación

Jairo Duque, Dr. rer. nat.
Director de Tesis

Jurado

Jurado

Santiago de Cali, Junio de 2014

Agradecimientos

Agradezco a todas las personas que hicieron parte en mi formación académica y personal; sin ninguna duda no estaría aquí de no ser por ellos, agradezco muy especialmente a mi familia que durante este largo camino fue mi apoyo incondicional, a mi mamá y papá por su amor y educación, a mis queridos hermanos por no dejarme sólo y ser una voz de ánimo, a mis amigos que durante toda mi formación me dieron la mano y palabras de fortaleza para enfrentar tantos problemas, a mi abuela que aunque ya no está con nosotros nunca saldrá de nuestros corazones, sus palabras y su sonrisa vivirán por siempre, a todos los profesores que participaron de mi formación por su gran esfuerzo. Un agradecimiento muy especial a el profesor Jairo Duque por su ayuda, por su paciencia, sus enseñanzas, consejos y amistad.

Resumen

En este trabajo se considera el problema de Stokes dos dimensional para fluidos incompresibles en el dominio acotado $\Omega = [0, 1] \times [0, 1]$. Se implementa un método de elementos finitos mixtos para aproximar numéricamente la solución al problema de Stokes en sus variables originales (u, p) por $(\tilde{u}, \tilde{p})$. De este modo, nos encontramos con un sistema de la forma

$$\begin{pmatrix} M & B^t \\ B & 0 \end{pmatrix} \begin{pmatrix} \tilde{u} \\ \tilde{p} \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix},$$

donde $M \in \mathbb{R}^{n \times n}$ es simétrica y definida positiva y $B \in \mathbb{R}^{m \times n}$ de rango completo. En este trabajo proponemos el uso de un preconditionador P para resolver el sistema lineal. El preconditionador P es inspirado por uno propuesto por M. Murphy, G. Golub, A. Wathen.

$$P = \begin{pmatrix} \widetilde{M} & 0 \\ 0 & \widetilde{S} \end{pmatrix}$$

donde $\widetilde{S}$ y $\widetilde{M}$ están definidas por:

$$\widetilde{S} = [B^t \widetilde{M}^{-1} B].$$

$$\widetilde{M} = (\text{diag} (M))$$

En este trabajo no se calcula la inversa de ninguna matriz, se implementa un algoritmo para resolver el producto de ellas por vectores.

Índice general

Introducción	VIII
1. El Modelo matemático de Stokes	1
2. Formulación variacional	3
3. Existencia y Unicidad	7
3.1. Existencia y unicidad para el problema de Stokes	8
3.2. Existencia y unicidad para el problema de Stokes discreto	11
4. Precondicionamiento	13
5. Fem Mixto e Implementación numérica Deal.II	17
6. Experimentos Numéricos	31
6.1. Test de Bercovier–Engelman (condición Dirichlet nula)	31
6.2. Test con condiciones de frontera no nula	40
7. Conclusiones	49
A. Stokes2D	52

Introducción

El flujo de fluidos viscosos incompresibles es clásicamente modelado por el problema de Stokes. En este trabajo se implementará un método de elementos finitos para aproximar la solución del problema de Stokes en sus variables originales.

Sea Ω un subconjunto abierto y acotado de $\mathbb{R}^2$ con frontera poligonal $\partial\Omega = \Gamma$ y sean $u: \Omega \rightarrow \mathbb{R}^2$ el campo de velocidades y $p: \Omega \rightarrow \mathbb{R}$ la presión del fluido en cada punto del dominio; el problema de Stokes en forma vectorial está dado por:

$$\begin{aligned} -\Delta u + \nabla p &= f \text{ en } \Omega \\ \operatorname{div}(u) &= 0 \text{ en } \Omega \\ u &= u_0 \text{ sobre } \Gamma \end{aligned} \tag{1}$$

Aquí f es la fuerza que actúa sobre el fluido, por ejemplo f puede ser la gravedad y u_0 es un campo de velocidades conocido [6]. Cuando se plantea la formulación variacional del problema de Stokes queremos encontrar $u \in V$ y $p \in X$ tales que

$$\begin{aligned} a(u, v) + b(p, v) &= f(v) \quad \forall v \in V \\ b(u, q) &= 0 \quad \forall q \in X, \end{aligned} \tag{2}$$

donde $a(\cdot, \cdot)$ y $b(\cdot, \cdot)$ son formas bilineales y V, X son los espacios funcionales donde se busca la solución débil. Se propone en el trabajo aproximar (u, p) con $(\tilde{u}, \tilde{p})$ mediante elementos finitos; así se obtiene un sistema lineal

$$\begin{pmatrix} M & B^t \\ B & 0 \end{pmatrix} \begin{pmatrix} \tilde{u} \\ \tilde{p} \end{pmatrix} = \begin{pmatrix} \tilde{f} \\ 0 \end{pmatrix}, \tag{3}$$

donde $M \in \mathbb{R}^{n \times n}$ es una matriz simétrica definida positiva y $B \in \mathbb{R}^{m \times n}$ de rango completo. Al implementar un método de elementos finitos para el problema de Stokes usando elementos lineales no hay convergencia [22], por lo tanto proponemos implemen-

tar un método de elementos finitos mixtos para aproximar la velocidad u y la presión p . Para la velocidad se usan elementos lagrangianos de grado tres y para la presión elementos lagrangianos de grado dos. Se propone el uso de estos elementos mixtos para que el método resultante sea estable y puedan converger las soluciones aproximadas a las soluciones exactas [20].

El bloque de ceros en el sistema (3) proviene de la restricción $\text{div}(u) = 0$ la cual introduce en la discretización del problema dificultades numéricas ya que métodos iterativos para solucionar el sistema lineal (3) como el Gradiente conjugado (CG) no funcionan [21], más aún, el bloque de ceros que aparece en la discretización del problema introduce problemas para el uso de preconditionadores clásicos como el de Jacobi o Gauss-Seidel. En vista de ésto, usaremos un esquema diferente para resolver el problema discreto.

La matriz M del problema discreto (3) es invertible ya que M es definida positiva. Por eliminación por bloques podemos desacoplar este sistema de la siguiente manera (multiplicar la primera fila del sistema por BM^{-1} y luego restar la segunda fila de la misma):

$$BM^{-1}B^t\tilde{p} = BM^{-1}\tilde{f}, \quad (4)$$

$$M\tilde{u} = \tilde{f} - B^t\tilde{p}. \quad (5)$$

donde la matriz $-BM^{-1}B^t$ se conoce como el complemento de Schur; en general tenemos la siguiente definición:

Definición 1. Supongamos que la matriz $A \in \mathbb{R}^{n \times n}$ se puede escribir de la forma

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

donde A_{11} es $r \times r$. Si se supone que A_{11} es no singular la matriz $W = A_{22} - A_{21}A_{11}^{-1}A_{12}$ se denomina el complemento de Schur de A_{11} en A .

Sea $S = BM^{-1}B^t$ simétrica y definida positiva, por lo tanto, si pudiéramos calcular S , podríamos aplicar el método del gradiente conjugado al sistema (4). Sin embargo, el cálculo de S es computacionalmente costoso además S probablemente es una matriz densa. Por otro lado, no es necesario tener S para aplicar el método del gradiente conjugado basta con tener un procedimiento que permita calcular Sv .

El objetivo del preconditionamiento es la reducción del número de iteraciones requerido para la convergencia, transformando el sistema original $Ax = b$ en un sistema $\tilde{A}x = \tilde{b}$. La

matriz $\tilde{A}$ y el vector $\tilde{b}$ se consiguen por premultiplicación por una matriz P llamada preconditionador, que debe ser fácilmente invertible [14]. Por medio del preconditionador se transforma el sistema $Ax = b$ en otro equivalente $\tilde{A}x = \tilde{b}$ (con la misma solución) con condiciones espectrales mas favorables, de tal suerte que el número de iteraciones se reduzca para la convergencia, sin incrementar significativamente la cantidad de cálculos por iteración. Para obtener la solución del sistema (4) sería de utilidad tener un preconditionador para la matriz S . En este trabajo proponemos un preconditionador para la matriz S que consiste en una aproximación de S^{-1} .

Murphy, Golub y Wather muestran en [15] un preconditionador bastante eficiente para sistemas lineales cuya matriz de coeficientes tiene la forma (3). Dicho preconditionador tiene la siguiente estructura cuando la matriz M es invertible:

$$P = \begin{pmatrix} M & 0 \\ 0 & BM^{-1}B^t \end{pmatrix}. \quad (6)$$

En este trabajo compararemos nuestro preconditionador con el propuesto por Murphy, Golub y Wather.

Capítulo 1

El Modelo matemático de Stokes

Normalmente la descripción matemática del flujo de un fluido Newtoniano se realiza con las *ecuaciones de Navier-Stokes*. En un fluido newtoniano se considera que los esfuerzos son proporcionales a la velocidad de deformación y al coeficiente de viscosidad. Mediante el reemplazo de las ecuaciones constitutivas de un fluido newtoniano en las ecuaciones de conservación (masa y momento) se obtienen las *ecuaciones de Navier-Stokes*, las cuales están dadas “sin dimensiones” por (para una completa descripción de la derivación de estas ecuaciones véase [8], [9], [4]):

$$\rho \left(\frac{\partial(u)}{\partial t} + (u \cdot \nabla)(u) \right) = -\nabla p + f + \frac{1}{\text{Re}} \Delta u, \quad (1.1)$$

las variables u y p en las *ecuaciones de Navier-Stokes* corresponden al vector de distribución de velocidades y a la presión del flujo respectivamente. El parámetros ρ corresponde a la densidad del fluido. Esta ecuación junto con la ecuación de continuidad:

$$\frac{\partial \rho}{\partial t} + \text{div}(\rho u) = 0, \quad (1.2)$$

describen el movimiento de un flujo viscoso incompresible. En el caso analizado en este trabajo se propone un flujo de fluido estacionario, así las ecuaciones (1.1) y (1.2) se reducen a

$$\rho(u \cdot \nabla u) = -\nabla p + f + \frac{1}{\text{Re}} \Delta u, \quad (1.3)$$

$$\text{div}(\rho u) = 0, \quad (1.4)$$

Aquí el número de Reynolds Re está definido como $Re = \frac{lk}{a}$, donde a es la viscosidad del fluido, l es una longitud característica que depende de la geometría del problema y k la velocidad promedio del fluido. Cuando el número de Reynolds es muy bajo, es decir, cuando la velocidad k o la longitud característica l del problema son muy bajos o la viscosidad a del fluido es alta, los términos de viscosidad $\frac{1}{Re}\Delta u$ son muy grandes y la velocidad es muy pequeña, así los términos no lineales $u \cdot \nabla u$ son despreciables frente a los términos viscosos. En el caso analizado en este trabajo, se consideran números de Re pequeños (menores que 10^3), así las ecuaciones (1.3) y (1.4) se reducen a

$$\frac{1}{Re}\Delta u - \nabla p + f = 0 \quad (1.5)$$

$$\operatorname{div}(\rho u) = 0, \quad (1.6)$$

donde f es el término proveniente de fuerzas externas que actúan sobre el fluido. Además suponiendo que ρ es constante se tiene que las ecuaciones (1.5) y (1.6) se reducen a

$$-\frac{1}{Re}\Delta u + \nabla p = f \quad (1.7)$$

$$\operatorname{div}(u) = 0. \quad (1.8)$$

Las ecuaciones (1.7) y (1.8) son conocidas como el problema de Stokes que clásicamente modelan el flujo de fluidos viscosos incompresibles. Para los ejemplos numéricos que se resolverán en este trabajo usaremos $Re = 1$. Por lo tanto sin pérdida de generalidad en lo que resta del trabajo $Re = 1$.

Capítulo 2

Formulación variacional

Sea $\Omega \subset \mathbb{R}^2$ una region abierta, acotada y simplemente conexa. Consideremos el problema de Stokes en su forma vectorial

$$\begin{aligned} -\Delta u + \nabla p &= f, \\ \operatorname{div}(u) &= 0, \end{aligned} \tag{2.1}$$

donde $u = (u_1, u_2)$ describe el campo de velocidades, p la presión y $f = (f_1, f_2)$ representa las fuerzas externas. Sea $W^{1,2}(\Omega)$ el espacio de Sobolev dado por el conjunto de funciones cuadrado integrables con derivadas hasta orden uno cuadrado integrables, véase [5], $W_0^{1,2}(\Omega) = \{u \in W^{1,2}(\Omega) : u|_{\partial\Omega} = 0\}$ el conjunto de funciones en el espacio de Sobolev $W^{1,2}(\Omega)$ que además son cero en la frontera de Ω y $L_0^2(\Omega) = \{q \in L^2(\Omega) : \int_{\Omega} q = 0\}$ el conjunto de funciones cuadrado integrables de promedio cero. (2.1) se puede escribir como un sistema de tres ecuaciones diferenciales parciales con tres incógnitas

$$\begin{aligned} -\Delta u_1 + \partial_{x_1} p &= f_1 \\ -\Delta u_2 + \partial_{x_2} p &= f_2 \\ \operatorname{div}(u) &= 0 \end{aligned} \tag{2.2}$$

Sean $v_1, v_2, q : \Omega \rightarrow \mathbb{R}$ funciones de prueba en $W_0^{1,2}(\Omega)$, $W_0^{1,2}(\Omega)$ y $L_0^2(\Omega)$, respectivamente. Multiplicando las ecuaciones del sistema (2.2) por v_1 , v_2 y q obtenemos:

$$\begin{aligned} -\Delta u_1 v_1 + \partial_{x_1} p v_1 &= f_1 v_1 \\ -\Delta u_2 v_2 + \partial_{x_2} p v_2 &= f_2 v_2 \\ \operatorname{div}(u) q &= 0 \end{aligned} \quad (2.3)$$

Ahora integrando las ecuaciones en Ω con respecto a x obtenemos

$$\begin{aligned} -\int_{\Omega} \Delta u_1 v_1 dx + \int_{\Omega} \partial_{x_1} p v_1 dx &= \int_{\Omega} f_1 v_1 dx \\ -\int_{\Omega} \Delta u_2 v_2 dx + \int_{\Omega} \partial_{x_2} p v_2 dx &= \int_{\Omega} f_2 v_2 dx \\ \int_{\Omega} \operatorname{div}(u) q dx &= 0 \end{aligned} \quad (2.4)$$

Aplicando integración por partes en los términos que contienen las derivadas de orden superior, se obtiene:

$$\begin{aligned} \int_{\Omega} \nabla u_1 \cdot \nabla v_1 dx - \int_{\partial\Omega} v_1 (\nabla u_1 \cdot \eta) ds + \int_{\Omega} \partial_{x_1} p v_1 dx &= \int_{\Omega} f_1 v_1 dx \\ \int_{\Omega} \nabla u_2 \cdot \nabla v_2 dx - \int_{\partial\Omega} v_2 (\nabla u_2 \cdot \eta) ds + \int_{\Omega} \partial_{x_2} p v_2 dx &= \int_{\Omega} f_2 v_2 dx \\ \int_{\Omega} \operatorname{div}(u) q dx &= 0 \end{aligned} \quad (2.5)$$

como v_1, v_2 son funciones de prueba en $W_0^{1,2}(\Omega)$ entonces

$$-\int_{\partial\Omega} v_1 (\nabla u_1 \cdot \eta) ds = -\int_{\partial\Omega} v_2 (\nabla u_2 \cdot \eta) ds = 0.$$

Por lo tanto (2.5) se puede escribir como

$$\begin{aligned} \int_{\Omega} \nabla u_1 \cdot \nabla v_1 dx + \int_{\Omega} \partial_{x_1} p v_1 dx &= \int_{\Omega} f_1 v_1 dx \\ \int_{\Omega} \nabla u_2 \cdot \nabla v_2 dx + \int_{\Omega} \partial_{x_2} p v_2 dx &= \int_{\Omega} f_2 v_2 dx \\ \int_{\Omega} \operatorname{div}(u) q dx &= 0. \end{aligned} \quad (2.6)$$

Usamos el teorema de Gauss para reescribir $\int_{\Omega} \partial_{x_i} p v_i dx$.

$$\int_{\Omega} \partial_{x_i} p v_i dx = \int_{\partial\Omega} p \cdot (v_i \eta) ds - \int_{\Omega} p \partial_{x_i} v_i dx.$$

Así tenemos que (2.6) se cambian por

$$\begin{aligned} \int_{\Omega} \nabla u_1 \cdot \nabla v_1 dx - \int_{\Omega} p \partial_{x_1} v_1 dx &= \int_{\Omega} f_1 v_1 dx \\ \int_{\Omega} \nabla u_2 \cdot \nabla v_2 dx - \int_{\Omega} p \partial_{x_2} v_2 dx &= \int_{\Omega} f_2 v_2 dx \\ \int_{\Omega} \operatorname{div}(u) q dx &= 0. \end{aligned} \quad (2.7)$$

Sumando las ecuaciones en (2.7) tenemos:

$$\begin{aligned} \sum_i \int_{\Omega} \nabla u_i \cdot \nabla v_i dx - \int_{\Omega} p \operatorname{div} v dx &= \sum_i \int_{\Omega} f_i v_i dx \quad \forall (v_1, v_2) \in [W_0^{1,2}(\Omega)]^2 \\ \int_{\Omega} \operatorname{div}(u) q dx &= 0, \quad \forall q \in L_0^2(\Omega) \end{aligned} \quad (2.8)$$

Definimos

$$\begin{aligned} a : [W_0^{1,2}(\Omega)]^2 \times [W_0^{1,2}(\Omega)]^2 &\rightarrow \mathbb{R} \\ a(u, v) &= \sum_{i=1}^2 \int_{\Omega} \nabla u_i \cdot \nabla v_i dx = \int_{\Omega} \nabla u \cdot \nabla v dx \end{aligned}$$

$$b : L_0^2(\Omega) \times [W_0^{1,2}(\Omega)]^2 \rightarrow \mathbb{R}$$

$$b(p, v) = - \int_{\Omega} p \operatorname{div} v dx,$$

claramente $a(\cdot, \cdot)$ y $b(\cdot, \cdot)$ son formas bilineales.

Formulación variacional: Sea $V = [W_0^{1,2}(\Omega)]^2$ y $X = L_0^2(\Omega)$. Encontrar $u \in V$ y $p \in X$ tales que:

$$\begin{aligned} a(u, v) + b(p, v) &= f(v) \quad \forall v \in V \\ b(q, u) &= 0 \quad \forall q \in X, \end{aligned} \tag{2.9}$$

donde

$$a(u, v) = \sum_{i=1}^2 \int_{\Omega} \nabla u_i \cdot \nabla v_i dx,$$

$$b(p, v) = - \int_{\Omega} p \operatorname{div} v dx,$$

$$f(v) = \sum_{i=1}^2 \int_{\Omega} f_i v_i dx.$$

Capítulo 3

Existencia y Unicidad

Se quiere probar la existencia y unicidad de las soluciones al problema (2.9); para ello se hace uso de la teoría general del métodos de elementos finitos mixtos. Los teoremas y demostraciones a continuación son bien conocidos de la literatura especializada [10], [7], [18], [3].

