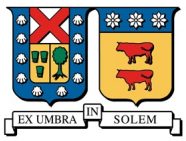


UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Estudio del Comportamiento de una Arteria al Interactuar con la Sangre

Dinámica de Fluidos Computacional

IPM-468 - 2022

Departamento Ing. Mecánica UTFSM

Martín Achondo Mercado, 201860005-9

Eduardo Hasbun Contreras, 201841031-4

Profesor: Dr. Olivier Skurtys

14 de Noviembre de 2022

Resumen

En este informe se intentó modelar el comportamiento de la pared arterial al interactuar con la sangre. Este problema se dividió en dos partes. La primera se despreciaron los efectos longitudinales y axiales. Con esto se obtuvo un sistema de EDO para la desviación de la pared y su velocidad. Este sistema obtenido fue resuelto con el método de Euler Implícito y el método de Crank-Nicolson para la solución transitoria y a grandes tiempos. En ambos casos se pudo modelar satisfactoriamente la ecuación, pero se demostró la diferencia de errores difusivos y la estabilidad a grandes tiempos de ambos métodos. Para la segunda parte, al no despreciar el término longitudinal se obtuvo una ecuación de onda para la desviación de la pared. Esta ecuación fue resuelta con los métodos de Leap-frog y Newmark. Ambos métodos entregaron buenas aproximaciones de la solución y se verificó su segundo orden de convergencia. De todas formas, se confirmó la condicionalidad e incondicionalidad en la estabilidad para el método de Leap-frog y de Newmark respectivamente.

Índice

1. Parte 1: Movimientos de la pared arterial

1.1. Introducción

En esta pregunta se modelará la pared de una arteria respecto a las pulsaciones del corazón. Este fenómeno, si se desprecian las interacciones longitudinales y axiales, genera una EDO de segundo orden. Para resolver este problema se utilizarán los métodos de Euler Implícito y de Crank-Nicolson con el fin de comparar la estabilidad de cada método. Los resultados serán presentados para estados transitorios y para grandes tiempos.

1.1.1. Presentación del problema

Se puede modelar una arteria por un cilindro flexible de base circular, de longitud L y de radio R_0. La pared del cilindro tiene un espesor H y es constituido de un material elástico incompresible, homogéneo e isotrópico.

Se obtiene un modelo simple que describe el comportamiento mecánico de la pared arterial en interacción con la sangre si se supone que el cilindro es constituido de un conjunto de anillos independientes los unos de los otros. Eso permite despreciar las acciones internas longitudinales y axiales a lo largo de la arteria y a suponer que la arteria se deforma solamente según la dirección radial. El radio de la arteria es dado por:

$$R(t) = R_0 + y(t) \tag{1.1}$$

donde t es el tiempo e y es el desplazamiento radial del anillo respecto a un radio de circunferencia R_0. La aplicación de la ley de Newton al sistema de anillos independientes conduce a una ecuación que modela el comportamiento mecánico de la pared en función del tiempo:

$$\frac{\mathrm{d}^2 y(t)}{\mathrm{d}t^2} + \beta \frac{\mathrm{d}y(t)}{\mathrm{d}t} + \alpha y(t) = \gamma(p(t) - p_0) \tag{1.2}$$

en donde:

$$\alpha = \frac{E}{\rho_w R_0^2} \quad \gamma = \frac{1}{\rho_w H} \quad \beta > 0 \tag{1.3}$$

donde: ρ_w es la masa específica y E es el módulo de Young de la pared cilíndrica. La función $p - p_0$ modela el esfuerzo generado por la diferencia de presión entre el interior de la arteria, donde se encuentra la sangre, y el exterior, donde se encuentra otros órganos. Al reposo, cuando $p = p_0$, la arteria tiene una configuración cilíndrica de radio R_0 $(y = 0)$.

Se trabajará con $y(0) = y'(0) = 0$ como condición inicial. Además, se tomarán los siguientes valores para los parámetros físicos: $L = 5 \times 10^{-2}$ m, $R_0 = 5 \times 10^{-3}$ m, $\rho_w = 10^3$ kg/m^3, $H = 3 \times 10^{-4}$ m,

$E = 9 \times 10^5$ N/m^2. La variación de la presión a lo largo de la arteria se modelará como:

$$p(t) - p_0 = x\Delta p(a + b\cos(\omega_0 t)) \tag{1.4}$$

donde: $b = 133.32$ N/m^2, $a = 10b$ N/m^2, $\Delta p = 0.25b$ N/m^2 y $\omega_0 = \frac{2\pi}{0.8}$. Las simulaciones se realizarán en $x = L/2$.

1.2. Metodología

Para resolver la ecuación 1.2 se utilizarán los métodos de Euler Implícito y Crank-Nicolson. Es por esta razón que esta ecuación al ser de segundo orden, será escrita como un sistema de ecuaciones diferenciales de primer orden. Para ello se utiliza la siguiente notación, en donde $z = \frac{\mathrm{d}y}{\mathrm{d}t}$. De esta manera, la ecuación 1.2 queda como:

$$\begin{cases} \dfrac{\mathrm{d}y}{\mathrm{d}t} = z \\ \dfrac{\mathrm{d}z}{\mathrm{d}t} = -\alpha y - \beta z + \gamma p_c(t) \end{cases} \tag{1.5}$$

En donde $p_c(t) = p(t) - p_0$, como notación. De esta manera, en forma matricial ($\mathbf{y}' = A\mathbf{y} + \mathbf{b}$) se puede escribir como:

$$\begin{pmatrix} y' \\ z' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -\alpha & -\beta \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} + \begin{pmatrix} 0 \\ \gamma p_c(t) \end{pmatrix} \tag{1.6}$$

La solución de este tipo de problemas tendrá la forma de:

$$\mathbf{y}(t) = \sum_{j=1}^{n} C_j \mathbf{v}_j \mathrm{e}^{\lambda_j} t + \mathbf{d}(t) \tag{1.7}$$

De donde $\mathbf{v}_j$ corresponde a los vectores propios asociados a los valores propios λ_j. Notar que el primer término corresponde a la solución transiente y se espera que tienda a cero con el tiempo asegurando estabilidad (ocurre si la parte real de los valores propios es negativa) y el segundo término corresponde a la solución cuasi-estacionaria a la cual tiende la solución $\mathbf{y}$ cuando $t \to \infty$.

De forma analítica, se pueden calcular los valores propios de la matriz A los cuales pueden dar indicios del comportamiento de la solución y la estabilidad de los métodos. Desarrollando, los valores propios corresponden a:

$$\lambda = \frac{-\beta \pm \sqrt{\beta^2 - 4\alpha}}{2} \tag{1.8}$$

Notar que si $\beta \geq 2\sqrt{\alpha}$ ambos valores propios son reales. Por otra parte, si $\beta < 2\sqrt{\alpha}$, ambos valores propios son complejos conjugados. Para las simulaciones se utilizarán dos valores de β, estos son: $\beta = \sqrt{\alpha}$ y $\beta = \alpha$. Con esto, los valores propios de la matriz A son:

1. Para $\beta = \sqrt{\alpha}$:
   - λ_1 = -3000 + 5192i
   - λ_2 = -3000 - 5192i

   Dado que son complejos con parte real negativa, la parte homogénea tiende a oscilaciones amortiguadas en la solución que decaen con el tiempo.

2. Para $\beta = \alpha$:
   - λ_1 = -1
   - $\lambda_2 = -3.6 \times 10^7$

   Dado que ambos son negativos y reales, la parte homogénea es asintóticamente estable para la solución decayendo exponencialmente con el tiempo.

1.2.1. Discretización de Euler Implícito

El método de Euler implícito discretiza la ecuación

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = F(\mathbf{y}, t) \tag{1.9}$$

de la forma:

$$\frac{\mathbf{y}^{n+1} - \mathbf{y}^n}{\Delta t} = F(\mathbf{y}^{n+1}, t^{n+1}) \tag{1.10}$$

Aplicado a la ecuación 1.6 obtenida anteriormente y usando la notación $f^n = \gamma p_c(t^n)$ y $h = \Delta t$, se obtiene:

$$\frac{1}{h}\begin{pmatrix} y^{n+1} - y^n \\ z^{n+1} - z^n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -\alpha & -\beta \end{pmatrix}\begin{pmatrix} y^{n+1} \\ z^{n+1} \end{pmatrix} + \begin{pmatrix} 0 \\ f^{n+1} \end{pmatrix} \tag{1.11}$$

Despejando las componentes de la solución en el paso $n+1$, se obtiene:

$$\begin{pmatrix} 1 & -h \\ h\alpha & 1 + h\beta \end{pmatrix}\begin{pmatrix} y^{n+1} \\ z^{n+1} \end{pmatrix} = \begin{pmatrix} y^n \\ z^n \end{pmatrix} + \begin{pmatrix} 0 \\ hf^{n+1} \end{pmatrix} \tag{1.12}$$

Esto lleva a resolver un sistema lineal en cada iteración. Para evitar tanto cálculo y programar una resolución del sistema lineal, se prefiere resolver analíticamente este sistema. Esto se puede realizar fácilmente dado que es un sistema de 2x2. De esta manera, en cada paso temporal se tiene:

$$\begin{cases} y^{n+1} = \dfrac{(\beta h + 1)y^n + hz^n + h^2 f^{n+1}}{\alpha h^2 + \beta h + 1} \\ z^{n+1} = \dfrac{-\alpha h y^n + z^n + hf^{n+1}}{\alpha h^2 + \beta h + 1} \end{cases} \tag{1.13}$$

Este método presentado tiene una convergencia de primer orden y es incondicionalmente estable.

