Estimación de la temperatura con la ecuacion del Bio-Calor usando DeepONet

Francisco Damián Escobar Candelaria
Yofre Hernán García Gómez

2025-04-21

Tabla de contenidos

Resumen

Aquí irá el resumen de la tesis.

1 Introducción

Part I

Preliminares

2 Ecuaciones diferenciales parciales

Las ecuaciones diferenciales parciales (EDPs), al igual que las ecuaciones diferenciales ordinarias (EDOs), se clasifican en lineales y no lineales. De forma análoga a una EDO lineal, la variable dependiente y sus derivadas parciales en una EDP lineal se elevan únicamente a la primera potencia (Zill y Cullen 2008).

2.1 Ecuación diferencial parcial lineal

Si dejamos que u denote la variable dependiente y que x e y representen las variables independientes, entonces la forma general de una **ecuación diferencial parcial lineal de segundo orden** está dada por:

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x\,\partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + Fu = G, \tag{2.1}$$

donde los coeficientes $A, B, C, \ldots, G$ son funciones de x e y. Cuando $G(x, y) = 0$, la ecuación 2.1 se denomina **homogénea**; de lo contrario, es **no homogénea**. Por ejemplo, las ecuaciones lineales:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \text{y} \quad \frac{\partial^2 u}{\partial x^2} - \frac{\partial u}{\partial y} = xy$$

son homogénea y no homogénea, respectivamente.

2.2 Solucion de una PDE

Una **solución** de una ecuación diferencial parcial es una función $u(x, y)$ de dos variables independientes que posee todas las derivadas parciales que aparecen en la ecuación y que satisface dicha ecuación en alguna región del plano xy.

No es lo habitual examinar los procedimientos para encontrar **soluciones generales** de ecuaciones diferenciales parciales lineales. No solo porque suele ser difícil obtener una solución general de una EDP lineal de segundo orden, sino que una solución general generalmente no es tan útil en aplicaciones prácticas. Por lo tanto, el enfoque común es el de encontrar **soluciones particulares** de algunas de las EDPs lineales más importantes, es decir, ecuaciones que aparecen en muchas aplicaciones.

2.3 Separación de variables

Aunque existen varios métodos que pueden intentarse para encontrar soluciones particulares de una EDP lineal, uno de los métodos más comunes se llama **método de separación de variables**. En este método buscamos una solución particular de la forma de un *producto* de una función de x y una función de y:

$$u(x, y) = X(x)Y(y).$$

Con esta suposición, *a veces* es posible reducir una EDP lineal en dos variables a dos ecuaciones diferenciales ordinarias (ODEs). Para este fin, observamos que:

$$\frac{\partial u}{\partial x} = X'Y, \quad \frac{\partial u}{\partial y} = XY', \quad \frac{\partial^2 u}{\partial x^2} = X''Y, \quad \frac{\partial^2 u}{\partial y^2} = XY'',$$

donde las comillas (*primes*) denotan derivación ordinaria.

2.4 Principio de superposición

Teorema 2.1. *Si $u_1, u_2, \dots, u_k$ son soluciones de una ecuación diferencial parcial lineal homogénea, entonces la combinación lineal*

$$u = c_1u_1 + c_2u_2 + \cdots + c_ku_k$$

donde las $c_1 = 1, 2, \dots, k$ son constantes. Es también una solución.

El teorema 2.1 se puede entender como: *siempre que tengamos un conjunto infinito de soluciones $u_1, u_2, u_3, \dots$ de una ecuación lineal homogénea, podemos construir otra solución u mediante la serie infinita*:

$$u = \sum_{k=1}^{\infty} c_k u_k,$$

donde las constantes c_i, con $i = 1, 2, \dots$, son coeficientes.

2.5 Clasificación de ecuaciones

Una ecuación diferencial parcial lineal de segundo orden con dos variables independientes y coeficientes constantes puede clasificarse en uno de tres tipos. Esta clasificación depende únicamente de los coeficientes de las derivadas de segundo orden. Por supuesto, asumimos que al menos uno de los coeficientes A, B o C es distinto de cero.

Definición 2.1. La ecuación diferencial parcial lineal de segundo orden

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x\,\partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + Fu = 0,$$

donde A, B, C, D, F son constantes reales, se dice que es:

- **Hiperbólica** si $B^2 - 4AC > 0$,
- **Parabólica** si $B^2 - 4AC = 0$,
- **Elíptica** si $B^2 - 4AC < 0$.