El problema (2.9) es un caso particular de los llamados problemas de punto silla los cuales tiene la siguiente estructura: Sean X y V dos espacios de Hilbert, sean además $a(u, v) : V \times V \rightarrow \mathbb{R}$ y $b(p, v) : X \times V \rightarrow \mathbb{R}$ dos formas bilineales continuas y $f \in V'$, $g \in X'$ funcionales lineales, entonces el problema de punto silla consiste en encontrar $u \in V$ y $p \in X$ tales que

$$\begin{aligned} a(u, v) + b(p, v) &= f(v) \quad \forall v \in V \\ b(q, u) &= g(q) \quad \forall q \in X. \end{aligned} \tag{3.1}$$

Para el problema (3.1) se puede demostrar la existencia y unicidad de la solución cuando la forma bilineal $a(\cdot, \cdot)$ es coerciva y $b(\cdot, \cdot)$ satisface las llamadas condiciones ínf – sup. Este resultado es conocido como el teorema de Brezzi que se cita a continuación

Teorema 1. *Sean X y V dos espacios de Hilbert, sean además $a(u, v) : V \times V \rightarrow \mathbb{R}$ y $b(p, v) : X \times V \rightarrow \mathbb{R}$ dos formas bilineales continuas. Sea*

$$K = \{v \in V, \forall p \in X, b(p, v) = 0\}.$$

Suponga que $a(\cdot, \cdot)$ es coerciva en K es decir:

$$\exists \alpha > 0, \forall v \in K, a(v, v) \geq \alpha \|v\|_V^2$$

y $b(\cdot, \cdot)$ satisface la llamada condición $\inf \sup$, es decir,

$$\exists \beta > 0, \inf_{p \in X} \sup_{v \in V} \frac{b(p, v)}{\|p\|_X \|v\|_V} \geq \beta$$

entonces el problema (3.1) tiene única solución

Definición 2. Sea $\Omega \subset \mathbb{R}^n$ un conjunto abierto, $k \geq 1$ un número de entero y $p \in [1, \infty)$. Se define

$$W^{k,p}(\Omega) = \{f \in L^p(\Omega) \mid D^\alpha f \in L^p(\Omega) \text{ para todo } |\alpha| \leq k\}.$$

Para cada $1 \leq p < \infty$ la norma $\|\cdot\|_{k,p}$ es definido como

$$\|\cdot\|_{k,p} = \left(\int_{\Omega} \sum_{|\alpha| \leq k} |D^\alpha f|^p dx \right)^{\frac{1}{p}} = \left(\sum_{|\alpha| \leq k} \|D^\alpha f\|_p^p \right)^{\frac{1}{p}}$$

Donde $\|\cdot\|_p$ es la norma en L^p y $\|Df\|_p = \|\nabla f\|_p = \|\partial_{x_1} f\|_p + \|\partial_{x_2} f\|_p$, para el caso en que $p = 2$ se tiene $W^{k,p}(\Omega) = H^k(\Omega)$ un espacio de Hilbert dotado con la norma $\|\cdot\|_{k,2}$

3.1. Existencia y unicidad para el problema de Stokes

Para demostrar la existencia y unicidad del problema variacional (2.9) probaremos que las formas bilineales cumplen las condiciones del Teorema 1.

Comenzamos probando que $a(\cdot, \cdot)$ es bilineal y acotada.

Claramente $a(\cdot, \cdot)$ es bilineal ya que la integral y la derivada son lineales.

$$a(u, v) = \sum_{i=1}^2 \int_{\Omega} \nabla u_i \cdot \nabla v_i dx$$

Entonces

$$|a(u, v)| = \left| \int_{\Omega} \nabla u \cdot \nabla v dx \right| = \left| \sum_{i=1}^2 \int_{\Omega} \nabla u_i \cdot \nabla v_i dx \right|$$

Por desigualdad triangular

$$\left| \sum_{i=1}^2 \int_{\Omega} \nabla u_i \cdot \nabla v_i dx \right| \leq \left| \int_{\Omega} \nabla u_1 \cdot \nabla v_1 dx \right| + \left| \int_{\Omega} \nabla u_2 \cdot \nabla v_2 dx \right|$$

por propiedades de la integral

$$\begin{aligned} & \left| \int_{\Omega} \nabla u_1 \cdot \nabla v_1 dx \right| + \left| \int_{\Omega} \nabla u_2 \cdot \nabla v_2 dx \right| \leq \int_{\Omega} |\nabla u_1 \cdot \nabla v_1| dx + \int_{\Omega} |\nabla u_2 \cdot \nabla v_2| dx = \\ & \int_{\Omega} |(\partial_{x_1} u_1, \partial_{x_2} u_1) \cdot (\partial_{x_1} v_1, \partial_{x_2} v_1)| dx + \int_{\Omega} |(\partial_{x_1} u_2, \partial_{x_2} u_2) \cdot (\partial_{x_1} v_2, \partial_{x_2} v_2)| dx = \\ & \int_{\Omega} |\partial_{x_1} u_1 \partial_{x_1} v_1 + \partial_{x_2} u_1 \partial_{x_2} v_1| dx + \int_{\Omega} |\partial_{x_1} u_2 \partial_{x_1} v_2 + \partial_{x_2} u_2 \partial_{x_2} v_2| dx \leq \\ & \int_{\Omega} |\partial_{x_1} u_1 \partial_{x_1} v_1| dx + \int_{\Omega} |\partial_{x_2} u_1 \partial_{x_2} v_1| dx + \int_{\Omega} |\partial_{x_1} u_2 \partial_{x_1} v_2| dx + \int_{\Omega} |\partial_{x_2} u_2 \partial_{x_2} v_2| dx \leq \\ & \|\partial_{x_1} u_1\|_2 \|\partial_{x_1} v_1\|_2 + \|\partial_{x_2} u_1\|_2 \|\partial_{x_2} v_1\|_2 + \|\partial_{x_1} u_2\|_2 \|\partial_{x_1} v_2\|_2 + \|\partial_{x_2} u_2\|_2 \|\partial_{x_2} v_2\|_2 \leq \\ & (\|\partial_{x_1} u_1\|_2 + \|\partial_{x_2} u_1\|_2)(\|\partial_{x_1} v_1\|_2 + \|\partial_{x_2} v_1\|_2) + (\|\partial_{x_1} u_2\|_2 + \|\partial_{x_2} u_2\|_2)(\|\partial_{x_1} v_2\|_2 + \|\partial_{x_2} v_2\|_2) \leq \\ & \|Du_1\|_2 \|Dv_1\|_2 + \|Du_1\|_2 \|Dv_2\|_2 + \|Du_2\|_2 \|Dv_1\|_2 + \|Du_2\|_2 \|Dv_2\|_2 \leq \\ & (\|u_1\|_2^2 + \|Du_1\|_2^2)^{\frac{1}{2}} (\|v_1\|_2^2 + \|Dv_1\|_2^2)^{\frac{1}{2}} + (\|u_1\|_2^2 + \|Du_1\|_2^2)^{\frac{1}{2}} (\|v_2\|_2^2 + \|Dv_2\|_2^2)^{\frac{1}{2}} + \\ & (\|u_2\|_2^2 + \|Du_2\|_2^2)^{\frac{1}{2}} (\|v_1\|_2^2 + \|Dv_1\|_2^2)^{\frac{1}{2}} + (\|u_2\|_2^2 + \|Du_2\|_2^2)^{\frac{1}{2}} (\|v_2\|_2^2 + \|Dv_2\|_2^2)^{\frac{1}{2}} = \\ & \left((\|u_1\|_2^2 + \|Du_1\|_2^2)^{\frac{1}{2}} + (\|u_2\|_2^2 + \|Du_2\|_2^2)^{\frac{1}{2}} \right) \left((\|v_1\|_2^2 + \|Dv_1\|_2^2)^{\frac{1}{2}} + (\|v_2\|_2^2 + \|Dv_2\|_2^2)^{\frac{1}{2}} \right) = \\ & \left(\|u_1\|_{W_0^{1,2}} + \|u_2\|_{W_0^{1,2}} \right) \left(\|v_1\|_{W_0^{1,2}} + \|v_2\|_{W_0^{1,2}} \right) = \\ & \|u\|_V \|v\|_V \end{aligned}$$

Por tanto $|a(u, v)| \leq \|u\|_V \|v\|_V$ es decir $a(\cdot, \cdot)$ es acotada y por tanto continua [20], [17].

A continuación se muestra que $a(\cdot, \cdot)$ es coerciva en K , es decir que existe $\alpha > 0$ tal que $\alpha \|u\|_V^2 \leq a(u, u)$ para todo $u \in K = \{v \in V : b(p, v) = 0, \forall p \in X\}$.

Recuerde que de la desigualdad de Poincaré afirma que para todo $u_i \in H_0^1(\Omega)$ existe una constante C positiva talque:

$$\|u_i\|_2 \leq C \|\nabla u_i\|_2 \quad (3.2)$$

de la desigualdad de Poincaré (3.2) y de $\|u_i\|_{H_0^1(\Omega)}^2 = \|u_i\|_2^2 + \|Du_i\|_2^2$ se sigue que

$$\|u_i\|_{H_0^1(\Omega)}^2 = \|u_i\|_2^2 + \|Du_i\|_2^2 \leq C\|\nabla u_i\|_2^2 + \|Du_i\|_2^2 = (C+1)\|\nabla u_i\|_2^2,$$

luego

$$\alpha\|u_i\|_{H_0^1(\Omega)}^2 \leq \|\nabla u_i\|_2^2 = (\nabla u_i \cdot \nabla u_i)_2 = \int_{\Omega} \nabla u_i \cdot \nabla u_i dx$$

con $\alpha = \frac{1}{c+1}$. Por otra parte

$$\|u\|_{[H_0^1(\Omega)]^2} = \|u_1\|_{H_0^1(\Omega)} + \|u_2\|_{H_0^1(\Omega)} \leq \alpha_1 \left(\int_{\Omega} \nabla u_1 \cdot \nabla u_1 dx \right)^{\frac{1}{2}} + \alpha_2 \left(\int_{\Omega} \nabla u_2 \cdot \nabla u_2 dx \right)^{\frac{1}{2}}$$

luego

$$\|u\|_{[H_0^1(\Omega)]^2} = \alpha_1 \|\nabla u_1\|_2 + \alpha_2 \|\nabla u_2\|_2 \leq \alpha_3 (\|\nabla u_1\|_2 + \|\nabla u_2\|_2)$$

con $\alpha_3 = \max(\alpha_1, \alpha_2)$ entonces

$$\|u\|_{[H_0^1(\Omega)]^2} \leq \alpha_3 ((\nabla u_1 \cdot \nabla u_1)_2^{\frac{1}{2}} + (\nabla u_2 \cdot \nabla u_2)_2^{\frac{1}{2}}) \leq \alpha_4 ((\nabla u_1 \cdot \nabla u_1)_2 + (\nabla u_2 \cdot \nabla u_2)_2)^{\frac{1}{2}} = \alpha_4 (a(u, u))^{\frac{1}{2}}$$

con $\alpha_4 = 2\alpha_3$ por lo tanto

$$\alpha_5 \|u\|_{[H_0^1(\Omega)]^2}^2 \leq a(u, u)$$

en consecuencia, la forma bilineal $a(\cdot, \cdot)$ es elíptica en V y por lo tanto es coerciva en K , [10]

Por último se demuestra a continuación que $b(\cdot, \cdot)$ cumple la condición ínf – sup para ello se recurre al resultado [1] sobre dominios Ω abiertos con frontera Lipschitz regular: Para $p \in L_0^2(\Omega)$ existe una constante $C > 0$ talque

$$C\|p\|_{L^2(\Omega)} \leq \|\nabla p\|_{H^{-1}(\Omega)}, \quad (3.3)$$

donde $\|\cdot\|_{H^{-1}}$ es la norma del dual topológico de $H_0^1(\Omega)$. La condición ínf – sup es equivalente a

$$C\|p\|_{L^2(\Omega)} \leq \sup_{v \in V} \frac{b(p, v)}{\|v\|_V}$$

para todo $p \in L_0^2(\Omega)$, es facil ver que la desigualdad anterior es equivalente a

$$C\|p\|_{L^2(\Omega)} \leq \sum_{i=1}^2 \sup_{v_i \in H_0^1(\Omega)} \frac{(p, \partial_{x_i} v_i)}{\|v_i\|_{H^1(\Omega)}}$$

para todo $p \in L_0^2(\Omega)$, la norma $H^{-1}(\Omega)$ es dada por

$$\|f\|_{H^{-1}(\Omega)} = \sup_{v_i \in H_0^1(\Omega)} \frac{(f, v_i)}{\|v_i\|_{H^1(\Omega)}}$$

Note que si $p \in L^2(\Omega)$ entonces $\partial_{x_i} p \in H^{-1}(\Omega)$ por lo tanto tendríamos que

$$C\|p\|_{L^2(\Omega)} \leq \sum_{i=1}^2 \|\partial_{x_i} p\|_{H^{-1}(\Omega)}$$

para todo $p \in L_0^2(\Omega)$ lo cual se tiene por (3.3), [2].

3.2. Existencia y unicidad para el problema de Stokes discreto

A fin de obtener aproximaciones numéricas de dichas soluciones se consideran espacios de dimension finita $V_h \subset V$ y $X_h \subset X$ y planteamos la correspondiente aproximación de Galarkin, que consiste en:

Hallar $(u_h, p_h) \in V_h \times X_h$ tales que

$$\begin{aligned} a(u_h, v) + b(p_h, v) &= f(v) \quad \forall v \in V_h \\ b(q, u_h) &= 0 \quad \forall q \in X_h. \end{aligned} \tag{3.4}$$

Para garantizar la estabilidad del método, los espacios V_h y X_h deben satisfacer una condición análoga a las condiciones ínf – sup, es decir, debe existir una constante $\tilde{\beta} > 0$ independiente de h tal que

$$\sup_{\substack{v \in V_h \\ v \neq 0}} \frac{b(q, v)}{\|v\|_1} \geq \tilde{\beta} \|q\|_0 \quad \forall q \in X_h; \tag{3.5}$$

bajo esta condición el problema (3.4) tiene una única solución [18].

Para elementos lagrangianos de grado 3 y elementos lagrangianos de grado 2, como los

que se usan en este trabajo para la aproximación de u y p respectivamente, hay que demostrar la existencia y unicidad de la solución, tal demostración se omite en este trabajo pero puede ser consultada en [18]

Capítulo 4

Precondicionamiento

Se quiere resolver el sistema lineal $Ax = b$ con A una matriz no singular real. Para resolverlo se usan métodos iterativos y métodos de precondicionamiento que buscan acelerar los métodos iterativos. La idea del precondicionamiento es la reducción del número de iteraciones requerido para la convergencia transformando el sistema original $Ax = b$ en un sistema $\tilde{A}x = \tilde{b}$ de tal forma que satisfaga las siguientes propiedades:

- Resolver $\tilde{A}x = \tilde{b}$ no debe incrementar considerablemente el número de operaciones que se requieren para resolver $Ax = b$.
- $Ax = b$ y $\tilde{A}x = \tilde{b}$ tienen la misma solución, es decir, $\tilde{A}^{-1}\tilde{b} = A^{-1}b$

La matriz $\tilde{A}$ y el vector $\tilde{b}$ se consiguen por pre-multiplicación o pos-multiplicación por matrices llamadas precondicionadores que deben ser fácilmente invertibles. El nuevo sistema se construye al tomar una matriz no singular P y formar el sistema $P^{-1}Ax = P^{-1}b$; a la matriz P se le llama precondicionador a izquierda. En las aplicaciones, frecuentemente las matrices deben considerarse por bloques. Una fuente de precondicionadores con estructura de bloques se origina cuando la matriz del sistema tiene la forma denominada de punto de ensilladura, es decir, matrices no singulares que tienen la siguiente estructura:

$$\mathcal{A} = \begin{pmatrix} M & B^t \\ C & 0 \end{pmatrix} \quad (4.1)$$

donde $M \in \mathbb{R}^{n \times n}$ y $B, C \in \mathbb{R}^{m \times n}$ con $n \geq m$. En muchas aplicaciones M es simétrica y $B = C$ en cuyo caso (4.1) sería simétrica; en todo caso, si M es o no simétrica, la matriz (4.1) es generalmente indefinida es decir las partes reales de sus valores propios son positivos o negativos. Murphy, Golub y Wather muestran en [15] un precondi-

cionador para sistemas lineales cuya matriz de coeficiente tienen la forma (4.1). Dicho preconditionador tiene la siguiente estructura cuando la matriz M es invertible:

$$P = \begin{pmatrix} M & 0 \\ 0 & CM^{-1}B^t \end{pmatrix}. \quad (4.2)$$

Un buen preconditionador se caracteriza por aproximar de manera no costosa a la inversa de la matriz de coeficientes; sin embargo, un buen preconditionador también se puede caracterizar si la matriz preconditionada tiene pocos valores propios distintos [16], El teorema que se presenta a continuación establece una importante relación entre la dimensión del subespacio de Krylov y el grado del polinomio minimal de la matriz del sistema, su demostración puede consultarse en [16].

Teorema 1: Sea A una matriz de orden n , m el grado del polinomio minimal de A y $K_k(A, r_0) = \text{gen}\{r_0, Ar_0, \dots, A^{k-1}r_0\}$ el subespacio de Krylov de orden k de A respecto a $r_0 = b - Ax_0$ donde x_0 es una primera aproximación a la solución de $Ax = b$ entonces la dimensión de $K_k(A, r_0)$ es $\min(m, k)$.

