UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Simulación de Líneas de Corriente Optimizada Mediante Cómputo Paralelo de Memoria Compartida

TESIS

Que para obtener el título de

Ingeniero Petrolero

P R E S E N T A

Rafael Alvarez Jiménez

DIRECTOR DE TESIS

Dr. Víctor Leonardo Teja Juárez

Ciudad Universitaria, Cd. Mx., 2022

Agradecimientos

A mis padres, Adriana Jiménez Mendoza y Eduardo Alvarez Vega, por su apoyo y cariño incondicional durante toda mi vida y por nunca dudar de mí. Sin ustedes nada de esto habría sido posible.

Al piloto Diego Alvarez, por la fortuna de que mi hermano sea también de mis amigos más preciados.

A Angelina Mendoza Martínez, Rafael Jiménez Valdovinos, Mariela Jiménez Mendoza y Gerardo Jiménez Mendoza, por sus ejemplos de vida, porque desde siempre han sido otros padres para mí y por todo el amor que he sentido de su parte hasta en el más mínimo detalle. Que sepan que todo es mutuo.

A mis tías y tíos Alicia Navarro, Jorge Ochoa, Nancy Ochoa Navarro, David Alvarez Vega, a mi hermanita Isa y a mis amigos Wil y Adri, por todas aquellas comidas que se volvieron cenas y hasta desayunos, por las buenas charlas, por los consejos, por querernos y procurarnos como lo hemos hecho.

A Zuri Benítez Luna, por ser mi mejor amiga, por crecer juntos, por tanto amor, comprensión y ternura y por la motivación para poder concluir este trabajo. Te amo.

A la UNAM, por la formación profesional y humana que me brindaron sus docentes desde el bachillerato y por todos los amigos y amigas que hice dentro de sus instalaciones.

Al Dr. Víctor Leonardo Teja Juárez, porque al discretizar el proyecto final de su clase finalmente comprendí cómo algo tan complejo como un yacimiento puede modelarse con matemáticas, porque como su tesista pude seguir aprendiendo de lo que me apasiona y por su invaluable tiempo y apoyo para la realización de esta tesis.

A mis sinodales, los profesores: Dr. Rodolfo Gabriel Camacho Velázquez, M.C. Luis Loera Barona, Dr. Erick Luna Rojero, Dr. Víctor Hugo Arana Ortíz, porque con sus comentarios hicieron de este un mejor trabajo y pude aprender un poco más.

A mis petroamigos Beto, Fer, Gaby, Memo, Martín, Ubi, Alexey, y todos aquellos que sin querer omito, por el compañerismo durante la licenciatura y por hacer del estrés universitario una experiencia mucho más llevadera.

A mi amiga peluda Blacky, por acompañarme en tantas noches de desvelos y por tanto amor en forma de lamidas, meneadas de cola, y bolas de pelo por toda la casa.

Al proyecto PAPIIT:IA106820 por facilitar los recursos computacionales para la realización de este trabajo.

A todos los que estuvieron ahí para mí, gracias. Este logro también es suyo.

Índice general

Índice de figuras iv

Índice de tablas v

Índice de códigos vi

Resumen vii

1. Introducción 1
- 1.1. Revisión histórica . . . 3
- 1.2. Objetivos de este trabajo . . . 4
- 1.3. Estructura de la tesis . . . 5

2. Líneas de Corriente: Modelo matemático y numérico 6
- 2.1. Modelo Físico Conceptual . . . 6
- 2.2. Trazado de las Líneas de Corriente . . . 7
  - 2.2.1. Método Analítico . . . 7
  - 2.2.2. Método Numérico: Algoritmo de Pollock . . . 9
- 2.3. Transporte en las líneas de corriente . . . 11
  - 2.3.1. Discretización espacial . . . 11
  - 2.3.2. Cálculos de inyección de agua . . . 12

3. Herramientas del modelo computacional 14
- 3.1. Cómputo paralelo . . . 14
- 3.2. Computadora paralela de memoria compartida . . . 15
- 3.3. Computadora paralela de memoria distribuida . . . 16
- 3.4. Numba . . . 17
  - 3.4.1. Uso básico de @jit . . . 18

4. Implementación computacional del algoritmo en serie 20
- 4.1. Funciones auxiliares . . . 20
- 4.2. Validación del simulador . . . 28
  - 4.2.1. Medio homogéneo . . . 28
  - 4.2.2. Medio heterogéneo . . . 30
  - 4.2.3. Alterando la configuración de pozos . . . 31
  - 4.2.4. Agregando pozos al dominio de flujo . . . 31

5. Implementación computacional del algoritmo en paralelo y análisis

de aceleración **36**
5.1. Análisis de aceleración 38

6. Conclusiones **41**

A. Conceptos **42**
A.1. Diferencias Finitas 42
A.2. Modelo de Pozos de Peaceman 43

B. IMPES bidimensional **45**
B.1. Modelo Matemático 45
B.2. Modelo Numérico 45
B.2.1. Ecuación de Presión 45
B.2.2. Ecuación de Saturación 46

Bibliografía **48**

Índice de figuras

1.1. Comparativa de métodos de simulación utilizados en la industria. Tomado de [1]. 3

2.1. Patrón *five-spot* clásico. Tomado de [2]. 6
2.2. Líneas de corriente isocronales 8
2.3. Líneas de corriente convergiendo a pozo productor 9
2.4. Resumen del algoritmo de Pollock 11

3.1. Arquitectura de John von Neumann. Tomado de [3]. 14
3.2. Arquitectura de un multiprocesador centralizado genérico. Tomado de [3]. 16
3.3. Arquitectura de una computadora multiprocesador de memoria distribuida. Tomado de [3]. 17

4.1. Resumen del programa principal 21
4.2. Resultados en una malla 30x30 29
4.3. Resultados en diferentes tamaños de malla 29
4.4. Heterogeneidad, ejemplo 1: Resultados en una malla 60x60 30
4.5. Heterogeneidad, ejemplo 1: Resultados en una malla 60x60 31
4.6. Heterogeneidad, ejemplo 2: Resultados en una malla 60x60 32
4.7. Heterogeneidad, ejemplo 3: Resultados en una malla 60x60 32
4.8. Configuraciones de pozos 1: medio homogéneo (malla 60x60) 33
4.9. Configuraciones de pozos 1: medio heterogéneo (malla 60x60) 33
4.10. Configuraciones de pozos 2: medio homogéneo (malla 60x60) 34
4.11. Configuraciones de pozos 2: medio heterogéneo (malla 60x60) 34
4.12. Configuraciones de pozos 3: medio homogéneo (malla 60x60) 35
4.13. Configuraciones de pozos 3: medio heterogéneo (malla 60x60) 35

5.1. Combinaciones de opciones de compilación utilizadas. 38
5.2. Comparativa de rendimiento entre las diversas configuraciones utilizadas. Resultados en segundos. 40
5.3. Comparativa de rendimiento entre las diversas configuraciones utilizadas. 40

Índice de tablas

1.1. Clasificación general de los simuladores de yacimientos. Modificado de [4] 2

2.1. Datos de simulación para líneas de corriente analíticas. 8

4.1. Datos de simulación para líneas de corriente numéricas. 28

5.1. Comparativa de tiempos de ejecución para los ejemplos en medio homogéneo. 39
5.2. Comparativa de tiempos de ejecución para los ejemplos en medio heterogéneo en mallas de 60x60. 39

Índice de códigos

3.1. Ejemplo de uso de @njit 18
3.2. Ejemplo de uso de diversas opciones de compilación 18
3.3. Ejemplo de varias funciones usando @njit 19
4.1. Programa principal del simulador en serie 21
4.2. *Init*: Función que calcula los puntos de partida de las líneas 22
4.3. *pseudotiempo*: Función que calcula el pseudotiempo 23
4.4. *velocidad*: Función que calcula la velocidad 23
4.5. *cara*: Función que determina la cara de entrada de la partícula 23
4.6. *algoritmoPollock*: Función donde se implementa el Algoritmo de Pollock 24
4.7. *kr*: Función que calcula la permeabilidad relativa 26
4.8. *fractionalflow*: Función que calcula el flujo fraccional 26
4.9. *transporte*: Función que hace los cálculos de transporte 26
5.1. *principal*: Función creada a partir del programa principal del simulador en serie. 36
5.2. Programa principal del simulador en paralelo. 37

Resumen

Las líneas de corriente son una herramienta valiosa para el modelado de yacimientos de hidrocarburos. Su uso se está adoptando como complemento a los simuladores numéricos de yacimientos convencionales.

Las líneas de corriente pueden trabajar en conjunto con un simulador de yacimientos convencional o bien pueden utilizarse en el post-procesamiento de datos, destacando principalmente dos ventajas: a) la interpretación rápida y sencilla de los canales preferentes de flujo en yacimiento, y b) el tiempo de simulación es considerablemente menor, gracias al uso de la coordenada *Tiempo de Vuelo*.

El Tiempo de Vuelo (en adelante llamado τ) se calcula mediante el Algoritmo de Pollock, el cual se encarga de rastrear una partícula desde la entrada de esta al dominio de flujo hasta su salida, o bien trazarla en sentido contrario con unas modificaciones menores al algoritmo. Necesita de las velocidades en cada una de las caras de los nodos para determinar por donde saldrá la partícula hasta que esta llegue a un pozo o se estanque. Por su parte, las velocidades son obtenidas de un simulador convencional de diferencias finitas; estas se calculan con los datos de potencial en cada nodo y la movilidad de cada cara mediante la ecuación de Darcy.

Los efectos de la heterogeneidad del medio son condensados dentro de la nueva coordenada τ, así como también lleva implícitas las coordenadas cartesianas convencionales. Este cambio de coordenadas hace que un problema 3D sea reducido a un problema 1D (pasamos de usar las coordenadas cartesianas x, y, z a usar solamente τ), lo cual simplifica los cálculos y reduce el tiempo de cómputo.

Otro punto relevante es que las líneas de corriente son independientes entre sí, es decir, no hay transferencia de masa entre ellas; de esto podemos concluir que podemos paralelizar los cálculos y reducir aún más el tiempo de cálculo.

Para este trabajo se construyó un simulador de líneas de corriente bidimensional para modelar la inyección de agua a un yacimiento, el cual considera flujo incompresible e inmiscible y desprecia efectos capilares y gravitacionales. En primer lugar se hizo un código en serie, el cual es el caso base de aceleración. Con el fin de optimizarlo se usó la biblioteca *Numba*, de donde tan solo el uso su decorador estrella *@njit*, redujo drásticamente el tiempo de cómputo.

Luego, con el objetivo de optimizar el tiempo de ejecución, se hicieron algunas modificaciones al código para poder prelocalizar memoria en la computadora. Posteriormente, se le añadieron al decorador diversas opciones que hacen uso de la técnica de memoria compartida, entre ellas *parallel=True*, con el fin de optimizarlo aún más y se obtuvieron menores tiempos de ejecución.

Capítulo 1

Introducción

Un yacimiento petrolero puede definirse como un medio poroso que contiene hidrocarburos. La forma eficiente de explotarlo requiere de una visión técnica-económica. Aunque producir un yacimiento a grandes gastos usualmente implica egresos, es primordial la correcta administración de su energía y de los recursos destinados para poder obtener el máximo valor presente neto del proyecto.

Según Wiggins y Startzman la *Administración Integral de Yacimientos* es: "la aplicación del estado del arte tecnológico a un sistema de depósito conocido, dentro de cierto entorno de administración"[5]. El puntero tecnológico incluye a los recursos humanos así como las herramientas que estos emplean en sus actividades dentro de cada uno de los tres grandes campos de la Ingeniería Petrolera.

Este trabajo se enfoca en el campo de la *Ingeniería de Yacimientos*, que Muskat definió como: "la parte fundamental de la Ingeniería Petrolera que, aplicando conocimientos científicos, permite una explotación racional de las acumulaciones de hidrocarburos, es decir, obtener su máxima recuperación al menor costo, beneficiando así no solo a las empresas productoras o a los países que poseen los campos petroleros, sino también a toda la humanidad al garantizar el suministro adecuado y oportuno de vitales productos como el petróleo y gas"[6]. Dentro de esta, el área de *Simulación de Yacimientos* tiene una enorme importancia; Abou-Kassem la define como: "el arte de combinar física, matemáticas, ingeniería de yacimientos y programación computacional con el fin de desarrollar una herramienta para predecir el comportamiento de un yacimiento de hidrocarburos bajo varias estrategias de operación"[7]. En la tabla 1.1 se ilustra la clasificación general de simuladores.

Dentro de los métodos de recuperación de hidrocarburos se encuentran los mecanismos primarios, secundarios y terciarios. La recuperación primaria hace referencia a la producción de fluidos con la energía disponible naturalmente en el yacimiento, la secundaria se refiere al desplazamiento (inmiscible) de aceite por inyección de agua/gas y, finalmente, la terciaria involucra métodos más avanzados como la inyección miscible de gas. En este trabajo se simulará la producción de un yacimiento de aceite por inyección de agua usando el método de Líneas de Corriente.

"Los simuladores de líneas de corriente aproximan los cálculos de flujo de fluidos 3D con una suma de soluciones 1D a lo largo de las líneas de corriente. La elección de dirección de las líneas de corriente para los cálculos 1D hace a la aproximación

Tipo de yacimiento	Fracturado No fracturado
Nivel de Simulación	Estudios de un pozo Región del yacimiento Escala completa del yacimiento
Tipo de Simulación	Gas Aceite Negro Geotérmico Aceite volátil Gas y condensado Inyección de químicos Inyección de miscibles Recuperación térmica
Tipo de Flujo	Monofásico Bifásico Trifásico Composicional
Número de Dimensiones	Cero Una Dos Tres
Geometría	x o z r x, y x, z r, z x, y, z r, θ, z

Tabla 1.1: Clasificación general de los simuladores de yacimientos. Modificado de [4]

extremadamente efectiva para el modelado del flujo convectivo en el yacimiento. Esto es típicamente el caso cuando la heterogeneidad es el factor predominante que gobierna el comportamiento de flujo. La geometría y densidad de las líneas de corriente reflejan el impacto geológico en las rutas de flujo, brindando mejor resolución en regiones donde el flujo es más rápido."[8]

En cuanto al esfuerzo que requiere para implementarse, el método de Líneas de Corriente está por encima del método de Curvas de Declinación y Balance de Materia y por debajo tan solo de la Simulación Numérica de Yacimientos convencional y el Modelo Integral de Producción, como lo muestra la figura 1.1. No obstante, recientes desarrollos en simulación de líneas de corriente han hecho que el nivel que ocupan en la figura poco a poco se acerque a aquel de la simulación numérica convencional.

La rapidez y versatilidad del método de líneas de corriente ha llevado a muchas aplicaciones, donde algunas de las más importantes son:

- Cálculos de volumen barrido.

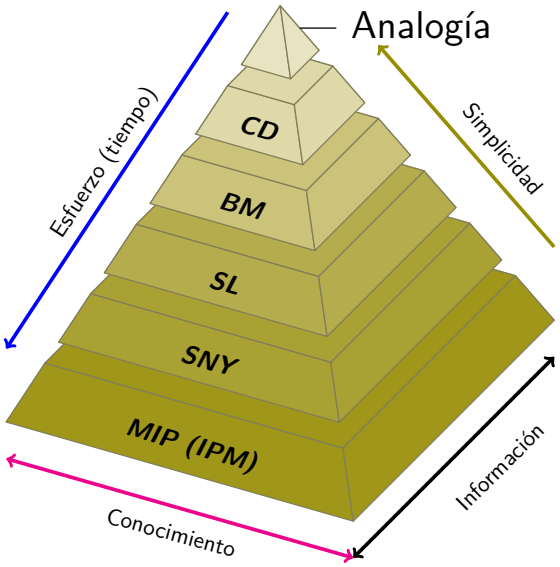


Figura 1.1: Comparativa de métodos de simulación utilizados en la industria. Tomado de [1].

- Asignación de gastos de producción y balanceo de patrones de flujo.
- Modelado de flujo de trazadores e inyección de agua.
- Ajuste histórico o integración de datos de producción.
- Simulación en mallas de escala fina.

En secciones posteriores se nota que cada línea de corriente es independiente a las demás, por lo que los cálculos pueden paralelizarse y de esta manera acelerar el código. Aunque en simulación usualmente se opta por usar Fortran o C debido al rendimiento que ofrecen para cómputo científico, en el presente trabajo se usó Python. Este lenguaje ha cobrado gran importancia en los últimos años en áreas como la Ciencia de Datos e Inteligencia Artificial donde se trabaja con grandes volúmenes de información, razón por la cual actualmente se cuenta con bibliotecas de gran calidad que optimizan los cálculos usando recursos cómo el cómputo paralelo a nivel CPU, como lo es Numba.

1.1. Revisión histórica

Esta sección incluye un breve repaso histórico de la simulación de líneas de corriente. La mayor parte de esta sección fue tomada del texto "*Streamline Simulation: Theory and Practice*"; sin embargo, se omiten los temas dedicados a métodos analíticos o semianalíticos, ya que no son objeto de este trabajo. Si el lector desea ahondar más en estos tópicos, se le recomienda leer la referencia [8].

El modelado de flujo de fluidos y transporte usando líneas y tubos de corriente se remonta al trabajo [9]. Desde entonces varios autores han aplicado y extendido los conceptos subyacentes para la aplicación para el modelado de yacimientos petroleros. Muchas de estas aplicaciones tempranas usaron aproximaciones analíticas o numéricas basadas en tubos de corriente para el modelado del desplazamiento mul-

tifásico, principalmente inyección de agua. El dominio de flujo es dividido en cierto número de tubos de corriente, y los cálculos de saturación del fluido se hacen a lo largo de los mismos. La motivación detrás del modelado con tubos de corriente fue la falta de dispersión numérica y la ventaja computacional asociada a un campo de velocidades que varía lentamente durante una inyección de agua. Sin embargo, la extensión directa de la aproximación de tubos de corriente al flujo 3D no es trivial debido a las complejidades asociadas al rastreo de la geometría de los tubos en el espacio tridimensional.