3 Problemas de valores en la frontera

Si, por ejemplo, $u(x,t)$ es una solución de una EDP, donde x representa una dimensión espacial y t representa el tiempo, entonces es posible prescribir el valor de u, o $\frac{\partial u}{\partial x}$, o una combinación lineal de u y $\frac{\partial u}{\partial x}$ en un valor x especificado, así como prescribir u y $\frac{\partial u}{\partial t}$ en un instante dado t (normalmente, $t = 0$). En otras palabras, un *problema de valores en la frontera* puede consistir en una EDP, junto con *condiciones de frontera* y *condiciones iniciales* (Zill y Cullen 2008).

3.1 Ecuaciones clásicas

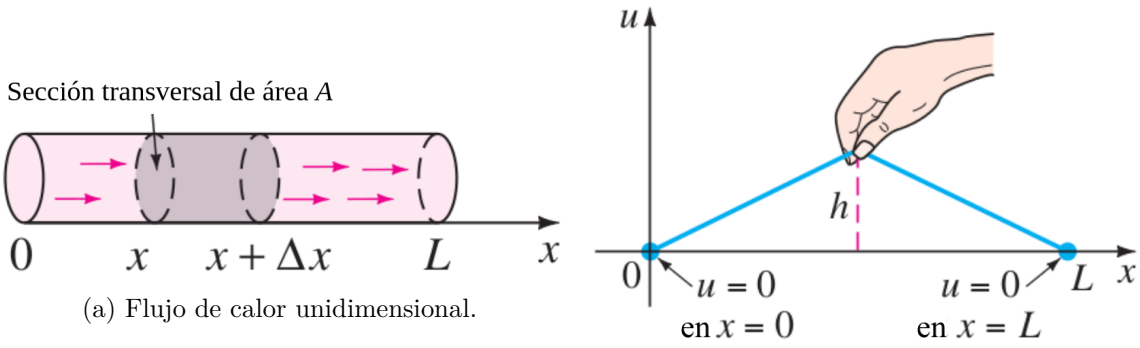
Aplicar el método de separación de variables para encontrar soluciones en forma de producto es muy común con las siguientes *ecuaciones clásicas* de la física matemática:

$$k\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}, \quad k > 0 \tag{3.1}$$

$$\alpha^2\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial t^2} \tag{3.2}$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{3.3}$$

o variantes ligeras de estas ecuaciones. Las EDPs 3.1, 3.2 y 3.3 se conocen, respectivamente, como la *ecuación del calor unidimensional*, la *ecuación de onda unidimensional* y la *forma bidimensional de la ecuación de Laplace*. El término "unidimensional" en el caso de las ecuaciones 3.1 y 3.2 se refiere al hecho de que x denota una variable espacial, mientras que t representa el tiempo; "bidimensional" en 3.3 significa que tanto x como y son variables espaciales. Si comparas 3.1-3.3 con la forma lineal en la Definición 2.1 (donde t juega el papel del símbolo y), observarás que la ecuación del calor 3.1 es parabólica, la ecuación de onda 3.2 es hiperbólica y la ecuación de Laplace 3.3 es elíptica.



(a) Flujo de calor unidimensional.

(b) Cuerda tensada.

Figura 3.1: Aplicaciones de las ecuaciones 3.1 y 3.2 (Zill y Cullen 2008).

3.2 Condiciones iniciales

Dado que las soluciones de las ecuaciones 3.1 y 3.2 dependen del tiempo t, es posible especificar lo que ocurre en $t = 0$; es decir, establecer **condiciones iniciales (CI)**. Si $f(x)$ representa la distribución inicial de temperatura en la varilla mostrada en la Figura 3.1a, entonces una solución $u(x, t)$ de 3.1 debe satisfacer la condición inicial única $u(x, 0) = f(x), \quad 0 < x < L$.

Por otro lado, para una cuerda vibrante podemos especificar tanto su desplazamiento inicial (o forma) $f(x)$ como su velocidad inicial $g(x)$. En términos matemáticos, buscamos una función $u(x, t)$ que satisfaga 3.2 y las dos condiciones iniciales:

$$u(x, 0) = f(x), \quad \left.\frac{\partial u}{\partial t}\right|_{t=0} = g(x), \quad 0 < x < L. \tag{3.4}$$

Por ejemplo, la cuerda podría ser tensada, como se muestra en la Figura 3.1b, y liberada desde el reposo ($g(x) = 0$).