Proposición 1: Si el preconditionador de la matriz (4.1), donde $M \in \mathbb{R}^{n \times n}$ y $B, C \in \mathbb{R}^{m \times n}$ con $n \geq m$, es de la forma (4.2) entonces la matriz preconditionada $T = P^{-1}A$ satisface $T(T - I)(T^2 - T - I) = 0$

Demostración:

Si $P = \begin{pmatrix} M & 0 \\ 0 & CM^{-1}B^t \end{pmatrix}$, entonces $P^{-1} = \begin{pmatrix} M^{-1} & 0 \\ 0 & (CM^{-1}B^t)^{-1} \end{pmatrix}$ luego

$$T = P^{-1}A = \begin{pmatrix} I & M^{-1}B^t \\ (CM^{-1}B^t)^{-1}C & 0 \end{pmatrix}$$

y

$$\left(T - \frac{1}{2}I\right)^2 - \frac{1}{4}I = \begin{pmatrix} M^{-1}B^t(CM^{-1}B^t)^{-1}C & 0 \\ 0 & I \end{pmatrix}$$

pero $M^{-1}B^t(CM^{-1}B^t)^{-1}C$ es idempotente por que

$$M^{-1}B^t(CM^{-1}B^t)^{-1}CM^{-1}B^t(CM^{-1}B^t)^{-1}C = M^{-1}B^t(CM^{-1}B^t)^{-1}C$$

entonces

$$\left[\left(T - \frac{1}{2}I \right)^2 - \frac{1}{4}I \right]^2 = \begin{pmatrix} M^{-1}B^t(CM^{-1}B^t)^{-1}C & 0 \\ 0 & I \end{pmatrix} = \left(T - \frac{1}{2}I \right)^2 - \frac{1}{4}I.$$

Por lo tanto

$$\begin{aligned} & \left[\left(T - \frac{1}{2}I \right)^2 - \frac{1}{4}I \right]^2 - \left(T - \frac{1}{2}I \right)^2 - \frac{1}{4}I = 0 \\ & \left[\left(T - \frac{1}{2}I \right)^2 - \frac{1}{4}I \right] \left\{ \left(T - \frac{1}{2}I \right)^2 - \frac{1}{4}I - I \right\} = 0 \\ & \left(T - \frac{1}{2}I + \frac{1}{2}I \right) \left(T - \frac{1}{2}I - \frac{1}{2}I \right) \left\{ \left(T - \frac{1}{2}I \right)^2 - \frac{5}{4}I \right\} = 0 \\ & T(T - I) \left(T - \frac{1 + \sqrt{5}}{2}I \right) \left(T - \frac{1 - \sqrt{5}}{2}I \right) = 0, \end{aligned}$$

luego el polinomio minimal de T tiene a lo mas grado 4, [16].

La proposición que se presenta a continuación tiene como consecuencia el garantizar que métodos iterativos de subespacio de Krylov como: Gradiente Conjugado, Minres, GMRES convergerán en a lo mas en 4 iteraciones. Suponiendo que se puede trabajar con matemática exacta.

Proposición 2:

Para cualquier vector r , el subespacio de Krylov

$$\text{gen}\{r, Tr, T^2r, T^3r, \dots\}$$

con T como en la proposición anterior es a lo más de dimensión 3 si T es no singular(o 4 si T es singular).

Demostración:

Sin perdida de generalidad suponga que T es singular. Como

$$Q(t) = t(t - 1) \left(t - \frac{1 + \sqrt{5}}{2} \right) \left(t - \frac{1 - \sqrt{5}}{2} \right) \quad (4.3)$$

es un polinomio mónico que anula a T , entonces el polinomio minimal de T tiene a lo más grado 4. Usando el teorema 1 se obtiene que la dimensión de

$$\text{gen}\{r, Tr, T^2r, T^3r, \dots\} \tag{4.4}$$

es a lo sumo 4. Si T es no singular, el polinomio (4.3) tendría a lo sumo grado 3 y por tanto el subespacio en (4.4) tiene a lo mas dimensión 3, [16].

Capítulo 5

Fem Mixto e Implementación numérica Deal.II

Supondremos en lo sucesivo que Ω es unión de rectángulos es decir $\mathfrak{T}_h = \cup_{i=1}^m T_i$ es una partición del dominio, véase [6], donde T_i son rectángulos.

En este trabajo se aproxima (u, p) con $(\tilde{u}, \tilde{p})$ mediante elementos finitos mixtos; a fin de obtener aproximaciones numéricas de dichas soluciones consideremos espacios de dimension finita $V_h \subset V$ y $X_h \subset X$ donde se aproxima la velocidad u mediante elementos Lagrangianos de grado 3 para cada componente y la presión p por elementos Lagrangianos de grado 2 [18], [7]. Al plantear la correspondiente aproximación de Galerkin se obtiene:

Hallar $(u_h, p_h) \in V_h \times X_h$ tales que cumpla

$$\begin{aligned} a(u_h, v) + b(p_h, v) &= f(v) \quad \forall v \in V_h \\ b(q, u_h) &= 0 \quad \forall q \in X_h. \end{aligned}$$

Sean

$$\begin{aligned} \{\varphi_1, \varphi_2, \varphi_3, \dots, \varphi_N\} &\text{ base de } V_h, \\ \{\mu_1, \mu_2, \mu_3, \dots, \mu_M\} &\text{ base de } X_h \end{aligned}$$

entonces

$$u_h = \sum_{j=1}^N \alpha_j \varphi_j \text{ y } p_h = \sum_{j=1}^M \beta_j \mu_j.$$

Luego el problema (3.4) es equivalente a:

$$\begin{aligned} a(u_h, \varphi_i) + b(p_h, \varphi_i) &= f(\varphi_i), \quad \forall \varphi_i \in V_h \\ b(\mu_i, u_h) &= 0, \quad \forall \mu_i \in X_h \end{aligned} \quad (5.1)$$

y por lo tanto

$$\begin{aligned} \sum_{j=1}^N \alpha_j a(\varphi_j, \varphi_i) + \sum_{j=1}^M \beta_j b(\mu_j, \varphi_i) &= f(\varphi_i), \quad \forall \varphi_i \in V_h \\ \sum_{j=1}^N \alpha_j b(\mu_i, \varphi_j) &= 0, \quad \forall \mu_i \in X_h \end{aligned}$$

Así se llega a un sistema con $N + M$ ecuaciones e incógnitas; podemos apreciar la estructura en bloques del sistema agrupando ecuaciones e incógnitas como sigue:

$$\begin{aligned} M &= (a_{ij}), \quad a_{ij} = a(\varphi_j, \varphi_i), \quad (N \times N) \\ B &= (b_{ij}), \quad b_{ij} = b(\mu_i, \varphi_j), \quad (M \times N) \\ F &= (f(\varphi_1), f(\varphi_2), \dots, f(\varphi_N))^t \end{aligned}$$

definiendo los vectores $p_h = (\beta_1, \beta_2, \dots, \beta_M)^t$, $u_h = (\alpha_1, \alpha_2, \dots, \alpha_N)^t$ obtenemos el sistema de la forma:

$$\begin{pmatrix} M & B^t \\ B & 0 \end{pmatrix} \begin{pmatrix} u_h \\ p_h \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix}. \quad (5.2)$$

donde $M \in \mathbb{R}^{n \times n}$ es una matriz simétrica definida positiva y $B \in \mathbb{R}^{m \times n}$ de rango completo. La matriz M es invertible ya que M es definida positiva. El sistema se puede desacoplar de la siguiente manera (multiplicar la primera fila del sistema por BM^{-1} y luego restar la segunda fila de la misma):

$$BM^{-1}B^tp_h = BM^{-1}F, \quad (5.3)$$

$$Mu_h = F - B^tp_h \quad (5.4)$$

donde la matriz $-BM^{-1}B^t$ se conoce como el complemento de Schur. Sea $S = BM^{-1}B^t$, S es simétrica y definida positiva ya que M lo es, por lo tanto, se podría aplicar el método del gradiente conjugado a (5.3) sin embargo, el cálculo de S es computacionalmente

costoso además S probablemente es una matriz densa; para resolver (5.3) y (5.4) también sería necesario calcular M^{-1} lo que es poco práctico. Estos obstáculos se pueden superar si se implementan dos pasos, el primero se basa en la observación de que para resolver con el gradiente conjugado (5.3) realmente no se necesita S sino como actúa S sobre un vector v , ésto se puede implementar por pasos.

1. Definamos $w = B^t v$;
2. solucionamos $My = w$ donde $y = M^{-1}w$, usando el método CG aplicado a la matriz M ;
3. Definimos $z = By$ para obtener $Sv = z$

El segundo paso se basa sobre el hecho que no se necesitan las inversas de manera explícita sino su acción sobre vectores, en general cuando se necesite el cálculo de una inversa por un vector no se hace de manera directa por el costo computacional sino que se procede como en (2), es decir, si se tiene una matriz P y se necesita $P^{-1}w$. Siendo w un vector, se resuelve $Py = w$ y en y se almacena el cálculo de $P^{-1}w$. En nuestro caso para estos cálculos con inversas siempre las matrices son simétricas definidas positivas y por tanto se usa el gradiente conjugado para resolverlas. En este trabajo se preconditiona (5.3) con una aproximación a la inversa de S y (5.4) con una aproximación a la inversa de M :

$$\begin{aligned}\tilde{S} &= [B^t \tilde{M}^{-1} B] \\ \tilde{M} &= (\text{diag} (M))\end{aligned}$$

Es importante notar que los cálculos con $(\text{diag} (M))$ en lugar de M son más favorables en cuestión de tiempo y operaciones

Implementación numérica Deal.II

La implementación numérica del problema de Stokes se realizó usando las rutinas del paquete deal.II, que es un conjunto de librerías de C++ dirigido a la solución numérica de las ecuaciones en derivadas parciales utilizando elementos finitos. El objetivo principal de deal.II es permitir el rápido desarrollo de modernos códigos de elementos finitos, utilizando entre otros aspectos, las mallas de adaptación y una amplia variedad de clases

de herramientas de uso frecuente en el programa de elementos finitos. Escribir este tipo de programas es una tarea no trivial, y programas exitosos tienden a ser muy grandes y complejos. Se cree que esto se hace mejor con una biblioteca de programas que se encarga de los detalles del manejo de la malla y el refinamiento, la manipulación de los grados de libertad, de las mallas de entrada y salida de los resultados en formatos gráficos, etc. Para este trabajo se usa la versión 6.3.1 para más información sobre Deal.II consulte la presente dirección <http://www.dealii.org/>

Stokes 2D Este programa está escrito para resolver el problema de Stokes 2D cuyas ecuaciones están representadas en (1); la formulación débil para este problema se describió en la sección 2. En la primera parte del código se define la clase `MixedLaplaceProblem`, que contiene las rutinas que ejecutan los pasos que el método de elementos finitos necesita para resolver numéricamente el problema: una rutina para hacer la triangulación del dominio, definir el elemento finito que se usa, ensamblar las matrices de rigidez, resolver numéricamente el sistema de ecuaciones lineales y finalmente exportar los resultados para ser procesados con un visualizador gráfico.

```
template <int dim>
class MixedLaplaceProblem
{
public:
    MixedLaplaceProblem (const unsigned int degree);
    void run ();
private:
    void make_grid_and_dofs ();
    void assemble_system ();
    void solve ();
    void compute_errors () const;
    void output_results () const;

    const unsigned int    degree;

    Triangulation<dim>    triangulation;
    FESystem<dim>         fe;
    DoFHandler<dim>       dof_handler;

    BlockSparsityPattern    sparsity_pattern;
    BlockSparseMatrix<double> system_matrix;
    ConstraintMatrix        constraints;
    BlockVector<double>      solution;
    BlockVector<double>      system_rhs;
};
```

En esta parte del código se define la función del lado derecho, que en el problema de Stokes representa las fuerzas externas, los valores de la función se agregan como vector.

```

template <int dim>
class RightHandSide : public Function<dim>
{
public:
    RightHandSide ();

    virtual void vector_value (const Point<dim> &p,
                               Vector<double> &values) const;
    virtual void vector_value_list (const std::vector<Point<dim> > &points,
                                     std::vector<Vector<double> > &value_list) const;
};

template <int dim>
RightHandSide<dim>::RightHandSide ()
:
Function<dim> (dim+1)
{}

template <int dim>
inline
void RightHandSide<dim>::vector_value (const Point<dim> &p,
                                       Vector<double> &values) const
{
    Assert (values.size() == dim+1,
            ExcDimensionMismatch (values.size(), dim+1));
    Assert (dim >= 2, ExcNotImplemented());

    values(0) = 256*(p[0]*p[0]*(p[0]-1)*(p[0]-1)*(12*p[1]-6)+
                    p[1]*(p[1]-1)*(2*p[1]-1)*(12*p[0]*p[0]-12*p[0]+2))+p[1]-0.5;
    values(1) = -256*(p[1]*p[1]*(p[1]-1)*(p[1]-1)*(12*p[0]-6)+
                    p[0]*(p[0]-1)*(2*p[0]-1)*(12*p[1]*p[1]-12*p[1]+2))+p[0]-0.5;
    values(2) = 0;
}

template <int dim>
void RightHandSide<dim>::vector_value_list (const std::vector<Point<dim> > &points,
                                             std::vector<Vector<double> > &value_list) const
{
    Assert (value_list.size() == points.size(),
            ExcDimensionMismatch (value_list.size(), points.size()));
    const unsigned int n_points = points.size();

    for (unsigned int p=0; p<n_points; ++p)
        RightHandSide<dim>::vector_value (points[p],
                                           value_list[p]);
}

```

Enseguida se definen los valores de frontera para el problema, al igual que en el paso anterior también se define como un vector.

```

template <int dim>
class BoundaryValues : public Function<dim>
{
public:
    BoundaryValues () : Function<dim>(dim+1) {}
}

```

```

virtual double value (const Point<dim> &p,
                    const unsigned int component = 0) const;
virtual void vector_value (const Point<dim> &p,
                        Vector<double> &value) const;

};

template <int dim>
double
BoundaryValues<dim>::value (const Point<dim> &p,
const unsigned int component) const
{
    Assert (component < this->n_components,
            ExcIndexRange (component, 0, this->n_components));

    if (component == 2)
        return (p[0]-0.5)*(p[1]-0.5);
    return 0;
}

template <int dim>
void
BoundaryValues<dim>::vector_value (const Point<dim> &p,
Vector<double> &values) const
{
    for (unsigned int c=0; c<this->n_components; ++c)
        values(c) = BoundaryValues<dim>::value (p, c);
}

```

Ahora se define la solución exacta, que se usara para calcular el error en la aproximación

```

template <int dim>
class ExactSolution : public Function<dim>
{
public:
    ExactSolution () : Function<dim>(dim+1) {}

    virtual void vector_value (const Point<dim> &p,
                            Vector<double> &value) const;
};

template <int dim>
void
ExactSolution<dim>::vector_value (const Point<dim> &p,
                                Vector<double> &values) const
{
    Assert (values.size() == dim+1,
            ExcDimensionMismatch (values.size(), dim+1));

    values(0) = -256*(p[1]*((p[1]-1)*(2*p[1]-1))*p[0]*p[0]*((p[0]-1)*(p[0]-1)));
    values(1) = 256* p[0]*((p[0]-1)*(2*p[0]-1))*p[1]*p[1]*(p[1]-1)*(p[1]-1);
    values(2) = (p[0]-0.5)*(p[1]-0.5);
}

```

Aquí se definen los elementos que vamos a usar, para este problema usamos elementos lagrangianos de orden 3 para la velocidad, que es un campo vectorial y elementos

lagrangianos de orden 2 para la presión, que es un campo escalar

```
template <int dim>
MixedLaplaceProblem<dim>::MixedLaplaceProblem (const unsigned int degree)
:
    degree (degree),
    fe (FE_Q<dim>(degree+1), dim,
        FE_Q<dim>(degree), 1),
    dof_handler (triangulation)
```

En lo siguiente se crea la malla y los grados de libertad para el problema, en nuestro caso es el rectángulo en $(0,1) \times (0,1)$ que cuando se refina hace rectángulos más pequeños dentro del rectángulo original, los grados de libertad se definen por bloques por los diferentes ordenes en los elementos

```
template <int dim>
void MixedLaplaceProblem<dim>::make_grid_and_dofs ()
{
    GridGenerator::hyper_cube (triangulation, 0, 1);
    triangulation.refine_global (5);
    std::ofstream out ("grid-1.eps");
    GridOut grid_out;
    grid_out.write_eps (triangulation, out);
    system_matrix.clear ();
    dof_handler.distribute_dofs (fe);
    DoFRenumbering::Cuthill_McKee (dof_handler);
    std::vector<unsigned int> block_component (dim+1,0);
    block_component[dim] = 1;
    DoFRenumbering::component_wise (dof_handler, block_component);
{
    constraints.clear ();
}
    constraints.close ();
    std::vector<unsigned int> dofs_per_block (2);
    DoFTools::count_dofs_per_block (dof_handler, dofs_per_block, block_component);
    const unsigned int n_u = dofs_per_block[0],
        n_p = dofs_per_block[1];
    std::cout << "    Number of active cells: "
        << triangulation.n_active_cells()
        << std::endl
        << "    Number of degrees of freedom: "
        << dof_handler.n_dofs()
        << " (" << n_u << '+' << n_p << ') '
        << std::endl;
{
    BlockCompressedSimpleSparsityPattern csp (2,2);
    csp.block(0,0).reinit (n_u, n_u);
    csp.block(1,0).reinit (n_p, n_u);
    csp.block(0,1).reinit (n_u, n_p);
    csp.block(1,1).reinit (n_p, n_p);
    csp.collect_sizes();
    DoFTools::make_sparsity_pattern (dof_handler, csp, constraints, false);
```

```

    sparsity_pattern.copy_from (csp);
}
system_matrix.reinit (sparsity_pattern);
solution.reinit (2);
solution.block(0).reinit (n_u);
solution.block(1).reinit (n_p);
solution.collect_sizes ();

system_rhs.reinit (2);
system_rhs.block(0).reinit (n_u);
system_rhs.block(1).reinit (n_p);
system_rhs.collect_sizes ();
}

```

Ahora se hace el ensamble del sistema, en esta parte hay que discretizar el problema de Stokes, además se interpola los valores de la frontera y se agregan a la matriz de rigidez

```

template <int dim>
void MixedLaplaceProblem<dim>::assemble_system ()
{
    QGauss<dim>    quadrature_formula(degree+2);
    FEValues<dim> fe_values (fe, quadrature_formula,
                           update_values | update_gradients |
                           update_quadrature_points | update_JxW_values);

    const unsigned int dofs_per_cell = fe.dofs_per_cell;
    const unsigned int n_q_points = quadrature_formula.size();
    FullMatrix<double> local_matrix (dofs_per_cell, dofs_per_cell);
    Vector<double> local_rhs (dofs_per_cell);
    std::vector<unsigned int> local_dof_indices (dofs_per_cell);
    const BoundaryValues<dim> boundaryValues;
    const RightHandSide<dim> right_hand_side;
    std::vector<Vector<double> > rhs_values (n_q_points,
                                             Vector<double>(dim+1));

    const FEValuesExtractors::Vector velocities (0);
    const FEValuesExtractors::Scalar pressure (dim);
    unsigned int comp_i, comp_j;
    typename DoFHandler<dim>::active_cell_iterator
        cell = dof_handler.begin_active(),
        endc = dof_handler.end();
    for (; cell!=endc; ++cell)
    {
        fe_values.reinit (cell);
        local_matrix = 0;
        local_rhs = 0;
        right_hand_side.vector_value_list(fe_values.get_quadrature_points(),
                                           rhs_values);

        for (unsigned int q=0; q<n_q_points; ++q)
            for (unsigned int i=0; i<dofs_per_cell; ++i)
                { comp_i = fe.system_to_component_index(i).first;
                  for (unsigned int j=0; j<dofs_per_cell; ++j)
                  {
                      comp_j = fe.system_to_component_index(j).first;
                      if ( (comp_i<dim) && (comp_j<dim) && (comp_i==comp_j) )