Otros dos métodos comúnmente utilizados para el transporte convectivo son el rastreo de partícula Lagrangiano [10] y métodos interfaciales como el rastreo de frentes ([11], [12]) y de métodos de ajuste de nivel ([13]). El algoritmo de rastreo de partícula reemplaza a los contornos frontales, por ejemplo, concentraciones de trazador son sustituidas por una colección de partículas estáticamente significativas. Cada partícula representa una parcela finita de fluido, ya sea másica o volumétrica. Después, las partículas son movidas resolviendo la ecuación de velocidad a lo largo de las rutas apropiadas. La dispersión es contabilizada con un algoritmo desarrollado por [14]; después de la convección, cada partícula es reposicionada con una variación en posición proporcional a la dispersión. En general, la aproximación Lagrangiana funciona bien cerca de "frentes empinados"(perfiles de saturación con cambios abruptos) pero no tan bien en perfiles suaves. Otro inconveniente asociado a estos esquemas es la pérdida de resolución del frente con la progresión del tiempo y la respuesta de la variación estática de la concentración.

Los métodos de líneas de corriente usan conceptos del rastreo de partícula para definir las rutas en el espacio 3D ([15]; [16]; [17]). La aproximación no requiere de geometría del tubo para evaluarse explícitamente y es así idealmente adecuado para modelar flujo y transporte en tres dimensiones ([18], [19]; [20]; [21]; [22]). Esto se ha facilitado bastante gracias a la introducción del *Tiempo de Vuelo* en las líneas de corriente como variable espacial ([16]). El tiempo de vuelo es simplemente el tiempo de viaje de un trazador neutro a lo largo de las líneas de corriente. La formulación del tiempo de vuelo desacopla los efectos de heterogeneidad geológica de los cálculos de transporte. Las ecuaciones de saturación multidimensional son reducidas a series de cálculos 1D a lo largo de las líneas de corriente, que son desacopladas de la malla geológica subyacente. Esto simplifica considerablemente los cálculos de saturación. Actualmente, estos cálculos son suficientemente generales para modelar la variación temporal del campo de velocidades, flujo compresible, gravedad, flujo composicional y flujo a través de fracturas, etcétera. La simplicidad, eficiencia computacional, y una generalización lista para tres dimensiones resumen el poder de la aproximación por líneas de corriente.

1.2. Objetivos de este trabajo

Objetivo general

Desarrollar una herramienta computacional mediante la utilización de cómputo paralelo de memoria compartida portable y sencilla, con el objetivo de calcular y mostrar las líneas de corriente para analizar la dinámica del flujo de fluidos agua-aceite que ocurre en un yacimiento.

Objetivos particulares

1. Investigar el modelo matemático y numérico de líneas de corriente.
2. Desarrollar un código numérico tipo serie para resolver el flujo bifásico en dos dimensiones.
3. Aplicar la metodología para calcular las líneas de corriente al código en serie.
4. Comparar los resultados del código con los resultados reportados en la literatura.
5. Desarrollar un código numérico en paralelo tomando como base el código en serie.
6. Llevar a cabo pruebas de mejora de rendimiento comparando la velocidad de ejecución del código en serie y del código en paralelo.

1.3. Estructura de la tesis

El Capítulo 2 contiene el modelo matemático y numérico para la simulación de Líneas de Corriente. Se incluyen al inicio algunos ejemplos de trazado analítico de líneas por medio de la solución línea-fuente y después se desarrolla el modelo numérico, llegando así al algoritmo de Pollock. Posteriormente, se desarrolla la solución numérica de la ecuación de transporte específicamente para el desplazamiento inmiscible de aceite por agua, considerando flujo incompresible.

En el Capítulo 3 se abordan una serie de conceptos clave para entender cómo es posible la aceleración del código, así como una breve introducción a la biblioteca *Numba*, la cual fue utilizada para optimizar el código automáticamente..

El Capítulo 4 contiene todas las funciones que componen al simulador en serie y la explicación de las mismas mediante diagramas de flujo. En este capítulo se incluyen varios ejemplos con los cuales se validó la funcionalidad el simulador considerando medios homogéneos, heterogéneos y diversas configuraciones de pozos.

En el Capítulo 5 se muestran las modificaciones que se le hicieron al código en serie para paralelizar, incluyendo las opciones de compilación utilizadas en el decorador. También se proporcionan los resultados del desempeño de los códigos generados y se comparan contra los resultados optimizados mediante @njit. Posteriormente, se extiende esta comparación añadiendo diversas formas de compilación con el fin de identificar la que mejor rendimiento presenta.

Finalmente, en el Capítulo 6 se hacen las conclusiones de los resultados y del trabajo. Se hacen comentarios de las limitantes del simulador y el trabajo futuro propuesto.

En cuanto a los apéndices, se incluye el modelo matemático y numérico del simulador IMPES utilizado en este trabajo, así como conceptos numéricos utilizados para la discretización de las ecuaciones.

Capítulo 2

Líneas de Corriente: Modelo matemático y numérico

En este capítulo se abordan las técnicas numéricas para llevar a cabo la simulación de inyección de agua para la recuperación de hidrocarburos por el método de Líneas de Corriente. La mayoría del contenido de esta sección fue tomada del texto "*Streamline Simulation: Theory and Practice*"[8].

2.1. Modelo Físico Conceptual

El modelo conocido como *five-spot* describe el desplazamiento de aceite por agua en un dominio isótropo, en el cual se coloca un pozo productor y un pozo inyector en las esquinas opuestas del dominio. Este modo de colocar los pozos emula la simetría de colocar 1 pozo productor y 4 pozos inyectores equidistantes al pozo productor. En el modelo conceptual, se toman en cuenta las siguientes consideraciones:

1. El desplazamiento ocurre en un medio bidimensional.
2. El medio es isótropo.
3. Se consideran los efectos de la presión capilar.
4. El efecto de la fuerza de gravedad es despreciable.
5. Existe una fuente (pozo inyector) y un sumidero (pozo productor).
6. Los fluidos son incompresibles e inmiscibles.

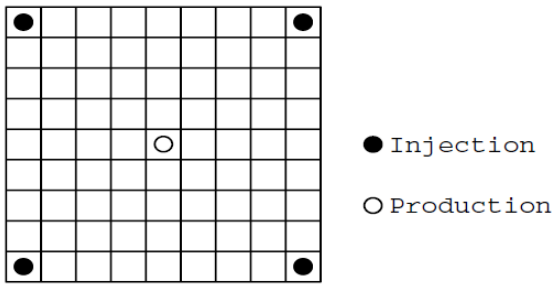


Figura 2.1: Patrón *five-spot* clásico. Tomado de [2].

2.2. Trazado de las Líneas de Corriente

2.2.1. Método Analítico

Para ilustrar el trazado de Líneas de Corriente, se hace uso de la Solución Línea Fuente y Sumidero. se debe recalcar que esta solución analítica considera:

- Flujo monofásico incompresible.
- Yacimiento infinito homogéneo de espesor constante.
- El radio de pozo es pequeño comparado con el tamaño del yacimiento.
- Múltiples pozos inyectores y productores.

Según Caudle y LeBlanc [23], la presión puede obtenerse de la solución a la ecuación de difusión y aplicando el principio de superposición:

$$p_{x,y} = \bar{p} - \frac{\mu}{4\pi kh} \sum_{i=1}^{n} q_i \ln\left((x - x_i)^2 + (y - y_i)^2\right) \tag{2.1}$$

donde $p_{x,y}$ es la presión del yacimiento y n es el número de pozos. El pozo i se ubica en (x_i, y_i) con un gasto q_i.

Sabiendo que la velocidad es la derivada de la posición, las velocidades se obtienen simplemente con las derivadas direccionales de la ecuación (2.1).

$$\vec{v_x} = \frac{1}{2\pi h\phi} \sum_{i=1}^{n} q_i \frac{x - x_i}{(x - x_i)^2 + (y - y_i)^2} \tag{2.2}$$

$$\vec{v_y} = \frac{1}{2\pi h\phi} \sum_{i=1}^{n} q_i \frac{y - y_i}{(x - x_i)^2 + (y - y_i)^2} \tag{2.3}$$

Las ecuaciones previas pueden extenderse al medio anisotrópico con una transformación de coordenadas propuesta por [9].

$$\bar{x} = \frac{x}{\sqrt{k_x}} \qquad\qquad \bar{y} = \frac{y}{\sqrt{k_y}} \tag{2.4}$$

y la solución, en unidades de campo, está dada por:

$$p_{x,y} = \bar{p} - \frac{70{,}6\mu B}{h\sqrt{k_x k_y}} \sum_{i=1}^{n} q_i \ln\left((x - x_i)^2 + \frac{k_x}{k_y}(y - y_i)^2\right) \tag{2.5}$$

$$\vec{v_x} = \frac{0{,}8936B}{h\phi}\sqrt{\frac{k_x}{k_y}} \sum_{i=1}^{n} q_i \frac{x - x_i}{(x - x_i)^2 + \frac{k_x}{k_y}(y - y_i)^2} \tag{2.6}$$

$$\vec{v_x} = \frac{0{,}8936B}{h\phi}\sqrt{\frac{k_x}{k_y}} \sum_{i=1}^{n} q_i \frac{y - y_i}{(x - x_i)^2 + \frac{k_x}{k_y}(y - y_i)^2} \tag{2.7}$$

Una vez calculadas las velocidades, es posible rastrear el avance de la Línea de Corriente dentro de un incremento de tiempo.

$$x_{i+1} = x_i + (\vec{v_{xi}})\Delta t \tag{2.8}$$

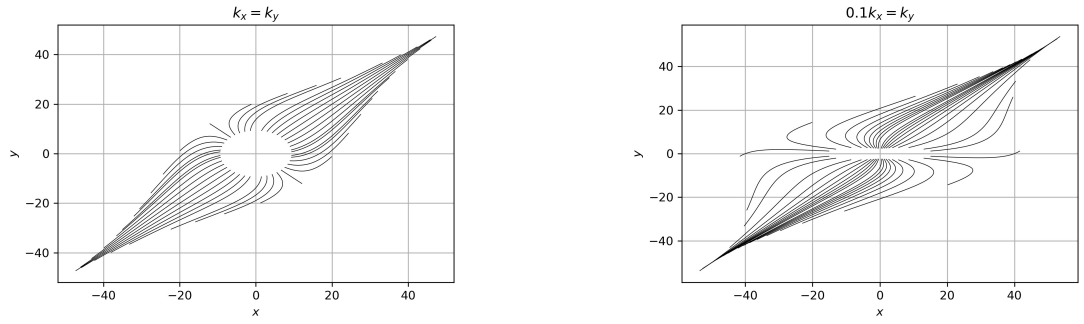
$$y_{i+1} = y_i + (\vec{v_{yi}})\Delta t \tag{2.9}$$

Ahora se cuenta con todos los elementos para programar una rutina que trace las trayectorias. Se puede usar un ciclo que tenga como límite un tiempo determinado o bien una posición (hasta que la línea converja a un pozo productor, por ejemplo). Ambos ejemplos fueron realizados usando los datos de la tabla 2.1 y los resultados se muestran en las figuras 2.2 y 2.3.

Es preciso aclarar que aunque todas las líneas de corriente surgen desde las coordenadas del pozo inyector, para programarlo se usó una aproximación diferente. Con el fin de crear diferentes trayectorias, se deben crear diferentes puntos iniciales (para cada línea de corriente). Con este fin, se hizo un círculo alrededor del pozo inyector que contiene a los puntos iniciales de cada línea, en este caso 40, y se repitió el algoritmo para cada una.

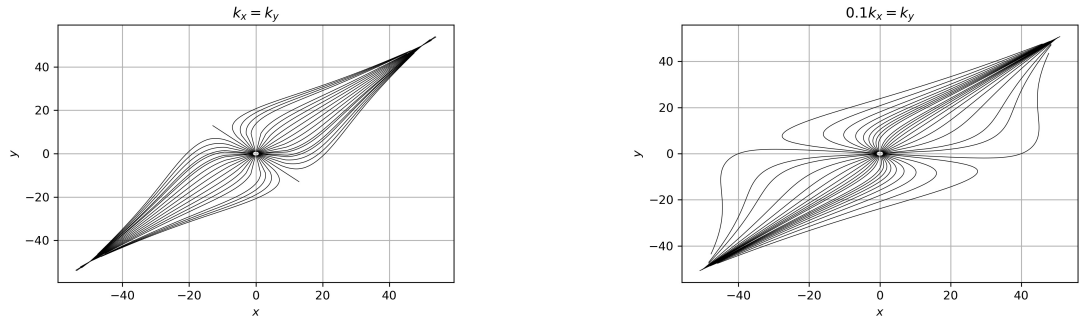
Parámetro	Valor
μ	1 [cp]
ϕ	30 %
h	3.28 [ft]
k_x	100 [mD]
B	1 [rb/STB]
q_{inj}	100 [BPD]
q_{prod}	800 [BPD]
Coord. inyector	(0,0)
Coord. productores	(50,50), (-50,-50)
dt	0.1 [d]
Tiempo Total	12, 36 [d]
Líneas	40

Tabla 2.1: Datos de simulación para líneas de corriente analíticas.



(a) Medio isótropo (b) Medio anisótropo

Figura 2.2: Líneas de corriente isocronales



(a) Medio isótropo (b) Medio anisótropo

Figura 2.3: Líneas de corriente convergiendo a pozo productor

2.2.2. Método Numérico: Algoritmo de Pollock

Para la implementación de este algoritmo, como primer paso se necesita la solución numérica de la ecuación de presión, ya sea durante la ejecución de este o como posprocesamiento del mismo. Estos cálculos no son diferentes a los de la simulación de diferencias finitas estándar.

La solución numérica proporciona presiones en el centro de las celdas y las velocidades del fluido en las caras. El algoritmo de Pollock parte de la hipótesis de que cada componente de velocidad varía linealmente entre los valores de velocidad en las caras. Esto es, la velocidad en x varía linealmente solo en esta dirección y es independiente de las velocidades en las otras direcciones, y así sucesivamente para los demás ejes, lo cual lleva al siguiente modelo de velocidades en las celdas:

$$\begin{aligned} v_x &= v_{x1} + c_x(x - x1) \\ v_y &= v_{y1} + c_y(y - y1) \\ v_z &= v_{z1} + c_z(z - z1) \end{aligned} \tag{2.10}$$

donde los coeficientes dependen de la diferencias de las velocidades de Darcy entre las caras:

$$\begin{aligned} c_x &= (v_{x2} - v_{x1})/\Delta x \\ c_y &= (v_{y2} - v_{y1})/\Delta y \\ c_z &= (v_{z2} - v_{z1})/\Delta z \end{aligned} \tag{2.11}$$

De las ecuaciones (2.10), se tiene

$$\nabla \cdot v = \sum_{i=1}^{3} c_i = c_x + c_y + c_z \tag{2.12}$$

Este es un resultado importante, ya que se ha verificado que si la solución discreta conserva flujo, también lo hace la velocidad local dentro de las celdas. De esta manera, **para flujo incompresible lejos de fuentes y sumideros, la solución de diferencias finitas quedará** $c_x+c_y+c_z = 0$. Sin embargo, para flujo compresible

el resultado será diferente de cero, ya que las compresibilidades del fluido y de la formación actuarán como términos fuente.

Las trayectorias y el tiempo de vuelo pueden calcularse por una integración directa de las velocidades en la celda.

$$\frac{d\tau}{\phi} = \frac{dx}{v_x} = \frac{dy}{v_y} = \frac{dz}{v_z} \tag{2.13}$$

Para las velocidades lineales, estas ecuaciones diferenciales pueden integrarse explícitamente e independientemente para cada dirección. Considérese una partícula partiendo de la ubicación arbitraria $(x0, y0, z0)$ dentro de una celda. La partícula puede salir de la celda a través de cualquiera de las seis caras. Se puede integrar la ec. (2.13) para obtener el tiempo de vuelo a cada una de las caras.

$$\begin{aligned}
\frac{\Delta\tau_{xi}}{\phi} &= \int_{x0}^{xi} \frac{dx}{v_{x0} + c_x(x - x0)} = \frac{1}{c_x} \ln\left(\frac{v_{xi}}{v_{x0}}\right) \\
\frac{\Delta\tau_{yi}}{\phi} &= \int_{y0}^{yi} \frac{dy}{v_{y0} + c_y(y - y0)} = \frac{1}{c_y} \ln\left(\frac{v_{yi}}{v_{y0}}\right) \\
\frac{\Delta\tau_{zi}}{\phi} &= \int_{z0}^{zi} \frac{dz}{v_{z0} + c_z(z - z0)} = \frac{1}{c_z} \ln\left(\frac{v_{zi}}{v_{z0}}\right)
\end{aligned} \tag{2.14}$$

El índice $i = 1, 2$ indica la cara del bloque en cada dirección. El algoritmo de Pollock especifica la cara de salida correcta como aquella que requiere menor tiempo de tránsito (positivo).

$$\Delta\tau = Min.Positivo(\Delta\tau_{x1}, \Delta\tau_{x2}, \Delta\tau_{y1}, \Delta\tau_{y2}, \Delta\tau_{z1}, \Delta\tau_{z2}) \tag{2.15}$$

Sabiendo el tiempo de vuelo de la partícula, es posible obtener las coordenadas de salida simplemente reorganizando la ec. (2.14).

$$\begin{aligned}
x &= x0 + v_{x0}\left(\frac{\exp\left(c_x\Delta\tau/\phi\right) - 1}{c_x}\right) = x0 + (v_{x0})\eta_x \\
y &= y0 + v_{y0}\left(\frac{\exp\left(c_y\Delta\tau/\phi\right) - 1}{c_y}\right) = y0 + (v_{y0})\eta_y \\
z &= z0 + v_{z0}\left(\frac{\exp\left(c_z\Delta\tau/\phi\right) - 1}{c_z}\right) = z0 + (v_{z0})\eta_z
\end{aligned} \tag{2.16}$$

Cuando la velocidad es uniforme (constante a lo largo del bloque en cierta dirección), por ejemplo en x, entonces $c_x = 0$. En este caso,

$$\begin{aligned}
&\frac{1}{c_x}\ln\left(\frac{v_x}{v_x 0}\right) \rightarrow \frac{x+x0}{v_{x0}} \\
&\eta_x = \left(\frac{\exp\left(c_x \Delta\tau/\phi\right)-1}{c_x}\right) \rightarrow \frac{\Delta\tau}{\phi} \\
&x = x0 + v_{x0}\left(\frac{\Delta\tau}{\phi}\right)
\end{aligned} \tag{2.17}$$

Es sencillo identificar líneas y puntos de estancamiento basados en el algoritmo de Pollock. Cuando cualquiera de las velocidades interpoladas cambien de signo a través de la celda, esta debe anularse en algún lugar dentro de la misma. El tiempo de vuelo calculado a lo largo de la celda en esa dirección será infinito, es decir, se tiene una línea de estancamiento. El tiempo de vuelo en las demás direcciones podría ser finito. Sin embargo, si todas las velocidades cambian de signo, entonces se tiene un punto de estancamiento. A manera de resumen la Figura 2.4 muestra el diagrama de flujo del algoritmo de Pollock.