3.3 Condiciones de frontera

La cuerda en la Figura 3.1b está fija al eje x en $x = 0$ y $x = L$ para todos los tiempos. Ésto se interpreta a través de dos **condiciones de frontera (CF):**

$$u(0, t) = 0, \quad u(L, t) = 0, \quad t > 0.$$

En éste contexto la función f en la ec 3.4 es continua y, en consecuencia, $f(0) = 0$ y $f(L) = 0$. En general, existen tres tipos de condicones de frontera asociadas con las ecuaciones 3.1, 3.2 y 3.3. En la frontera es posible especificar los valores de *una* de las siguientes:

$$\text{(i)} \quad u, \qquad \text{(ii)} \quad \frac{\partial u}{\partial n}, \qquad \text{or} \qquad \text{(iii)} \quad \frac{\partial u}{\partial n} + hu, \quad \text{con } h \text{ constante.}$$

Aquí $\frac{\partial u}{\partial n}$ denota la derivada normal de u (la derivada de u en dirección perpendicular a la frontera). Una condición de frontera del primer tipo (i) es llamada **condición de Dirichlet**; una condición de frontera del segundo tipo (ii) es llamada **condición de Neumann**; y una condición de frontera del tercer tipo (iii) es conocida como **condición de Robin**. Por ejemplo, para $t > 0$ una condición típica al extremo derecho de la varilla de la Figura 3.1a puede ser:

$$\text{(i)}' \quad u(L,t) = u_0, \text{ con } u_0 \text{ constante}$$

$$\text{(ii)}' \quad \left.\frac{\partial u}{\partial x}\right|_{x=L} = 0$$

$$\text{(iii)}' \quad \left.\frac{\partial u}{\partial x}\right|_{x=L} = -h(u(L,t) - u_m), \text{ con } h > 0 \text{ y } u_m \text{ constantes}$$

La condición (i)' simplemente establece que el límite $x = L$ se mantiene, por algún medio, a una temperatura constante u_0 durante todo el tiempo $t > 0$. La condición (ii)' indica que el contorno $x = L$ está *aislado*. Según la ley empírica de la transferencia de calor, el flujo de calor a través del borde (es decir, la cantidad de calor por unidad de área por unidad de tiempo conducida a través la frontera) es proporcional al valor de la derivada normal $\frac{\partial u}{\partial n}$ de la temperatura u. Por lo tanto, cuando el límite $x = L$ está aislado térmicamente, no fluye calor hacia dentro ni hacia fuera de la varilla, por lo que

$$\left.\frac{\partial u}{\partial x}\right|_{x=L} = 0.$$

Es posible interpretar (iii)' como que el calor se pierde del extremo derecho de la varilla al estar en contacto con un medio, como el aire o el agua, que se mantiene a temperatura constante. Según la ley de enfriamiento de Newton, el flujo de calor hacia afuera de la varilla es proporcional a la diferencia entre la temperatura $u(L,t)$ en la frontera y la temperatura u_m del medio circundante. Se observa que si se pierde calor por el extremo izquierdo de la varilla, la condición de contorno es

$$\left.\frac{\partial u}{\partial x}\right|_{x=0} = h(u(0,t) - u_m).$$

El cambio de signo respecto de (iii)' corresponde con el supuesto de que la varilla está a una temperatura más alta que el medio que rodea los extremos, de modo que $u(0,t) > u_m$ y $u(L,t) > u_m$. Para $x = 0$ y $x = L$, las pendientes $u_x(0,t)$ y $u_x(L,t)$ deben ser positivas y negativas, respectivamente.

Por supuesto, en los extremos de la varilla se pueden especificar diferentes condiciones al mismo tiempo. Por ejemplo, podríamos tener

$$\left.\frac{\partial u}{\partial x}\right|_{x=0} = 0 \quad \text{y} \qquad u(L,t) = u_0, \quad t > 0.$$

4 Problemas de valor inicial

Las ecuaciones diferenciales son utilizadas para modelar problemas en ciencia e ingeniería que implican el cambio de una variable con respecto a otra. La mayoría de estos problemas requieren la solución de un problema de valor inicial, es decir, la solución de una ecuación diferencial que satisface una condición inicial dada.

En situaciones reales comunes, la ecuación diferencial que modela el problema es demasiado compleja para resolverse con exactitud, y se adopta uno de dos enfoques para aproximar la solución. El primer enfoque consiste en modificar el problema simplificando la ecuación diferencial a una que pueda resolverse con exactitud y luego utilizar la solución de la ecuación simplificada para aproximar la solución del problema original. El otro enfoque utiliza métodos para aproximar la solución del problema original. Este es el enfoque más común porque los métodos de aproximación proporcionan resultados más precisos e información de error realista (Burden y Faires 2010).