```

```

        local_matrix(i,j) += fe_values.shape_grad(i,q) *
                             fe_values.shape_grad(j,q) *
                             fe_values.JxW(q);
    if ( (comp_i<dim) && (comp_j==dim) )
        local_matrix(i,j) += -fe_values.shape_grad(i,q)[comp_i] *
                             fe_values.shape_value(j,q) *
                             fe_values.JxW(q);
    if ( (comp_i==dim) && (comp_j<dim) )
        local_matrix(i,j) += fe_values.shape_value(i,q) *
                             fe_values.shape_grad(j,q)[comp_j] *
                             fe_values.JxW(q);
    }

    local_rhs(i) += fe_values.shape_value( i, q)*
                  rhs_values[q](comp_i) *
                  fe_values.JxW(q);
}

cell->get_dof_indices (local_dof_indices);

for (unsigned int i=0; i<dofs_per_cell; ++i)
    for (unsigned int j=0; j<dofs_per_cell; ++j)
        system_matrix.add (local_dof_indices[i],
                           local_dof_indices[j],
                           local_matrix(i,j));
for (unsigned int i=0; i<dofs_per_cell; ++i)
    system_rhs(local_dof_indices[i]) += local_rhs(i);

std::map<unsigned int,double> boundary_valuess;
std::vector<bool> component_mask (dim+1, true);
component_mask[dim] = false;
VectorTools::interpolate_boundary_values (dof_handler,
                                          0,
                                          boundaryValues,
                                          boundary_valuess,
                                          component_mask);
MatrixTools::apply_boundary_values (boundary_valuess,
                                    system_matrix,
                                    solution,
                                    system_rhs);
}
}

```

Enseguida se define la clase de la matriz inversa, note que no se calcula la matriz inversa. Esta se aproxima mediante un método CG.

```

template <class Matrix>
class InverseMatrix : public Subscriptor
{
public:
    InverseMatrix (const Matrix &m);

    void vmult (Vector<double> &dst,
               const Vector<double> &src) const;

```

```

private:
    const SmartPointer<const Matrix> matrix;
};
template <class Matrix>
InverseMatrix<Matrix>::InverseMatrix (const Matrix &m)
:
    matrix (&m)
{}
template <class Matrix>
void InverseMatrix<Matrix>::vmult (Vector<double> &dst,
                                   const Vector<double> &src) const
{
    SolverControl solver_control (src.size(), 1e-07);
    SolverCG<>      cg (solver_control);

    dst = 0;

    cg.solve (*matrix, dst, src, PreconditionIdentity());
}

```

Para el caso en el que hay que operar con el complemento de Schur, se define la siguiente clase la cual hace las operaciones de derecha a izquierda usando también la clase que aproxima la inversa de una matrix

```

class SchurComplement : public Subscriptor
{
public:
    SchurComplement (const BlockSparseMatrix<double> &A,
                     const InverseMatrix<SparseMatrix<double> > &Minv);
    void vmult (Vector<double> &dst,
                const Vector<double> &src) const;

private:
    const SmartPointer<const BlockSparseMatrix<double> > system_matrix;
    const SmartPointer<const InverseMatrix<SparseMatrix<double> > > m_inverse;
    mutable Vector<double> tmp1, tmp2;
};

SchurComplement::SchurComplement (const BlockSparseMatrix<double> &A,
                                   const InverseMatrix<SparseMatrix<double> > &Minv):
    system_matrix (&A),
    m_inverse (&Minv),
    tmp1 (A.block(0,0).m()),
    tmp2 (A.block(0,0).m())
{}

void SchurComplement::vmult (Vector<double> &dst,
                             const Vector<double> &src) const
{
    system_matrix->block(0,1).vmult (tmp1, src);
    m_inverse->vmult (tmp2, tmp1);
    system_matrix->block(1,0).vmult (dst, tmp2);
}

```

Una vez se desacopla el sistema se busca la solución aproximada p , para esto se pre-

condiciona y se resuelve con CG. Para preconditionar este sistema que contiene a p se implementa la clase aproximación al complemento de Schur usando una aproximación de la matriz M la cual consiste de su diagonal

```
class ApproximateSchurComplement : public Subscriptor
{
public:
    ApproximateSchurComplement (const BlockSparseMatrix<double> &A);

    void vmult (Vector<double> &dst,
               const Vector<double> &src) const;

private:
    const SmartPointer<const BlockSparseMatrix<double> > system_matrix;

    mutable Vector<double> tmp1, tmp2;
};

ApproximateSchurComplement::ApproximateSchurComplement (const BlockSparseMatrix<double> &A)
:
    system_matrix (&A),
    tmp1 (A.block(0,0).m()),
    tmp2 (A.block(0,0).m())
{}

void ApproximateSchurComplement::vmult (Vector<double> &dst,
                                       const Vector<double> &src) const
{
    system_matrix->block(0,1).vmult (tmp1, src);
    system_matrix->block(0,0).precondition_Jacobi (tmp2, tmp1);
    system_matrix->block(1,0).vmult (dst, tmp2);
}
```

Igual que en el paso anterior, se quiere buscar la solución aproximada u para esto se toma el sistema desacoplado que tiene a u y se preconditiona y soluciona con CG. Para preconditionarlo usamos una aproximación de M la cual consiste en su diagonal

```
class ApproximateM : public Subscriptor
{
public:
    ApproximateM (const BlockSparseMatrix<double> &A);

    void vmult (Vector<double> &dst,
               const Vector<double> &src) const;

private:
    const SmartPointer<const BlockSparseMatrix<double> > system_matrix;

    mutable Vector<double> tmp1, tmp2;
};

ApproximateM::ApproximateM (const BlockSparseMatrix<double> &A)
:
    system_matrix (&A),
    tmp1 (A.block(0,0).m()),
```

```

        tmp2 (A.block(0,0).m())
    {}
    void ApproximateM::vmult (Vector<double> &dst,
                             const Vector<double> &src) const
    {
        system_matrix->block(0,0).precondition_Jacobi (dst, src);}

```

La siguiente clase calcula la solución aproximada para el problema de Stokes ya discreto, por medio del desacople del sistema; además se preconditionan los sistemas lineales que permiten el cálculo de u y p aproximados

```

template <int dim>
void MixedLaplaceProblem<dim>::solve ()
{
    const InverseMatrix<SparseMatrix<double> >
        m_inverse (system_matrix.block(0,0));
    Vector<double> tmp (solution.block(0).size());
    Vector<double> tmp2 (solution.block(0).size());
    {
        Vector<double> schur_rhs (solution.block(1).size());

        m_inverse.vmult (tmp, system_rhs.block(0));
        system_matrix.block(1,0).vmult (schur_rhs, tmp);
        schur_rhs -= system_rhs.block(1);

        SchurComplement
            schur_complement (system_matrix, m_inverse);

        ApproximateSchurComplement
            approximate_schur_complement (system_matrix);

        InverseMatrix<ApproximateSchurComplement>
            preconditioner1 (approximate_schur_complement);

        SchurComplement
            preconditioner2 (schur_complement);

        SolverControl solver_control (solution.block(1).size(),
                                       1e-04);
        SolverCG<> cg (solver_control);

        cg.solve (schur_complement, solution.block(1), schur_rhs,
                  preconditioner1);
        constraints.distribute (solution);
        std::cout << solver_control.last_step()
                   << " CG Schur complement iterations to obtain convergence."
                   << std::endl;
    }
    {
        system_matrix.block(0,1).vmult (tmp, solution.block(1));
        tmp *= -1;
    }
}

```

```

    tmp += system_rhs.block(0);

    ApproximateM
        approximateM(system_matrix);

    ApproximateM
        preconditioner3 (approximateM);

    SolverControl solver_control (solution.block(0).size(),
                                   1e-04);
    SolverCG<>      cg (solver_control);

    cg.solve (system_matrix.block(0,0), solution.block(0), tmp,
              preconditioner3);
    constraints.distribute (solution);

    std::cout << solver_control.last_step()
               << " CG Schur complement iterations to obtain convergence."
               << std::endl;
}
}

```

Por último se implementa la clase que permite calcular el error en la aproximación usando la solución exacta ya conocida para el problema de prueba y la solución aproximada.

```

template <int dim>
void MixedLaplaceProblem<dim>::compute_errors () const
{
    const ComponentSelectFunction<dim>
        pressure_mask (dim, dim+1);
    const ComponentSelectFunction<dim>
        velocity_mask(std::make_pair(0, dim), dim+1);

    ExactSolution<dim> exact_solution;
    Vector<double> cellwise_errors (triangulation.n_active_cells());

    QTrapez<1>      q_trapez;
    QIterated<dim>  quadrature (q_trapez, degree+2);

    VectorTools::integrate_difference (dof_handler, solution, exact_solution,
                                       cellwise_errors, quadrature,
                                       VectorTools::L2_norm,
                                       &pressure_mask);
    const double p_l2_error = cellwise_errors.l2_norm();

    VectorTools::integrate_difference (dof_handler, solution, exact_solution,
                                       cellwise_errors, quadrature,
                                       VectorTools::L2_norm,
                                       &velocity_mask);
    const double u_l2_error = cellwise_errors.l2_norm();
}

```

```
std::cout << "Errors: ||e_p||_L2 = " << p_l2_error
          << ",    ||e_u||_L2 = " << u_l2_error
          << std::endl;
}
```

Capítulo 6

Experimentos Numéricos

6.1. Test de Bercovier–Engelman (condición Dirichlet nula)

A continuación consideramos el Test de Bercovier–Engelman, el cual resuelve las ecuaciones de Stokes (2.1) en $\Omega = (0, 1)^2$. En este caso $u = 0$ sobre $\partial\Omega$ y la función del lado derecho $f(x, y)$ tiene componentes $f_1(x, y), f_2(x, y)$ definidos de la siguiente forma:

$$\begin{aligned}f_1(x, y) &= g(x, y) + y - 0.5 \\f_2(x, y) &= -g(y, x) + x - 0.5\end{aligned}$$

donde $g(x, y) = 256(x^2(x - 1)^2(12y - 6) + y(y - 1)(2y - 1)(12x^2 - 12x + 2))$. Las componentes de la velocidad $u(x, y)$ son $u_1(x, y), u_2(x, y)$ y la solución analítica del sistema es

$$\begin{aligned}u_1(x, y) &= -256y(y - 1)(2y - 1)x^2(x - 1)^2 \\u_2(x, y) &= -u_1(y, x) \\p(x, y) &= (x - 0.5)(y - 0.5).\end{aligned}$$

En el cuadro 6.1 se presentan las iteraciones al resolver el test de Bercovier–Engelman con y sin preconditionamiento refinando el dominio varias veces; también se indican los grados de libertad y los números de celdas. Se puede observar que cuando se refina un número pequeño de veces la diferencia en las iteraciones es pequeña, aunque siempre se hacen menos iteraciones cuando se preconditiona el problema. Al aumentar el número de

refinamientos la diferencia en las iteraciones se hace mayor, evidenciando que resolver el problema con preconditionamiento reduce significativamente el número de iteraciones, que es lo que se espera del modelo teórico. El tiempo de ejecución del programa para el método sin preconditionar fue mejor que el método preconditionado cuando se refina 6 y 7 veces que es el caso cuando la reducción de las iteraciones es significativa y por tanto el de interés (cuadro 6.2). Para los cálculos se usó un procesador Intel xeon quad core 3ghz. El error en la aproximación de u y p preconditionando es menor al error de la aproximación de u y p sin preconditionar a medida que la malla se hace más fina, refinado 6 y 7 (Cuadro 6.3). A continuación se muestran las gráficas de las soluciones aproximadas y exactas para este test.

Ref	Dof	Cells	Itera CG u y p precon	Itera CG u y p sin precon	Dif
1	123 (98+25)	4	6+2	6+4	2
2	419 (338+81)	16	20+3	20+9	6
3	1539 (1250+289)	64	41+3	43+5	4
4	5891 (4802+1089)	265	80+3	84+4	5
5	23043 (18818+4225)	1024	161+2	169+2	8
6	91139 (74498+16641)	4096	320+1	334+17	30
7	362499 (296450+66049)	16384	606+1	670+286	349

Abreviaturas. Ref.: Refinado, Itera.: Iteraciones, precon.: Precondicionado, Dif.: Diferencia

Cuadro 6.1: Iteraciones del método de elementos finitos

Ref	Cells	Tiempo en u y p precon	Tiempo en u y p sin precon
1	4	0m7.790s	0m7.776s
2	16	0m7.778s	0m7.859s
3	64	0m7.893s	0m7.904s
4	265	0m8.518s	0m8.341s
5	1024	0m17.023s	0m13.969s
6	4096	1m21.189s	0m46.741s
7	16384	7m56.614s	4m15.452s

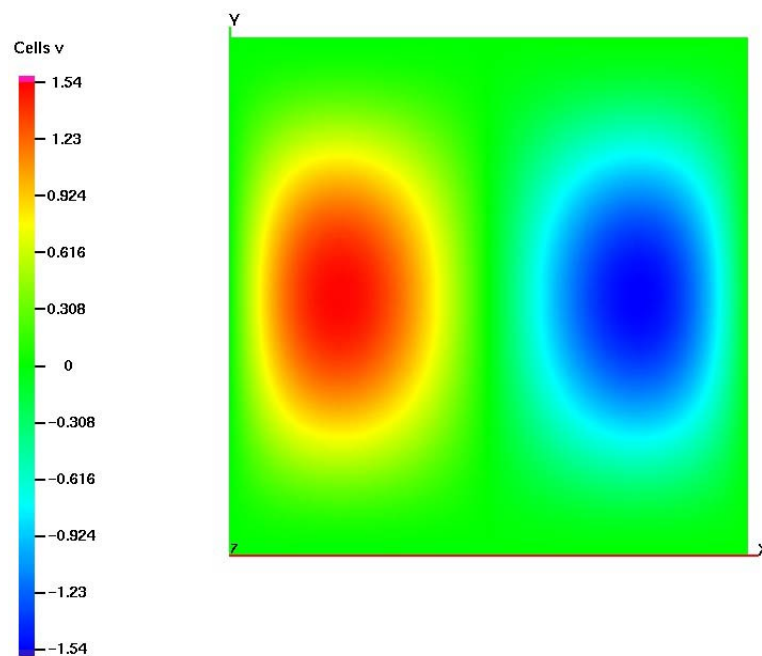
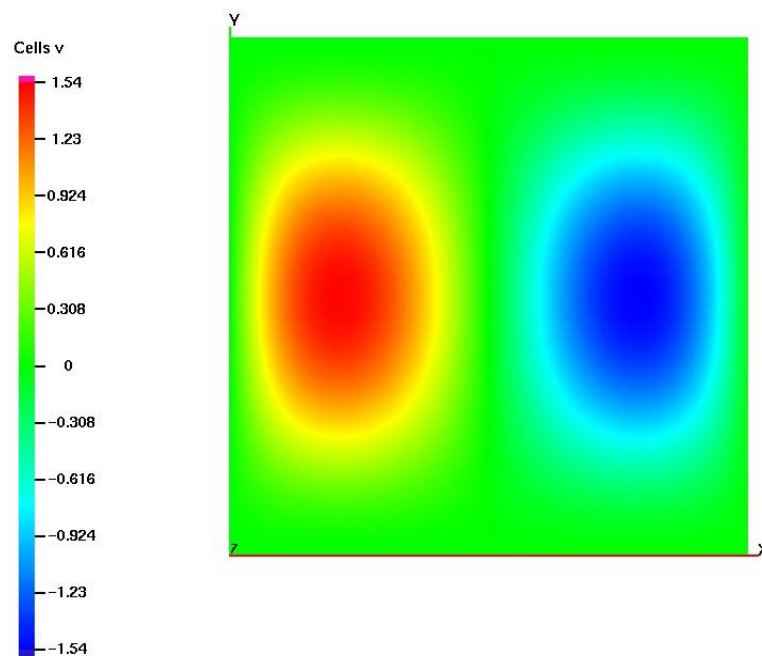
Abreviaturas. Ref.: Refinado, precon.: Precondicionado

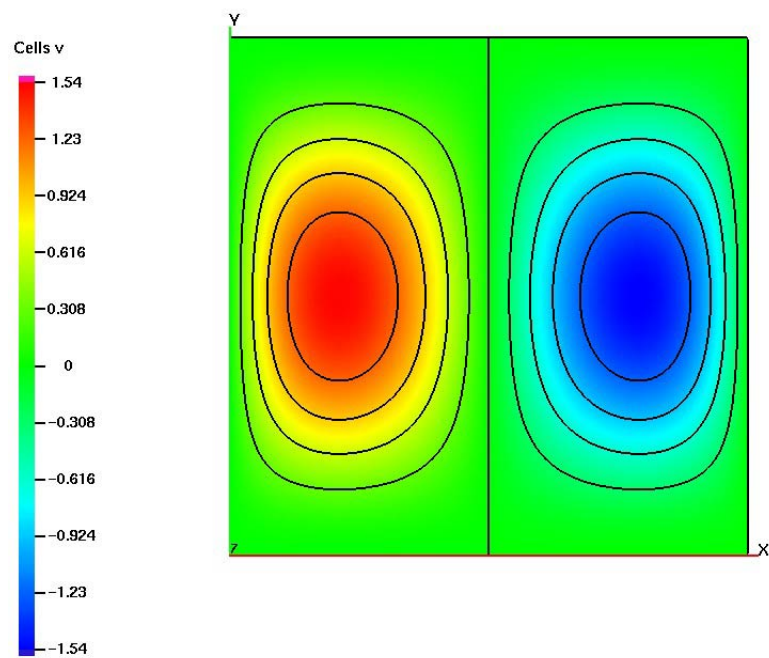
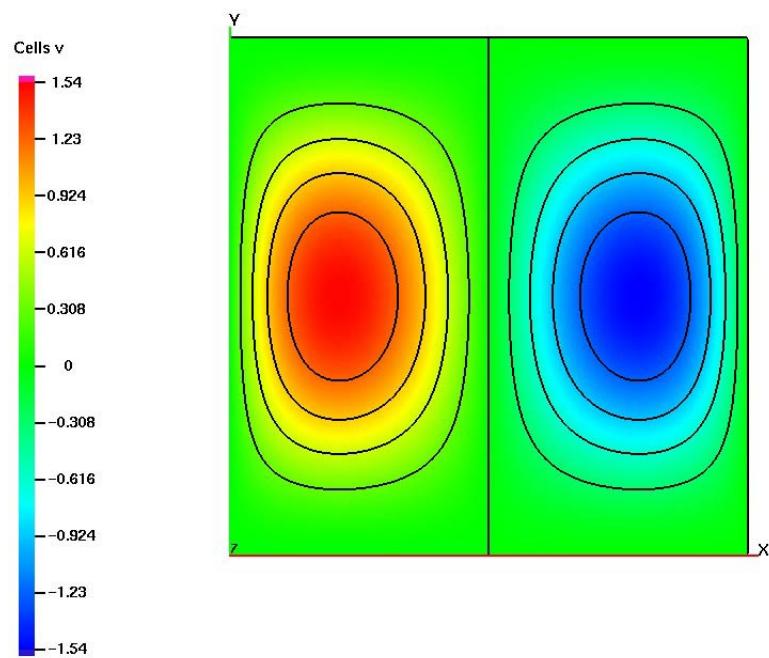
Cuadro 6.2: Tiempo CPU de ejecución