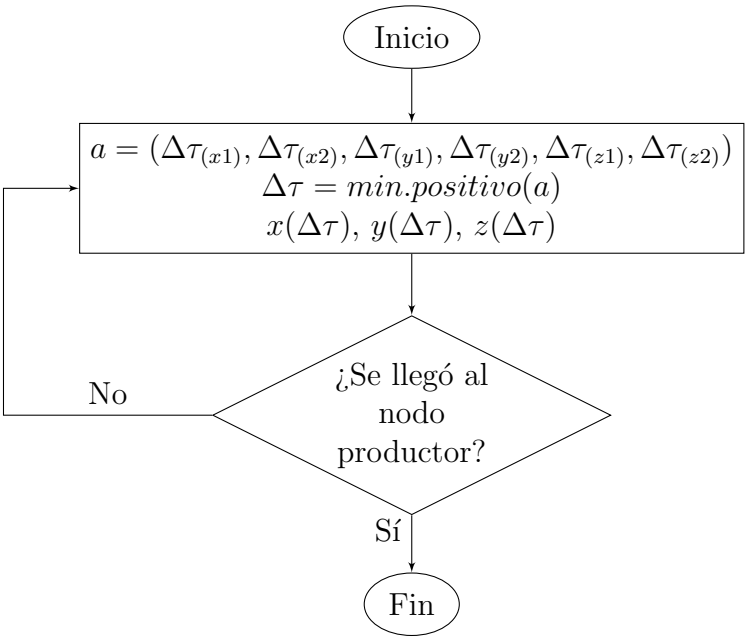


Figura 2.4: Resumen del algoritmo de Pollock

2.3. Transporte en las líneas de corriente

2.3.1. Discretización espacial

Considérese una línea de corriente en un 1/4 de modelo *five-spot*. La solución numérica de la ecuación de transporte se lleva a cabo mediante τ como coordenada espacial. Para discretizar la línea, naturalmente se opta por seleccionar los intervalos $\Delta\tau$ generados durante la integración del campo de velocidades. Esto tiene la ventaja de tener las propiedades constitutivas constantes para cada una de las celdas $\Delta\tau$;

sin embargo, la exactitud y estabilidad de la solución numérica estarán afectadas por la distribución desigual de $\Delta\tau$. En general, $\Delta\tau$ será mínimo cerca del inyector y productor y será mucho mayor en la mitad del dominio debido a las variaciones de velocidad. Como resultado, el paso temporal en la solución numérica estará limitado por el menor $\Delta\tau$.

Una de las principales fortalezas de la simulación de líneas de corriente viene de la naturaleza unidimensional de los cálculos de saturación. Nótese que los cálculos de saturación están completamente desacoplados de la malla espacial subyacente en coordenadas (x, y, z). En la simulación de líneas de corriente, las soluciones 1D se llevan a cabo independientemente de la malla física e independiente de cada línea, usando la discretización τ elegida para cada cual. Adicionalmente, la discretización a lo largo de cada línea puede optimizarse para para obtener pasos de tiempo largos para las soluciones 1D. Por ejemplo, para flujo bifásico la estabilidad de la solución estará dada por:

$$Max\left(\frac{df_w}{dS_w}\frac{\Delta t}{\Delta\tau}\right) \leq 1 \tag{2.18}$$

De esta manera, pequeños $\Delta\tau$ implicarán pasos de tiempo más pequeños. Por otro lado, si $\Delta\tau$ es muy grande, puede llevar a una pérdida de resolución espacial y distorsión de los frentes. La optimización de la solución 1D podría ser crítica para la eficiencia computacional. La habilidad de tomar pasos largos de tiempo para los cálculos de saturación sin importar la malla física subyacente es una de las grandes ventajas que no tiene la simulación de diferencias finitas.

2.3.2. Cálculos de inyección de agua

La siguiente identidad del operador es una relación muy importante en la simulación de líneas de corriente. Se utiliza para transformar las ecuaciones del espacio físico a coordenadas de tiempo de vuelo.

$$\vec{v}\cdot\nabla = \phi\frac{\partial}{\partial\tau} \tag{2.19}$$

Considérese la ecuación de conservación para la fase agua en el caso de flujo bifásico incompresible, despreciando efectos capilares y gravitatorios:

$$\phi\frac{\partial S_w}{\partial t} + \nabla\cdot(\bar{k}\lambda_w\nabla p_o) = \frac{q_w}{\rho_w} \tag{2.20}$$

Haciendo uso de la ley de Darcy y el flujo fraccional definido como $f_w = \dfrac{\lambda_w}{\lambda_t}$, la ecuación queda.

$$\phi\frac{\partial S_w}{\partial t} + \nabla\cdot(f_w\vec{v_t}) = \frac{q_w}{\rho_w} \tag{2.21}$$

Ahora, si se aplica la identidad (2.19) al flujo fraccional,

$$\nabla \cdot (f_w \vec{v_t}) = \vec{v_t} \cdot \nabla f_w = \phi \frac{\partial f_w}{\partial \tau} \tag{2.22}$$

Finalmente, sustituyendo la ec. (2.22) en la ec. (2.21) se llega a:

$$\phi \left(\frac{\partial S_w}{\partial t} + \frac{\partial f_w}{\partial \tau} \right) = \frac{q_w}{\rho_w} \tag{2.23}$$

Y lejos de fuentes y sumideros:

$$\frac{\partial S_w}{\partial t} + \frac{\partial f_w}{\partial \tau} = 0 \tag{2.24}$$

Esta ecuación puede discretizarse de la siguiente manera:

$$\frac{(S_w)^{n+1} - (S_w)^n}{\Delta t} + \frac{(\bar{f}_w)_{i+1/2} - (\bar{f}_w)_{i-1/2}}{\Delta \tau} = 0 \tag{2.25}$$

donde $(\bar{f}_w)_{i+1/2}$ es el flujo promedio a la frontera entre los nodos i e $i+1$ y n es el salto temporal.

Para el flujo entre bloques la ponderación corriente arriba es la elección más simple, que para este caso es estable y proporciona la siguiente ecuación:

$$\frac{(S_w)^{n+1} - (S_w)^n}{\Delta t} + \frac{(\bar{f}_w)_i - (\bar{f}_w)_{i-1}}{\Delta \tau} = 0 \tag{2.26}$$

Finalmente, los cálculos para la saturación del siguiente salto temporal quedan de la siguiente manera:

$$(S_w)^{n+1} = (S_w)^n - \frac{\Delta t}{\Delta \tau} \left[(\bar{f}_w)_i - (\bar{f}_w)_{i-1} \right] \tag{2.27}$$

Capítulo 3

Herramientas del modelo computacional

La primera parte de esta sección fue tomada del texto "*Cómputo paralelo para la solución de flujo en medios porosos aplicando funciones de base radial* "[3].

La parte correspondiente a la biblioteca Numba fue tomada directamente de su documentación; sin embargo, aquí solo se reúnen aquellas funcionalidades utilizadas dentro del simulador. Si el lector desea ahondar más en la biblioteca, se le recomienda revisar directamente la documentación de la bibloteca en cuestión.

3.1. Cómputo paralelo

Remontándose a la década de los 40, las computadoras secuenciales tradicionales se basan en el modelo introducido por John von Neumann (figura 3.1), el cual consiste de una unidad central de procesamiento (CPU por sus siglas en inglés), una memoria principal para almacenar información, un camino (BUS) sobre el cual fluyen los datos y un mecanismo de sincronización (reloj).

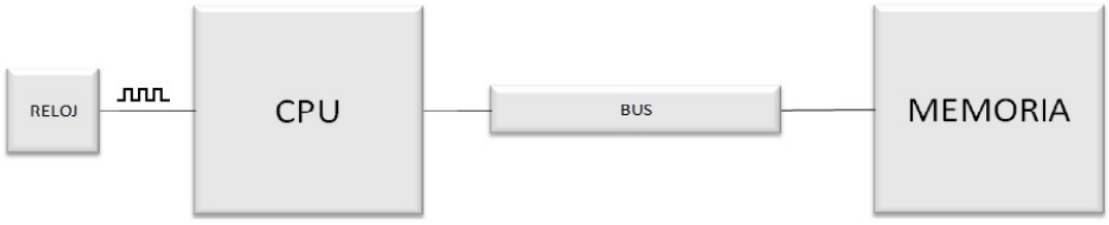


Figura 3.1: Arquitectura de John von Neumann. Tomado de [3].

El número de ciclos de reloj por segundo medidos en Hertz [Hz], caracteriza la velocidad a la cual se ejecutan las instrucciones en la computadora. La velocidad en este tipo de arquitectura está limitada por dos factores:

- razón de ejecución de las instrucciones (ciclo de reloj) y
- la velocidad a la cual se intercambia la información entre la memoria y el CPU (ancho de banda del bus de memoria).

En el primer factor, la frecuencia no puede incrementarse indefinidamente: el consumo de energía de un chip está dado por la ecuación

$$P = C_L * V_{dd}^2 * f$$

donde P es la potencia, C_L el cambio de capacitancia por ciclo de reloj (proporcional al número de transistores cuyas entradas cambian), V_{dd} es la tensión de alimentación y f es la frecuencia de reloj. Un aumento en la frecuencia aumenta la cantidad de energía utilizada en un procesador [24]; aún así no se considera en el consumo de energía, la máxima velocidad a la cual puede viajar la información a lo largo de las conexiones entre componentes, es la velocidad de la luz, en este aspecto, reducir el tiempo de ejecución implicaría acortar la longitud de las conexiones, las cuales también tienen un límite.

En el segundo factor, la velocidad está limitada por el ancho de banda del BUS de datos. Para mejorarla existen varias técnicas como incrementar el número de canales sobre los cuales el CPU accede a la memoria (Memory interleaving), incrementar la razón de cambio de información mediante el uso de una memoria relativamente pequeña y muy rápida (memoria caché) y el uso de segmentación encauzada (pipeline), donde múltiples instrucciones pueden ejecutarse simultáneamente, no significando esto, que se reduzca el número de instrucciones que se ejecutan al mismo tiempo [25].

Todas estas técnicas tienen limitaciones físicas y económicas. Una manera alterna de incrementar la razón de ejecución de instrucciones, es usando múltiples procesadores y unidades de memoria conectadas a ellos de alguna manera, es decir, una computadora paralela.

Una computadora paralela es un sistema de cómputo multiprocesador que soporta la programación paralela. De ella puede hacerse una clasificación en dos categorías importantes: multicomputadoras y multiprocesadores centralizados. Como su nombre lo implica, una multicomputadora es una computadora construida de múltiples computadoras y una red de interconexión. Los procesadores en diferentes computadoras interactúan pasándose mensajes o información entre ellos; esta clasificación es comúnmente conocida como **computadora paralela de memoria distribuida**. En contraste, un multiprocesador centralizado, también llamado multiprocesador simétrico (SMP), es un sistema más altamente integrado, en el cual todos los CPUs comparten el espacio físico de la memoria global, el que a la vez, es su medio de comunicación. De ahí que a esta clasificación se le conoce como **computadora paralela de memoria compartida** [26].

3.2. Computadora paralela de memoria compartida

Un multiprocesador centralizado es una extensión del uniprocesador de John von Neumann. CPUs adicionales se adjuntan en el bus y todos los procesadores comparten la misma memoria principal (figura 3.2). Esta arquitectura es también llamada

SMP, porque toda la memoria está en un solo lugar y tiene el mismo tiempo de acceso para cada procesador. Los datos privados son aquellos que solo un procesador usa, mientras que los datos compartidos son aquellos que múltiples procesadores usan. En un procesador centralizado, los procesadores se comunican entre ellos a través de los datos compartidos. Los diseñadores de este tipo de arquitectura deben tener en cuenta dos problemas asociados a los datos compartidos: *cache-coherency* y *sinconización* [26], donde en el primero, al haber agregado la memoria caché para reducir la contención entre los procesadores para los datos compartidos y dado que el caché es una vista de la memoria principal; en el segundo problema, deben asegurar la exclusión mutua: un solo proceso excluye temporalmente a todos los demás para usar la(s) locaclidad(es) de memoria compartida de forma que garantice la integridad de la actividad específica.

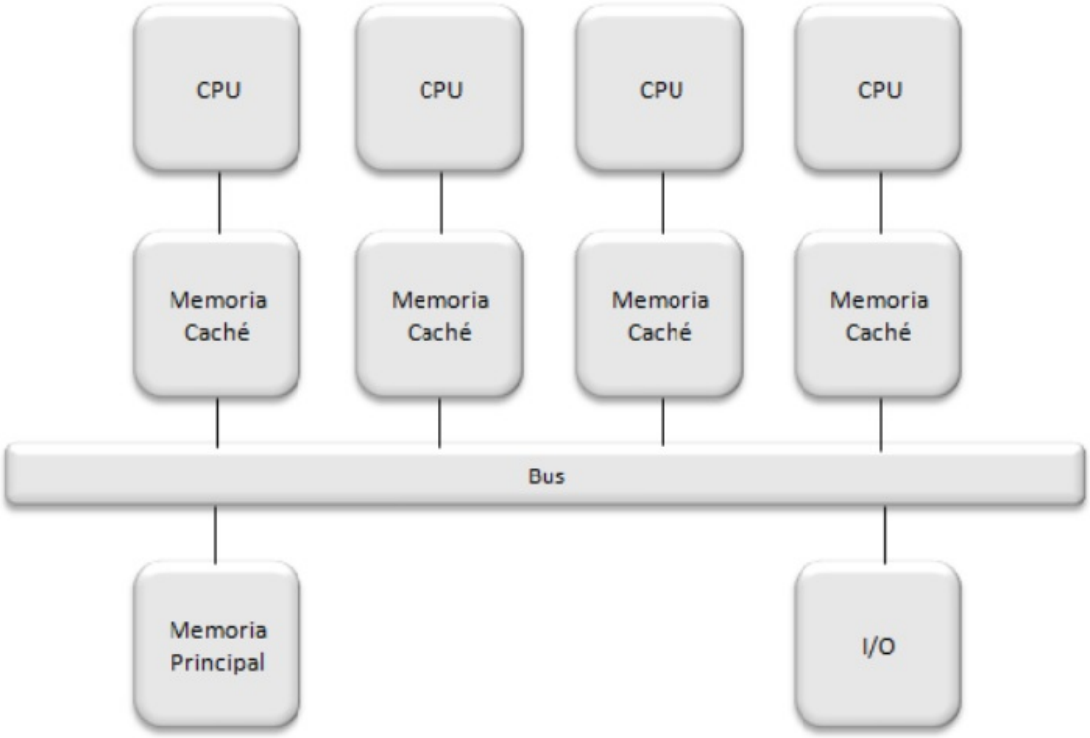


Figura 3.2: Arquitectura de un multiprocesador centralizado genérico. Tomado de [3].

3.3. Computadora paralela de memoria distribuida

En este tipo de arquitectura, cada procesador tiene su propia memoria local, lo que conlleva las siguientes ventajas: que no exista bus de memora compartida (evita problemas de ancho de banda y contención); que no haya límite para el número de procesadores (depende de la red de interconexión) y que no haya problemas de cache-coherency. Sin embargo, se tiene la desventaja de que las tareas que se ejecutan en cada procesador solo operan sobre datos locales, por lo que si se requieren datos remotos, se debe realizar una comunicación con otros procesadores, esto

implica, mayor tiempo de consumo para construir y enviar un mensaje, así como para recibirlo y desempaquetarlo. La arquitectura de una computadora distribuida aparece en la figura 3.3. Si la colección distribuida de memorias forma un espacio lógico de direcciones, la computadora paralela es llamada un multiprocesador distribuido, pero si los espacios locales de direcciones son disjuntos, es llamada una multicomputadora.

Una forma perfectamente satisfactoria para comunicar a los procesadores que se encuentran en una computadora paralela de memoria distribuida, es la interfaz de paso de mensajes (MPI), la cual es una especificación estándar que comprende un conjunto de rutinas para el manejo de procesos e intercambio de mensajes.

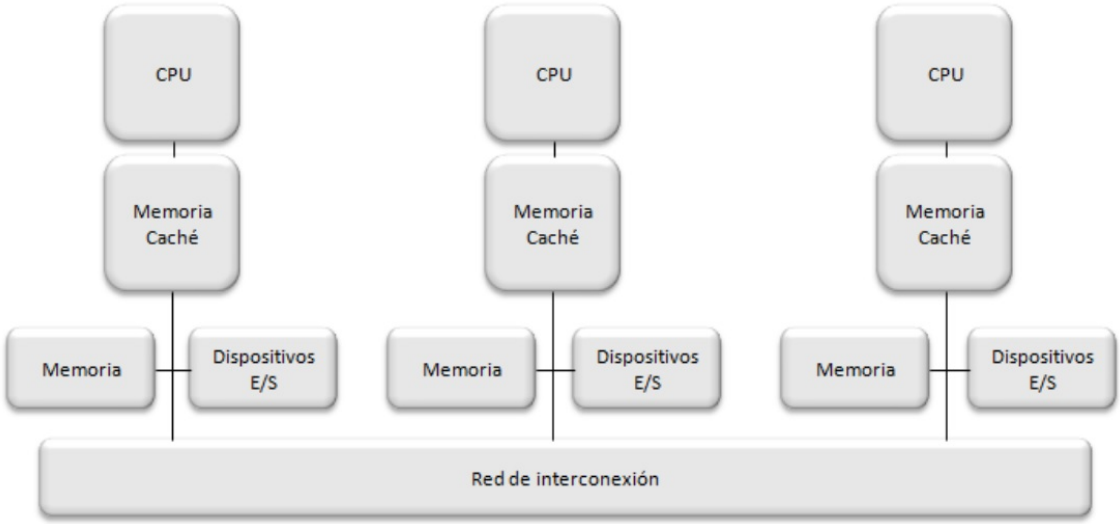


Figura 3.3: Arquitectura de una computadora multiprocesador de memoria distribuida. Tomado de [3].