1.2.2. Discretización de Crank-Nicolson

El método de Crank-Nicolson discretiza la ecuación

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = F(\mathbf{y}, t) \tag{1.14}$$

de la forma:

$$\frac{\mathbf{y}^{n+1} - \mathbf{y}^n}{\Delta t} = \frac{1}{2}\left(F(\mathbf{y}^{n+1}, t^{n+1}) + F(\mathbf{y}^n, t^n)\right) \tag{1.15}$$

Aplicado a la ecuación 1.6 obtenida anteriormente y usando la notación $f^n = \gamma p_c(t^n)$ y $h = \Delta t$, se obtiene:

$$\frac{1}{h}\begin{pmatrix} y^{n+1} - y^n \\ z^{n+1} - z^n \end{pmatrix} = \frac{1}{2}\left(\begin{pmatrix} 0 & 1 \\ -\alpha & -\beta \end{pmatrix}\begin{pmatrix} y^{n+1} \\ z^{n+1} \end{pmatrix} + \begin{pmatrix} 0 \\ f^{n+1} \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ -\alpha & -\beta \end{pmatrix}\begin{pmatrix} y^n \\ z^n \end{pmatrix} + \begin{pmatrix} 0 \\ f^n \end{pmatrix}\right) \tag{1.16}$$

Despejando las componentes de la solución en el paso $n+1$, se obtiene:

$$\begin{pmatrix} 1 & -\frac{1}{2}h \\ \frac{1}{2}h\alpha & 1 + \frac{1}{2}h\beta \end{pmatrix}\begin{pmatrix} y^{n+1} \\ z^{n+1} \end{pmatrix} = \begin{pmatrix} y^n + \frac{1}{2}hz^n \\ z^n - \frac{1}{2}h(\alpha y^n + \beta z^n) \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{1}{2}h(f^{n+1} + f^n) \end{pmatrix} \tag{1.17}$$

Esto lleva a resolver un sistema lineal en cada iteración. Para evitar tanto cálculo y programar una resolución del sistema lineal, se prefiere resolver analíticamente este sistema. Esto se puede realizar fácilmente dado que es un sistema de 2x2. De esta manera, en cada paso temporal se tiene:

$$\begin{cases} y^{n+1} = \dfrac{(2\beta h + 4)y^n + 2hz^n + h(2z^n - \alpha h y^n) + h^2(f^{n+1} + f^n)}{\alpha h^2 + 2\beta h + 4} \\ z^{n+1} = \dfrac{-2\alpha h y^n + 4z^n - h(2\alpha y^n + (\alpha h + 2\beta)z^n) + 2h(f^{n+1} + f^n)}{\alpha h^2 + 2\beta h + 4} \end{cases} \tag{1.18}$$

Este método presentado tiene una convergencia de segundo orden y es incondicionalmente estable.

1.2.3. Discretización θ

Para condensar ambas discretizaciones anteriores en solo 1, se utilizará el método θ. Este discretiza la ecuación:

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = F(\mathbf{y}, t) \tag{1.19}$$

de la forma:

$$\frac{\mathbf{y}^{n+1} - \mathbf{y}^n}{\Delta t} = \theta F(\mathbf{y}^{n+1}, t^{n+1}) + (1 - \theta)F(\mathbf{y}^{n+1}, t^{n+1}) \tag{1.20}$$

Notar que si $\theta = 1$ se recupera el método de Euler Implícito y si $\theta = \frac{1}{2}$ se recupera el método de Crank-Nicolson. Dado esto, se programará el método θ y se variará el parámetro dependiendo la discretización que se quiera utilizar.

Aplicado a la ecuación 1.6 obtenida anteriormente y usando la notación $f^n = \gamma p_c(t^n)$ y $h = \Delta t$, se obtiene:

$$\frac{1}{h}\begin{pmatrix} y^{n+1} - y^n \\ z^{n+1} - z^n \end{pmatrix} = \theta\left(\begin{pmatrix} 0 & 1 \\ -\alpha & -\beta \end{pmatrix}\begin{pmatrix} y^{n+1} \\ z^{n+1} \end{pmatrix} + \begin{pmatrix} 0 \\ f^{n+1} \end{pmatrix}\right) + (1-\theta)\left(\begin{pmatrix} 0 & 1 \\ -\alpha & -\beta \end{pmatrix}\begin{pmatrix} y^n \\ z^n \end{pmatrix} + \begin{pmatrix} 0 \\ f^n \end{pmatrix}\right) \tag{1.21}$$

Despejando las componentes de la solución en el paso $n+1$, se obtiene:

$$\begin{pmatrix} 1 & -h\theta \\ h\alpha\theta & 1 + h\beta\theta \end{pmatrix}\begin{pmatrix} y^{n+1} + h(1-\theta)z^n \\ z^{n+1} - h(1-\theta)(\alpha y^n + \beta z^n) \end{pmatrix} = \begin{pmatrix} y^n \\ z^n \end{pmatrix} + \begin{pmatrix} 0 \\ h\left(\theta f^{n+1} + (1-\theta)f^n\right) \end{pmatrix} \tag{1.22}$$

Esto lleva a resolver un sistema lineal en cada iteración. Para evitar tanto cálculo y programar una resolución del sistema lineal, se prefiere resolver analíticamente este sistema. Esto se puede realizar fácilmente dado que es un sistema de 2x2. De esta manera, en cada paso temporal se tiene:

$$\begin{cases} y^{n+1} = \dfrac{(\beta h\theta + 1)y^n + h\theta z^n + h(1-\theta)(z^n - \alpha h\theta y^n) + h^2\theta\left(\theta f^{n+1} + (1-\theta)f^n\right)}{\alpha h^2\theta^2 + \beta h\theta + 1} \\ z^{n+1} = \dfrac{-\alpha h\theta y^n + z^n - h(1-\theta)(\alpha y^n + (\alpha h\theta + \beta)z^n) + h\left(\theta f^{n+1} + (1-\theta)f^n\right)}{\alpha h^2\theta^2 + \beta h\theta + 1} \end{cases} \tag{1.23}$$

1.2.4. Análisis de estabilidad

La parte transitoria de la solución a grandes tiempos será evaluada a partir de una aproximación por medio de sus valores propios y el paso temporal para analizar su estabilidad. Para el método de Euler Implícito se obtiene:

$$\mathbf{y}^{n+1} = \mathbf{y}^n + h\lambda\mathbf{y}^{n+1} \tag{1.24}$$

De esta forma, resolviendo la recursividad:

$$\mathbf{y}^n_{\mathrm{BE}} = \left(\frac{1}{1-h\lambda}\right)^n \tag{1.25}$$

Se puede realizar lo mismo para el método de Crank-Nicolson:

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \frac{1}{2}h\lambda(\mathbf{y}^{n+1} + \mathbf{y}^n) \tag{1.26}$$

Nuevamente, resolviendo la recursividad:

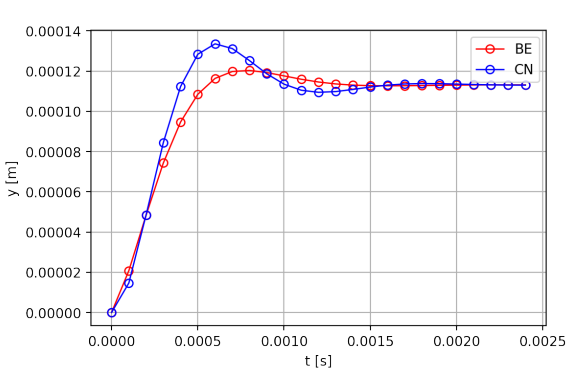
$$\mathbf{y}^n_{\mathrm{CN}} = \left(\frac{1+h\lambda/2}{1-h\lambda/2}\right)^n \tag{1.27}$$

Estos resultados entregan un indicio de la estabilidad a grandes tiempos.