Ref	Cells	Error en u Precon	Error en p Precon	Error en u	Error en p
1	4	9,84203E-03	0,0223204	9,84203E-03	0,0223204
2	16	6,23E-04	0,0006102	6,23E-04	0,0006067
3	64	3,93E-05	0,00007607	3,93E-05	0,00003992
4	265	2,58E-06	6,31E-05	2,70E-06	1,29E-04
5	1024	2,50E-06	8,88E-05	1,25E-06	1,55E-04
6	4096	4,81E-06	1,61E-04	6,53E-06	7,99E-04
7	16384	1,23E-05	3,06E-04	1,81E-05	1,45E-03

Abreviaturas. Ref.: Refinado, precon.: Precondicionado

Cuadro 6.3: Error $\|(u - \tilde{u}, p - \tilde{p})\|_{L^2_\Omega}$

Figura 6.1: Velocidad Aproximada v Figura 6.2: Velocidad Exacta v

Figura 6.3: Velocidad Aproximada v Figura 6.4: Velocidad Exacta v

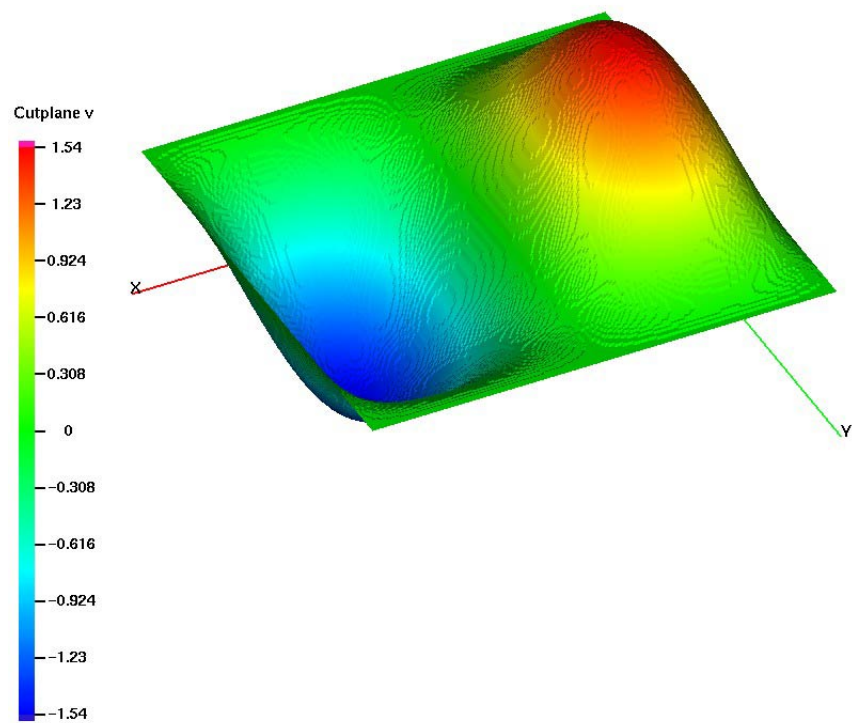


Figura 6.5: Velocidad Aproximada v

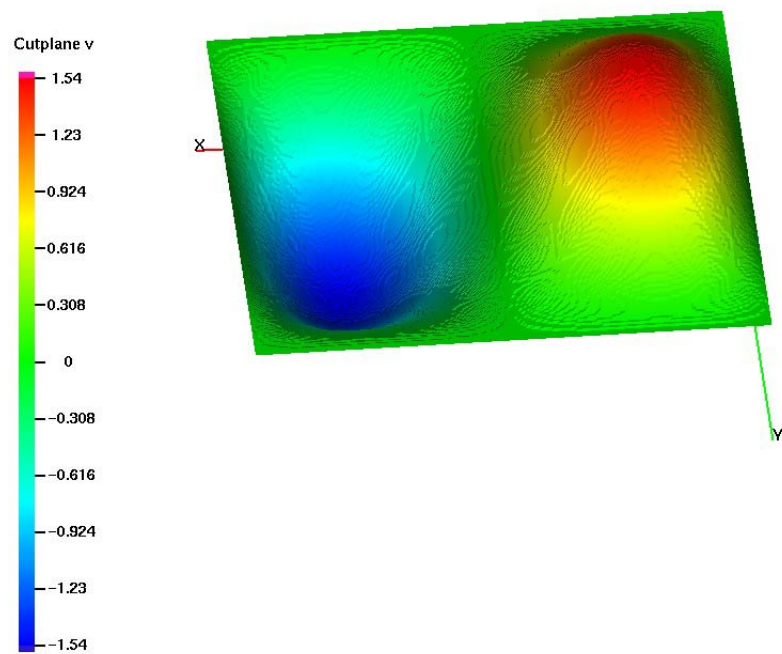
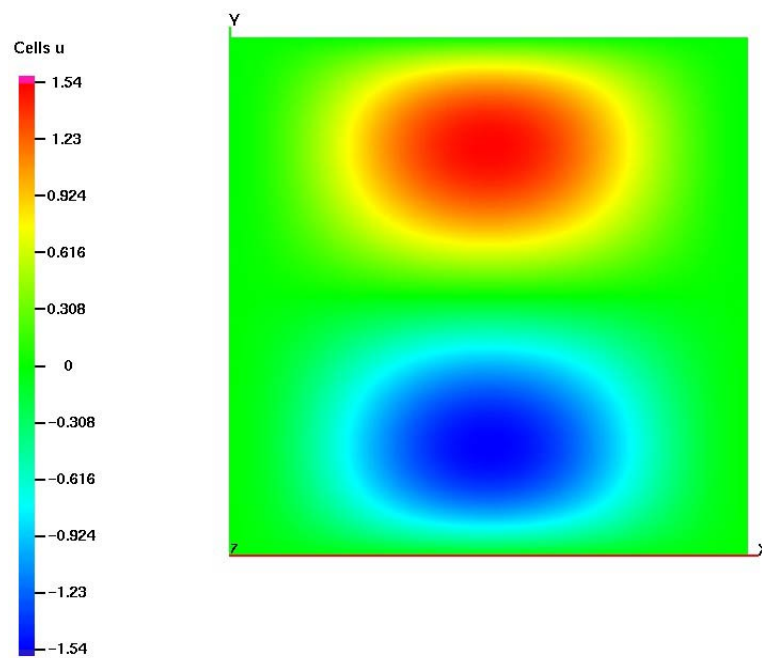
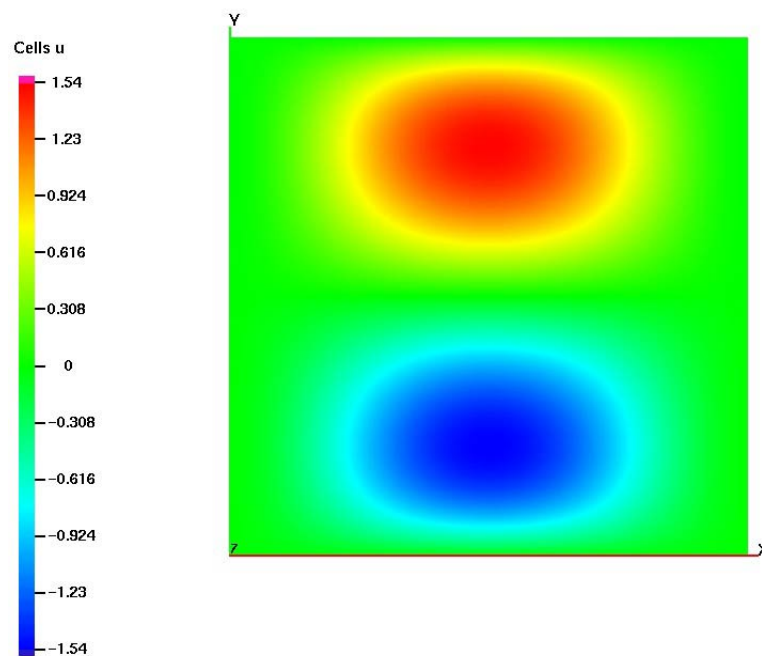
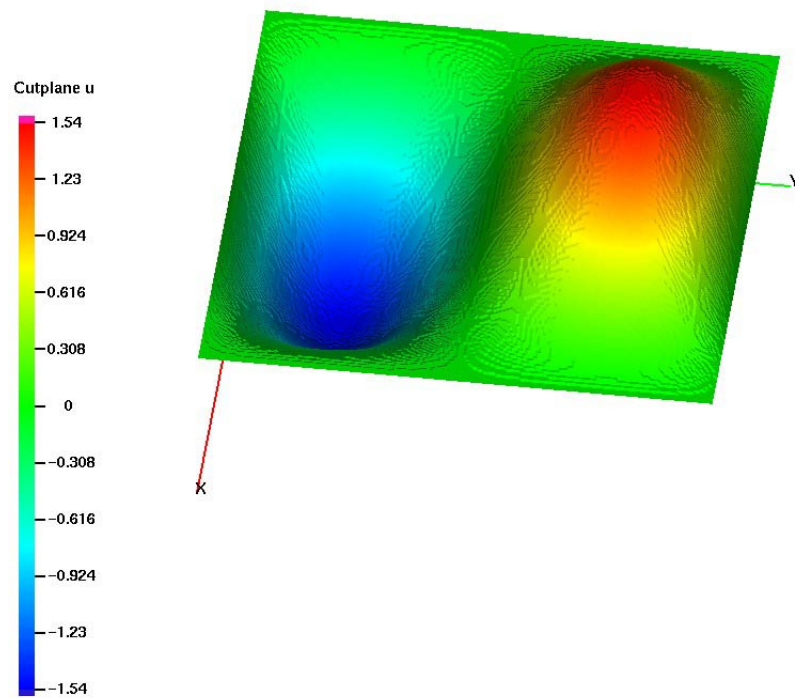
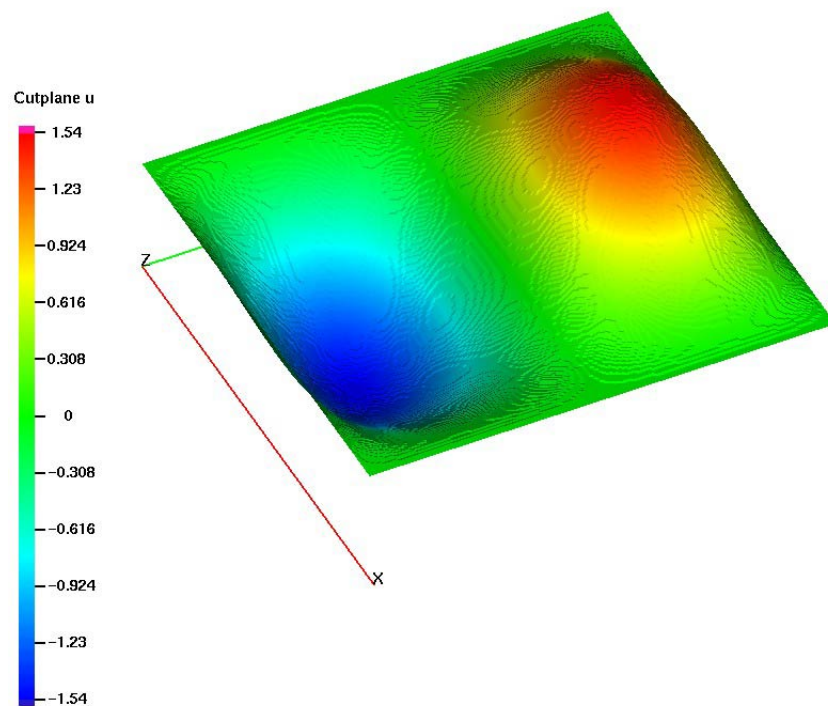


Figura 6.6: Velocidad Exacta V

Figura 6.7: Velocidad Aproximada U Figura 6.8: Velocidad Exacta U

Figura 6.9: Velocidad Aproximada U Figura 6.10: Velocidad Exacta U

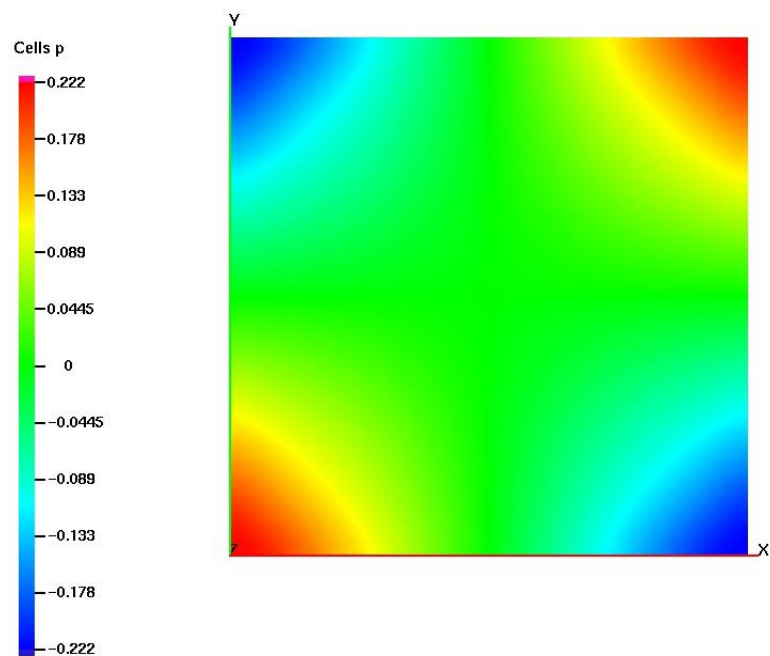


Figura 6.11: Presión Aproximada

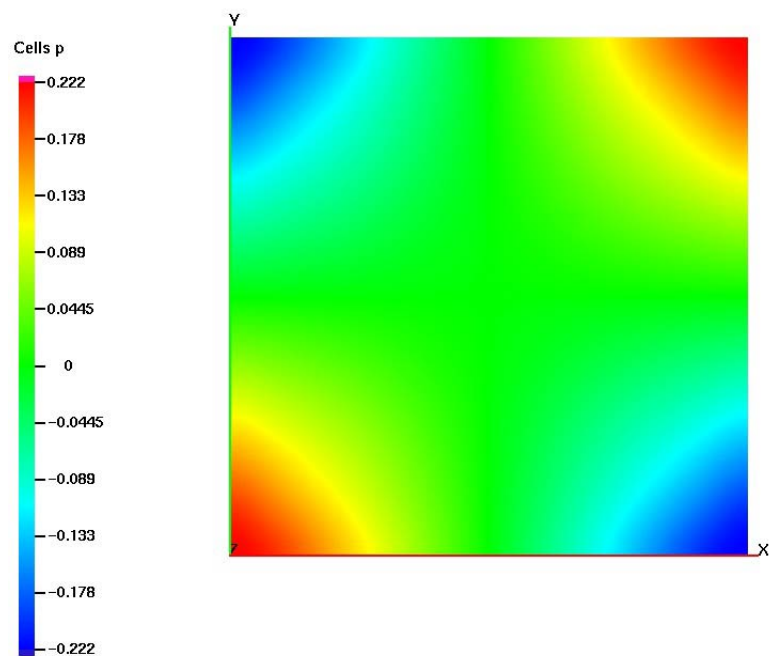


Figura 6.12: Presión Exacta

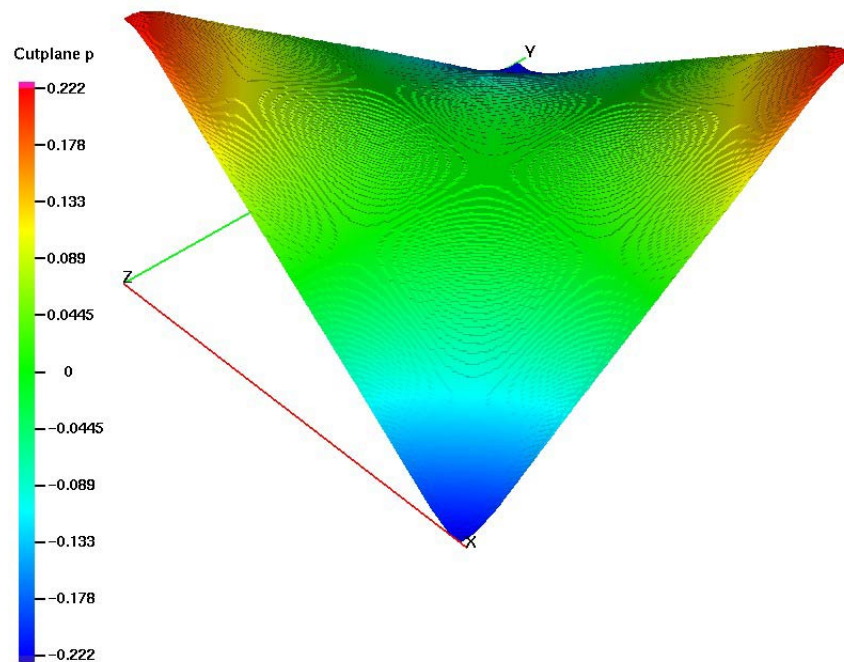


Figura 6.13: Presión Aproximada

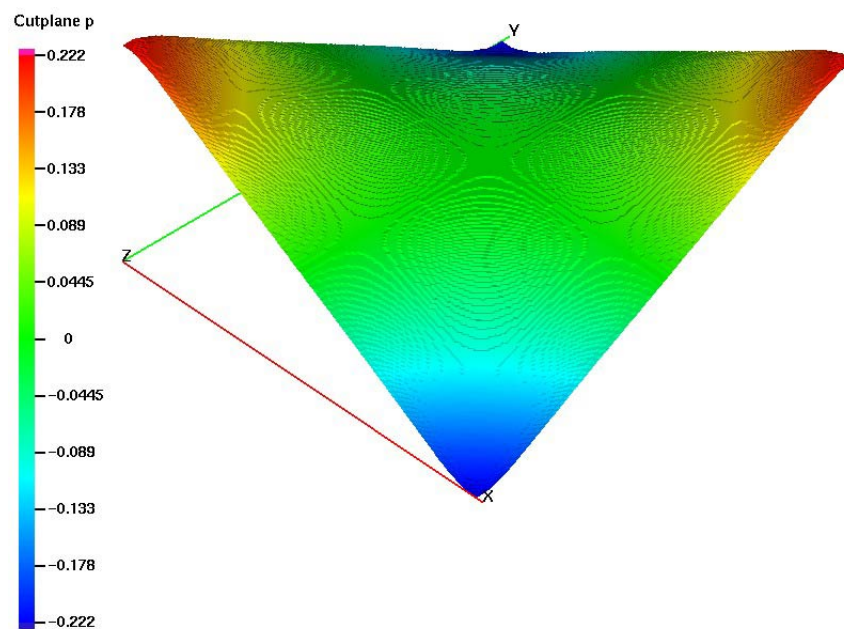


Figura 6.14: Presión Exacta

6.2. Test con condiciones de frontera no nula

Este ejemplo soluciona las ecuaciones de Stokes (2.1) en $\Omega = (0,1)^2$, además $u \neq 0$ sobre $\partial\Omega$ y la fuerza externa es nula,

$$f(x, y) = 0.$$

Los componentes de la velocidad u y la presión p para este test están definidos como:

$$\begin{aligned} u_1(x, y) &= 20xy^3 \\ u_2(x, y) &= 5x^4 - 5y^4 \\ p(x, y) &= 60x^2y - 20y^3 \end{aligned}$$