3.4. Numba

Numba traduce funciones de Python a código de máquina optimizado en tiempo de ejecución usando la biblioteca compiladora LLVM, estándar en la industria. Algoritmos numéricos en Python compilados con Numba pueden acercarse a las velocidades de C o FORTRAN.

No es necesario reemplazar el interpretador de Python, ni hacer nada adicional con respecto a éste. Sólo hay que aplicar uno de los decoradores de Numba a la función y Numba hace el resto.

Numba ofrece variedad de opciones para paralelizar código para CPUs y GPUs, usualmente con tan solo unos cambios menores en el código. Su funcionalidad central, y el que más se usó en el simulador, es el decorador numba.jit(). Usando este decorador, puede marcar una función para la optimización por el compilador JIT (Just In Time) de Numba. Varios modos de invocación desencadenan diferentes opciones de compilación y comportamientos.

3.4.1. Uso básico de @jit

Compilación sencilla

La forma recomendada para usar al decorador @jit es dejando decidir a Numba donde y cómo optimizar. En este modo, la compilación será diferida hasta la primera ejecución de la función. Numba inferirá el tipo de argumentos al momento que llama la función, y generará código optimizado basado en esta información.

Código 3.1: Ejemplo de uso de @njit

```
1 from numba import jit
2
3 @jit(nopython=True)
4 def f(x,y):
5     return x+y
```

Más opciones de compilación

Se pueden pasar una serie de argumentos de palabras clave al decorador @jit.

- **Nopython:** Numba tiene dos modos de compilación: modo nopython y modo objeto. El primero produce código mucho más rápido, pero tiene limitaciones que pueden obligar a Numba a recurrir al segundo. Para evitar que Numba retroceda y, en su lugar, genere un error, hay que usar nopython=True (@njit equivale a @jit(nopython=True)).
- **Nogil:** Cada vez que Numba optimiza el código Python a código nativo que solo funciona en tipos y variables nativos (en lugar de objetos Python), ya no es necesario mantener el bloqueo de intérprete global (GIL) de Python. Numba liberará el GIL al ingresar a dicha función compilada si pasó nogil=True.

  El código que se ejecuta con el GIL liberado se ejecuta simultáneamente con otros subprocesos que ejecutan código Python o Numba (ya sea la misma función compilada u otra), lo que le permite aprovechar los sistemas de múltiples núcleos. Esto no será posible si la función se compila en modo objeto.

  Al usar nogil=True, se habrá que tener cuidado con las trampas habituales de la programación multiproceso (consistencia, sincronización, condiciones de carrera, etc.).
- **Cache:** Para evitar los tiempos de compilación cada vez que invoca un programa Python, se puede indicar a Numba que escriba el resultado de la compilación de funciones en una memoria caché basada en archivos. Esto se hace pasando cache=True.
- **Parallel:** Permite la paralelización automática (y las optimizaciones relacionadas) para aquellas operaciones en la función que se sabe que tienen semántica paralela. Esta característica se habilita pasando parallel=True y debe usarse junto con nopython=True.

Código 3.2: Ejemplo de uso de diversas opciones de compilación

```
1 @njit(nogil=True, cache=True, parallel=True)
2 def f(x,y):
```

```
3     return x+y
```

Llamar e incorporar otras funciones

Las funciones compiladas por Numba pueden llamar a otras funciones compiladas. Las llamadas a la función pueden incluso estar alineadas en el código nativo, dependiendo de la heurística del optimizador. Por ejemplo:

Código 3.3: Ejemplo de varias funciones usando @njit

```
1 @njit
2 def square(x):
3     return x ** 2
4
5 @njit
6 def hypot(x, y):
7     return math.sqrt(square(x) + square(y))
```

El decorador @jit debe agregarse a cualquier función de biblioteca de este tipo, de lo contrario Numba puede generar un código mucho más lento.

Capítulo 4

Implementación computacional del algoritmo en serie

Esta sección describe la metodología computacional para implementar la solución del flujo de fluidos bifásico e incompresible en medios porosos.

Algoritmo general e implementación

El simulador se compone de tres archivos: un archivo donde se insertan los datos de simulación (importado al archivo principal como d), otro archivo que contiene todas las funciones utilizadas en el programa principal (importado al archivo principal como A), y el archivo principal. La razón de dividirlo de esta manera fue acortar el archivo principal y para que al modificar los datos de simulación, no modifiquemos alguna otra parte del código accidentalmente.

Comencemos explicando el simulador mediante la figura 4.1. Se requieren como insumo las propiedades de la roca y los fluidos, un campo de velocidad total calculado por un simulador de diferencias finitas y finalmente datos de simulación como el número de líneas a calcular y el tiempo de simulación. Nótese que no se requieren datos de permeabilidad, ya que esta viene implícita en el campo de velocidad.

A grandes rasgos la simulación es llevada a cabo por dos ciclos. Un ciclo *for* se encarga repetir el algoritmo para las líneas de corriente definidas por el usuario; dentro de este se calcula la trayectoria y se obtiene la discretización natural de τ mediante el algoritmo de Pollock. Posteriormente, aún dentro del *for*, se entra a un ciclo *while* que controla el tiempo de simulación y es en donde se hacen los cálculos de transporte con la nueva coordenada.

4.1. Funciones auxiliares

En esta sección se enlistan y explican todas las funciones que hacen la simulación posible. Se omitirá el archivo de datos, ya que solo es un documento donde se vacía información, así como también a la función que lee las velocidades de un archivo de texto. Vale la pena recordar que la librería *numpy* es importada como *np*, tal como la comunidad de Python lo ha estandarizado. Las funciones se enlistan en orden de aparición del programa principal (código 4.1), el cual es básicamente el algoritmo

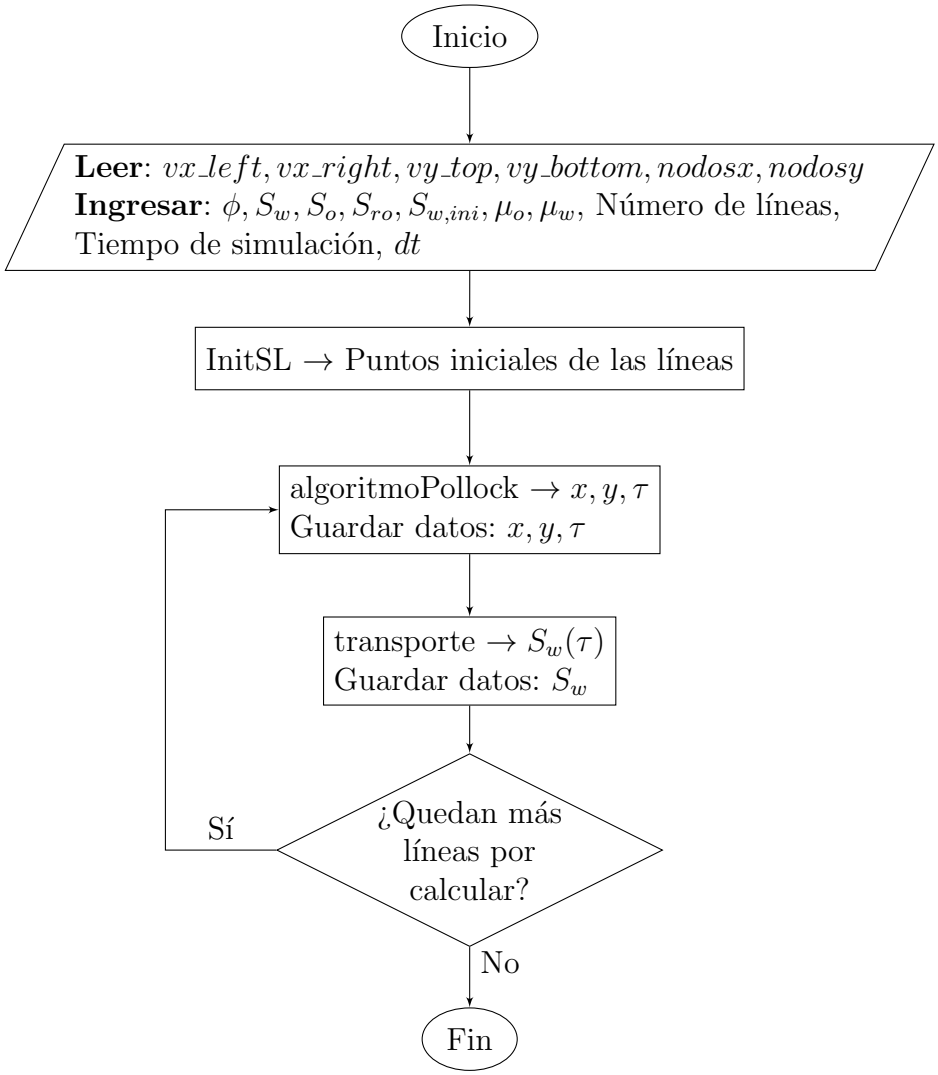


Figura 4.1: Resumen del programa principal

de Pollock.

Código 4.1: Programa principal del simulador en serie

```
1  (nx_cell, ny_cell, U_xl, U_xr, U_yt, U_yb) = A.LecturaVelocidades()
2
3  # Aquí guardaremos las trayectorias, saturaciones y DeltaTau de
       todas las streamlines
4  X_coord     = [] # Coordenadas x
5  Y_coord     = [] # Coordenadas y
6  Sw_SL       = [] # Saturación
7  Tau_SL      = [] # Tiempo de vuelo
8
9  # Cálculo de puntos iniciales de las streamlines
10 Xg,Yg = A.InitSL(d.DX, d.DY, d.NSL)
11
12 for k in range (d.NSL): # Para todas las  SL
13     # INICIA EL ALGORITMO DE POLLOCK
14     if (k != 0):
15         del csp, cea
16     s = 0
17     i = d.xcell_inj; j = d.ycell_inj # índices de la celda de
```

```
        partida
18      X = np.array([]); Y = np.array([]); DTAU = np.array([0])
19      aux1 = Xg[k]; aux2 = Yg[k] # Punto inicial de la línea
20      X = np.append(X,aux1); Y = np.append(Y,aux2) # Se inicia la
            trayectoria
21
22      while (i*j != (nx_cell-1)*(nx_cell-2) ): # mientras no se
            llegue al nodo productor
23          if s==0: # Condicional para asignar cara de entrada, al
                inicio del algoritmo para una SL
24              if   X[-1] == d.DX:
25                  csp=2
26              elif Y[-1] == d.DY:
27                  csp=4
28          cea,i,j =A.cara(csp,i,j)
29          aux1,aux2,aux3,csp = A.algoritmoPollock(X[-1], Y[-1], U_xl[
                i][j], U_xr[i][j], U_yb[i][j], U_yt[i][j], cea, i, j, d.
                DX, d.DY, d.phi)
30          X = np.append(X,aux1) # x_new
31          Y = np.append(Y,aux2) # y_new
32          DTAU = np.append(DTAU,aux3) # Dtau*phi
33          s = s+1
34
35      X=np.array(X) ; X_coord.append(X)
36      Y=np.array(Y) ; Y_coord.append(Y)
37      Tau_SL.append(DTAU)
38
39      # INICIAN CÁLCULOS DE TRANSPORTE
40      DTAU = np.array(DTAU)
41      fw = np.zeros(len(DTAU))
42      Sw = np.ones (len(DTAU))*d.Sw_ini
43      A.transporte(Sw, fw, d.Srw, d.Sor, d.Muo, d.Muw, DTAU, d.dt, d.
            TiempoTotal)
44      Sw_SL.append(Sw)
```

Aunque por definición todas las líneas de corriente parten de la misma fuente, para poder obtener diferentes trayectorias cada línea debe partir de un punto diferente. Esto se logra haciendo que las líneas partan de los bordes del nodo o nodos fuente de la malla de diferencias finitas. Estos bordes corresponden a un área total de flujo, el cual será dividida igualmente entre el número de líneas a trazar y en el centro de cada área, se ubicará el punto de partida de la línea en cuestión; de esta manera, conceptualmente cada línea acarreará el mismo flujo a través de sí. La función **InitSL** (código 4.2) se encarga de calcular con esta lógica todos los puntos iniciales para un pozo ubicado en la esquina inferior izquierda.

Código 4.2: *Init*: Función que calcula los puntos de partida de las líneas

```
1   def InitSL(dx,dy,n):
2   """
3       Función que nos da los puntos iniciales de las lineas de
            corriente, dando como input:
4       - n: el numero de líneas deseado (debe ser par) y,
5       - dx,dy: las dimensiones de la celda
6
7       Pensada para un modelo five-spot, donde de la celda del pozo
            inyector, que es el punto de partida, solo dos de sus caras
            llevaran líneas. Se asume una celda cuadrada.
```

```
8  """
9      Xg = np.zeros(n)
10     Yg = np.zeros(n)
11
12     # Para los puntos de partida de las SL
13     aux1 = (n/2)
14     # Para dividir una cara en (aux1) partes iguales
15     aux2 = (dx/aux1)
16     aux3 = aux2/2
17
18     for i in range (n):
19         if i < aux1: # para la primera mitad
20             Xg[i] = dx
21             Yg[i] = aux2*(i)+aux3
22         else:         # para la segunda mitad
23             Xg[i] = aux2*(i)+aux3-dx
24             Yg[i] = dy
25
26     return (Xg,Yg)
```

Una vez que se tienen los puntos de partida de las líneas, se procede a trazar su trayectoria con la función **algoritmoPollock** (código 4.6, figura 2.4), que ocupa a las funciones **velocidad** (código 4.4) y **pseudotiempo** (código 4.3). El rastreo de la partícula requiere de un especial cuidado al ser programado, ya que no hay manera de predecir por qué celdas irá atravesando. Se optó por crear la función **cara** (código 4.5) para determinar por qué cara la partícula entra en cada celda que cruce, tomando como referencia a la cara de salida de la celda anterior (csp); este sistema de orientación nos permite ir variando los índices i, j según corresponda. Este proceso se repite hasta que se llegue a una cara del nodo productor o bien a un punto de estancamiento.

Código 4.3: *pseudotiempo*: Función que calcula el pseudotiempo

```
1  def pseudotiempo(c,Dtau):
2      # Función que se ocupa para el cálculo de posición de la partí
           cula, dentro del algoritmo de Pollock
3      if c==0:
4          y=Dtau
5      else:
6          y=(np.exp(c*Dtau)-1)/c
7      return (y)
```

Código 4.4: *velocidad*: Función que calcula la velocidad

```
1  def velocidad(vp1,cp,p1,p):
2      # vp1=velocidad a la izquierda o al fondo de la celda
3      # cp=coeficiente
4      # p1=coordenada a la izquierda o al fondo de la celda
5      # p=punto de partida, (x/y)
6      v=vp1+cp*(p-p1)
7      return (v)
```

Código 4.5: *cara*: Función que determina la cara de entrada de la partícula

```
1  def cara(csp,i,j):
2      """
```

```
3       Función que devuelve la cara entrada actual (cea) de la partí
            cula, también se encarga de modificar los índices de celda (
            i,j) para el siguiente cálculo
4
5       1 = cara oeste
6       2 = cara este
7       3 = cara sur
8       4 = cara norte
9
10      la cara de salida pasada (csp) es la cara de entrada actual (
            cea)
11      """
12      if   csp==1: #Sale al oeste
13          cea=2
14          i=i-1
15      elif csp==2: #Sale al este
16          cea=1
17          i=i+1
18      elif csp==3: #Sale al sur
19          cea=4
20          j=j-1
21      elif csp==4: #Sale al norte
22          cea=3
23          j=j+1
24      return (cea,i,j)
```

Código 4.6: *algoritmoPollock*: Función donde se implementa el Algoritmo de Pollock

```
1   def algoritmoPollock(x0,y0,ux1,ux2,uy1,uy2,cea,i,j,DX,DY,phi):
2       """
3       cea (cara de entrada actual), csp (cara de salida pasada)
4       1 = cara oeste
5       2 = cara este
6       3 = cara sur
7       4 = cara norte
8
9       Recordando la definición de los coeficientes
10      cx = (ux2-ux1)/DX
11      cy = (uy2-uy1)/DY
12      """
13      # Si la velocidad en x,y es constante
14      if cx == 0 and cy == 0:
15          if cea == 1:   # Cara oeste
16              ux0 = ux1
17              uy0 = uy1
18          elif cea == 2: # Cara este
19              ux0 = ux2
20              uy0 = uy1
21          elif cea == 3: # Cara sur
22              uy0 = uy1
23              ux0 = ux1
24          elif cea == 4: # Cara norte
25              uy0 = uy2
26              ux0 = ux1
27
28          Dtau_x1 = (DX*i    -x0)/ux0
29          Dtau_x2 = (DX*(i+1)-x0)/ux0
30          Dtau_y1 = (DY*j    -y0)/uy0
```

```
31          Dtau_y2 = (DY*(j+1)-y0)/uy0
32
33      # Si cx,cy son diferentes de 0
34      else:
35          if cea == 1:   # Cara oeste
36              ux0 = ux1
37              uy0 = velocidad(uy1, cy, j*DY, y0)
38          elif cea == 2: # Cara este
39              ux0 = ux2
40              uy0 = velocidad(uy1, cy, j*DY, y0)
41          elif cea == 3: # Cara sur
42              uy0 = uy1
43              ux0 = velocidad(ux1, cx, i*DX, x0)
44          elif cea == 4: # Cara norte
45              uy0 = uy2
46              ux0 = velocidad(ux1, cx, i*DX, x0)
47
48          # Cálculo de tiempo de vuelo a cada cara para velocidades
               en x
49          if ux0 == 0: # Para evitar la división entre cero
50              Dtau_x1 = np.NaN
51              Dtau_x2 = np.NaN
52          else:
53              if ux1 == 0: # Para evitar el log(cero)
54                  Dtau_x1 = np.NaN
55              else:
56                  Dtau_x1 = (1/cx)*np.log(ux1/ux0)
57
58              if ux2 == 0: # Para evitar el log(cero)
59                  Dtau_x2 = np.NaN
60              else:
61                  Dtau_x2 = (1/cx)*np.log(ux2/ux0)
62
63          # Cálculo de tiempo de vuelo a cada cara para velocidades
               en y
64          if uy0 == 0: # Para evitar la división entre cero
65              Dtau_y1 = np.NaN
66              Dtau_y2 = np.NaN
67          else:
68              if uy1 == 0: # Para evitar el log(cero)
69                  Dtau_y1 = np.NaN
70              else:
71                  Dtau_y1=(1/cy)*np.log(uy1/uy0)
72
73              if uy2 == 0: # Para evitar el log(cero)
74                  Dtau_y2 = np.NaN
75              else:
76                  Dtau_y2 = (1/cy)*np.log(uy2/uy0)
77
78      # Aquí se sale de los condicionales de cx,cy
79      Dtau_array=np.array([Dtau_x1,Dtau_x2,Dtau_y1,Dtau_y2])
80
81      for i in range (len(Dtau_array)):
82          if Dtau_array[i]<=1:
83              Dtau_array[i] = np.NaN
84
85      Dtau=np.nanmin(Dtau_array)
86
```

```
87      # Para devolver la cara de salida
88      if Dtau == Dtau_array[0]:
89          csp = 1
90      elif Dtau == Dtau_array[1]:
91          csp = 2
92      elif Dtau == Dtau_array[2]:
93          csp = 3
94      elif Dtau == Dtau_array[3]:
95          csp = 4
96
97      x_new = x0+ux0*pseudotiempo(cx,Dtau)
98      y_new = y0+uy0*pseudotiempo(cy,Dtau)
99
100     return (x_new, y_new, Dtau*phi, csp)
```

Ahora que se calculó la trayectoria desde el pozo inyector al productor y simultáneamente se obtuvo la discretización de τ, es posible realizar los cálculos de transporte. En primer lugar las permeabilidades relativas de las fases son calculadas con la función **kr** (código 4.7), y posteriormente el flujo fraccional de agua con **fractionalflow** (código 4.8).