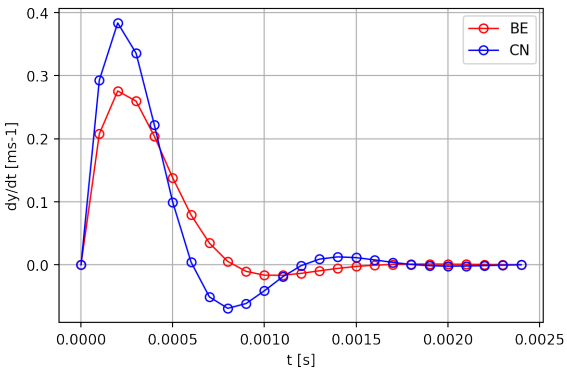
1.3. Resultados

1.3.1. Simulación transitoria

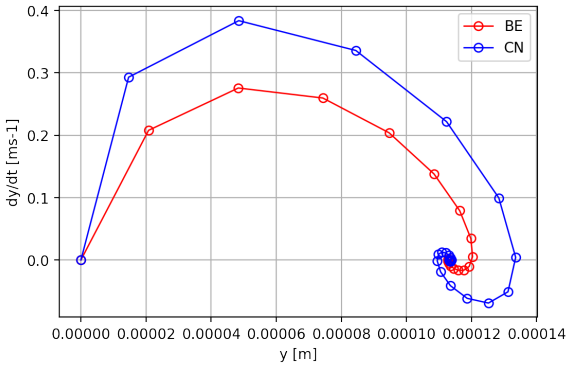
Se presentan los resultados para $\beta = \sqrt{\alpha}$:



(a) $y(t)$ vs t



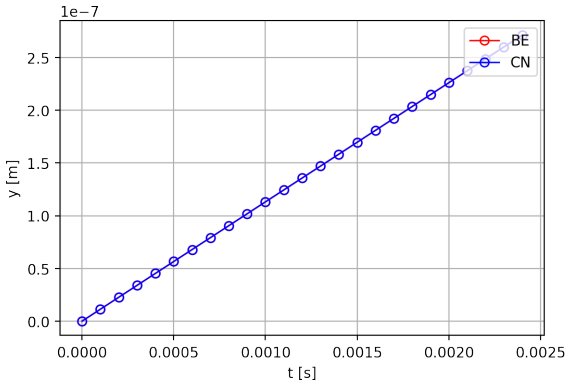
(b) $y'(t)$ vs t



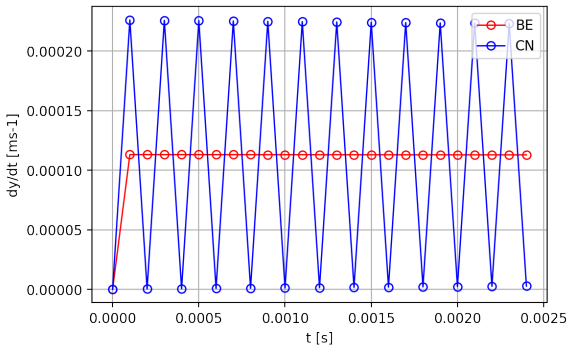
(c) $y'(t)$ vs $y(t)$

Figura 1.1: Resultados transitorios para $\beta = \sqrt{\alpha}$

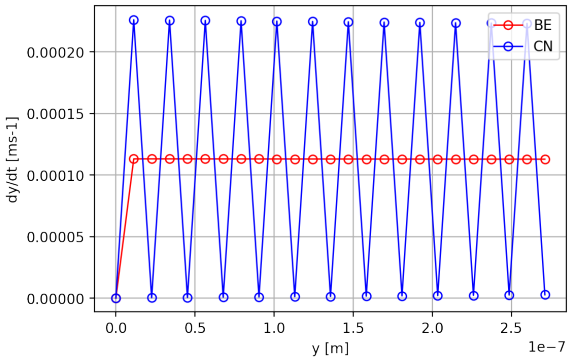
Se presentan los resultados para $\beta = \alpha$:



(a) $y(t)$ vs t



(b) $y'(t)$ vs t

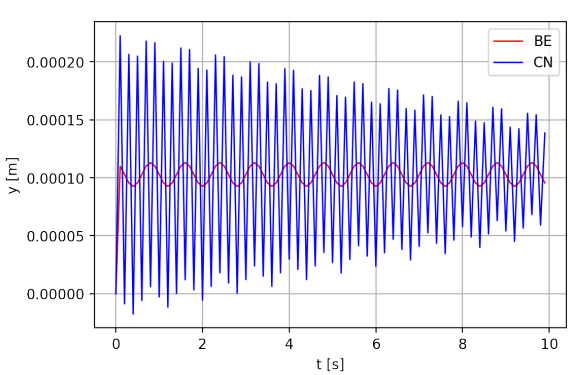


(c) $y'(t)$ vs $y(t)$

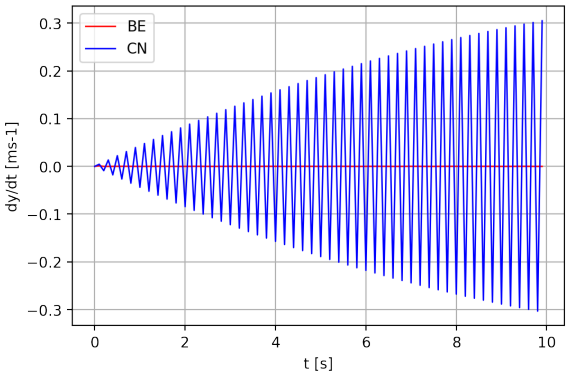
Figura 1.2: Resultados transitorios para $\beta = \alpha$

1.3.2. Simulación para grandes tiempos

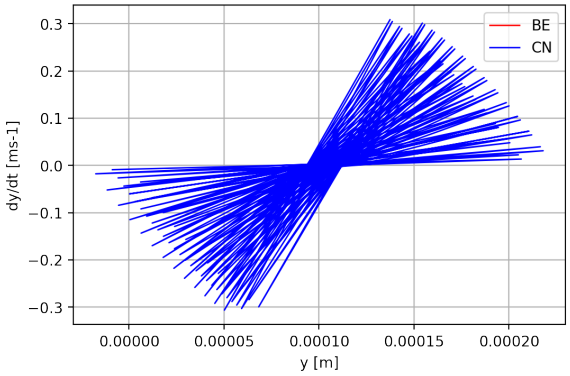
Se presentan los resultados para $\beta = \sqrt{\alpha}$:



(a) $y(t)$ vs t



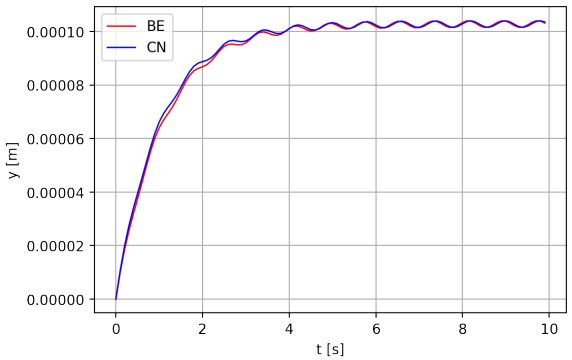
(b) $y'(t)$ vs t



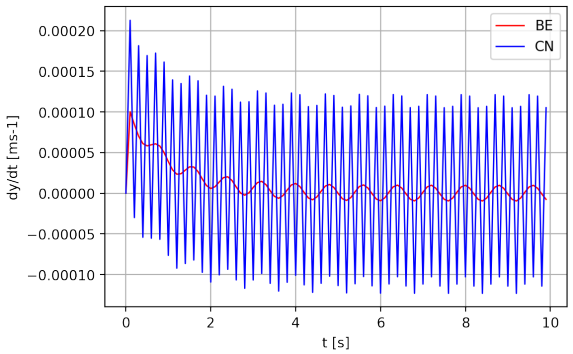
(c) $y'(t)$ vs $y(t)$

Figura 1.3: Resultados transitorios para $\beta = \sqrt{\alpha}$

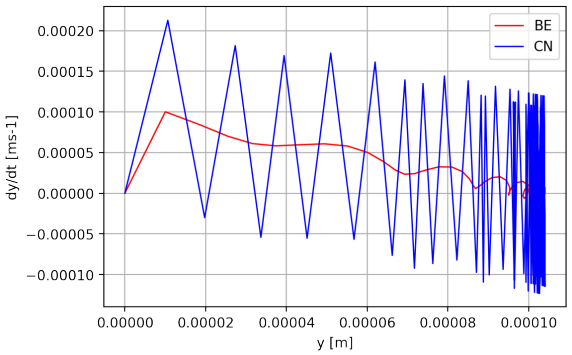
Se presentan los resultados para $\beta = \alpha$:



(a) $y(t)$ vs t



(b) $y'(t)$ vs t



(c) $y'(t)$ vs $y(t)$

Figura 1.4: Resultados transitorios para $\beta = \alpha$

1.4. Análisis de resultados

Con los resultados obtenidos, se nota que se pudo modelar satisfactoriamente la ecuación 1.6 mediante los métodos de Crank-Nicolson y Euler Implícito. Ambos métodos fueron aplicados para la solución transitoria y la estacionaria.