En el cuadro 6.4 se presentan las iteraciones al resolver el test con condiciones de frontera no nula con y sin preconditionamiento refinando el dominio varias veces, también se indican los grados de libertad y los números de celdas. Se puede observar que el número de iteraciones es menor en el problema preconditionado pero la diferencia en las iteraciones se hace mayor a medida que se refina muchas veces. El tiempo de ejecución del programa fue casi siempre mayor cuando se preconditiono la matriz, el único caso en el que no fue así es cuando se refina una sola vez aunque su diferencia es muy pequeña, la cual pudo ser causada por algún tipo de proceso paralelo en el computador que tomara gasto de cpu y memoria (cuadro 6.5). Los errores cuando se preconditiona son mejores a medida que se refina tanto para la velocidad como para la presión (cuadro 6.6).

Ref	Dof	Cells	Itera CG u y p precon	Itera CG u y p sin precon	Dif
1	123 (98+25)	4	16+9	18+15	8
2	419 (338+81)	16	39+15	39+25	10
3	1539 (1250+289)	64	77+20	80+27	10
4	5891 (4802+1089)	265	156+28	161+24	1
5	23043 (18818+4225)	1024	311+30	321+24	4
6	91139 (74498+16641)	4096	616+27	636+16	9
7	362499 (296450+66049)	16384	1221+20	1256+30	45
8	1445891 (1182722+263169)	65536	2411+13	2482+89	147
Abreviaturas. Ref.: Refinado, Itera.: Iteraciones, precon.: Precondicionado, Dif.: Diferencia					

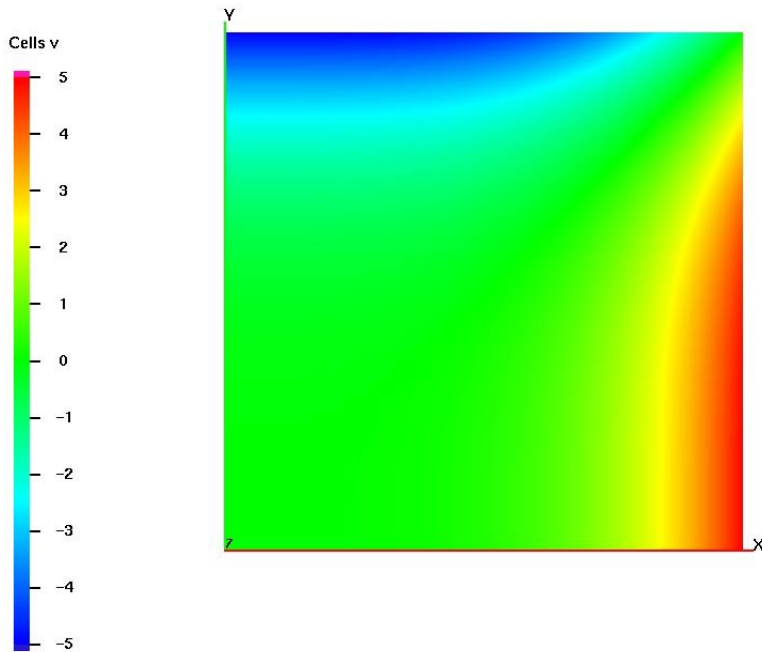
Cuadro 6.4: Iteraciones

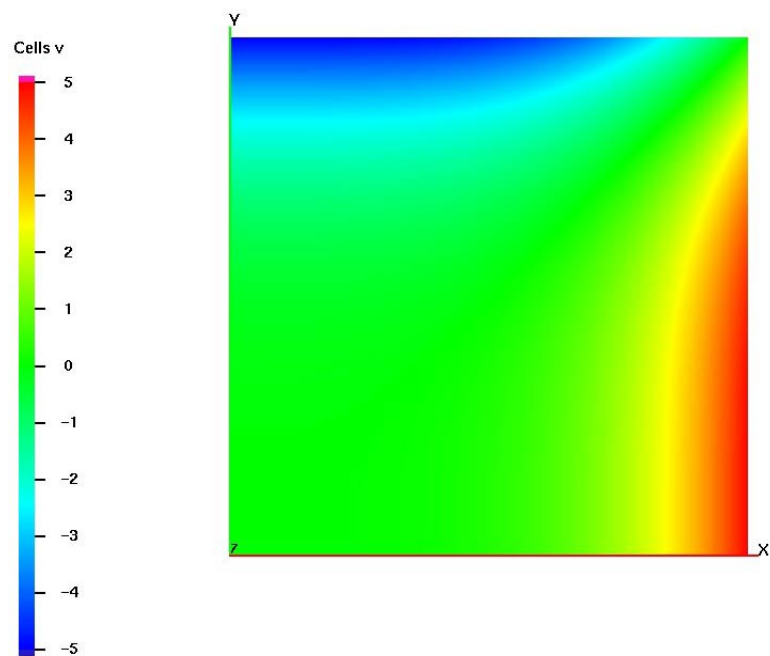
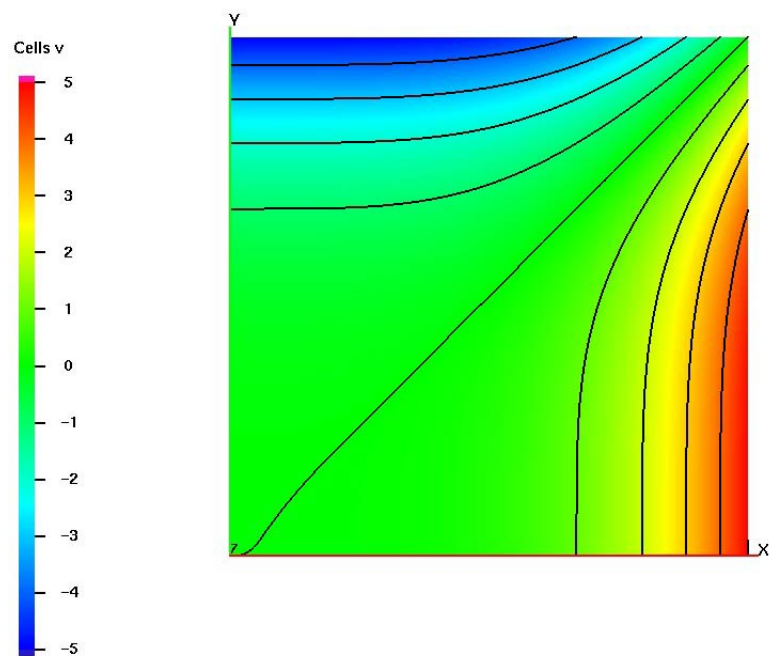
Ref	Cells	Tiempo en u y p precon	Tiempo en u y p sin precon
1	4	0m7.772s	0m7.827s
2	16	0m8.018s	0m7.730s
3	64	0m7.938s	0m7.815s
4	265	0m8.854s	0m8.278s
5	1024	0m16.764s	0m11.031s
6	4096	2m12.118s	0m35.145s
7	16384	13m18.546s	3m30.199s
8	65536	80m5.302s	29m58.523s
Abreviaturas. Ref.: Refinado, precon.: Precondicionado			

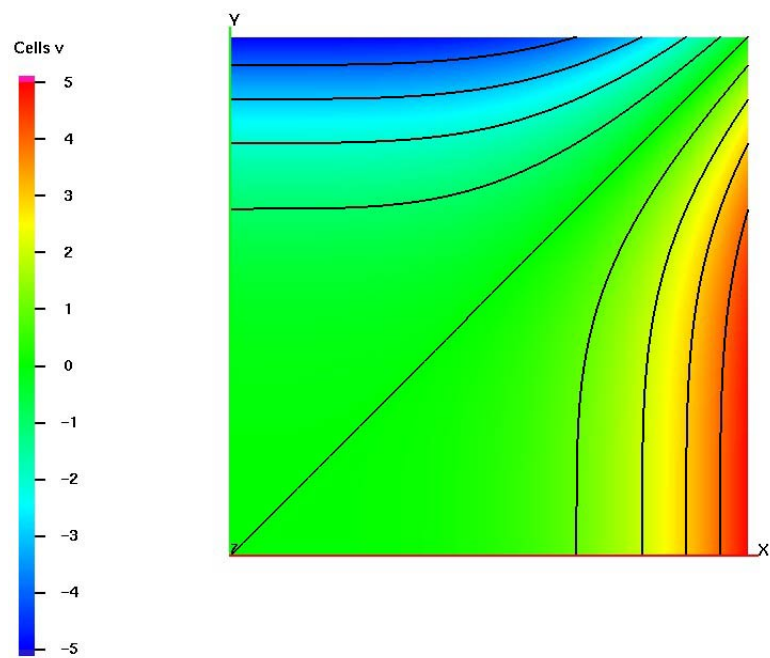
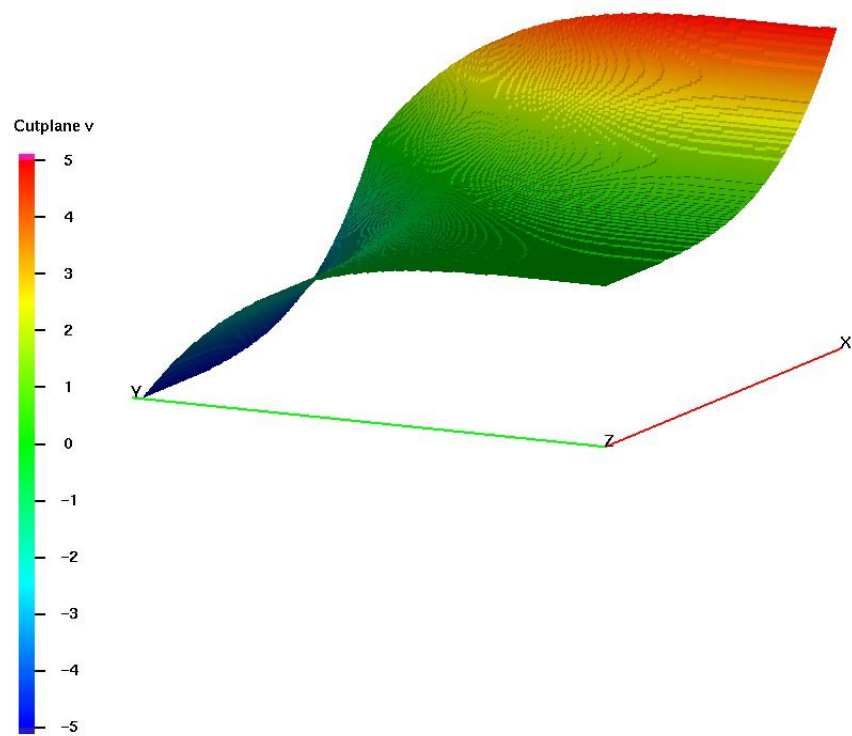
Cuadro 6.5: Tiempo

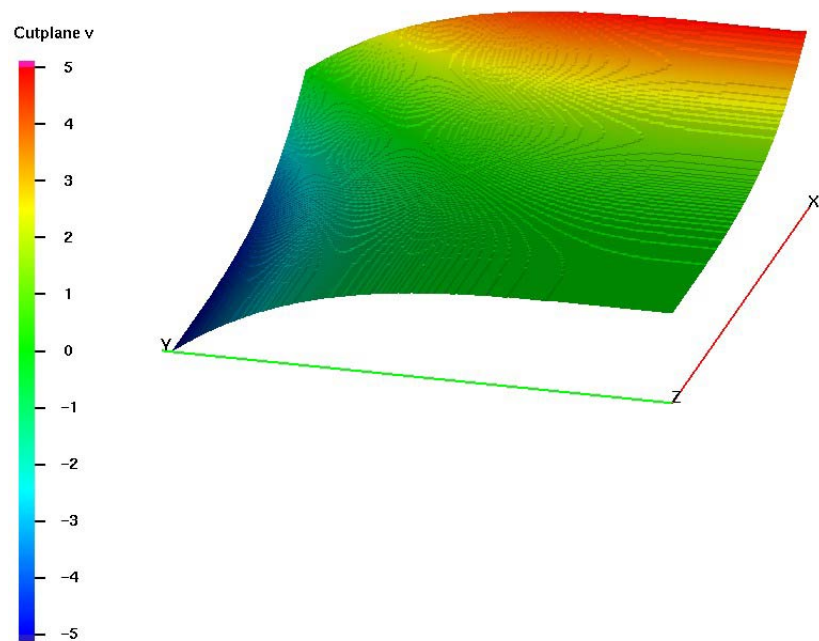
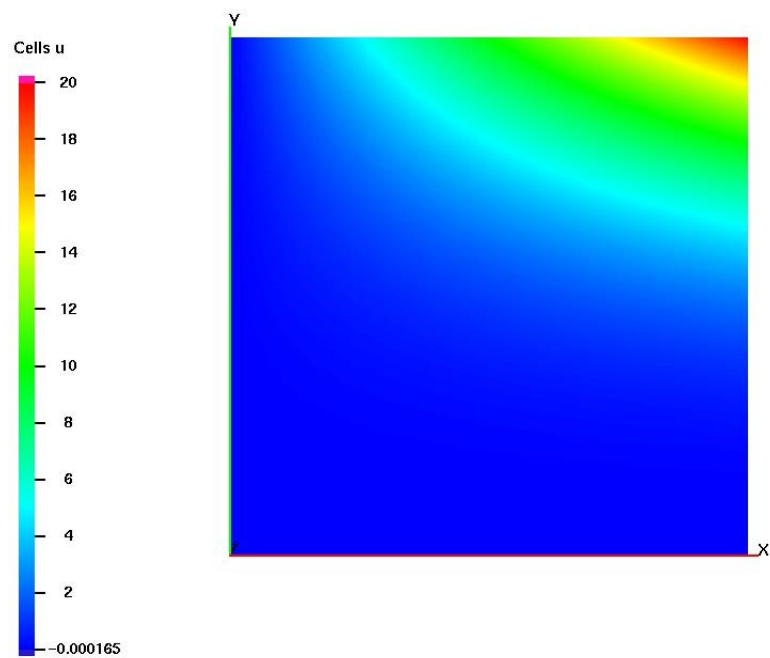
Ref	Cells	Error en u Precon	Error en p Precon	Error en u	Error en p
1	4	0,00424731	5,00062	0,00424673	5,00062
2	16	0,000372697	5	0,000364068	5,00002
3	64	0,00011168	5	0,000106013	5
4	265	0,00011825	4,99998	0,00038812	5,00001
5	1024	0,000138009	5,00001	0,000883999	5,00003
6	4096	0,00024929	5,00002	0,0105302	5,00113
7	16384	0,000941124	5,00007	0,161946	5,11671
8	65536	0,00101878	5,00023	0,967286	8,76825
Abreviaturas. Ref.: Refinado, precon.: Precondicionado					