Dentro de la función **transporte** (código 4.9) hay un ciclo *while* que repite los cálculos hasta que se cumpla el tiempo total de simulación. Dentro de éste, cada una de las saturaciones se calcula mediante un ciclo *for*. Una vez finalizados los cálculos para la línea, se guardan los datos de su trayectoria, saturación y τ, y se repite el algoritmo para las siguientes líneas.

Código 4.7: *kr*: Función que calcula la permeabilidad relativa

```
1  def kr(Sw, Srw, Sro):
2      # Función que calcula las permeabilidades relativas
3      omega = 2 # Exponente
4      Scf = (Sw-Srw)/(1-Srw-Sro) # saturación efectiva
5      krw = Scf**omega
6      kro = (1-Scf)**omega
7
8      return(krw, kro)
```

Código 4.8: *fractionalflow*: Función que calcula el flujo fraccional

```
1  def fractionalflow(kro, krw, Muo, Muw):
2      # Función que calcula el flujo fraccional de agua
3
4      if krw == 0 :
5          fw = 0
6      else:
7          fw = 1/(1+(kro*Muw)/(krw*Muo))
8
9      return(fw)
```

Código 4.9: *transporte*: Función que hace los cálculos de transporte

```
1  def transporte(Sw, fw, Srw, Sro, Muo, Muw, DTAU, dt, TiempoTotal):
2      """
3      Función que calcula la saturación explícitamente.
4      Sw, DTAU, fw son arreglos. Las demás variables son escalares.
```

```
5       """
6       t=0
7       while t <= TiempoTotal:
8           # A lo largo de la streamline.
9           for i in range (len(DTAU)):
10              if i==0: # Condición de inyección
11                  Sw[i] = (1-Sro)
12                  # kro, krw se actualizan en cada ciclo
13                  krw, kro = kr(Sw[i], Srw, Sro)
14                  fw[i] = fractionalflow(kro, krw, Muo, Muw)
15
16              else: # Cálculo IMPES
17                  if fw[i] == 0:
18                      break
19                  # kro, krw se actualizan en cada ciclo
20                  krw, kro = kr(Sw[i], Srw, Sro)
21                  fw[i] = fractionalflow(kro, krw, Muo, Muw)
22
23                  aux = (dt/DTAU[i])*(fw[i]-fw[i-1])
24                  Sw[i] = Sw[i]-aux
25          t = t+dt
```

4.2. Validación del simulador

Con el fin de validar al simulador de líneas de corriente, se compararon cualitativamente los perfiles de saturación que presenta contra los calculados por un simulador de diferencias finitas. Los datos de simulación se enlistan en la tabla 4.1.

Propiedad	Valor
Líneas	40
μ_o	1.14 [cp]
μ_w	0.096 [cp]
ϕ	20 %
L_x	1,000 [ft]
L_y	1,000 [ft]
k_{xx}	100 [mD]
k_{yy}	100 [mD]
S_{ro}	20 %
S_{rw}	22 %
$S_{w,ini}$	22 %
Tiempo Total	100 [d]

Tabla 4.1: Datos de simulación para líneas de corriente numéricas.

Se hicieron diferentes corridas de simulación variando el tamaño de malla, ubicación de pozos y permeabilidad, esta última aleatoriamente. Al variar las permeabilidades se aprovecha al máximo la simulación de líneas de corriente, pues se observan los canales preferentes de flujo y el avance del frente de inyección con gran claridad y detalle.

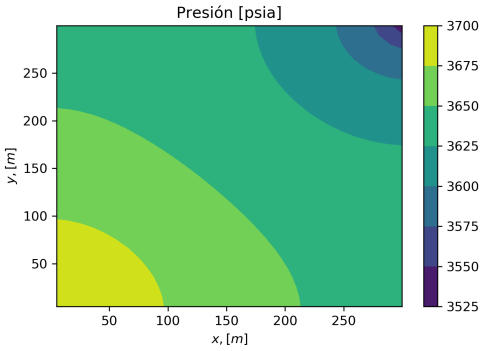
A continuación se encuentran los resultados calculados por los simuladores, así como la discusión de sus resultados.

4.2.1. Medio homogéneo

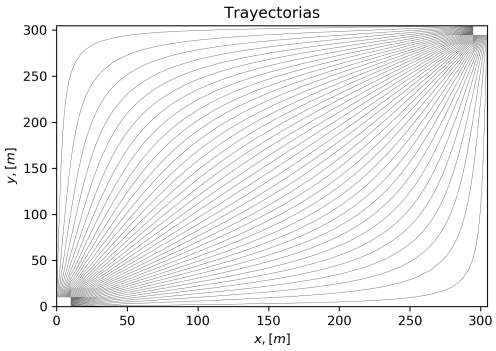
En la figura 4.2 se tienen imágenes que nos permiten observar cómo se relacionan la simulación de diferencias finitas con la de líneas de corriente. Partimos con la figura 4.2a donde tenemos al perfil de presión del yacimiento y sus líneas de isopotencial; por definición, las líneas de corriente son ortogonales a éstas y se comprueba con la figura 4.2b.

Posteriormente, si se agrega la saturación al gráfico de líneas de corriente, se obtiene la figura 4.2c. Cada uno de los puntos que se observan corresponden a la discretización natural de τ proporcionada por el Algoritmo de Pollock. Se observa que las figuras 4.2c y 4.2d presentan cualitativamente los mismos resultados, por lo que se concluye que el simulador entrega resultados satisfactorios.

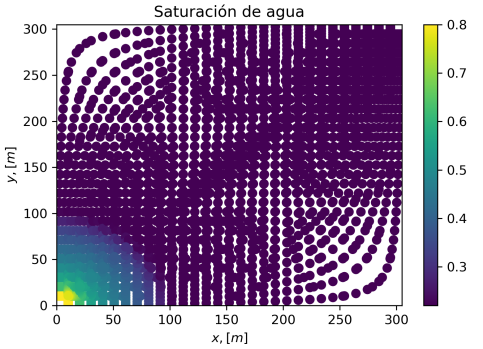
Al aumentar el tamaño de malla a 60x60 y 200x200, se obtienen los resultados de la figura 4.3. Es conveniente aclarar que para todos los casos se fijó el número de líneas a cuarenta. Se observa que a medida que refinamos la malla, el frente de inyección retrocede y los puntos comienzan a desaparecer para dar la impresión de que se tiene una línea continua, aunque no es el caso.



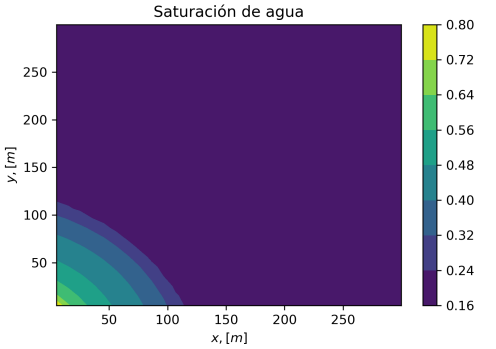
(a) Perfil de presión



(b) Líneas de corriente

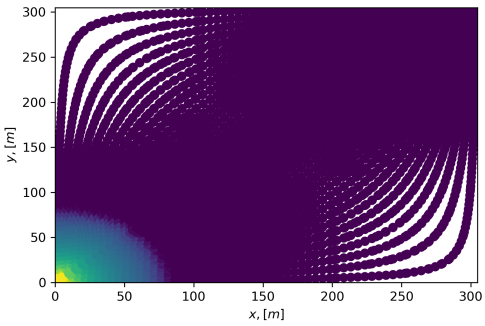


(c) Perfil de saturación

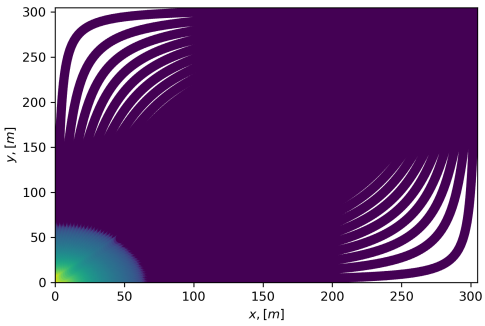


(d) Validación con simulador IMPES

Figura 4.2: Resultados en una malla 30x30



(a) 60x60



(b) 200x200

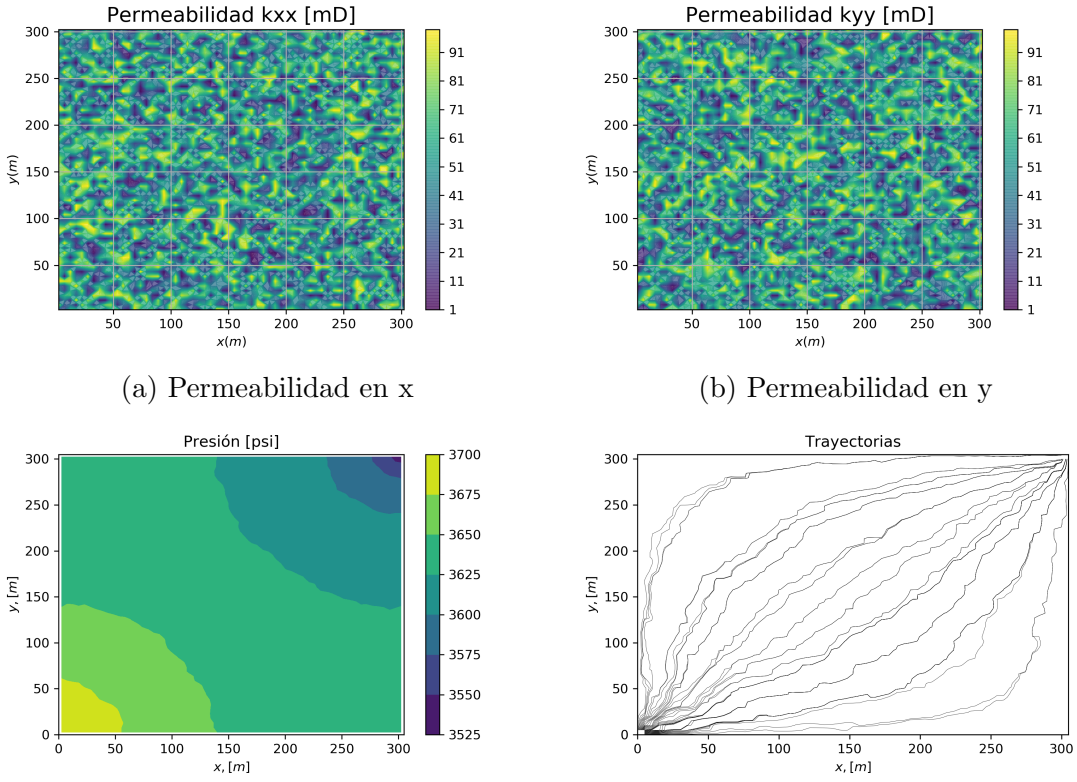
Figura 4.3: Resultados en diferentes tamaños de malla

4.2.2. Medio heterogéneo

Para este ejemplo se siguió utilizando la misma configuración de pozos, pero se modificó el valor de las permeabilidades aleatoriamente de 1-100 [mD] (figuras 4.4a y 4.4b); haciendo esto, el modelo se acerca más a la realidad de cualquier yacimiento. Como se ha dicho anteriormente, las líneas de corriente son extremadamente útiles en medios heterogéneos, y es ahora donde se demuestra.

En la figura 4.5, análogo a la figura 4.2, se encuentran los resultados en una malla de 60x60 con permeabilidades variables. Observando sólo los mapas de permeabilidad (figuras 4.4a y 4.4b) es complicado, por no decir imposible, predecir los canales preferentes de flujo. Sin embargo, con líneas de corriente es sencillo identificar zonas de alta permeabilidad (aquellas zonas en donde las líneas casi se empalman), así como también zonas en donde podría quedar aceite sin desplazar.

Adicionalmente, con el fin de hacer una comparación, se cuenta con los perfiles de saturación reportados por ambos simuladores. Una vez más se observa que entregan resultados similares; sin embargo, vemos que el simulador de diferencias finitas (figura 4.5b) pierde resolución en el avance del frente de inyección cerca de la frontera sur del dominio, mientras que las líneas de corriente (figura 4.5a) proporcionan gran detalle en esta área y en todas, aún en zonas de alta velocidad.



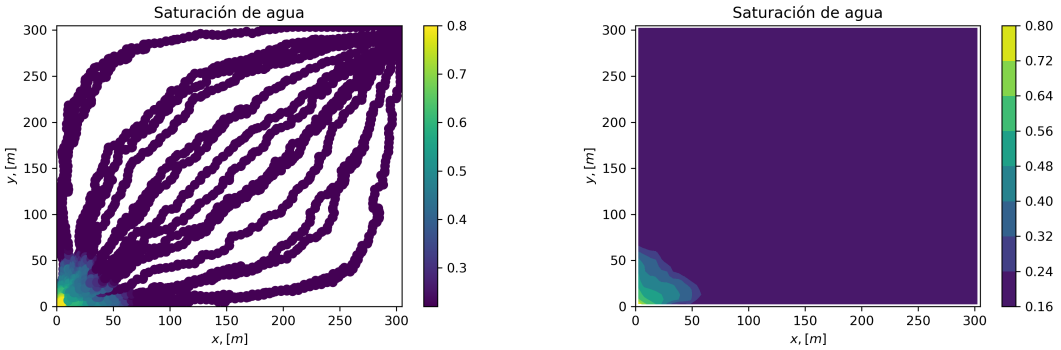
(a) Permeabilidad en x

(b) Permeabilidad en y

(c) Perfil de presión

(d) Líneas de corriente

Figura 4.4: Heterogeneidad, ejemplo 1: Resultados en una malla 60x60



(a) Perfil de saturación (b) Validación con simulador IMPES

Figura 4.5: Heterogeneidad, ejemplo 1: Resultados en una malla 60x60

4.2.3. Alterando la configuración de pozos

Con el fin de observar diversos patrones de flujo, para estos ejemplos se modificó la ubicación de los pozos tal como lo muestran las figuras 4.6 y 4.7. Al igual que en el ejemplo anterior, el medio es heterogéneo y se puede comprobar observando que las líneas de isopotencial (figura 4.6a) y las líneas de corriente (figura 4.6b) no son simétricas.

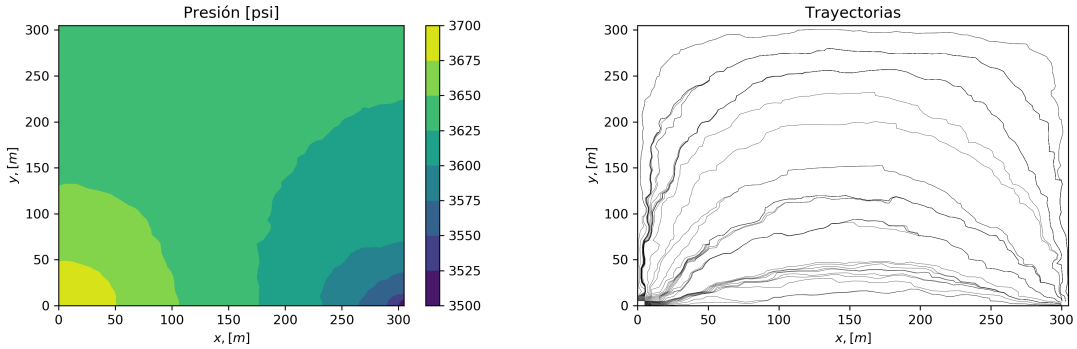
Una vez más observamos cómo las líneas de corriente no siguen trayectorias simples, algunas tardan más que otras (es decir, tienen mayor τ), pero finalmente todas convergen al pozo productor.

4.2.4. Agregando pozos al dominio de flujo

Con el objetivo de visualizar los resultados del simulador en escenarios más complejos, en esta sección se crearon tres nuevas configuraciones de pozos, en la que cada una va adicionando un pozo productor respectivamente.