Para la simulación transitoria, se pueden apreciar diferencias respecto a los métodos. En el caso $\beta = \sqrt{\alpha}$, el método de Crank-Nicolson genera una oscilación mayor respecto al método de Euler Implícito en la solución de y y z. Pareciera ser que el método de Euler Implícito genera un error de difusión en donde se disipa esta oscilación, a diferencia del método de Crank-Nicolson. Esto principalmente se puede deber a que el método de CN es de orden 2 a diferencia del BE. De todas formas, ambos siguen la misma tendencia. Respecto al comportamiento de la solución, dado que los valores propios son complejos y la parte real es negativa, la solución tiende a un punto fijo (sumidero). Por otra parte, para $\beta = \alpha$, ambos métodos modelan bien la solución, entregando una solución constante. Sin embargo, el método de Crank Nicolson pareciera añadir oscilaciones respecto al método de Euler Implícito en la solución de la derivada, z. Esto se puede deber a la diferencia grande existente entre el módulo de los valores propios.

Adicionalmente, ambos métodos se comportan de manera similar al disminuir el paso temporal. De todas formas, es importante señalar que ambos métodos son incondicionalmente estable, por lo que al aumentar el paso temporal también convergían a la solución. Por último, para el paso temporal de 10^{-4} utilizado, se podría haber usado el método de Euler Explícito dado que cumple con su condición de estabilidad y dado el valor del módulo de los valores propios.

Respecto a la solución para grandes tiempos, se realizó el siguiente análisis para la estabilidad, explicado en la metodología. Para $\beta = \alpha$:

$$\begin{aligned} y_k^{BE} &\approx (0.9090)^k \\ z_k^{BE} &\approx (0.2777)^k \\ y_k^{CN} &\approx (0.9048)^k \\ z_k^{CN} &\approx (-0.9999)^k \end{aligned} \tag{1.28}$$

Claramente se nota que y_k^{BE}, z_k^{BE}, y_k^{CN} son estables. Esto se ve demostrado en las figuras presentadas. Por esta razón, en la figura 3.4a, el método de Euler y Crank-Nicolson generan una solución bastante similar que se termina estabilizando cercano a los 0.00010. Por otra parte, solo z_k^{CN} contiene un término oscilante. Es por esto que en la figura 3.4b el método de CN genera esas grandes oscilaciones a diferencia de BE. Por último, en el plano de fase se demuestra la gran diferencia dada esta oscilación en el método de CN. En resumen, ambos métodos logran simular a grandes el valor de y, a diferencia del valor de z para el cual el método de Crank-Nicolson genera oscilaciones incluidas por los valores propios.

Para $\beta = \sqrt{\alpha}$:

$$\begin{aligned} y_k^{BE} &\approx (0.0033)^k \\ z_k^{BE} &\approx (0.0033)^k \\ y_k^{CN} &\approx (-0.9867)^k \\ z_k^{CN} &\approx (-0.9867)^k \end{aligned} \tag{1.29}$$

Para estos valores se obtiene que para el método de Euler Implícito, ambos componentes de la solución son estables. Por otra parte, para el método de Crank-Nicolson, ambos componentes entregan una oscilación. Esta se visualiza en las gráficas 3.3 presentadas. Este hecho demuestra que es de suma importancia verificar la estabilidad de los métodos utilizados a partir de los valores propios y el paso temporal antes de sacar cualquier conclusión de la solución obtenida. Para este caso, se prefiere entonces el método de BE, el que claramente entrega el comportamiento de la solución tendiendo al término fuente.

1.5. Conclusiones

Para finalizar, ambos métodos, BE y CN, fueron utilizados para resolver el sistema de ecuaciones 1.6. Dado el término fuente sinusodial, se esperaba que la solución fuera oscilatoria. Ambos métodos fueron aplicados para 2 configuraciones de valores propios, en la cual se obtuvieron diferencias para su estabilidad.

El método de Crank-Nicolson pareció simular de mejor manera la parte transiente. Esto se puede deber a su segundo orden de convergencia. Lo anterior respecto al método de Euler Implícito dado que introdujo un error difusivo. Sin embargo, a grandes tiempos la situación se invirtió. El método de Euler Implícito se comportó asintóticamente estable, a diferencia del método de Crank-Nicolson, el cual agregaba grandes oscilaciones. Es por esto que hay que ser bastante cuidadoso al elegir un método, ya que depende de que tipo de simulación se quiere realizar y la ecuación a modelar.

1.6. Referencias

[1] Chapra, Steven. Canale, Raymond. (2007). *Métodos numéricos para ingenieros*. McGraw-Hill.

[2] Quarteroni A. Sacco, R. Saleri, F. (2000). *Numerical Mathematics*. Springer

[3] Skurtys, Olivier. (2022). *Computación Científica*. UTFSM.

1.7. Anexos

1.7.1. Códigos elaborados

Código 1.1: Clase principal para resolver sistema de EDOs

```
1
2  from f90compiler import f90_compiler
3  import numpy as np
4
5  class Theta_Method():
6
7      def __init__(self):
8
9          function_name = 'theta_method'
10         file_name = 'sub_theta.so'
11
12         self.theta_class = f90_compiler(function_name, file_name)
13
14     def set_params(self):
15         L = 5*10**-2
16         R0 = 5*10**-3
17         rho_w = 10**3
18         H = 3*10**-4
19         E = 9*10**5
20
21         self.alpha = E/(rho_w*R0**2)
22         self.beta = 0
23         self.gamma = 1.0/(rho_w*H)
24         self.x = L/2
25
26     def solve_system(self,theta,tf,h):
27
28         y0 = np.array([[0],[0]])
29         t = 0.0
30         N = int(tf/h)
31
32         y1 = np.zeros((N,1), order='F')
33         y2 = np.zeros((N,1), order='F')
34         t = np.zeros((N,1), order='F')
35
36         y1,y2,t = self.theta_class.theta_func(theta,y0,h,self.x,tf,t,y1,y2,N,
               self.alpha,self.beta,self.gamma)
37
38         return y1,y2,t
39
40     def eigen_values(self):
41         lambda_1 = (-self.beta + np.sqrt(self.beta**2-4*self.alpha + 0j))/2
42         lambda_2 = (-self.beta - np.sqrt(self.beta**2-4*self.alpha + 0j))/2
43         return lambda_1,lambda_2
```

Código 1.2: Subrutina método θ

```
1
2  subroutine theta_method(theta,y0,h,x,tf,t,y1,y2,N,alpha,beta,gamma) bind(C,
       name="theta_method")
3      use iso_c_binding
4      implicit none
5
6      integer(c_int), intent(in), value :: N
7      real(c_double), intent(in), value :: theta,h,x,alpha,beta,gamma,tf
8      real(c_double), intent(in) :: y0(2)
9      real(c_double), intent(out) :: y1(N),y2(N)
10     real(c_double), intent(inout) :: t(N)
11
12     real(c_double) :: y_old(2),y(2),v1,z,tx
13     integer(c_int), parameter :: imax=5000
14     integer(c_int) :: iter
15
16     iter = 2
17     tx = t(1)
18     v1 = h*theta
19     z = 1.0d0/(alpha*(v1)**2+beta*v1 +1)
20
21     y_old(:) = y0(:)
22
23     do while(tx.lt.tf)
24
25         y(1) = z*((beta*v1+1)*y_old(1)+v1*y_old(2) + h*(1-theta)*(-alpha*v1*
               y_old(1)+y_old(2)) + h*gamma*v1*Pt_f(x,tx,theta,h))
26
27         y(2) = z*(-alpha*v1*y_old(1)+y_old(2) + h*(1-theta)*(-alpha*y_old(1)-(
               alpha*v1+beta)*y_old(2))+ h*gamma*Pt_f(x,tx,theta,h))
28
29         y1(iter) = y(1)
30         y2(iter) = y(2)
31         t(iter) = tx + h
32
33         y_old(:) = y(:)
34
35         tx = tx + h
36         iter = iter + 1
37         write(*,*) iter,N
38
39         if (iter.gt.N) then
40             exit
41         end if
42
43     end do
44
45     contains
46
47         real(c_double) function Pt_f(x,t,theta,h)
48             implicit none
49             real(c_double), intent(in) :: x,t,theta,h
50
```

```
51              Pt_f = theta*Pt(x,t+h) + (1-theta)*Pt(x,t)
52
53          end function Pt_f
54
55          real(c_double) function Pt(x,t)
56              implicit none
57
58              real(c_double), intent(in) :: x,t
59              real(c_double) :: b,a,dP,omega_0
60              real(c_double), parameter :: pi = 4.0d0*atan(1.0d0)
61
62              b = 133.32
63              a = 10*b
64              dP = 0.25*b
65              omega_0 = 2.0d0*pi/0.8
66
67              Pt = x*dP*(a+b*cos(omega_0*t))
68
69          end function Pt
70
71  end subroutine theta_method
```

2. Parte 2: Modelo hiperbólico para la interacción de la sangre con la pared arterial

2.1. Introducción

En esta pregunta se modelará la interacción de la sangre con la pared de una arteria respecto a las pulsaciones del corazón. Si se sigue trabajando la ecuación 1.2 sin despreciar la interacción axial entre los anillos la solución para y puede verse modelada con una ecuación de onda. Para resolver este problema se utilizarán los métodos de Leap-frog y Newmark con el fin de comparar la estabilidad y convergencia de cada método. Los resultados serán presentados 2 casos propuestos.