Ejemplo

El movimiento de un péndulo oscilante bajo ciertas suposiciones se describe mediante la ecuación diferencial de segundo orden:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin\theta = 0$$

Donde L es la longitud del péndulo, $g \approx 9.81\frac{m}{s^2}$ es la constante gravitacional terrestre y θ es el ángulo que forma el péndulo con la vertical. Si, además, especificamos la posición del péndulo al inicio del movimiento, $\theta(t_0) = \theta_0$, y su velocidad en ese punto, $\theta'(t_0) = \theta'_0$. Tenemos un *problema de valor inicial*.

Para dar una idea más clara acerca de los problemas de valor inicial Burden y Faires (2010) brinda las siguientes definicones y teoremas:

Definición 4.1. Se dice que una función $f(t, y)$ satisface una **Condición de Lipschitz** en la variable y en un conjunto $D \subset \mathbb{R}^2$ si existe una constante $L > 0$ tal que

$$|f(t, y_1) - f(t, y_2)| \leq L|y_1 - y_2|$$

donde $f(t, y_1)\text{y}f(t, y_2)$ están en D. La constante L es llamada **constante de Lipschitz** para f.

Definición 4.2. Se dice que un conjunto $D \subset \mathbb{R}^2$ es **convexo** si para cualesquiera $f(t, y_1), f(t, y_2) \in D$, entonces $((1-\lambda)t_1 + \lambda t_2, (1-\lambda)y_1 + \lambda y_2)$ también pertenece a D para cada $\lambda \in [0, 1]$.

En términos geométricos, la Definición 4.2 establece que un conjunto es convexo siempre que, para cualesquiera dos puntos dentro del conjunto, todo el segmento recto entre ellos también pertenezca al conjunto Figura 4.1.

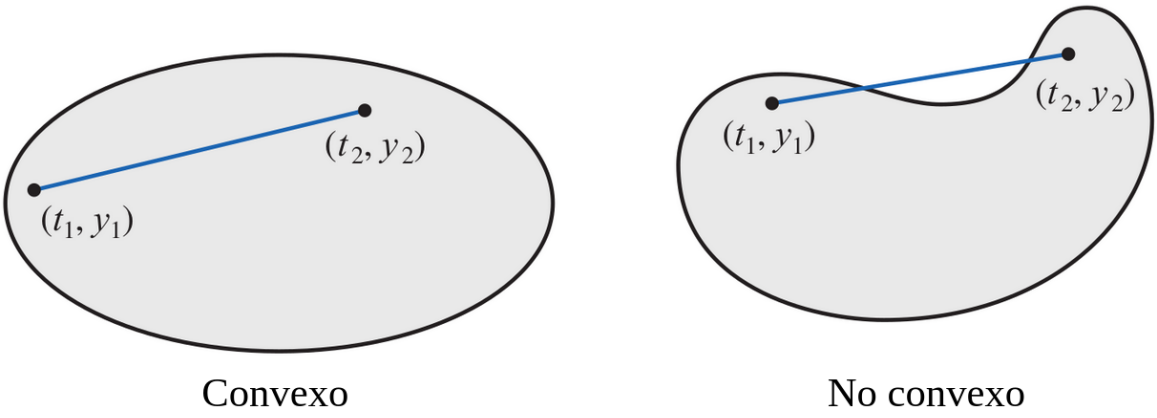


Figura 4.1: Ejemplo geométrico de un conjunto *convexo* y *no convexo* (Burden y Faires 2010).

Teorema 4.1. *Supongamos que $f(t, y)$ está definida en un conjunto convexo $D \in \mathbb{R}^2$. Si existe una constante $L > 0$ con*

$$\left|\frac{\partial f}{\partial y}(t, y)\right| \leq L, \quad \textit{para todo} (t, y) \in D, \tag{4.1}$$

entonces f satisface una condición de Lipschitz en D en la variable y con una constante de Lipschitz L.

Como se mostrará en el siguiente teorema, suele ser de gran interés determinar si la función involucrada en un problema de valor inicial satisface una condición de Lipschitz en su segunda variable, y la condición 4.1 suele ser más fácil de aplicar que la definición. Cabe destacar, sin embargo, que el Teorema 4.1 solo proporciona condiciones suficientes para que se cumpla una condición de Lipschitz.