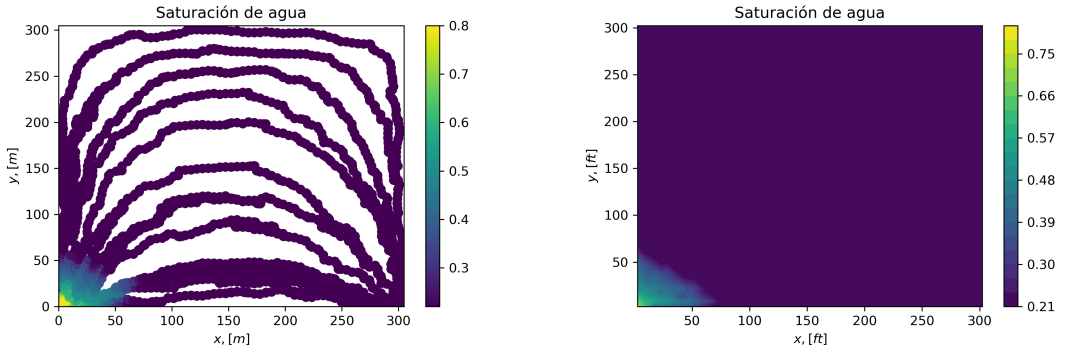
Cada una de las configuraciones fue simulada tanto para un medio homogéneo (figuras 4.8, 4.10, 4.12) como para un medio heterogéneo (figuras 4.9, 4.11, 4.13), variando la permeabilidad aleatoriamente como en los ejemplos anteriores.

El simulador no requirió de modificaciones para poder simular estos dominios de flujo, el algoritmo de Pollock automáticamente se detiene una vez que una línea converge a un pozo. Se aprecia que las líneas siguen trayectorias cada vez más complejas cuando se añade la heterogeneidad del medio; sin embargo, esto no es ninguna limitante o contratiempo, pues sigue dando resultados bastante buenos.



(a) Perfil de presión

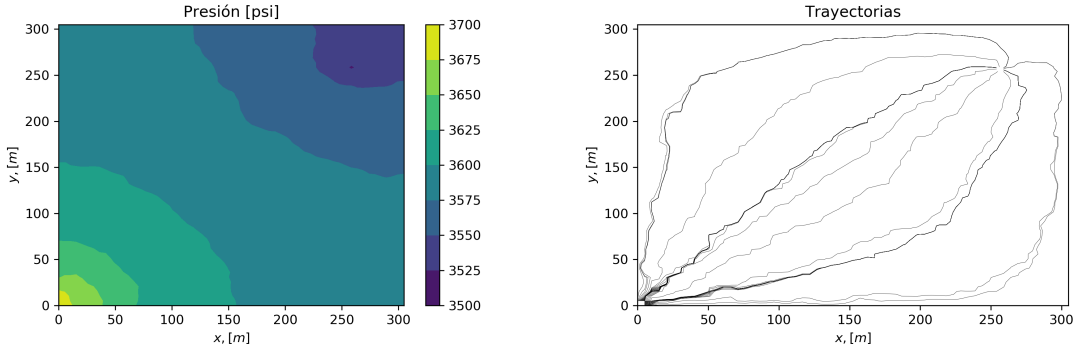
(b) Líneas de corriente



(c) Perfil de saturación

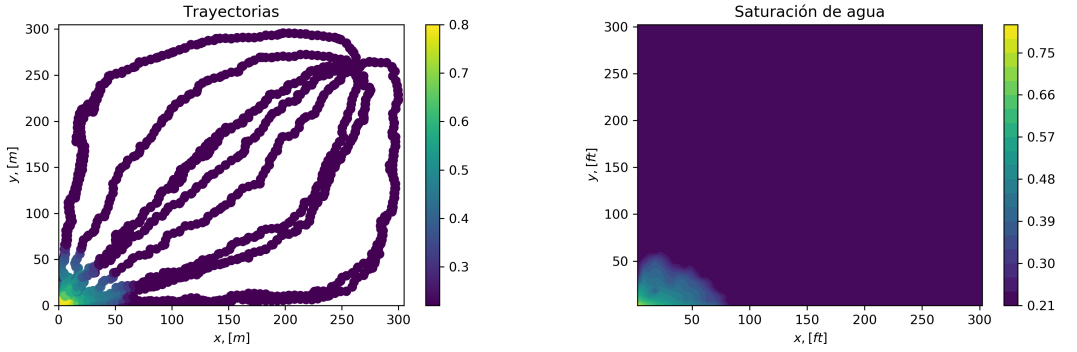
(d) Perfil de saturación

Figura 4.6: Heterogeneidad, ejemplo 2: Resultados en una malla 60x60



(a) Perfil de presión

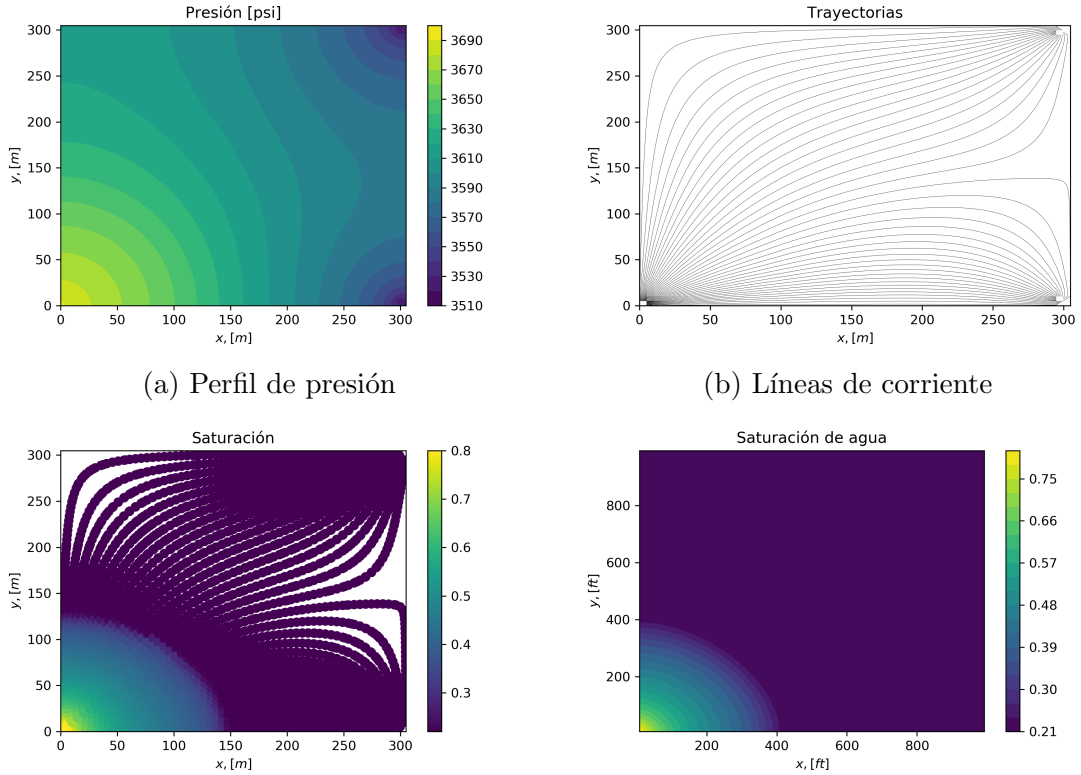
(b) Líneas de corriente



(c) Perfil de saturación

(d) Perfil de presión

Figura 4.7: Heterogeneidad, ejemplo 3: Resultados en una malla 60x60



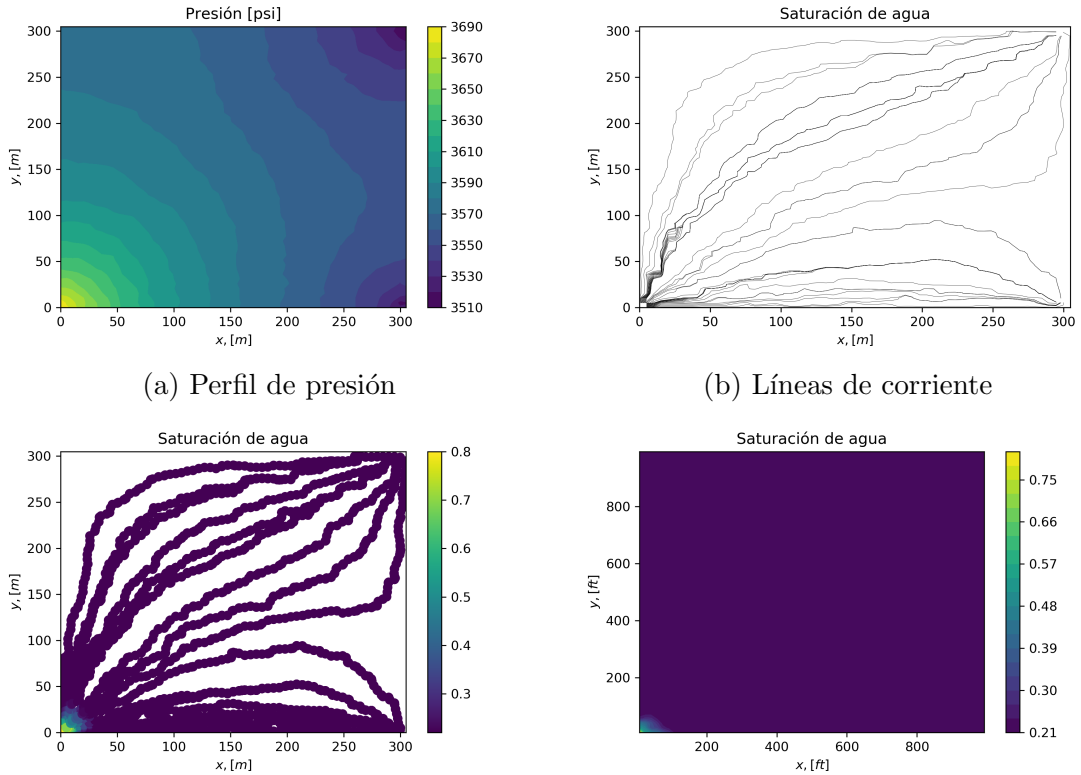
(a) Perfil de presión

(b) Líneas de corriente

(c) Perfil de saturación

(d) Validación con simulador IMPES

Figura 4.8: Configuraciones de pozos 1: medio homogéneo (malla 60x60)



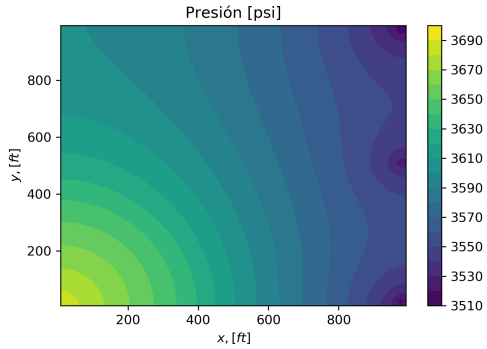
(a) Perfil de presión

(b) Líneas de corriente

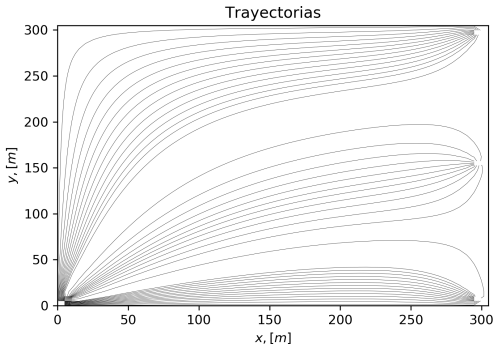
(c) Perfil de saturación

(d) Validación con simulador IMPES

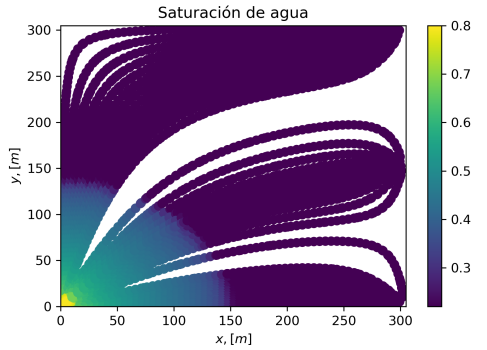
Figura 4.9: Configuraciones de pozos 1: medio heterogéneo (malla 60x60)



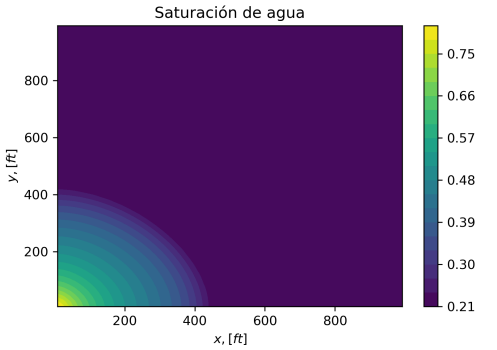
(a) Perfil de presión



(b) Líneas de corriente

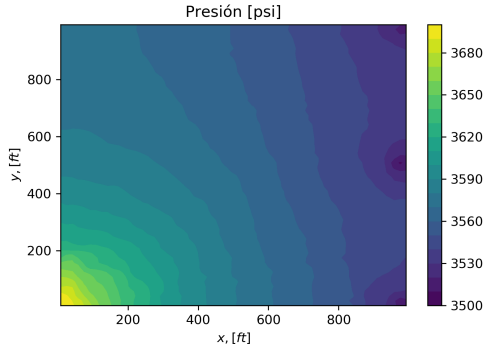


(c) Perfil de saturación

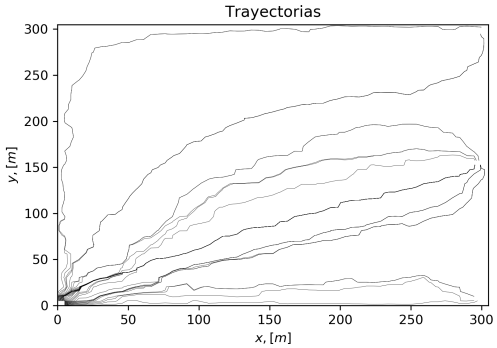


(d) Validación con simulador IMPES

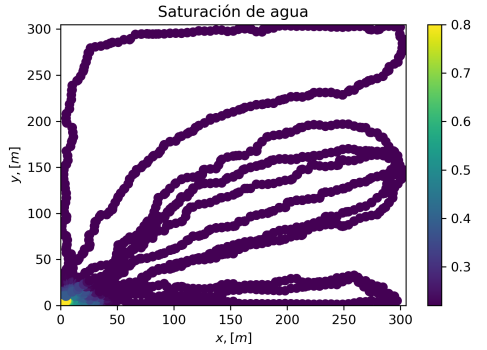
Figura 4.10: Configuraciones de pozos 2: medio homogéneo (malla 60x60)



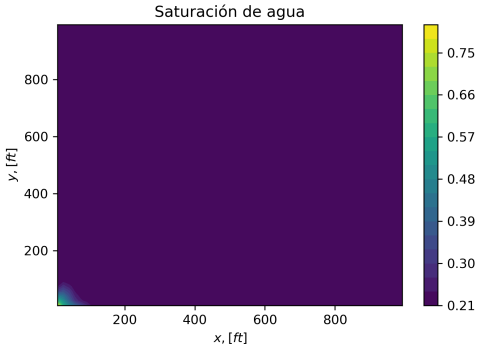
(a) Perfil de presión



(b) Líneas de corriente

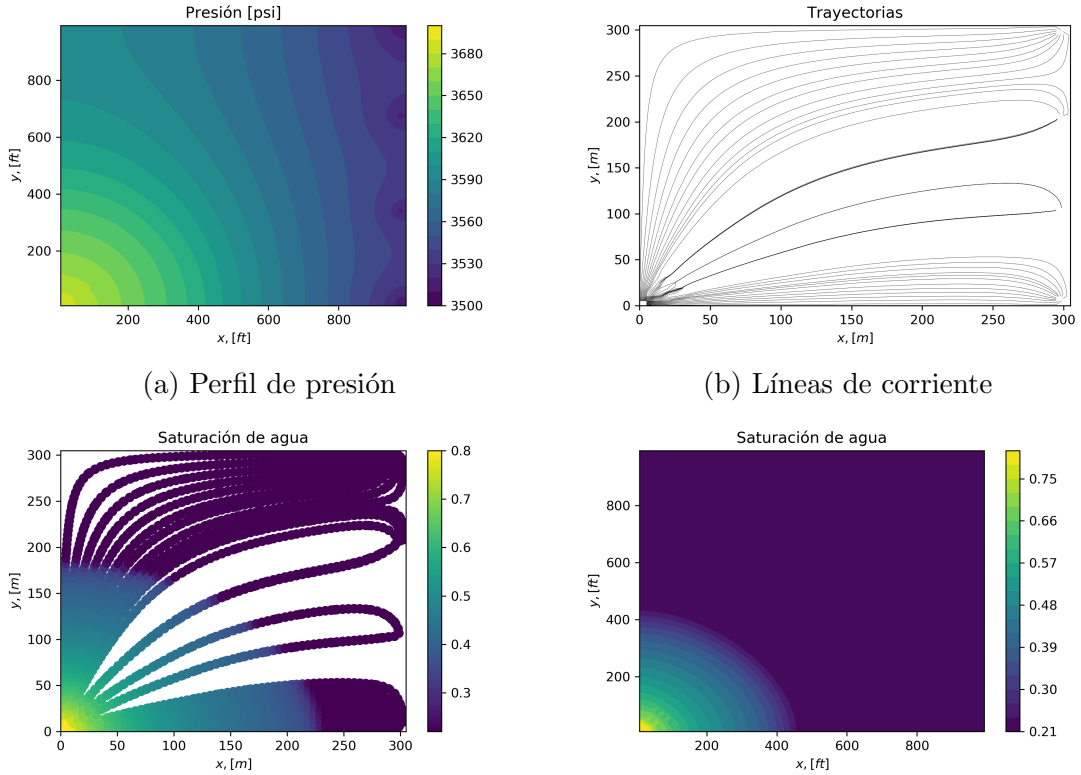


(c) Perfil de saturación



(d) Validación con simulador IMPES

Figura 4.11: Configuraciones de pozos 2: medio heterogéneo (malla 60x60)