2.1.1. Presentación del problema

Si ahora no se desprecia la interacción axial entre los anillos, la ecuación 1.2 se escribe de la manera siguiente, con x la coordenada longitudinal.

$$\rho_w H \frac{\partial^2 y}{\partial t^2} - \sigma_x \frac{\partial^2 y}{\partial x^2} + \frac{HE}{R_0^2} y = p - p_0 \tag{2.1}$$

donde σ_x es la componente radial del esfuerzo axial y L es el largo del cilindro considerado. En particular si se desprecia el tercer término de la parte izquierda de la ecuación anterior se encuentra una ecuación de onda de la forma siguiente:

$$\frac{\partial^2 u}{\partial t^2} - \gamma^2 \frac{\partial^2 u}{\partial x^2} = f \tag{2.2}$$

Esta ecuación de onda se deberá modelar en base a dos discretizaciones y términos fuentes distintos, con el fin de estudiar el comportamiento de la solución y su convergencia.

2.2. Metodología

La ecuación 2.2 será resuelta bajo las discretizaciones de Leap-frog y Newmark. Se prefieren estos métodos dado que la ecuación es hiperbólica. A continuación se presentan estas discretizaciones.

2.2.1. Discretización de Leap-frog

Este esquema discretiza las derivadas de segundo orden temporales y espaciales de la ecuación 2.2 con diferencias centradas. De esta manera se obtiene:

$$\frac{u_j^{n+1} - 2u_j^n + u_j^{n-1}}{\Delta t^2} - \gamma^2 \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2} = f_j^n \tag{2.3}$$

Despejando los términos de la solución en el paso $n+1$, se obtiene:

$$u_j^{n+1} = 2u_j^n - u_j^{n-1} + \varphi\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right) + \Delta t^2 f_j^n \tag{2.4}$$

donde $\varphi = (\gamma \Delta t/\Delta x)^2$. Dadas las discretizaciones empleadas, el esquema de Leap-frog es de segundo orden en el espacio y tiempo. Notar que el método necesita los valores de la solución de los dos pasos anteriores. Esto implica que el método es explícito. Esto provoca que el método sea condicionalmente estable, por ende lleva a existir la siguiente condición CFL de estabilidad:

$$\gamma \frac{\Delta t}{\Delta x} \leq 1 \tag{2.5}$$

Por otra parte, dado que a la partida se tiene la condición inicial: $\mathbf{u}^0$ y $\mathbf{v}^0$, se necesita una forma de poder adaptar el primer paso del método ya que no se tiene $\mathbf{u}^{-1}$. Esto se se arregla aplicando diferencias centradas a la velocidad inicial. Así:

$$v_j^0 = \frac{u_j^1 - u_j^{-1}}{2\Delta x} \tag{2.6}$$

De esta manera se despeja:

$$u_j^{-1} = u_j^1 - 2\Delta t v_j^0 \tag{2.7}$$

Reemplazando en la ecuación 2.4 para el $n = 0$, se obtiene:

$$u_j^1 = u_j^0 + \Delta t v_j^0 + \frac{\varphi}{2}\left(u_{j+1}^0 - 2u_j^0 + u_{j-1}^0\right) + \frac{1}{2}\Delta t^2 f_j^0 \tag{2.8}$$

De esta manera se puede tener el primer paso temporal. Los siguientes pasos son modelados con la ecuación 2.4 dado que se conocerán los dos pasos anteriores.

2.2.2. Discretización de Newmark

El esquema de Newmark lo que hace es discretizar la solución u y su velocidad paralelamente en base a dos parámetros β y θ. De esta manera en cada paso temporal se necesita calcular u y v. La discretización para ambas variables entonces toma la siguiente forma:

$$u_j^{n+1} - u_j^n = \Delta t v_j^n + \varphi \left[\beta w_j^{n+1} + \left(\frac{1}{2} - \beta\right) w_j^n\right] + \Delta t^2 \left[\beta f_j^{n+1} + \left(\frac{1}{2} - \beta\right) f_j^n\right] \tag{2.9}$$

$$v_j^n = v_j^{n-1} + \frac{\varphi}{2}\left[(1-\theta) w_j^{n-1} + \theta w_j^n\right] + \Delta t \left[\theta f_j^{n+1} + (1-\theta) f_j^n\right] \tag{2.10}$$

En donde:

$$w_j^n = u_{j+1}^n - 2u_j^n + u_{j-1}^n \tag{2.11}$$

Notar que la ecuación 2.9 para u es implícita. Por ende, se despejarán los u_j^{n+1}. De esta forma, la ecuación transforma en:

$$-\varphi\beta u_{j-1}^{n+1} + (1 + 2\varphi\beta) u_j^{n+1} - \varphi\beta u_{j+1}^{n+1} = u_j^n + \Delta t v_j^n + \varphi\left(\frac{1}{2} - \beta\right) w_j^n + +\Delta t^2 \left[\beta f_j^{n+1} + \left(\frac{1}{2} - \beta\right) f_j^n\right] \tag{2.12}$$

La cual se nota que forma un sistemas de ecuaciones trigonal. Este sistema será resuelto en cada paso temporal con el algoritmo de Thomas para sistemas trigonales dado su bajo costo. Es importante destacar que el sistema lineal queda de $(N_x - 2) \times (N_x - 2)$ dado que solo se incluyen los nodos interiores. No es necesario incluir expresiones para los bordes ya que se tomarán condiciones de dirichlet en el borde con valor 0, es decir $u_1 = u_{N_x} = 0$.

En resumen, cada iteración del método de Newmark queda como:

- Se calcula u_j^{n+1} con la ecuación 2.12 resolviendo el sistema con el algoritmo de Thomas.
- Se calcula w_j^{n+1} con la ecuación 2.11 y el resultado anterior.
- Se calcula v_j^{n+1} con la ecuación 2.10.

En general los parámetros β y θ deben satisfacer $0 \leq \beta \leq \frac{1}{2}$, $0 \leq \theta \leq 1$. Además, para que el método de Newmark sea incondicionalmente estable, se debe cumplir que: $2\beta \geq \theta \geq \frac{1}{2}$. Por último, este método es de segundo orden en el tiempo si $\theta = \frac{1}{2}$. Para cualquier otro valor de θ el método queda de segundo orden.

2.2.3. Simulaciones

Para ambas discretizaciones se realizarán 2 simulaciones:

1. Para la primera simulación se considerará el dominio espacio-tiempo de $]0,1[\times]0,1[$ y como término fuente $f(x,t) = (1+\pi^2\gamma^2)\mathrm{e}^{-t}\sin(\pi x)$. Junto a esto, se usarán las condiciones iniciales: $u(x,0) = \sin(\pi x)$ y $v(x,0) = -\sin(\pi x)$. Con esto, la solución exacta adopta la forma de: $u(x,t) = \mathrm{e}^{-t}\sin(\pi x)$. Se utilizarán 4 de las siguientes discretizaciones para el espacio y el tiempo:

$$\Delta x = \Delta t = \frac{1}{2^k \times 10} \tag{2.13}$$

   Con $k = 0, 1, 2, 3$. Con esta malla se calculará el error e_i^k, dado como la máxima diferencia entre la solución aproximada y la solución exacta en el tiempo dado por: $t_i = i/10$ para $i = 1, \cdots, 10$. De esta forma se podrá calcular el orden de convergencia de los métodos como:

$$p_i^k = \frac{\ln\left(\frac{e_i^0}{e_i^k}\right)}{\ln(2^k)} \tag{2.14}$$

   Notar que para esta simulación la solución es independiente de γ. Por esta razón se tomará un $\gamma < 1$ para que cumpla con la condición CFL del método de Leap-frog.