Teorema 4.2. *Supóngase que $D = \{(t, y) | \ a \leq t \leq b, \ -\infty < y < \infty\}$ y que $f(t, y)$ es continua en D. Si f satisface una condición de Lipschitz en D en la variable y, entonces el problema del valor inicial*

$$y'(t) = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha,$$

tiene una solución única $y(t)$ para $a \leq t \leq b$.

Ejemplo

Use el teorema Teorema 4.2 para mostrar que hay una única solución al problema de valor inicial:

$$y'(t) = 1 + t\sin(ty), \quad 0 \le t \le 2, \quad y(0) = 0.$$

Solución: Mantieniendo a t constante y usando el *Teorema de valor medio* a la función

$$f(t, y) = 1 + t\sin(ty),$$

notamos que cuando $y_1 < y_2$, un número ξ existe en (y_1, y_2) tal que:

$$\frac{f(t, y_2) - f(t, y_1)}{y_2 - y_1} = \frac{\partial}{\partial y} f(t, \xi) = t^2 \cos(\xi t).$$

De este modo:

$$|f(t, y_2) - f(t, y_1)| = |y_2 - y_1||t^2 \cos(\xi t)| \le 4|y_2 - y_1|,$$

y f satisface una condición de Lipschitz en la variable y con constante de Lipschitz $L = 4$. Además, $f(t, y)$ es continua cuando $0 \le t \le 2$ y $-\infty < y < \infty$, por lo que el Teorema 4.2 implica que existe una solución única para este problema de valor inicial.

4.1 Problemas bien plantedos

Ahora que hemos abordado, hasta cierto punto, la cuestión de cuándo los problemas de valor inicial tienen soluciones únicas, podemos pasar a la segunda consideración importante: cuándo aproximar la solución de un problema de valor inicial. Los problemas de valor inicial obtenidos mediante la observación de fenómenos físicos generalmente solo se aproximan a la situación real, por lo que necesitamos saber si pequeños cambios en el plamteamiento del problema introducen cambios correspondientemente pequeños en la solución (Burden y Faires 2010).

A continiación se presentan otras definiciones así como teoremas que brindaŕan un conocimiento más sólido acerca de los problemas bien planteados. Se usará como referencia a Burden y Faires (2010).

Definición 4.3. Se dice que el problema de valor inicial

$$\frac{dy}{dt} = f(t, y), \quad a \le t \le b, \quad y(a) = \alpha, \tag{4.2}$$

es un **problema bien planteado** si:

- Existe una única solución $y(t)$ para el problema, y

- Existen constantes $\varepsilon_0 > 0$ y $k > 0$ tales que para cualquier ε, con $\varepsilon_0 > \varepsilon > 0$, siempre que $\delta(t)$ sea continua con $|\delta(t)| < \varepsilon$ para todo t en $[a, b]$, y cuando $|\delta_0| < \varepsilon$, el problema del valor inicial

$$\frac{dz}{dt} = f(t, z) + \delta(t), \quad a \leq t \leq b, \quad z(a) = \alpha + \delta_0 \tag{4.3}$$

tenga una única solución $z(t)$ que satisface:

$$|z(t) - y(t)| < k\varepsilon \quad \forall t \in [a, b]$$

El problema especificado por al Ecuación 4.3 se denomina **problema perturbado** asociado al problema original Ecuación 4.2. Se asume la posibilidad de que se introduzca un error en el planteamiento de la ecuación diferencial, así como la presencia de un error δ_0 en la condición inicial.

Los métodos numéricos siempre se centrarán en la solución de un problema perturbado, ya que cualquier error de redondeo introducido en la representación perturba el problema original. A menos que el problema original esté bien planteado, hay pocas razones para esperar que la solución numérica de un problema perturbado se aproxime con precisión a la solución del problema original.

Teorema 4.3. *Supongamos* $D = \{(t, y) | \quad a \leq t \leq b, \quad -\infty < y < \infty\}$. *Si* f *es continua y satisface una condición de Lipschitz en la variable* y *en el conjunto* D, *entonces el problema de valor inicial*

$$\frac{dy}{dt} = f(t, y), \quad a \leq t \leq b, \quad y(a) = \alpha$$

es bien planteado.