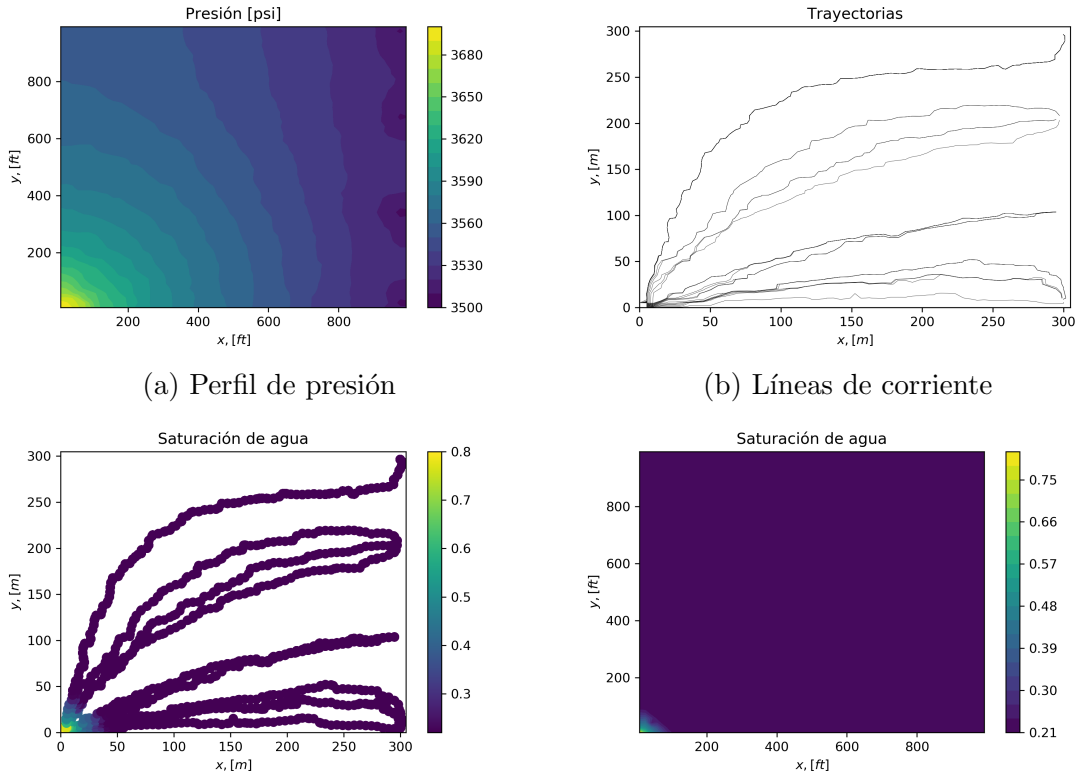
(a) Perfil de presión

(b) Líneas de corriente

(c) Perfil de saturación

(d) Validación con simulador IMPES

Figura 4.12: Configuraciones de pozos 3: medio homogéneo (malla 60x60)



(a) Perfil de presión

(b) Líneas de corriente

(c) Perfil de saturación

(d) Validación con simulador IMPES

Figura 4.13: Configuraciones de pozos 3: medio heterogéneo (malla 60x60)

Capítulo 5

Implementación computacional del algoritmo en paralelo y análisis de aceleración

Con el objetivo de optimizar el código, la primera aproximación fue agregar el decorador @njit a todas las funciones del simulador (en adelante llamada Configuración A). Aunque los resultados de aceleración con esta ligera modificación fueron drásticos (tabla 5.1), todavía es posible sacar más provecho a Numba haciendo uso de sus diferentes formas de compilación.

Como segunda aproximación, se modificaron aquellas partes del código que no favorecen a la paralelización, como el uso de *append* y el ciclo *while*, que hacían que los arreglos de coordenadas y τ de las líneas tuvieran la longitud necesaria para que se terminen de trazar; de esta manera, los arreglos de cada línea de corriente podían tener diferente longitud entre ellas. Ahora, los arreglos de todas las líneas tienen una longitud fija y sus valores se almacenan en una matriz.

También se aprovecharon los ciclos que contenía el archivo principal para crear una nueva función auxiliar llamada *principal* (código 5.1). El nuevo archivo principal se muestra en el código 5.2.

De igual manera, tomando en cuenta que cuando Numba optimiza una función, la primera vez que la ejecuta es muy lenta con respecto a las siguientes, se le quitó @njit a las funciones que se ejecutan solamente una vez y que no contienen funciones dentro de sí. Aquellas que conservaron el decorador se encuentran en la figura 5.1.

Código 5.1: *principal*: Función creada a partir del programa principal del simulador en serie.

```
1 def principal(Xg, Yg, X_coord, Y_coord, Tau_SL, Sw_SL, limit, U_xl,
      U_xr, U_yb, U_yt, NSL, xcell_inj, ycell_inj, DX, DY, phi,
     Sw_ini, Srw, Sor, Muo, Muw, dt, TiempoTotal):
2     for k in range (NSL): # Para todas las  SL
3
4         s = 0 #Contador para los loops del ciclo para cada SL
5         i = xcell_inj; j = ycell_inj # es la celda de partida de
              cada SL (0,0)
6
```

```
7           X_coord[k][0] = Xg[k] # Se inicia la trayectoria de la
               streamline
8           Y_coord[k][0] = Yg[k] # Se inicia la trayectoria de la
               streamline
9
10          for l in range (1,limit-2):
11              if s==0: # Condicional para asignar cara de entrada, al
                    inicio del algoritmo para una SL
12                  if   X_coord[k][0] == DX:
13                      csp=2
14                  elif Y_coord[k][0] == DY:
15                      csp=4
16
17              cea,i,j =cara(csp,i,j) #aquí se obtiene la cara de
                   entrada actual (cea) de la partícula
18              aux1,aux2,aux3,csp = algoritmoPollock(X_coord[k][l-1],
                   Y_coord[k][l-1],     U_xl[i][j], U_xr[i][j], U_yb[i
                   ][j], U_yt[i][j], cea, i,j, DX, DY, phi)
19
20              X_coord[k][l] = aux1 # x_new
21              Y_coord[k][l] = aux2 # y_new
22              Tau_SL [k][l] = aux3 # Dtau*phi
23
24              s=s+1
25              if s>500: #Máximo de loops por SL
26                  break
27
28          # Aquí se hacen los cálculos de transporte por streamline
29          DTAU = Tau_SL[k][:]
30          fw = np.zeros(len(DTAU))
31          Sw = np.ones (len(DTAU))*Sw_ini
32          transporte(Sw, fw, Srw, Sor, Muo, Muw, DTAU, dt,
               TiempoTotal)
33          Sw_SL[k][:]=Sw
```

Código 5.2: Programa principal del simulador en paralelo.

```
1  (nx_cell, ny_cell, U_xl, U_xr, U_yt, U_yb) = A.LecturaVelocidades()
2
3  limit = 400 # maxima longitud de una línea de corriente
4
5  X_coord     = np.zeros([d.NSL,limit]) # Coordenadas x de las líneas
6  Y_coord     = np.zeros([d.NSL,limit]) # Coordenadas y de las líneas
7  Sw_SL       = np.zeros([d.NSL,limit]) # Saturación    de las líneas
8  Tau_SL      = np.zeros([d.NSL,limit]) # DeltaTau      de las líneas
9
10 # Cálculo de puntos iniciales de las líneas
11 Xg,Yg = A.InitSL(d.DX, d.DY, d.NSL)
12
13
14 A.principal(Xg, Yg, X_coord, Y_coord, Tau_SL, Sw_SL, limit, U_xl,
      U_xr, U_yb, U_yt, d.NSL, d.xcell_inj, d.ycell_inj, d.DX, d.DY, d
      .phi, d.Sw_ini, d.Srw, d.Sor, d.Muo, d.Muw, d.dt, d.TiempoTotal)
```

Debido a que el objetivo es paralelizar, en primer lugar se agregó la opción *parallel=True* a las funciones que permitían el uso de ésta (aquellas que contienen algún ciclo *for*) y se midió su tiempo de cómputo. Sin embargo, se observó que el

paralelizado automático no mejoró el rendimiento del código para ninguna malla. Este resultado se obtuvo debido a que se utilizaron mallas relativamente gruesas, además de que la velocidad de un solo hilo de procesamiento es superior a utilizar varios hilos para luego unir la solución.

Posteriormente se probaron más opciones de compilación como nogil, cache y parallel, hasta que finalmente se encontró la mejor. Las siete configuraciones comparadas se ilustran en la figura 5.1.

Función	@njit							nogil							cache					parallel				
pseudotiempo	A	B	C	D	E	F	G	B				F	G				E	F	G					
velocidad	A	B	C	D	E	F	G	B	C	D	E	F	G					F	G					
cara	A	B	C	D	E	F	G	B					G						G					
algoritmoPollock	A	B	C	D	E	F	G	B					G	C	D	E	F	G		C	D	E	F	
kr	A	B	C	D	E	F	G	B					G	C	D	E	F	G						
fractionalflow	A	B	C	D	E	F	G	B					G	C	D	E	F	G						
transporte	A	B	C	D	E	F	G	B					G	C	D	E	F	G						
principal		B	C	D	E	F	G	B		D	E	F	G	C				G						

Figura 5.1: Combinaciones de opciones de compilación utilizadas.

5.1. Análisis de aceleración

En esta sección se comparan los tiempos de ejecución del simulador utilizando como caso base a la implementación que usa solamente la biblioteca *Numpy*, altamente optimizada por sus desarrolladores, contra aquellas implementaciones que hacen uso de @njit y sus diversas formas de compilación (ilustradas en la figura 5.1).

@njit

Se tienen dos tablas de resultados para comparar la simple adición del decorador @njit (tablas 5.1 y 5.2), donde la primera contiene a los ejemplos realizados para un medio homogéneo y la segunda, aquellos de medio heterogéneo. Para todos los ejemplos se simularon 40 líneas de corriente con su respectivo perfil de saturación a 100 días con el fin de poder comparar los resultados. Sin embargo, el paso del tiempo no pudo ser el mismo debido a las restricciones impuestas por el τ de cada simulación (se trabajó con un dt fijo).

En la tabla 5.1 se muestra cómo a medida que se refina la malla, el tiempo de ejecución con Numpy aumenta considerablemente hasta alcanzar casi una hora en la malla de 200x200. Posteriormente, se observa que el desempeño del simulador mejora drásticamente con el uso de @njit, obteniendo cocientes de aceleración de hasta 532 para el ejemplo más extremo.

Es conveniente mencionar que aunque los tiempos de cálculo sin Numba parecieran ser muy lentos, el simulador de diferencias finitas para la malla de 200x200 tardó media hora tan solo para simular un salto temporal. Esto es una muestra de cómo aún sin optimizar, la simulación de líneas de corriente es mucho más rápida que la

de diferencias finitas, puesto que en esta no se resuelve un sistema de ecuaciones lineales.

Tamaño de Malla	dt [d]	Numpy [s]	Numpy+@njit [s]	Aceleración
30x30	0.1	7.618	0.944	8.069x
60x60	0.1	15.194	1.007	15.088x
90x90	0.01	219.072	1.279	171.283x
200x200	0.001	3491.93	6.557	532.556x

Tabla 5.1: Comparativa de tiempos de ejecución para los ejemplos en medio homogéneo.

Ejemplo	dt [d]	Numpy [s]	Numpy+@njit [s]	Aceleración
1	0.1	16.56	1.021	16.219x
2	0.01	115.12	2.225	51.739x
3	0.1	21.13	1.516	33.727x

Tabla 5.2: Comparativa de tiempos de ejecución para los ejemplos en medio heterogéneo en mallas de 60x60.

En cuanto a los ejemplos de la tabla 5.2, todos fueron realizados en mallas de 60x60. Se observa que los resultados del primer ejemplo son muy parecidos a su contraparte de medio homogéneo (en la tabla 5.1), esto puede atribuirse a que las líneas tienen relativamente el mismo τ aunque las permeabilidades cambien, ya que tienen los pozos ubicados en el mismo lugar.

Con respecto a los últimos dos ejemplos, se tiene que los tiempos de ejecución en la implementación con Numpy simple varían seis veces entre sí. En primer lugar, esto se debe a que se usaron *dt* distintos y en segundo, a que el ejemplo 2 tiene mayor τ en todas sus líneas que los ejemplos anteriores (es decir, las líneas tienen trayectorias más largas), lo que a su vez implica más cálculos de transporte y mayor tiempo de cómputo.

Añadiendo opciones de compilación

Se amplió la tabla 5.1 con los resultados de desempeño de las diversas configuraciones utilizadas, obteniendo la figura 5.2. A esta se le aplicaron escalas de color que permiten observar rápidamente qué implementación arroja los mejores resultados.

Las configuraciones se fueron creando a medida que se probaba el efecto que tenían *cache* y *nogil* en el rendimiento global del simulador. Cada nueva configuración no necesariamente era mejor que la anterior; por ejemplo, algunas mejoraban el rendimiento en las mallas más finas, pero no en el resto o viceversa (configuraciones B y C). Sin embargo, se notaba que había cierta correlación.

Se revisó de nuevo la documentación de Numba, específicamente de las opciones de compilación (incluidas en el Capitulo 3), y fue así como se llega a la Configuración G. En resumen, *cache* permite ahorrar tiempo de compilación cada que se invoca una función y *nogil* libera el bloqueo de intérprete global (GIL) de Python, que ya no

		Configuración						
	Numpy	A	B	C	D	E	F	G
Malla: 30x30	7.618	0.944	1.5194	2.1039	2.7386	3.1471	2.9549	0.09904
Malla: 60x60	15.194	1.007	1.5198	2.1065	2.7403	2.9105	3.038	0.1028
Malla: 90x90	219.072	1.279	2.1092	1.3333	1.9563	2.3137	2.1706	0.6793
Malla: 200x200	3491.93	6.557	5.5113	4.814	5.3113	5.554	5.615	3.9859

Figura 5.2: Comparativa de rendimiento entre las diversas configuraciones utilizadas. Resultados en segundos.

es necesario una vez que las funciones están optimizadas. Sabiendo que el simulador tiene varios programas recurrentes, se añadieron ambas opciones de compilación a todas las funciones esperando una mejoría.

Es menester mencionar que aunque *parallel* no se usó en esta configuración, *nogil* aprovecha los múltiples núcleos del CPU, ya que el código ejecutado con el GIL liberado se ejecuta simultáneamente con otros subprocesos que ejecutan código Python o Numba (ya sea la misma función compilada u otra).

Con el fin de observar adecuadamente los resultados en la figura 5.3 fue necesario hacer uso de un eje logarítmico. Se observa que los resultados de las mallas forman, a grandes rasgos, líneas decrecientes. Los tiempos de ejecución de la implementación con Python simple, fueron mejorados varios órdenes de magnitud con la Configuración A. Posteriormente, este desempeño fue mejorado aún más profundizando tan solo un poco más en conceptos computacionales y en el propio Python. Numba automáticamente hizo el resto con tan solo añadir las opciones de compilación *nogil* y *cache* a las funciones del simulador.

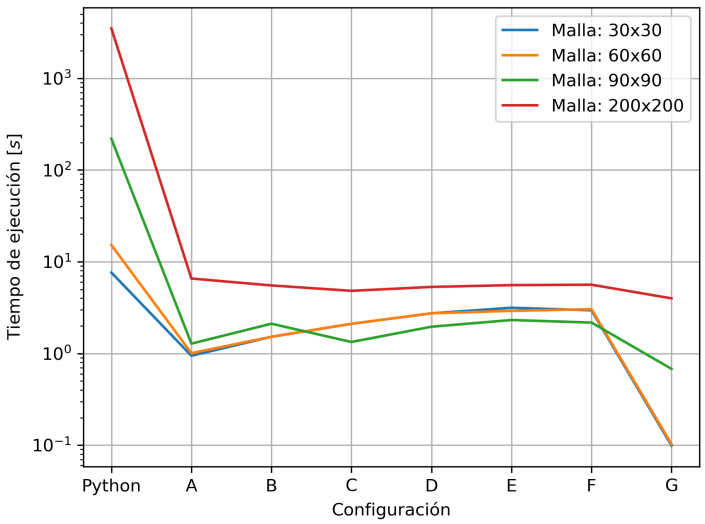


Figura 5.3: Comparativa de rendimiento entre las diversas configuraciones utilizadas.

Finalmente, cabe aclarar que estos resultados variarán según el equipo que ejecute al simulador. Para este trabajo se usó una computadora con un procesador Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz y una RAM de 8 GB.

Capítulo 6

Conclusiones

Se investigó el modelo matemático y numérico de líneas de corriente. Con base en estos, se desarrolló un código numérico tipo serie que resuelve el flujo bifásico en dos dimensiones. Se compararon los resultados del código con aquellos reportados por un simulador de diferencias finitas en diferentes escenarios, esto con el fin de validar el simulador.

- Las líneas de corriente demostraron ser especialmente útiles al dibujar con detalle el avance del frente de inyección aún en zonas de alta permeabilidad.
- Se modificó el código en serie partiendo del decorador @njit para optimizar utilizando las opciones de compilación que paralelizan ciertas funciones, donde la Configuración G obtuvo el mejor rendimiento con cocientes de aceleración que van desde 1.64x hasta 9.53x.
- Se desarrolló un simulador mediante cómputo paralelo de memoria compartida portable y sencillo, que calcula y muestra las líneas de corriente con su perfil de saturación para analizar la fenomenología de flujo de fluidos bifásico que ocurre en la simulación de yacimientos.

Trabajo futuro

1. Como trabajo futuro se propone modificar las funciones del simulador para que este pueda también ser utilizado en flujo compresible, yacimientos naturalmente fracturados, así como también en flujo tridimensional.
2. Se propone incorporar un dt adaptativo en lugar del fijo que se implementó en este trabajo. El uso de éste podría acelerar considerablemente los cálculos aún cuando no se involucren los decoradores de Numba en los códigos.
3. Desarrollar una interfaz gráfica que permita el uso del simulador bifásico y del simulador de líneas de corriente desarrollado en este trabajo de una manera fácil e intuitiva.
4. Finalmente, también se sugiere incluir a la GPU en la paralelización del código. En el presente se paralelizó únicamente a nivel CPU debido a las limitantes actuales de Numba. Sin embargo, actualmente en Python existen bibliotecas para explotar la potencia de cálculo de las GPU, por ejemplo *Cupy*.