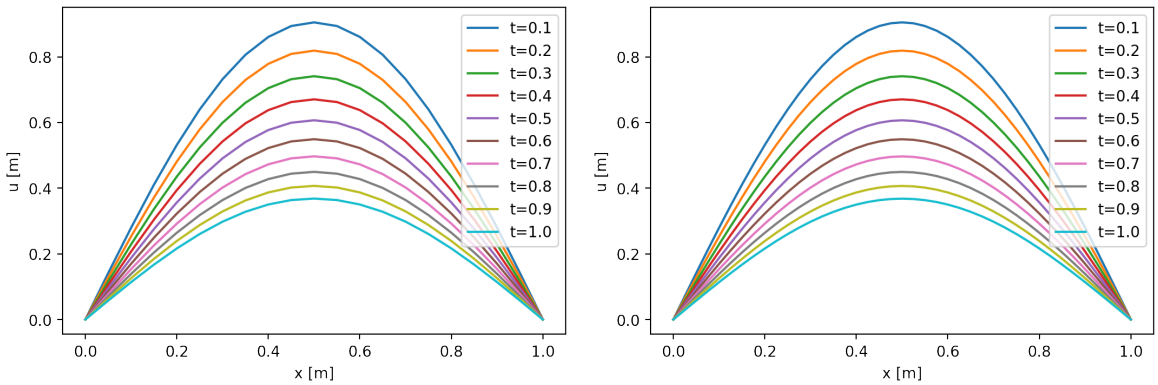
2. Para la segunda simulación se utilizará el dominio espacio-temporal $]0, L[\times]0, T[$, en donde $T = 1$ s. Además, se tomará $\gamma^2 = \frac{\sigma_x}{\rho_w H}$ con $\sigma_x = 1$ kg/s y el término fuente $f = \frac{x\Delta p \sin(\omega_0 t)}{\rho_w H}$. Los valores de estás constantes físicas serán iguales al de la sección pasada dado que se seguirá modelando la interacción sangre - arteria.

   Adicionalmente se utilizarán dos mallas, la primera: $\Delta x = L/10$ y $\Delta t = T/100$ y la segunda: $\Delta x = L/10$ y $\Delta t = T/400$.

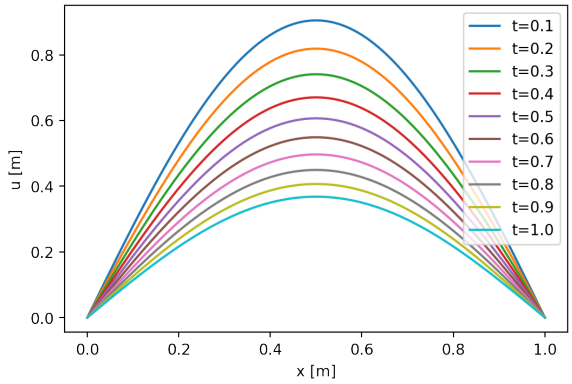
2.3. Resultados

2.3.1. Simulación 1

Dado que para esta simulación la solución no depende de γ, se utilizó $\gamma = 0.9$ para asegurar la estabilidad del método de Leap-frog. El método de Newmark no tiene problemas al ser incondicionalmente estable. Para el método de Newmark s utilizó $\beta = 0.25$ y $\theta = 0.5$.

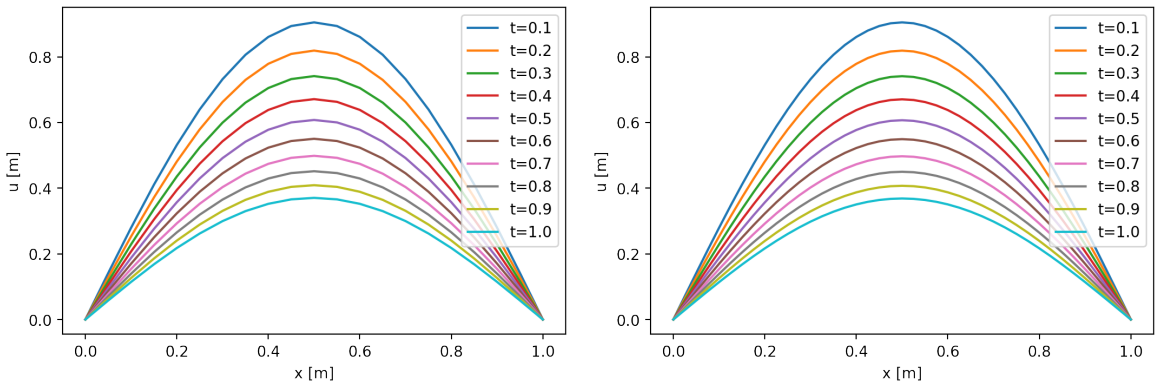


(a) $k = 1$ (b) $k = 2$

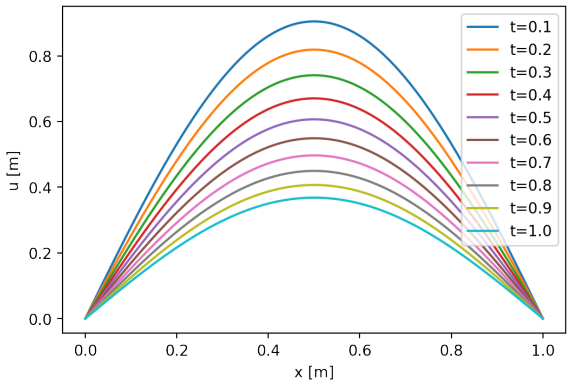


(c) $k = 3$

Figura 2.1: Resultados para esquema de Leap-frog



(a) $k = 1$ (b) $k = 2$



(c) $k = 3$

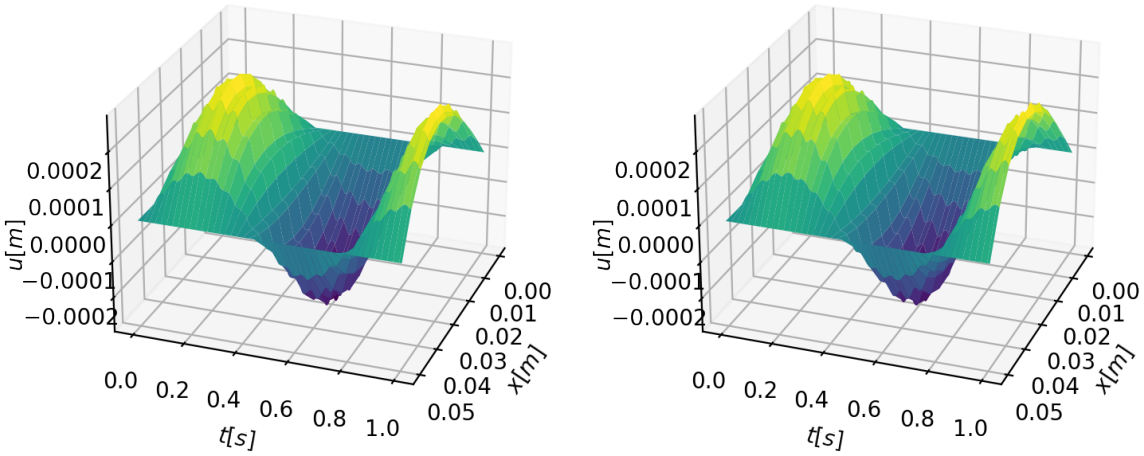
Figura 2.2: Resultados para esquema de Newmark

Tabla 2.1: Convergencia de esquemas de Leap-frog y Newmark

$t_j^{(0)}$	$p_{LF}^{(1)}$	$p_{LF}^{(2)}$	$p_{LF}^{(3)}$	$t_j^{(0)}$	$p_{NW}^{(1)}$	$p_{NW}^{(2)}$	$p_{NW}^{(3)}$
0.1	2.034	2.022	2.012	0.1	1.955	1.972	1.980
0.2	2.022	2.014	2.010	0.2	1.970	1.981	1.987
0.3	2.017	2.011	2.008	0.3	1.975	1.985	1.989
0.4	2.014	2.009	2.006	0.4	1.979	1.987	1.991
0.5	2.012	2.007	2.005	0.5	1.983	1.989	1.992
0.6	2.010	2.006	2.004	0.6	1.987	1.992	1.994
0.7	2.009	2.005	2.004	0.7	1.991	1.994	1.996
0.8	2.007	2.005	2.003	0.8	1.997	1.998	1.999
0.9	2.006	2.004	2.003	0.9	2.003	2.002	2.002
1.0	2.004	2.003	2.002	1.0	2.013	2.008	2.006