Ejemplo

Demostrar que el problema de valor inicial

$$\frac{dy}{dt} = y - t^2 + 1, \quad 0 \leq t \leq 2, \quad y(0) = 0.5,$$

está bien planteado en el dominio $D = \{(t, y) | \quad 0 \leq t \leq 2 \quad \text{y} \quad -\infty < y < \infty\}$.
Solución: Dado que

$$\left|\frac{\partial(y - t^2 + 1)}{\partial y}\right| = |1| = 1, \tag{4.4}$$

el Teorema 4.1 implica que la función $f(t, y) = y - t^2 + 1$ satisface una condición de Lipschitz en y sobre D con constante de Lipschitz igual a 1. Además, como f es continua en D, el Teorema 4.3 garantiza que el problema está bien planteado.
A modo de ilustración, consideremos ahora la solución del problema perturbado:

$$\frac{dz}{dt} = z - t^2 + 1 + \delta, \quad 0 \leq t \leq 2, \quad z(0) = 0.5 + \delta_0, \tag{4.5}$$

donde δ y δ_0 son constantes pequeñas, las soluciones respectivas de las ecuaciones 4.4 y 4.5 son:

$$y(t) = (t+1)^2 - 0.5e^t$$

$$z(t) = (t+1)^2 + (\delta + \delta_0 - 0.5)e^t - \delta$$

Sea ε un número positivo. Si $|\delta| < \varepsilon$ y $|\delta_0| < \varepsilon$, entonces

$$|y(t) - z(t)| = |(\delta + \delta_0)e^t - \delta| \leq |\delta + \delta_0|e^2 + |\delta| \leq (2e^2 + 1)\varepsilon,$$

para todo t. Esta desigualdad demuestra que 4.4 está bien planteado, con una constante de estabilidad $k(\varepsilon) = 2e^2 + 1$ para cualquier $\varepsilon > 0$.

5 Método de Crank Nickolson

Existen métodos implícitos de diferencias finitas para resolver ecuaciones diferenciales parciales parabólicas ec 3.1. Estos métodos requieren la resolución de un sistema de ecuaciones para determinar los valores aproximados de u en la línea de tiempo $(j+1)$. Sin embargo, los métodos implícitos no presentan problemas de inestabilidad (Zill y Cullen 2008).

El algoritmo introducido por **J. Crank** y **P. Nicholson** en 1947 se utiliza principalmente para resolver la ecuación del calor. El algoritmo consiste en sustituir la segunda derivada parcial en $c\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}$ por el promedio de dos cocientes de diferencias centrales, uno evaluado en t y el otro en $t+k$:

$$\begin{aligned} &\frac{c}{2}\left[\frac{u(x+h,t)-2u(x,t)+u(x-h,t)}{h^2}\right]+ \\ &+\frac{c}{2}\left[\frac{u(x+h,t+k)-2u(x,t+k)+u(x-h,t+k)}{h^2}\right] \\ &\quad=\frac{1}{k}[u(x,t+k)-u(x,t)] \end{aligned} \tag{5.1}$$

si se define $\lambda = \frac{ck}{h^2}$ y

$$\begin{aligned} u(x+h,t)=u_{i+1,j}, \quad u(x,t)=u_{ij}, \quad u(x-h,t)=u_{i-1,j}, \\ u(x+h,t+k)=u_{i+1,j+1}, \quad u(x,t+k)=u_{i.j+1}, \quad u(x-h,t+k)=u_{i-1,j+1}, \end{aligned}$$

es posible reescribir a la eq 5.1 como:

$$-u_{i-1,j+1}+\alpha u_{i,j+1}-u_{i+1,j+1}=u_{i+1,j}-\beta u_{ij}+u_{i-1,j}, \tag{5.2}$$

donde $\alpha = 2(1+\frac{1}{\lambda})$, $\beta = 2(1-\frac{1}{\lambda})$, $j = 0,1,\ldots m-1$ e $i = 0,1,\ldots n-1$.

Para cada elección de j la ecuación diferencial eq 5.2 para $i = 0,1,\ldots n-1$ da $n-1$ ecuaciones en $n-1$ incógnitas $u_{i,j+1}$. Debido a las condiciones de contorno preestablecidas, los valores de $u_{i,j+1}$ se conocen para $i = 0$ y para $i = n$. Por ejemplo, en el caso $n = 4$, el sistema de ecuaciones para determinar los valores aproximados de u en la línea de tiempo $(j+1)$ es:

$$\begin{aligned} -u_{0,j+1}+\alpha u_{1,j+1}-u_{2,j+1} &= u_{2,j}-\beta u_{1,j}+u_{0,j} \\ -u_{1,j+1}+\alpha u_{2,j+1}-u_{3,j+1} &= u_{3,j}-\beta u_{2,j}+u_{1,j} \\ -u_{2,j+1}+\alpha u_{3,j+1}-u_{4,j+1} &= u_{4,j}-\beta u_{3,j}+u_{2,j} \end{aligned}$$