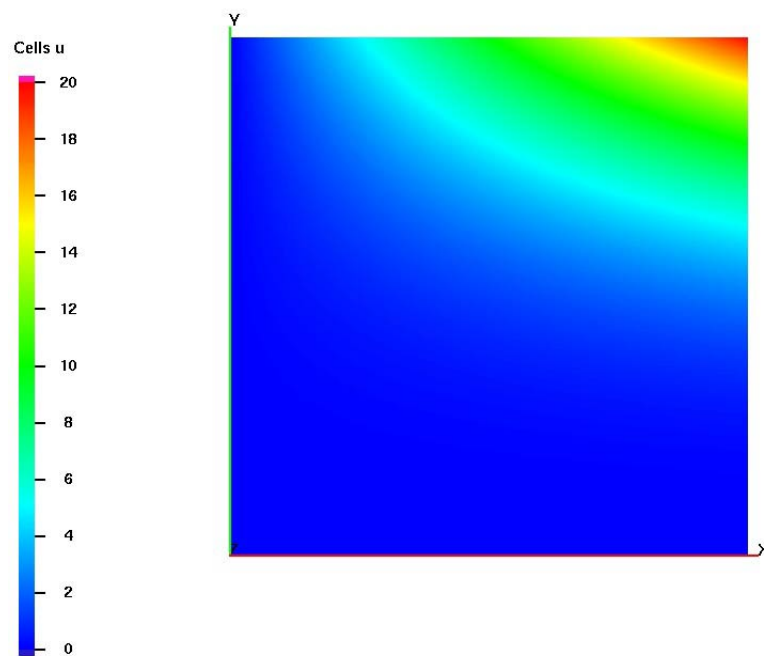
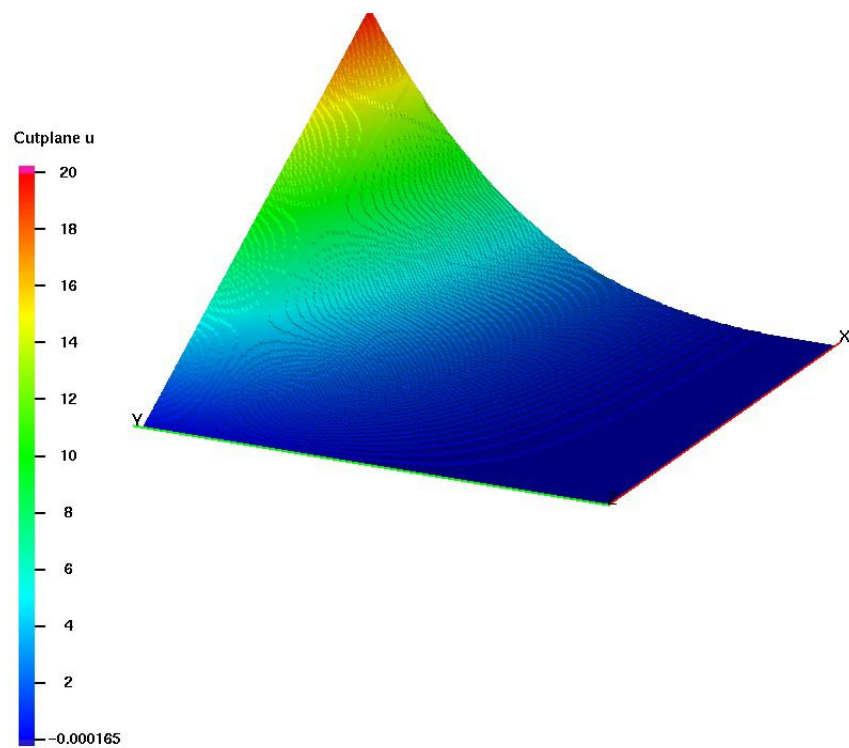
Cuadro 6.6: Error

Figura 6.15: Velocidad Aproximada v

Figura 6.16: Velocidad Exacta v Figura 6.17: Velocidad Aproximada v

Figura 6.18: Velocidad Exacta v Figura 6.19: Velocidad Aproximada v

Figura 6.20: Velocidad Exacta V Figura 6.21: Velocidad Aproximada U

Figura 6.22: Velocidad Exacta U Figura 6.23: Velocidad Aproximada U

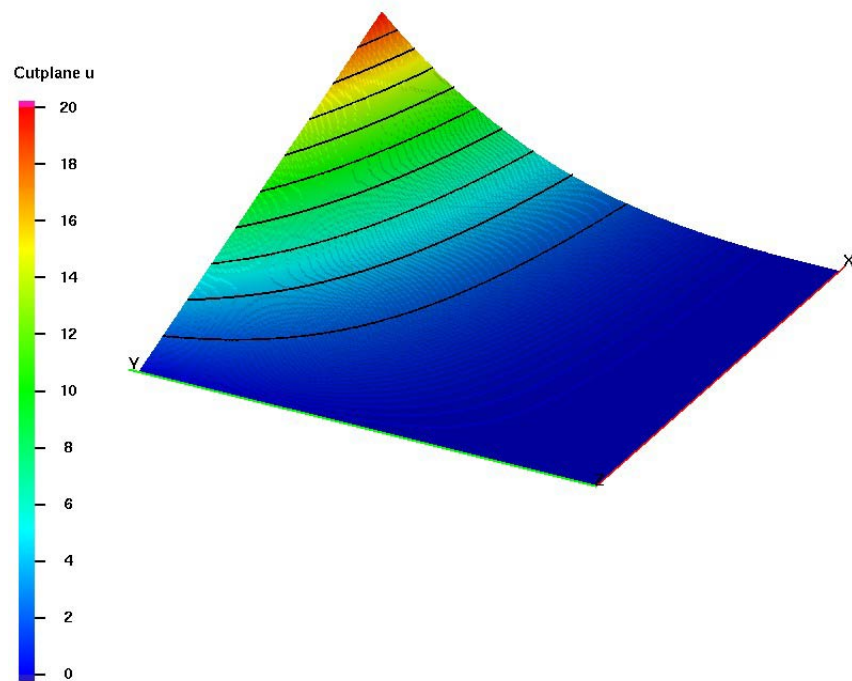
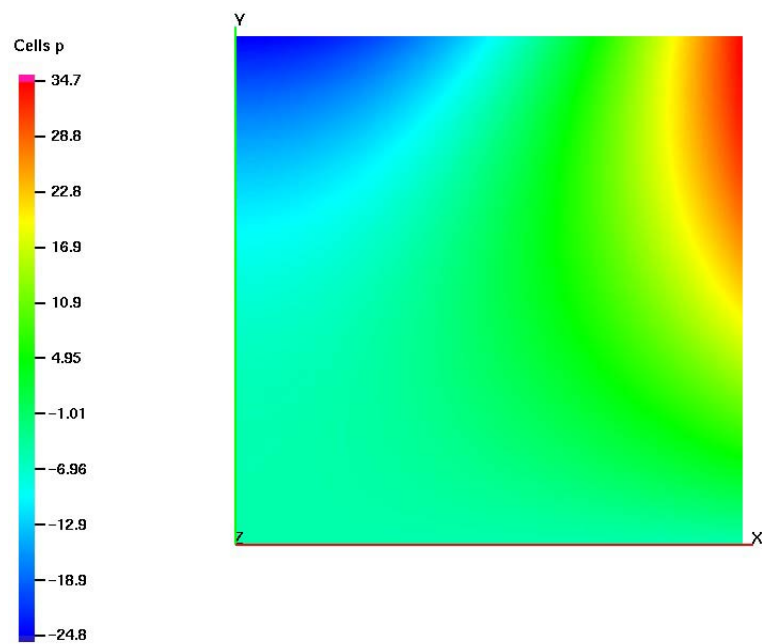
Figura 6.24: Velocidad Exacta U 

Figura 6.25: Presión Aproximada

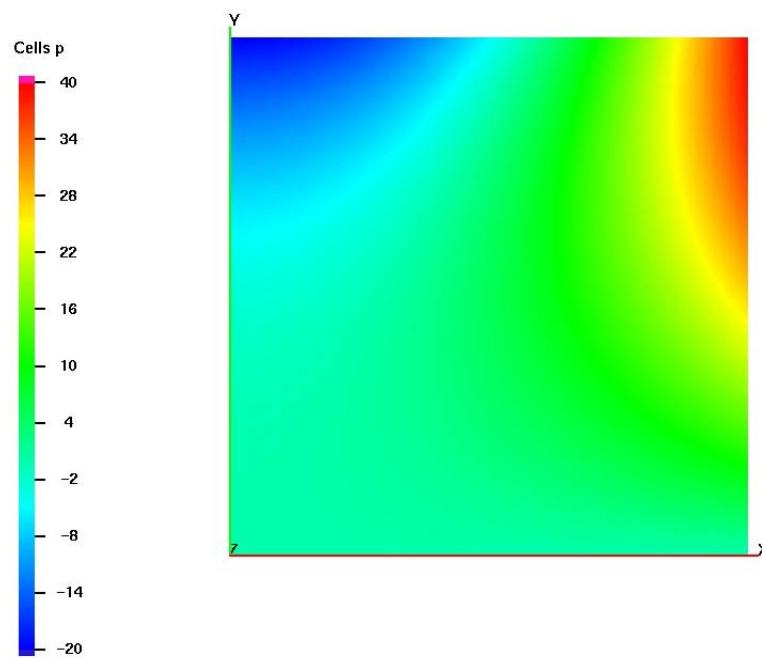


Figura 6.26: Presión Exacta

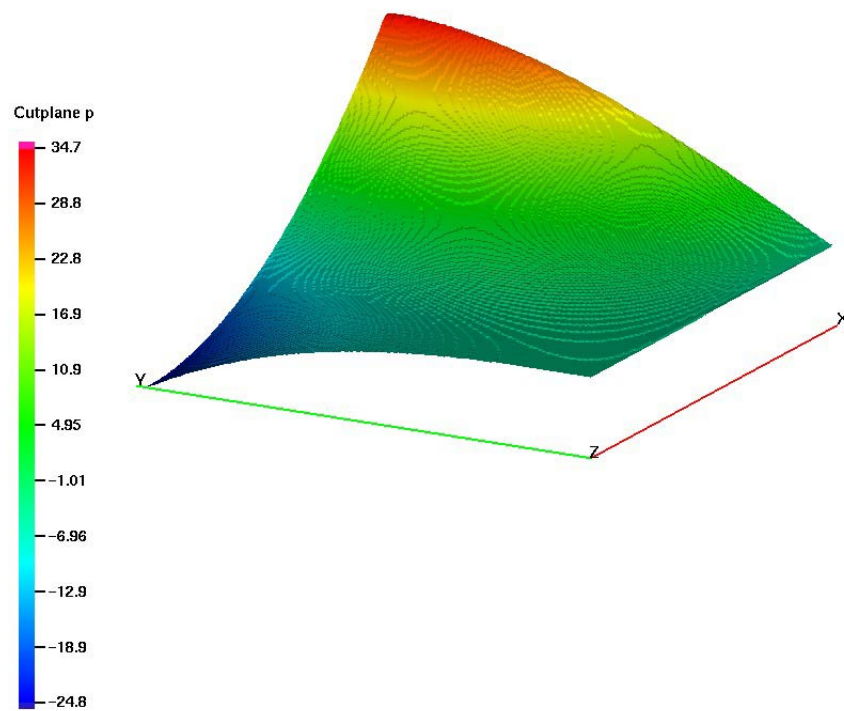


Figura 6.27: Presión Aproximada

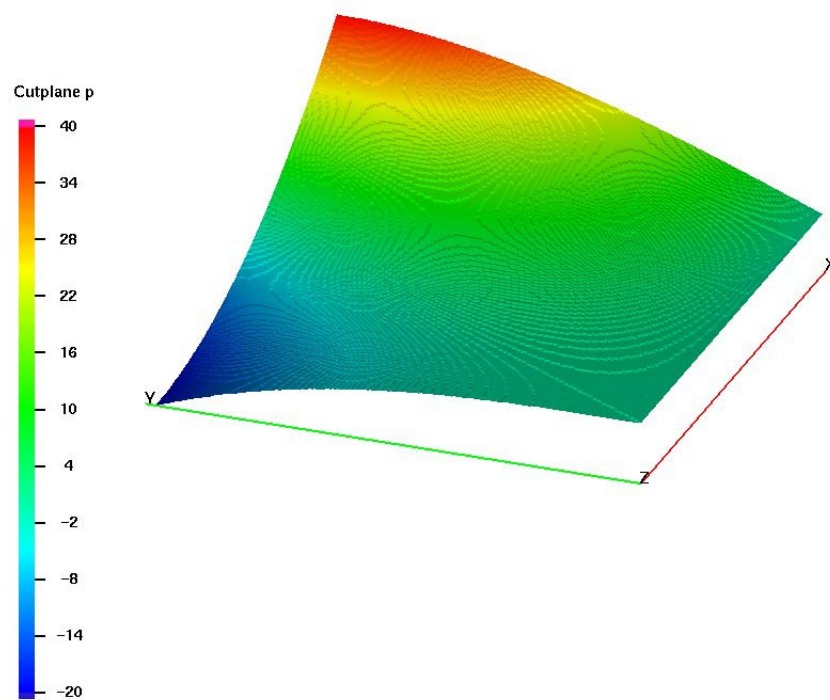


Figura 6.28: Presión Exacta

Capítulo 7

Conclusiones

En este trabajo se muestra que el problema de Stokes puede ser resuelto numéricamente en sus variables originales mediante elementos finitos mixtos, además, al preconditionar el sistema discreto resultante de aplicar el método de elementos finitos se observa que se reducen las iteraciones del método iterativo que resuelve el sistema discreto desacoplado, pero son significativas cuando se refina muchas veces. El preconditionador propuesto en este trabajo es una aproximación no costosa computacionalmente hablando del preconditionador propuesto por Murphy, Golub y Wather en [15] ya que se trabaja con la diagonal de la matriz M y no con toda la matriz. En la clase matriz inversa se implementa el uso del método iterativo CG, el residual en este método debe ser mas preciso que cuando se usa el CG para resolver los sistemas lineales desacoplados que permiten el calculo de las aproximaciones u y p .

Bibliografía

- [1] J. Nečas; *Les Méthodes Directes en Théorie des Équations Elliptiques*, Masson, 1967.
- [2] James H. Bramble; *A proof of the INF-SUP condition for THE STOKES EQUATIONS ON LIPSCHITZ DOMAINS*, Department of Mathematics, Texas A&M University, College Station
- [3] Juddy Heliana Arias Castro; *Existencia de soluciones de problemas con estructura de punto silla doble*, Tesis de grado. Universidad del Valle, Facultad de ciencias, Cali, (2010).
- [4] Bruce R. Munson, Donald F. Young, Theodore H. Okiishi; *Fundamentos de mecánica de fluidos*, Limusa Wiley, 1ª ed, (2004), Mexico DF.
- [5] M. B, Davis y G. F, Carey; *Interactive solution of the stream function-vorticity equations using a multigrid solver with finite*, Communications in Numerical Methods in Engineering, Vol 9, No. 3, (2003).
- [6] P. G. Ciarlet, *The finite element methods for elliptic problems*, North-Holland, 1ª ed, (1997), Amsterdam
- [7] W. Hackbusch; *Elliptic differential Equations: Theory and Numerical Treatment*, Springer-Verlag, Berlin, (1992).
- [8] I G Currie; *Fundamental mechanics of fluids*, Marcel Dekker, 3ª ed, (2003), Estados Unidos.
- [9] Yunus A. Cengel, John M. Cimbala; *Mecánica de fluidos*, McGraw-Hill interamericana, 1ª ed, (2006), Mexico DF.

- [10] Alexander Gutiérrez; *Estabilización de un método de elementos finitos para problemas elípticos*, Tesis de grado. Universidad del Valle, Facultad de ciencias, Cali, (2003).
- [11] Ga Hyung Jo, Hi Jun Choe; *Finite Element approximation of the vorticity-stream function equations for incompressible 2-dimensional Navier-Stokes flows*, Department of Mathematics Yonsei University, Korea.
- [12] F. Dubois; *First vorticity-velocity-pressure numerical scheme for the Stokes problem*, Computer Methods In Applied. Mechanics and Engineering, (2003).
- [13] V. Girault, P. Raviart; *Finite element Methods for the Navier-Stokes Equations. Theory and Algorithms*, Springer-Verlag, Berlin, Germany, (1986).
- [14] G. Cervantes, C. Mejia; *Precondicionamiento de Métodos Iterativos*, Revista de la Academia Colombiana de Ciencias, (2004).
- [15] M. Murphy, G. Golub, A. Wathen; *A note on preconditioning for indefinite linear systems*, SIAMJ. Sci. Compt., 21 (6) (2000).
- [16] G. Cervantes; *Introducción a los preconditionadores*, Tesis de Maestría, Universidad Nacional de Colombia, Medellín (2003).
- [17] Pavel Solin; *Partial Differential Equations and the Finite Element Method*, JOHN WILEY & SONS, INC, (2006).
- [18] F. Brezzi, M. Fortin; *Mixed and Hybrid Finite element Methods*, Springer-Verlag, Berlin, Germany, (1991).
- [19] Raviart, P.-A. and Thomas, J.-M; *Introduction à L'Analyse Numérique des Equations aux Dérivées Partielles*. Masson, 1983.
- [20] Gabriel N. Gatica; *Análisis Numérico de EDPs. El Método de Elementos Finitos Mixtos*. 2012.
- [21] Francisco Javier Sayas; *Elementos Finitos avanzados*. 2008
- [22] D. Boffi, F. Brezzi, L. Demkowicz, R. G. Duran, R. Falk and M. Fortin, *Mixed Finite Elements, Compatibility Conditions, and Applications*, Lectures Notes in Mathematics vol. 1939, 2008.