2.3.2. Simulación 2

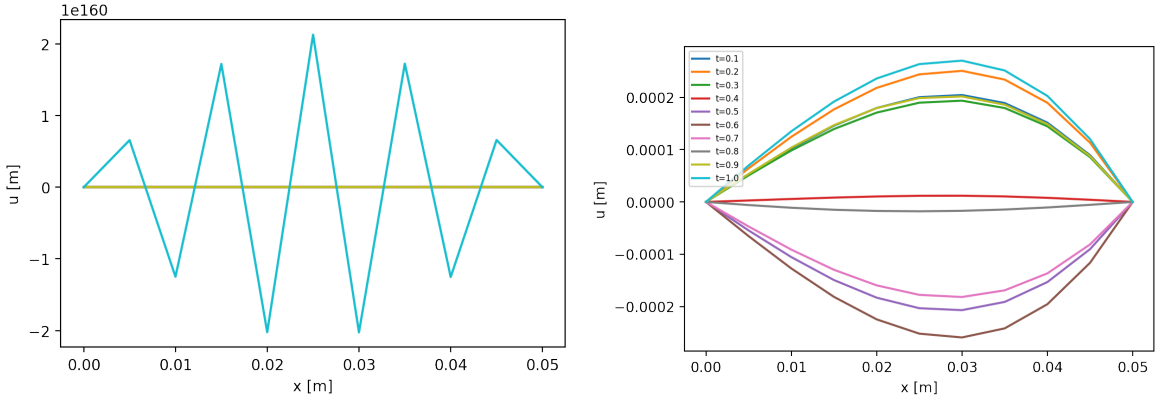
Se presentan los resultados para ambas mallas y esquemas. Para el método de Newmark s utilizó $\beta = 0.25$ y $\theta = 0.5$.



(a) Leap-frog

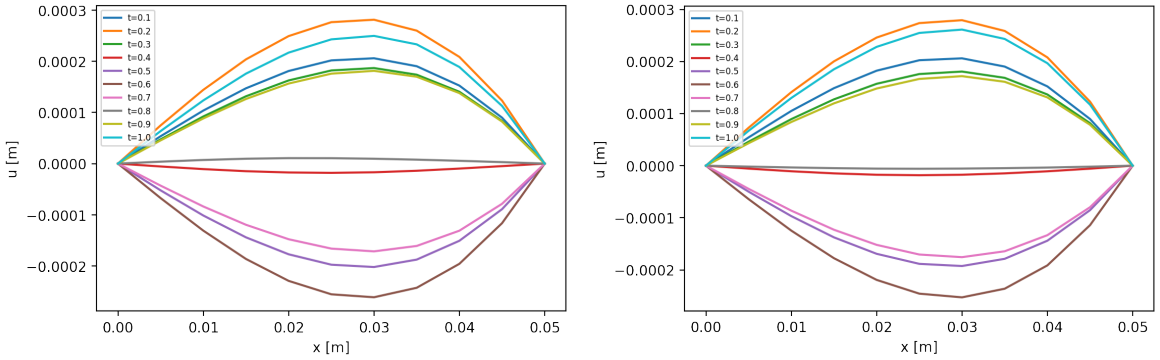
(b) Newmark

Figura 2.3: Resultados para esquemas de Leap-frog y Newmark para malla 2



(a) Leap-frog (b) Newmark

Figura 2.4: Resultados para esquemas de Leap-frog y Newmark para malla 1



(a) Leap-frog (b) Newmark

Figura 2.5: Resultados para esquemas de Leap-frog y Newmark para malla 2

2.4. Análisis de resultados

Con los resultados anteriores, se nota que se pudo modelar la ecuación 2.2 mediante los métodos de Newmark y Leap-frog satisfactoriamente. Junto a esto, se pudo obtener sus convergencias y comparar con la teoría.

Respecto a la simulación 1, ambos métodos lograron representar la solución de la ecuación, concordando con la solución teórica presentada. Sin embargo, es importante destacar que se utilizó un valor de $\gamma = 0.9$ para asegurar la convergencia del método de Leap-frog. Lo anterior se puede realizar dado que la solución analítica no depende de γ. Para ambos métodos, al variar el orden de la malla, el parámetro k, la solución se torna más suave y más cercana a la analítica. De todas formas, para todos los 3 valores de k, las soluciones son bastante similares. Para ambos métodos se calculó la convergencia p_j^k detallada anteriormente. Con sus resultados, se pudo visualizar que ambos métodos son de segundo orden, como se esperaba. Además, se obtuvo que esta convergencia temporal no depende de la malla. Esto se evidencia al entregar valores cercanos a 2 para todos los casos. De todas formas, para el método de Newmark es de orden 2 para valores de $\beta = 0.25$ y $\theta = 0.5$, los cuales fueron usados.

Pasando a la simulación 2, ambos métodos fueron nuevamente usados para modelar la ecuación 2.2. Sin embargo, se modificó la condición inicial y el parámetro γ. Para la malla 1, se nota que el método de Leap-frog diverge. Esto se debe a que no se cumple su condición CFL, $\gamma\lambda > 1$, donde $\lambda = \Delta t/\Delta x$, condición necesaria para asegurar su estabilidad. Por otra parte, el método de Newmark logra simular de buena forma la ecuación. Si se utiliza la malla 2, $\Delta t = T/400$, la condición CFL del método de Leap-frog se cumple y se obtienen resultados similares al método de Newmark. Este problema evidencia la clara estabilidad condicional del método de Leap-frog y la incondicionalidad en la estabilidad del método de Newmark.

Por último, en la figura 3.7, se puede visualizar el comportamiento ondulatorio que tiene la pared arterial respecto al tiempo a lo largo de su longitud. Ambos métodos lograron modelar de buena forma este fenómeno ya que son métodos que se suelen utilizar para resolver numéricamente la ecuación de onda. Como se demostró anteriormente, ambos métodos son de orden 2 en el tiempo. Por esta razón, se obtienen resultados bastante similares para ambos. Es importante destacar que no se asegura que el mismo resultado sea obtenido para otros términos fuentes. Esto se debe a la característica de ambos métodos. Leap-frog utiliza los valores de los pasos n y $n-1$ para obtener los valores en la siguiente iteración. Esto puede provocar cierta inestabilidad o divergencia dependiendo de las oscilaciones del término fuente. El método de Newmark pareciera ser menos sensible al término fuente dado que solo utiliza los valores en el paso n y es implícito. Además, la velocidad se va calculando conjuntamente a la posición, lo cual hace sentido para fenómenos ondulatorios. Además, se pueden variar los parámetros de β y θ dependiendo el caso sacrificando orden de convergencia.

2.5. Conclusiones

Los métodos de Newmark y Leap-frog suelen ser usados para resolver numéricamente la ecuación de onda. Ambos métodos, al ser de orden 2, suelen dar resultados bastante similares entre si. De todas formas, el método de Leap-frog es condicionalmente estable. Por esta razón, hay que elegir cuidadosamente la discretización para que cumpla con su condición CFL. Por otra parte, el método de Newmark es incondicionalmente estable. Dado lo anterior, el método convergió para las mallas utilizadas.

Para finalizar, con ambas discretizaciones, se pudo modelar la oscilación que tiene la pared arterial dada por la ecuación 2.2. Los resultados obtenidos coinciden con la teoría física y resuelven el problema planteado.

2.6. Referencias

[1] Chapra, Steven. Canale, Raymond. (2007). *Métodos numéricos para ingenieros*. McGraw-Hill.

[2] Quarteroni A. Sacco, R. Saleri, F. (2000). *Numerical Mathematics*. Springer

[3] Skurtys, Olivier. (2022). *Computación Científica*. UTFSM.