reordenando se llega a

$$\begin{array}{ccccccc} \alpha u_{1,j+1} & - & u_{2,j+1} & & & & = b_1 \\ -u_{1,j+1} & + & \alpha u_{2,j+1} & - & u_{3,j+1} & & = b_2 \\ & - & u_{2,j+1} & + & \alpha u_{3,j+1} & & = b_3 \end{array} \tag{5.3}$$

donde

$$\begin{aligned} b_1 &= u_{2,j} - \beta u_{1,j} + u_{0,j} + u_{0,j+1}, \\ b_2 &= u_{3,j} - \beta u_{2,j} + u_{1,j}, \\ b_3 &= u_{4,j} - \beta u_{3,j} + u_{2,j} + u_{4,j+1}. \end{aligned}$$

En general, si utilizamos la ecuación diferencial eq 5.2 para determinar valores de u en la línea de tiempo $(j-1)$, es necesario resolver un sistema lineal $\mathbf{AX} = \mathbf{B}$, donde la matriz de coeficientes $\mathbf{A}$ es una **matriz tridiagonal**,

$$A = \begin{pmatrix} \alpha & -1 & 0 & 0 & 0 & \cdots & 0 & \\ -1 & \alpha & -1 & 0 & 0 & \cdots & 0 & \\ 0 & -1 & \alpha & -1 & 0 & \cdots & 0 & \\ 0 & 0 & -1 & \alpha & -1 & \cdots & 0 & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & 0 & 0 & 0 & \cdots & \alpha & -1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & -1 & \alpha \end{pmatrix},$$

y las componentes de la matriz columna $\mathbf{B}$ son

$$\begin{aligned} b_1 &= u_{2,j} - \beta u_{1,j} + u_{0,j} + u_{0,j+1}, \\ b_2 &= u_{3,j} - \beta u_{2,j} + u_{1,j}, \\ b_3 &= u_{4,j} - \beta u_{3,j} + u_{2,j}, \\ &\vdots \\ b_{n-1} &= u_{n,j} - \beta u_{n-1,j} + u_{n-2,j} + u_{n,j+1}. \end{aligned}$$

Part II

Redes neuronales

6 Physic Informed Neural Networks (PINNs)

Las Physics-Informed Neural Networks (PINNs) son un enfoque innovador que combina redes neuronales con ecuaciones diferenciales gobernantes para resolver problemas complejos de física (Blechschmidt y Ernst 2021). A diferencia de métodos tradicionales, las PINNs incorporan directamente las ecuaciones físicas en su función de pérdida mediante diferenciación automática, lo que permite minimizar simultáneamente el error en los datos y el residual de las PDEs (Karniadakis et al. 2021). Esta característica las hace particularmente valiosas en escenarios con datos limitados, donde el conocimiento físico actúa como un regularizador efectivo . La capacidad de aproximación de las PINNs se fundamenta en el teorema de aproximación universal de las redes neuronales, adaptado para incorporar restricciones físicas a través de términos de penalización en la función de optimización (Karniadakis et al. 2021).

Como ejemplo, se considera la **ecuación de Burgers para viscocidad**:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

Con una condición inicial adecuada y condiciones de contorno de Dirichlet. En la figura Figura 6.1, la red izquierda (physics-uninformed) representa el sustituto de la solución de EDP $u(x,t)$, mientras que la red derecha (physics-informed) describe el residuo de EDP $\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2}$. La función de pérdida incluye una pérdida supervisada de las mediciones de datos de u de las condiciones iniciales y de contorno, y una pérdida no supervisada de EDP:

$$\mathcal{L} = w_{\text{data}}\mathcal{L}_{\text{data}} + w_{\text{PDE}}\mathcal{L}_{\text{PDE}} \tag{6.1}$$

donde:

$$\mathcal{L}_{\text{data}} = \frac{1}{N_{\text{data}}}\sum_{i=1}^{N_{\text{data}}} \left(u(x_i,t_i) - u_i\right)^2$$

$$\mathcal{L}_{\text{PDE}} = \frac{1}{N_{\text{PDE}}}\sum_{j=1}^{N_{\text{PDE}}} \left(\frac{\partial u}{\partial t}(x_j,t_j) + u\frac{\partial u}{\partial x}(x_j,t_j) - \nu\frac{\partial^2 u}{\partial x^2}(x_j,t_j)\right)^2$$

Aquí, (x_i,t_i) representan puntos donde se conocen valores de la solución y (x_j,t_j) son puntos interiores del dominio. Los pesos w_{data} y w_{PDE} equilibran la contribución de cada término. La

red se entrena minimizando $\mathcal{L}$ usando optimizadores como Adam o L-BFGS hasta alcanzar un umbral ε (Karniadakis et al. 2021).