Apéndice A

Stokes2D

```
/* -----
2 * Copyright (C) 2005 - 2014 by the deal.II
  *
4 * Este programa se basa en el tutorial de deal.II en especial el
  ejemplo-20
  *
6 * The deal.II library is free software; you can use it, redistribute
  * it, and/or modify it under the terms of the GNU Lesser General
8 * Public License as published by the Free Software Foundation; either
  * version 2.1 of the License, or (at your option) any later version.
10 * The full text of the license can be found in the file LICENSE at
  * the top level of the deal.II distribution.
12 * -----
  * Author: Luis fernando Mejia Rodriguez, Universidad del Valle, 2013
14 */
#include <base/quadrature_lib.h>
16 #include <base/logstream.h>
#include <base/function.h>
18 #include <lac/block_vector.h>
#include <lac/full_matrix.h>
20 #include <lac/block_sparse_matrix.h>
#include <lac/solver_cg.h>
22 #include <lac/precondition.h>
#include <grid/tria.h>
24 #include <grid/grid_generator.h>
#include <grid/tria_accessor.h>
26 #include <grid/tria_iterator.h>
#include <dofs/dof_handler.h>
```

```

28 #include <dofs/dof_renumbering.h>
    #include <dofs/dof_accessor.h>
30 #include <dofs/dof_tools.h>
    #include <fe/fe_dgq.h>
32 #include <fe/fe_system.h>
    #include <fe/fe_values.h>
34 #include <numerics/vectors.h>
    #include <numerics/matrices.h>
36 #include <numerics/data_out.h>
    #include <fstream>
38 #include <iostream>
    #include <base/tensor_function.h>
40 #include <base/utilities.h>
    #include <lac/constraint_matrix.h>
42 #include <grid/tria_boundary_lib.h>
    #include <grid/grid_tools.h>
44 #include <grid/grid_refinement.h>
    #include <fe/fe_q.h>
46 #include <fe/mapping_q1.h>
    #include <grid/grid_out.h>
48 #include <numerics/error_estimator.h>
    #include <lac/sparse_direct.h>
50 #include <lac/sparse_ilu.h>

52 using namespace dealii;

54 template <int dim>
    class MixedLaplaceProblem
56 {
    public:
58     MixedLaplaceProblem (const unsigned int degree);
        void run ();
60     private:
        void make_grid_and_dofs ();
62        void assemble_system ();
        void solve ();
64        void compute_errors () const;
        void output_results () const;
66
        const unsigned int    degree;
68
        Triangulation<dim>    triangulation;

```

```

70     FESystem<dim>          fe;
       DoFHandler<dim>      dof_handler;

72

       BlockSparsityPattern    sparsity_pattern;
74     BlockSparseMatrix<double> system_matrix;
       ConstraintMatrix        constraints;
76     BlockVector<double>      solution;
       BlockVector<double>      system_rhs;
78 };

template <int dim>
80     class RightHandSide : public Function<dim>
    {
82     public:
       RightHandSide ();

84

       virtual void vector_value (const Point<dim> &p,
86                               Vector<double> &values) const;
       virtual void vector_value_list (const std::vector<Point<dim> > &
88                                     std::vector<Vector<double> > &
       value_list) const;
    };

90     template <int dim>
       RightHandSide<dim>::RightHandSide ()
92         :
       Function<dim> (dim+1)
94     {}

96     template <int dim>
       inline
98     void RightHandSide<dim>::vector_value (const Point<dim> &p,
                                              Vector<double> &values)
       const
100    {
       Assert (values.size() == dim+1,
102       ExcDimensionMismatch (values.size(), dim+1));
       Assert (dim >= 2, ExcNotImplemented());

104

       values(0) = 256*(p[0]*p[0]*(p[0]-1)*(p[0]-1)*(12*p[1]-6)+
106       p[1]*(p[1]-1)*(2*p[1]-1)*(12*p[0]*p[0]-12*p[0]+2))+p
       [1]-0.5;
       values(1) = -256*(p[1]*p[1]*(p[1]-1)*(p[1]-1)*(12*p[0]-6)+

```

```

108         p[0]*(p[0]-1)*(2*p[0]-1)*(12*p[1]*p[1]-12*p[1]+2))+p
        [0]-0.5;
        values(2) = 0;
110     }

    template <int dim>
112     void RightHandSide<dim>::vector_value_list (const std::vector<
        Point<dim> > &points,

                                                    std::vector<Vector<
        double> > &value_list) const
114     {
        Assert (value_list.size() == points.size(),
116             ExcDimensionMismatch (value_list.size(), points.size()));
        const unsigned int n_points = points.size();

118         for (unsigned int p=0; p<n_points; ++p)
120             RightHandSide<dim>::vector_value (points[p],
                                                    value_list[p]);

122     }

    template <int dim>
124     class BoundaryValues : public Function<dim>
    {
126     public:
        BoundaryValues () : Function<dim>(dim+1) {}
128     virtual double value (const Point<dim> &p,
                            const unsigned int component = 0) const;
130     virtual void vector_value (const Point<dim> &p,
                                Vector<double> &value) const;
132 };

    template <int dim>
134     double
        BoundaryValues<dim>::value (const Point<dim> &p,
136     const unsigned int component) const
    {
138         Assert (component < this->n_components,
            ExcIndexRange (component, 0, this->n_components));

140         if (component == 2)
142             return (p[0]-0.5)*(p[1]-0.5);
            return 0;
144     }

    template <int dim>
146     void

```

```

BoundaryValues<dim>::vector_value (const Point<dim> &p,
148 Vector<double> &values) const
{
150   for (unsigned int c=0; c<this->n_components; ++c)
       values(c) = BoundaryValues<dim>::value (p, c);
152 }

template <int dim>
154 class ExactSolution : public Function<dim>
{
156   public:
       ExactSolution () : Function<dim>(dim+1) {}
158
       virtual void vector_value (const Point<dim> &p,
160                               Vector<double> &value) const;
};

template <int dim>
void
164 ExactSolution<dim>::vector_value (const Point<dim> &p,
                                   Vector<double> &values) const
166 {
    Assert (values.size() == dim+1,
168           ExcDimensionMismatch (values.size(), dim+1));

170   values(0) = -256*(p[1]*((p[1]-1)*(2*p[1]-1))*p[0]*p[0]*((p[0]-1)*(p
[0]-1)));
       values(1) = 256* p[0]*((p[0]-1)*(2*p[0]-1))*p[1]*p[1]*(p[1]-1)*(p
[1]-1);
172   values(2) = (p[0]-0.5)*(p[1]-0.5);
}

template <int dim>
MixedLaplaceProblem<dim>::MixedLaplaceProblem (const unsigned int
degree)
176 :
       degree (degree),
178       fe (FE_Q<dim>(degree+1), dim,
           FE_Q<dim>(degree), 1),
180       dof_handler (triangulation)
{}

template <int dim>
void MixedLaplaceProblem<dim>::make_grid_and_dofs ()
184 {
    GridGenerator::hyper_cube (triangulation, 0, 1);

```

```

186 triangulation.refine_global (5);
    std::ofstream out ("grid-1.eps");
188 GridOut grid_out;
    grid_out.write_eps (triangulation, out);
190 system_matrix.clear ();
    dof_handler.distribute_dofs (fe);
192 DoFRenummering::Cuthill_McKee (dof_handler);
    std::vector<unsigned int> block_component (dim+1,0);
194 block_component[dim] = 1;
    DoFRenummering::component_wise (dof_handler, block_component);
196 {
    constraints.clear ();
198 }
    constraints.close ();
200 std::vector<unsigned int> dofs_per_block (2);
    DoFTools::count_dofs_per_block (dof_handler, dofs_per_block,
        block_component);
202 const unsigned int n_u = dofs_per_block[0],
        n_p = dofs_per_block[1];
204 std::cout << "    Number of active cells: "
        << triangulation.n_active_cells()
206        << std::endl
        << "    Number of degrees of freedom: "
208        << dof_handler.n_dofs()
        << " (" << n_u << '+' << n_p << ') '
210        << std::endl;
    {
212        BlockCompressedSimpleSparsityPattern csp (2,2);
        csp.block(0,0).reinit (n_u, n_u);
214        csp.block(1,0).reinit (n_p, n_u);
        csp.block(0,1).reinit (n_u, n_p);
216        csp.block(1,1).reinit (n_p, n_p);
        csp.collect_sizes();
218        DoFTools::make_sparsity_pattern (dof_handler, csp, constraints,
            false);
        sparsity_pattern.copy_from (csp);
220    }
    system_matrix.reinit (sparsity_pattern);
222 solution.reinit (2);
    solution.block(0).reinit (n_u);
224 solution.block(1).reinit (n_p);
    solution.collect_sizes ();

```

```

226     system_rhs.reinit (2);
228     system_rhs.block(0).reinit (n_u);
        system_rhs.block(1).reinit (n_p);
230     system_rhs.collect_sizes ();
    }
232 template <int dim>
    void MixedLaplaceProblem<dim>::assemble_system ()
234 {
    QGauss<dim>    quadrature_formula(degree+2);
236     FEValues<dim> fe_values (fe, quadrature_formula,
        update_values      | update_gradients |
238        update_quadrature_points |
        update_JxW_values);
    const unsigned int    dofs_per_cell    = fe.dofs_per_cell;
240     const unsigned int    n_q_points      = quadrature_formula.size();
    FullMatrix<double>    local_matrix (dofs_per_cell, dofs_per_cell);
242     Vector<double>        local_rhs (dofs_per_cell);
    std::vector<unsigned int> local_dof_indices (dofs_per_cell);
244     const BoundaryValues<dim> boundaryValues;
    const RightHandSide<dim>      right_hand_side;
246     std::vector<Vector<double> >    rhs_values (n_q_points,
        Vector<double>(dim+1))

    ;

248     const FEValuesExtractors::Vector velocities (0);
    const FEValuesExtractors::Scalar pressure (dim);
250     unsigned int comp_i, comp_j;
    typename DoFHandler<dim>::active_cell_iterator
252     cell = dof_handler.begin_active(),
        endc = dof_handler.end();
254     for (; cell!=endc; ++cell)
    {
256         fe_values.reinit (cell);
        local_matrix = 0;
258         local_rhs = 0;
        right_hand_side.vector_value_list(fe_values.
            get_quadrature_points(),
260            rhs_values);

        for (unsigned int q=0; q<n_q_points; ++q)
262         for (unsigned int i=0; i<dofs_per_cell; ++i)
            { comp_i = fe.system_to_component_index(i).first;
264             for (unsigned int j=0; j<dofs_per_cell; ++j)

```

```

{
266   comp_j = fe.system_to_component_index(j).first;
   if ( (comp_i < dim) && (comp_j < dim) && (comp_i == comp_j) )
268       local_matrix(i,j) += fe_values.shape_grad(i,q) *
                           fe_values.shape_grad(j,q) *
270                           fe_values.JxW(q);
       if ( (comp_i < dim) && (comp_j == dim) )
272   local_matrix(i,j) += -fe_values.shape_grad(i,q)[comp_i] *
                           fe_values.shape_value(j,q) *
274                           fe_values.JxW(q);
       if ( (comp_i == dim) && (comp_j < dim) )
276   local_matrix(i,j) += fe_values.shape_value(i,q) *
                           fe_values.shape_grad(j,q)[comp_j] *
278                           fe_values.JxW(q);
   }
280   local_rhs(i) += fe_values.shape_value( i, q)*
   rhs_values[q](comp_i) *
282   fe_values.JxW(q);
}

284   cell->get_dof_indices (local_dof_indices);

286

   for (unsigned int i=0; i<dofs_per_cell; ++i)
288       for (unsigned int j=0; j<dofs_per_cell; ++j)
           system_matrix.add (local_dof_indices[i],
290                           local_dof_indices[j],
                           local_matrix(i,j));
292   for (unsigned int i=0; i<dofs_per_cell; ++i)
       system_rhs(local_dof_indices[i]) += local_rhs(i);

294

   std::map<unsigned int, double> boundary_valuess;
296   std::vector<bool> component_mask (dim+1, true);
   component_mask[dim] = false;
298   VectorTools::interpolate_boundary_values (dof_handler,
       0,
300       boundaryValues,
       boundary_valuess,
302       component_mask);
   MatrixTools::apply_boundary_values (boundary_valuess,
304       system_matrix,
       solution,
306       system_rhs);

```

```

    }
308 }
    template <class Matrix>
310 class InverseMatrix : public Subscriptor
    {
312     public:
        InverseMatrix (const Matrix &m);
314
        void vmult (Vector<double>          &dst,
316                  const Vector<double> &src) const;

318     private:
        const SmartPointer<const Matrix> matrix;
320 };
    template <class Matrix>
322 InverseMatrix<Matrix>::InverseMatrix (const Matrix &m)
        :
324         matrix (&m)
    {}
326 template <class Matrix>
    void InverseMatrix<Matrix>::vmult (Vector<double>          &dst,
328                                     const Vector<double> &src) const
    {
330     SolverControl solver_control (src.size(), 1e-07);
        SolverCG<>      cg (solver_control);
332
        dst = 0;
334
        cg.solve (*matrix, dst, src, PreconditionIdentity());
336 }
    class SchurComplement : public Subscriptor
338 {
    public:
340     SchurComplement (const BlockSparseMatrix<double> &A,
                      const InverseMatrix<SparseMatrix<double> > &Minv)
        ;
342     void vmult (Vector<double>          &dst,
                  const Vector<double> &src) const;
344     private:
        const SmartPointer<const BlockSparseMatrix<double> > system_matrix
        ;

```

```

346     const SmartPointer<const InverseMatrix<SparseMatrix<double> > >
      m_inverse;
      mutable Vector<double> tmp1, tmp2;
348 };
      SchurComplement::SchurComplement (const BlockSparseMatrix<double> &A,
350                                         const InverseMatrix<SparseMatrix<
      double> > &Minv):
          system_matrix (&A),
352          m_inverse (&Minv),
          tmp1 (A.block(0,0).m()),
354          tmp2 (A.block(0,0).m())
      {}
356 void SchurComplement::vmult (Vector<double>          &dst,
      const Vector<double> &src) const
358 {
      system_matrix->block(0,1).vmult (tmp1, src);
360 m_inverse->vmult (tmp2, tmp1);
      system_matrix->block(1,0).vmult (dst, tmp2);
362 }
      class ApproximateSchurComplement : public Subscriptor
364 {
      public:
366     ApproximateSchurComplement (const BlockSparseMatrix<double> &A);

368     void vmult (Vector<double>          &dst,
      const Vector<double> &src) const;

370     private:
372     const SmartPointer<const BlockSparseMatrix<double> > system_matrix
      ;

374     mutable Vector<double> tmp1, tmp2;
      };
376 ApproximateSchurComplement::ApproximateSchurComplement (const
      BlockSparseMatrix<double> &A)
      :
378     system_matrix (&A),
      tmp1 (A.block(0,0).m()),
380     tmp2 (A.block(0,0).m())
      {}
382 void ApproximateSchurComplement::vmult (Vector<double>          &dst,

```

```

                                const Vector<double> &src)

    const
384 {
    system_matrix->block(0,1).vmult (tmp1, src);
386 system_matrix->block(0,0).precondition_Jacobi (tmp2, tmp1);
    system_matrix->block(1,0).vmult (dst, tmp2);
388 }

    class ApproximateM : public Subscriptor
390 {
    public:
392     ApproximateM (const BlockSparseMatrix<double> &A);

394     void vmult (Vector<double>          &dst,
                const Vector<double> &src) const;

396 private:
398     const SmartPointer<const BlockSparseMatrix<double> > system_matrix
        ;

400     mutable Vector<double> tmp1, tmp2;
    };

402 ApproximateM::ApproximateM (const BlockSparseMatrix<double> &A)
        :
404     system_matrix (&A),
        tmp1 (A.block(0,0).m()),
406     tmp2 (A.block(0,0).m())
    {}

408 void ApproximateM::vmult (Vector<double>          &dst,
                        const Vector<double> &src)

    const
410 {

412     system_matrix->block(0,0).precondition_Jacobi (dst, src);
    }

414 template <int dim>
    void MixedLaplaceProblem<dim>::solve ()
416 {
    const InverseMatrix<SparseMatrix<double> >
418     m_inverse (system_matrix.block(0,0));
    Vector<double> tmp (solution.block(0).size());
420 Vector<double> tmp2 (solution.block(0).size());
    {

```

```

422     Vector<double> schur_rhs (solution.block(1).size());

424     m_inverse.vmult (tmp, system_rhs.block(0));
     system_matrix.block(1,0).vmult (schur_rhs, tmp);
426     schur_rhs -= system_rhs.block(1);

428     SchurComplement
         schur_complement (system_matrix, m_inverse);
430
     ApproximateSchurComplement
432     approximate_schur_complement (system_matrix);

434     InverseMatrix<ApproximateSchurComplement>
         preconditioner1 (approximate_schur_complement);
436
     SchurComplement
438     preconditioner2 (schur_complement);

440     SolverControl solver_control (solution.block(1).size(),
         1e-04);
442     SolverCG<>      cg (solver_control);

444     cg.solve (schur_complement, solution.block(1), schur_rhs,
         preconditioner1);
446     constraints.distribute (solution);
     std::cout << solver_control.last_step()
448             << " CG Schur complement iterations to obtain
convergence."
             << std::endl;
450 }
{
452     system_matrix.block(0,1).vmult (tmp, solution.block(1));
     tmp *= -1;
454     tmp += system_rhs.block(0);

456     ApproximateM
         approximateM(system_matrix);
458
     ApproximateM
460     preconditioner3 (approximateM);

462     SolverControl solver_control (solution.block(0).size(),

```

```

1e-04);
464 SolverCG<>      cg (solver_control);

466 cg.solve (system_matrix.block(0,0), solution.block(0), tmp,
           preconditioner3);
468 constraints.distribute (solution);

470 std::cout << solver_control.last_step()
           << " CG Schur complement iterations to obtain
           convergence."
472           << std::endl;
    }
474 }

template <int dim>
476 void MixedLaplaceProblem<dim>::compute_errors () const
{
478     const ComponentSelectFunction<dim>
           pressure_mask (dim, dim+1);
480     const ComponentSelectFunction<dim>
           velocity_mask(std::make_pair(0, dim), dim+1);
482
           ExactSolution<dim> exact_solution;
484     Vector<double> cellwise_errors (triangulation.n_active_cells());

486     QTrapez<1>      q_trapez;
           QIterated<dim> quadrature (q_trapez, degree+2);
488
           VectorTools::integrate_difference (dof_handler, solution,
           exact_solution,
490           cellwise_errors, quadrature,
           VectorTools::L2_norm,
492           &pressure_mask);
           const double p_l2_error = cellwise_errors.l2_norm();
494
           VectorTools::integrate_difference (dof_handler, solution,
           exact_solution,
496           cellwise_errors, quadrature,
           VectorTools::L2_norm,
498           &velocity_mask);
           const double u_l2_error = cellwise_errors.l2_norm();
500
           std::cout << "Errors: ||e_p||_L2 = " << p_l2_error

```

```

502         << " , ||e_u||_L2 = " << u_l2_error
          << std::endl;
504     }
    template <int dim>
506 void MixedLaplaceProblem<dim>::output_results () const
    {
508         std::vector<std::string> solution_names;
          switch (dim)
510         {
              case 2:
512                 solution_names.push_back ("u");
                    solution_names.push_back ("v");
514                 solution_names.push_back ("p");
                    break;
516
              case 3:
518                 solution_names.push_back ("u");
                    solution_names.push_back ("v");
520                 solution_names.push_back ("w");
                    solution_names.push_back ("p");
522                 break;
524
              default:
                    Assert (false, ExcNotImplemented());
526         }
          DataOut<dim> data_out;
528         data_out.attach_dof_handler (dof_handler);
          data_out.add_data_vector (solution, solution_names);
530         data_out.build_patches (degree+1);
          std::ofstream output ("solution.gmv");
532         data_out.write_gmv (output);
          }
534 template <int dim>
    void MixedLaplaceProblem<dim>::run ()
536 {
          make_grid_and_dofs();
538         assemble_system ();
          solve ();
540         compute_errors ();
          output_results ();
542     }
    int main ()

```

```

544 {
    try
546     {
        deallog.depth_console (0);

548         MixedLaplaceProblem<2> mixed_laplace_problem(2);
550         mixed_laplace_problem.run ();
    }
552 catch (std::exception &exc)
    {
554         std::cerr << std::endl << std::endl
        << "-----"
556         << std::endl;
        std::cerr << "Exception on processing: " << std::endl
558         << exc.what() << std::endl
        << "Aborting!" << std::endl
560         << "-----"
        << std::endl;

562         return 1;
    }
564 catch (...)
    {
566         std::cerr << std::endl << std::endl
568         << "-----"
        << std::endl;
570         std::cerr << "Unknown exception!" << std::endl
        << "Aborting!" << std::endl
572         << "-----"
        << std::endl;
574         return 1;
    }
576 return 0;
}

```

Stokes-2D.cc