2.7. Anexos

2.7.1. Códigos elaborados

Código 2.1: Clase principal para resolver la ecuación de onda

```
1
2  from f90compiler import f90_compiler
3  import numpy as np
4
5  class Wave_Eq_Solver():
6
7      def __init__(self,problem=1,method=None):
8
9          function_name = 'leap_frog'
10         file_name = 'leap_frog.so'
11         self.leapfrog_class = f90_compiler(function_name, file_name)
12         function_name = 'newmark'
13         file_name = 'newmark.so'
14         self.newmark_class = f90_compiler(function_name, file_name)
15
16         self.prob = problem
17         self.method = method
18         self.set_params()
19
20     def set_params(self):
21         rho_w = 10**3
22         H = 3*10**-4
23         sigma = 1
24         self.L = 5*10**-2
25         self.T = 1
26         self.gamma = np.sqrt(sigma/(rho_w*H))
27         if self.prob == 1:
28             self.gamma = 0.9
29         self.f_source = self.prob
30
31     def calculate_uk(self,k):
32
33         if self.prob==1:
34             tf = 1
35             xmax = 1
36             dx,dt = self.get_mesh_1(k)
37         elif self.prob==2:
38             tf = self.T
39             xmax = self.L
40             dx,dt = self.get_mesh_2(k)
41
42         Nt = int(tf/dt) +1
43         Nx = int(xmax/dx) +1
44         x = np.linspace(0,xmax,Nx)
45
46         u_0 = np.zeros((2,Nx), order='F')
47         if self.prob==1:
48             u_0[0,:] = np.sin(np.pi*x)
49             u_0[1,:] = -np.sin(np.pi*x)
```

```
50
51          u_out = np.zeros((Nt,Nx), order='F')
52          if self.method=='leapfrog':
53              u_out = self.leapfrog_class.leap_frog_func(u_out,dt,dx,self.gamma,
                    Nt,Nx,self.f_source,u_0)
54          elif self.method=='newmark':
55              beta = 0.25
56              theta = 0.5
57              u_out = self.newmark_class.newmark_func(u_out,dt,dx,self.gamma,Nt,
                    Nx,self.f_source,beta,theta,u_0)
58
59          if self.prob==2:
60              return u_out,x,np.linspace(0,self.T,Nt)
61
62          t = np.linspace(0.1,tf,10)
63          n_p = int(tf/(10*dt))
64          u_j = np.zeros((10,Nx))
65          for j in range(1,11):
66              u_j[j-1,:] = u_out[j*n_p,:]
67
68          return u_j,x,t
69
70      def analytic(self,x,t):
71          an = np.exp(-t)*np.sin(np.pi*x)
72          return an
73
74      def get_mesh_1(self,k):
75          dx = 1/(2**k*10)
76          dt = 1/(2**k*10)
77          return dx,dt
78
79      def get_mesh_2(self,k):
80          dx = self.L/10
81          dt = self.T/(100*(k)**2)
82          return dx,dt
83
84      def get_error(self,u_an,u_ap):
85          return np.max(np.abs(u_an-u_ap))
86
87      def p_convergence(self,k):
88          ujk,xk,t = self.calculate_uk(k)
89          uj0,x0,t2 = self.calculate_uk(0)
90          ejk = np.zeros(len(t))
91          ej0 = np.zeros(len(t2))
92
93          for j in range(len(t)):
94              u_an = self.analytic(xk,t[j])
95              ejk[j] = self.get_error(u_an,ujk[j,:])
96          for j in range(len(t2)):
97              u_an = self.analytic(x0,t2[j])
98              ej0[j] = self.get_error(u_an,uj0[j,:])
99
100         pj = np.log(ej0/ejk)/np.log(2**k)
101
102         return pj
```

Código 2.2: Subrutina para utilizar método de Leap-frog

```
1
2  subroutine leap_frog(u_out,dt,dx,gamma,Nt,Nx,f,u_0) bind(C, name="leap_frog")
3      use iso_c_binding
4      implicit none
5
6      integer(c_int), intent(in), value :: Nt,Nx,f
7      real(c_double), intent(in), value :: dt,dx,gamma
8      real(c_double), intent(in) :: u_0(2,Nx)
9      real(c_double), intent(inout) :: u_out(Nt,Nx)
10     real(c_double) :: u(Nx),u_old(Nx),u_old2(Nx)
11
12     real(c_double) :: t,x,v0(Nx),phi
13     integer(c_int) :: i,j,n_t
14
15     t = 0.0d0
16     x = 0.0d0
17
18     v0(:) = u_0(2,:)
19     u_old2(:) = u_0(1,:)
20     u_old = 0.0d0
21     u = 0.0d0
22     u_out = 0.0d0
23     u_out(1,:) = u_old2(:)
24
25     phi = (gamma*dt/dx)**2
26
27     do i=2,Nx-1
28         x = (i-1)*dx
29         u_old(i) = u_old2(i) + dt*v0(i) +(phi/2.0d0)*(u_old2(i+1) - 2.0d0*
               u_old2(i) + u_old2(i-1)) + (dt**2/2)*func(t,x,f,gamma)
30         u_out(2,i) = u_old(i)
31     end do
32
33     t = t + dt
34
35     do n_t=3,Nt
36         x = 0
37         do j=2,Nx-1
38             x = (j-1)*dx
39             u(j) = 2.0d0*u_old(j) - u_old2(j) + phi*(u_old(j+1)-2.0d0*u_old(j)
                   +u_old(j-1)) + (dt**2)*func(t,x,f,gamma)
40             u_out(n_t,j) = u(j)
41         end do
42
43         u_old2(:) = u_old(:)
44         u_old(:) = u(:)
45
46         t = t + dt
47     end do
48
49     contains
50         real(c_double) function func(t,x,f,gamma)
51             implicit none
```

```
52
53              real(c_double), intent(in) :: x,t,gamma
54              integer(c_int), intent(in) :: f
55              real(c_double), parameter :: pi = 4.0d0*atan(1.0d0)
56              real(c_double) :: dP,rho_w,H,omega_0,a,b,fx
57
58              rho_w = 10**3
59              H = 3.0d-4
60              b = 133.32
61              a = 10*b
62              dP = 0.25*b
63              omega_0 = 2.0d0*pi/0.8
64
65              if (f==1) then
66                  fx = (1+(pi*gamma)**2)*exp(-t)*sin(pi*x)
67              else if (f==2) then
68                  fx = x*dP*sin(omega_0*t)/(rho_w*H)
69              end if
70              func = fx
71
72          end function func
73
74  end subroutine leap_frog
```

Código 2.3: Subrutina para utilizar método de Newmark

```
1
2   subroutine newmark(u_out,dt,dx,gamma,Nt,Nx,f,beta,theta,u_0) bind(C, name="
        newmark")
3       use iso_c_binding
4       implicit none
5
6       integer(c_int), intent(in), value :: Nt,Nx,f
7       real(c_double), intent(in), value :: dt,dx,gamma,theta,beta
8       real(c_double), intent(in) :: u_0(2,Nx)
9       real(c_double), intent(inout) :: u_out(Nt,Nx)
10
11      real(c_double) :: u(Nx),u_old(Nx),w(Nx),w_old(Nx),v(Nx),v_old(Nx)
12      real(c_double) :: b(Nx-2), xx(Nx-2)
13      real(c_double) :: t,x,phi,c1,c2,lambda
14      integer(c_int) :: j,n_t
15
16      t = 0.0d0
17      x = 0.0d0
18
19      xx = 0.0d0
20      b = 0.0d0
21      v_old(:) = u_0(2,:)
22      u_old(:) = u_0(1,:)
23
24      w_old = 0.0d0
25      u = 0.0d0
26      w = 0.0d0
27      v = 0.0d0
28      u_out = 0.0d0
```

```
29      u_out(1,:) = u_old(:)
30
31      lambda = dt/dx
32      phi = (gamma*lambda)**2
33      c1 = -phi*beta
34      c2 = 1.0d0 + 2.0d0*phi*beta
35
36      t = t
37
38      do j=2,Nx-1
39          w_old(j) = u_old(j+1)-2.0d0*u_old(j)+u_old(j-1)
40      end do
41
42      do n_t=2,Nt
43
44          x = 0
45          do j=2,Nx-1
46              x = (j-1)*dx
47
48              b(j-1) = u_old(j) + dt*v_old(j)  + phi*(0.5d0-beta)*w_old(j) + &
49              &dt**2*beta*func(t+dt,x,f,gamma) + dt**2*(0.5d0-beta)*func(t,x,f,
                    gamma)
50
51          end do
52
53          call Thomas(xx,Nx-2,c1,c2,c1,b)
54
55          do j=2,Nx-1
56              u(j) = xx(j-1)
57          end do
58
59          do j=2,Nx-1
60              x = (j-1)*dx
61              w(j) = u(j+1)-2.0d0*u(j)+u(j-1)
62
63              v(j) = v_old(j) + (phi/dt)*( (1-theta)*w_old(j) + theta*w(j)) +&
64              & dt*(theta*func(t+dt,x,f,gamma) + (1-theta)*func(t,x,f,gamma))
65          end do
66
67          w_old(:) = w(:)
68          v_old(:) = v(:)
69          u_old(:) = u(:)
70
71          u_out(n_t,:) = u(:)
72
73          t = t + dt
74      end do
75
76      contains
77          real(c_double) function func(t,x,f,gamma)
78              implicit none
79
80              real(c_double), intent(in) :: x,t,gamma
81              integer(c_int), intent(in) :: f
82              real(c_double), parameter :: pi = 4.0d0*atan(1.0d0)
```

```
83              real(c_double) :: dP,rho_w,H,omega_0,a,b,fx
84
85              rho_w = 10**3
86              H = 3.0d-4
87              b = 133.32
88              a = 10*b
89              dP = 0.25*b
90              omega_0 = 2.0d0*pi/0.8
91
92              if (f==1) then
93                  fx = (1+(pi*gamma)**2)*exp(-t)*sin(pi*x)
94              else if (f==2) then
95                  fx = x*dP*sin(omega_0*t)/(rho_w*H)
96              end if
97              func = fx
98
99          end function func
100
101 end subroutine newmark
```