Apéndice A

Conceptos

A.1. Diferencias Finitas

La Técnica de Diferencias Finitas se utiliza generalmente cuando se plantean problemas que no se basan en las leyes de conservación, sino en principios puramente matemáticos y las Ecuaciones Diferenciales Parciales se resuelven para encontrar puntos discretos. O bien, el sistema de ecuaciones diferenciales es bien comportado, hablando numérica y computacionalmente.

La metodología numérica de diferencias fintas se resume como sigue:

Cuando se usa una aproximación por diferencias finitas, el dominio del problema es **discretizado** tal que los valores de la variable dependiente desconocida se consideran solo en un número finito de puntos en lugar de cada punto sobre la región. Si n nodos son seleccionados, n ecuaciones algebraicas son desarrolladas discretizando las ecuaciones gobernantes y sus condiciones de frontera, esto es, el problema de resolver las ecuaciones diferenciales ordinarias o parciales sobre su dominio, se convierte en la tarea de desarrollar un grupo de ecuaciones algebraicas y resolverlas en los puntos discretos por un algoritmo recomendable.

Diferencias Finitas Centrales de Primer Orden

$$\frac{\partial f(x_i)}{\partial x} = \frac{f(x_i + h) - f(x_i - h)}{2h} \tag{A.1}$$

Diferencias Finitas Centrales de Segundo Orden

$$\frac{\partial^2 f(x_i)}{\partial^2 x} = \frac{f(x_i + h) - 2f(x_i) + f(x_i - h)}{h^2} \tag{A.2}$$

Caso Especial

$$\begin{aligned}\frac{\partial}{\partial x}\left(\lambda \frac{\partial f(x)}{\partial x}\right) =& \frac{1}{h_x}\left[\left(\frac{\lambda}{h_{x,e}}\right)_{x_1+\frac{h_1}{2}} [f(x_i + h_x) - f(x_i)]\right] - \\ & \frac{1}{h_x}\left[\left(\frac{\lambda}{h_{x,w}}\right)_{x_1-\frac{h_x}{2}} [f(x_i - f(x_i - h_x)]\right]\end{aligned} \tag{A.3}$$

A.2. Modelo de Pozos de Peaceman

Con el fin de implementar los términos fuente/sumidero en el punto P de una manera adecuada, se considera el modelo de Peaceman.

$$q_\alpha = WI_\alpha \frac{\rho_\alpha k_{r\alpha}}{\mu_\alpha} \left[p_{bh} - p_\alpha - p_\alpha \gamma (p_{bh} - p_\alpha) \right] \tag{A.4}$$

Donde el Radio Equivalente, r_e, y el Índice de Pozo, WI, se definen de la siguiente manera:

$$r_e = \frac{0{,}14}{0{,}5} \frac{\sqrt{\left(\frac{k_{yy}}{k_{xx}}\right)^{\frac{1}{2}} \Delta x^2 + \left(\frac{k_{xx}}{k_{yy}}\right)^{\frac{1}{2}} \Delta y^2}}{\left(\frac{k_{yy}}{k_{xx}}\right)^{\frac{1}{4}} + \left(\frac{k_{xx}}{k_{yy}}\right)^{\frac{1}{4}}} \tag{A.5}$$

$$WI = \frac{2\pi \Delta z \sqrt{k_{xx} k_{yy}}}{\left(ln \frac{r_e}{r_{well}} + s \right)} \tag{A.6}$$

Para este caso se desprecian los efectos gravitacionales y de daño en el pozo. De este modo, los términos fuente quedan de la forma:

$$\frac{q_\alpha}{\rho_\alpha} = Q_\alpha = WI_\alpha \frac{k_{r\alpha}}{\mu_\alpha} (p_{bh} - p_\alpha) \tag{A.7}$$

El modelo de Peaceman, para el escenario con un pozo inyector y otro productor, se usará una vez para el nodo inyector (inyección de agua) y dos veces para el nodo productor (eventual producción de agua y aceite). Previamente se utilizó el subíndice α para que éste sea sustituido con las correspondientes fases agua y aceite. Sin embargo, en la ecuación (A.6) se observa que el WI es el único parámetro que no depende de los fluidos, sino de la geometría del pozo exclusivamente. Es por esto que será necesario definir dos WI (del pozo inyector y pozo productor) y tres términos fuente (inyección de un fluido y producción de dos fluidos).

Índice de Pozo de Inyección

$$WI_{inj} = \frac{2\pi \Delta z \sqrt{k_{xx} k_{yy}}}{\left(ln \frac{r_e}{r_{well,inj}} + s \right)} \tag{A.8}$$

Índice de Pozo de Producción

$$WI_{prod} = \frac{2\pi \Delta z \sqrt{k_{xx} k_{yy}}}{\left(ln \frac{r_e}{r_{well,prod}} + s \right)} \tag{A.9}$$

Ahora, de la ecuación (A.7),sustituyendo α por las respectivas fases agua (w) y aceite (o), y usando la definición $p_w = p_o - p_c$, los términos fuente quedan:

Agua(inyección)

$$\frac{q_w}{\rho_w} = Q_w = (WI_{inj}) \frac{k_{rw}}{\mu_w} (p_{bh} + p_c - p_o) \tag{A.10}$$

Aceite(producción)

$$\frac{q_o}{\rho_o} = Q_o = (WI_{prod}) \frac{k_{ro}}{\mu_o} (p_{bh} - p_o) \tag{A.11}$$

Agua(producción)

$$\frac{q_w}{\rho_w} = Q_w = (WI_{prod}) \frac{k_{rw}}{\mu_w} (p_{bh} + p_c - p_o) \tag{A.12}$$

Apéndice B

IMPES bidimensional

B.1. Modelo Matemático

- Se tiene un sistema de dos fases, agua (w) y aceite (o).
- Las fases se consideran incompresibles.
- La porosidad de la roca es constante.

Presión

$$\nabla \cdot \left(-\bar{k}\lambda \nabla p_o + \bar{k}\lambda_w \frac{dp_c}{dS_w} \nabla S_w \right) = Q_w + Q_o \tag{B.1}$$

Saturación

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot \left(-\bar{k}\lambda_w \nabla p_o + \bar{k}\lambda_w \frac{dp_c}{dS_w} \nabla S_w \right) = Q_w \tag{B.2}$$

Donde k es el tensor de permeabilidad, λ es la movilidad total, p_o es la presión del aceite, p_c es la presión capilar, S_w es la saturación de agua y Q_α es el término fuente/sumidero de la fase α.

B.2. Modelo Numérico

B.2.1. Ecuación de Presión

Discretizando por diferencias finitas la ecuación de presión, esta queda de la siguiente manera. Es preciso mencionar que para esta discretización y la de la ecuación de saturación, ya se implementó el modelo de Peaceman.

$$TE_{i,j}(p_E) + TW_{i,j}(p_W) + TN_{i,j}(p_N) + TS_{i,j}(p_S) - TP_{i,j}(p_P) = B_P \tag{B.3}$$

Los coeficientes que acompañan a las presiones son comúnmente llamados *Transmisibilidades*, y estas se definen:

$$\begin{aligned} TE_{i,j} &= \left(\frac{k_{xx}\lambda}{\delta x}\right)_e \Delta y \Delta z & TW_{i,j} &= \left(\frac{k_{xx}\lambda}{\delta x}\right)_w \Delta y \Delta z \\ TN_{i,j} &= \left(\frac{k_{yy}\lambda}{\delta y}\right)_e \Delta x \Delta z & TS_{i,j} &= \left(\frac{k_{yy}\lambda}{\delta y}\right)_w \Delta x \Delta z \end{aligned} \tag{B.4}$$

$$TP_{i,j} = (TE_{i,j} + TW_{i,j} + TN_{i,j} + TS_{i,j}) + \lambda_w(WI_{inj} + WI_{prod}) + \lambda_o(WI_{prod}) \tag{B.5}$$

Debido a que el problema en cuestión es bidimensional, en la discretización aparecen 5 transmisibilidades. Cada una de estas corresponderá a una diagonal en la matriz de coeficientes, haciendo a la misma una matriz pentadiagonal. Análogamente, si se escalara el problema a 3D, se tendría una matriz heptadiagonal.

Ahora, el vector B del sistema queda:

$$\begin{aligned} B_{i,j} = &\left(T_e\left[\frac{dp_c}{dS}\right]_e + T_w\left[\frac{dp_c}{dS}\right]_w + T_n\left[\frac{dp_c}{dS}\right]_n + T_s\left[\frac{dp_c}{dS}\right]_s\right) S_P - \\ &\left(T_e\left[\frac{dp_c}{dS}\right]_e S_E + T_w\left[\frac{dp_c}{dS}\right]_w S_W + T_n\left[\frac{dp_c}{dS}\right]_n S_N + T_s\left[\frac{dp_c}{dS}\right]_s S_S\right) + \\ &WI_{inj}(\lambda_w)(p_{bh,inj} + p_c) + WI_{prod}(\lambda_o + \lambda_w)(p_{bh,prod} + p_c) \end{aligned} \tag{B.6}$$

$$\begin{aligned} T_e &= \left(\frac{k_{xx}\lambda_w}{\delta x}\right)_e \Delta y \Delta z & T_w &= \left(\frac{k_{xx}\lambda_w}{\delta x}\right)_w \Delta y \Delta z \\ T_n &= \left(\frac{k_{yy}\lambda_w}{\delta y}\right)_n \Delta x \Delta y & T_s &= \left(\frac{k_{yy}\lambda_w}{\delta y}\right)_s \Delta x \Delta y \end{aligned} \tag{B.7}$$

Cabe aclarar que los coeficientes (B.7) no son los definidos previamente en (B.4). La única diferencia entre ellos es que en primer lugar se ocupa en los cálculos la movilidad total (λ) y en este último la movilidad del agua (λ_w).

B.2.2. Ecuación de Saturación

Para la discretización temporal se usa un esquema explícito, lo que significa que aunque la discretización también tenga coeficientes, no se deberá resolver un sistema de ecuaciones como se hace con la presión. Esto es debido a que en el esquema explícito, la saturación al tiempo $n+1$ se calcula con los datos del tiempo n, los cuales son conocidos. Esto resulta en que el cálculo de la nueva saturación es una simple sustitución.

$$Sw_{i,j}^{n+1} = -AP_{i,j}^n(p_P^n) + AE_{i,j}^n(p_E^n) + AW_{i,j}^n(p_W^n) + AN_{i,j}^n(p_N^n) + AS_{i,j}^n(p_S^n) + B_P^n \tag{B.8}$$

$$AE^n_{i,j} = \left(\frac{k_{xx}\lambda_w}{dx}\right)^n_e \frac{\Delta t}{\Delta x\phi} \qquad AW^n_{i,j} = \left(\frac{k_{xx}\lambda_w}{dx}\right)^n_w \frac{\Delta t}{\Delta x\phi}$$
$$AN^n_{i,j} = \left(\frac{k_{yy}\lambda_w}{dy}\right)^n_n \frac{\Delta t}{\Delta y\phi} \qquad AS^n_{i,j} = \left(\frac{k_{yy}\lambda_w}{dy}\right)^n_s \frac{\Delta t}{\Delta y\phi} \tag{B.9}$$

$$AP^n_{i,j} = (AE^n_{i,j} + AW^n_{i,j} + AN^n_{i,j} + AS^n_{i,j}) + \lambda^n_w(WI_{inj} + WI_{prod})\frac{dt}{\Delta V\phi} \tag{B.10}$$

$$\begin{aligned} B_{i,j} = &\left(A^n_e \left[\frac{dp_c}{dS}\right]^n_e + A^n_w \left[\frac{dp_c}{dS}\right]^n_w + A^n_n \left[\frac{dp_c}{dS}\right]^n_n + A^n_s \left[\frac{dp_c}{dS}\right]^n_s\right) S_P - \\ &\left(A^n_e \left[\frac{dp_c}{dS}\right]^n_e S_E + A^n_w \left[\frac{dp_c}{dS}\right]^n_w S_W + A^n_n \left[\frac{dp_c}{dS}\right]^n_n S_N + A^n_s \left[\frac{dp_c}{dS}\right]^n_s S_S\right) + \\ &(Sw^n_{i,j} + \lambda^n_w[(p_{bh,inj} + p_c)WI_{inj} + (p_{bh,prod} + p_c)WI_{prod}])\frac{dt}{\Delta V\phi} \end{aligned} \tag{B.11}$$

$$A^n_e = \frac{k_{xx}(\lambda_w)^n_e}{\delta x_e\phi\Delta x} \qquad A^n_w = \frac{k_{xx}(\lambda_w)^n_w}{\delta x_w\phi\Delta x}$$
$$A^n_n = \frac{k_{yy}(\lambda_w)^n_n}{\delta y_n\phi\Delta y} \qquad A^n_s = \frac{k_{yy}(\lambda_w)^n_s}{\delta y_s\phi\Delta y} \tag{B.12}$$

Cuando los cálculos involucran los efectos de la presión capilar, los coeficientes se vuelven más grandes y a su vez más complicados de manejar computacionalmente. Esto último se traduce en que es probable que el dt deba reducirse hasta tal punto que la simulación converja.

Bibliografía

[1] RAHXION, "Curso de modelado de yacimientos de hidrocarburos. módulo 3." www.rahxion.com, Cd. de México, 2022.

[2] Z. Chen, G. Huan, and Y. Ma, *Computational methods for multiphase flows in porous media.* Computational science & engineering: 2, Society for Industrial and Applied Mathematics, 2006.

[3] G. Strempler, "Cómputo paralelo para la solución de flujo en medios porosos aplicando funciones de base radial," Master's thesis, Universidad Nacional Autónoma de México, 2016.

[4] D. E. Trujillo Escalona, J. J. Sanchez Vela, and M. C. Velazquez Franco, *Apuntes de simulacion numerica de yacimientos.* UNAM, 2007.

[5] M. Wiggins and R. Startzman, "An Approach to Reservoir Management," *Society of Petroleum Engineers Journal*, vol. All Days, 09 1990. SPE-20747-MS.

[6] M. Paris de Ferrer, *Fundamentos de ingeniería de yacimientos.* Maracaibo: Ediciones Astro Data, 2009.

[7] J. H. Abou-Kassem, S. M. Farouq-Ali, and M. R. Islam, *Petroleum reservoir simulations.* Elsevier, 2013.

[8] A. Datta-Gupta and M. J. King, *Streamline simulation : theory and practice.* SPE textbook series: v. 11, Society of Petroleum Engineers, 2007.

[9] M. Muskat, *The Flow of Homogeneous Fluids Through Porous Media.* McGraw-Hill Book Company, 1937.

[10] A. L. Schafer-Perini and J. L. Wilson, "Efficient and accurate front tracking for two-dimensional groundwater flow models," *Water Resources Research*, vol. 27, no. 7, pp. 1471–1485, 1991.

[11] J. Glimm, E. Isaacson, D. Marchesin, and O. McBryan, "Front tracking for hyperbolic systems," *Advances in Applied Mathematics*, vol. 2, no. 1, pp. 91–119, 1981.

[12] F. Bratvedt, K. Bratvedt, C. F. Buchholz, L. Holden, H. Holden, and N. H. Risebro, "A New Front-Tracking Method for Reservoir Simulation," *SPE Reservoir Engineering*, vol. 7, pp. 107–116, 02 1992.

[13] R. S. MacKay, "Level set methods. by j. a. sethian. cambridge university press, 1996. 218 pp. isbn 052187202 9. £27.95," *Journal of Fluid Mechanics*, vol. 345, p. 412–413, 1997.

[14] A. J. Chorin, "Numerical study of slightly viscous flow," *Journal of Fluid Mechanics*, vol. 57, no. 4, p. 785–796, 1973.

[15] D. W. Pollock, "Semianalytical computation of path lines for finite-difference models," *Groundwater*, vol. 26, no. 6, pp. 743–750, 1988.

[16] A. Datta-Gupta and M. J. King, "A semianalytic approach to tracer flow modeling in heterogeneous permeable media," *Advances in Water Resources*, vol. 18, no. 1, pp. 9–24, 1995.

[17] M. J. King and A. Datta-Gupta, "Streamline simulation: A current perspective," *In Situ*, vol. 22, no. 1, pp. 91–140, 1998.

[18] F. Bratvedt, K. Bratvedt, C. Buchholz, T. Gimse, H. Holden, L. Holden, and N. H. Risebro, "Frontline and frontsim, two full scale, two-phase, black oil reservoir simulators based on front tracking," *Surv. Math. Ind*, vol. 3, no. 3, pp. 185–215, 1993.

[19] F. Bratvedt, T. Gimse, and C. Tegnander, "Streamline computations for porous media flow including gravity," *Transport in Porous Media*, vol. 25, no. 1, pp. 63–78, 1996.

[20] *Rapid Simulation of Multiphase Flow Through Fine-Scale Geostatistical Realizations Using a New, 3-D, Streamline Model: A Field Example*, vol. All Days of *SPE Petroleum Computer Conference*, 06 1996. SPE-36008-MS.

[21] *Multiphase Streamline Modeling in Three Dimensions: Further Generalizations and a Field Application*, vol. All Days of *SPE Reservoir Simulation Conference*, 06 1997. SPE-38003-MS.

[22] R. P. Batycky, M. J. Blunt, and M. R. Thiele, "A 3D Field-Scale Streamline-Based Reservoir Simulator," *SPE Reservoir Engineering*, vol. 12, pp. 246–254, 11 1997.

[23] J. LeBlanc and B. Caudle, "A Streamline Model for Secondary Recovery," *Society of Petroleum Engineers Journal*, vol. 11, pp. 7–12, 03 1971.

[24] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Digital integrated circuits*, vol. 2. Prentice Hall Englewood Cliffs, 2002.

[25] J. Savage Carmona, G. Vázquez Rodríguez, and N. E. Chávez Rodríguez, "Diseño de microprocesadores," 2017.

[26] M. J. Quinn, "Parallel programming," *TMH CSE*, vol. 526, p. 105, 2003.