Este enfoque permite resolver EDPs (clásicas, fraccionarias o estocásticas) sin mallas, en dominios complejos o con datos escasos y ruidosos, siendo una herramienta flexible y poderosa para la modelación científica.

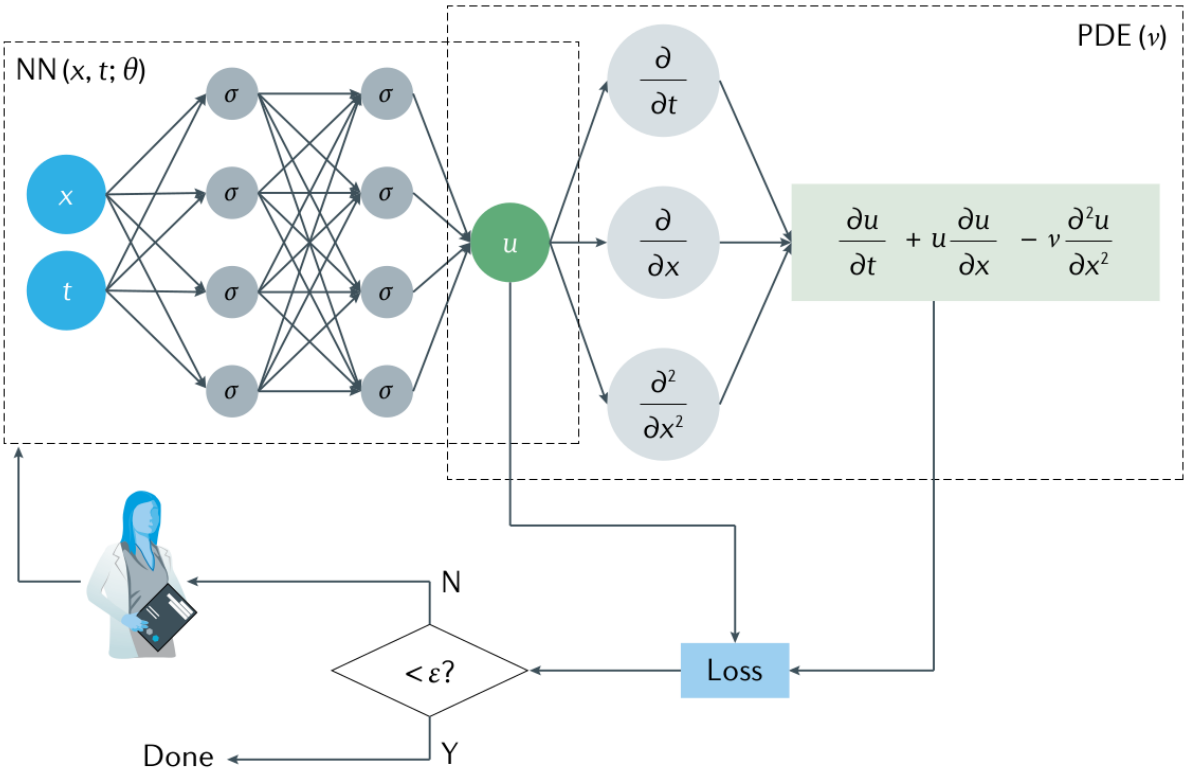


Figura 6.1: **El algoritmo de una PINN.** Construya una red neuronal (NN) $u(x,t;\theta)$ donde θ representa el conjunto de pesos entrenables w y sesgos b, y σ representa una función de activación no lineal. Especifique los datos de medición x_i, t_i, u_i para u y los puntos residuales x_j, t_j para la EDP. Especifique la pérdida $\mathcal{L}$ en la ecuación Ecuación 6.1 sumando las pérdidas ponderadas de los datos y la EDP. Entrene la NN para encontrar los mejores parámetros θ^* minimizando la pérdida $\mathcal{L}$ (Karniadakis et al. 2021).

6.1 Ejemplo de resolución de la ecuación de Burger 1D con deepxde

Dada la ecuación:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = v\frac{\partial^2 u}{\partial x^2} \quad , x \in [-1, 1],\ t \in [0, 1]$$

con la condición de frontera de Dirichlet y condición inicial:

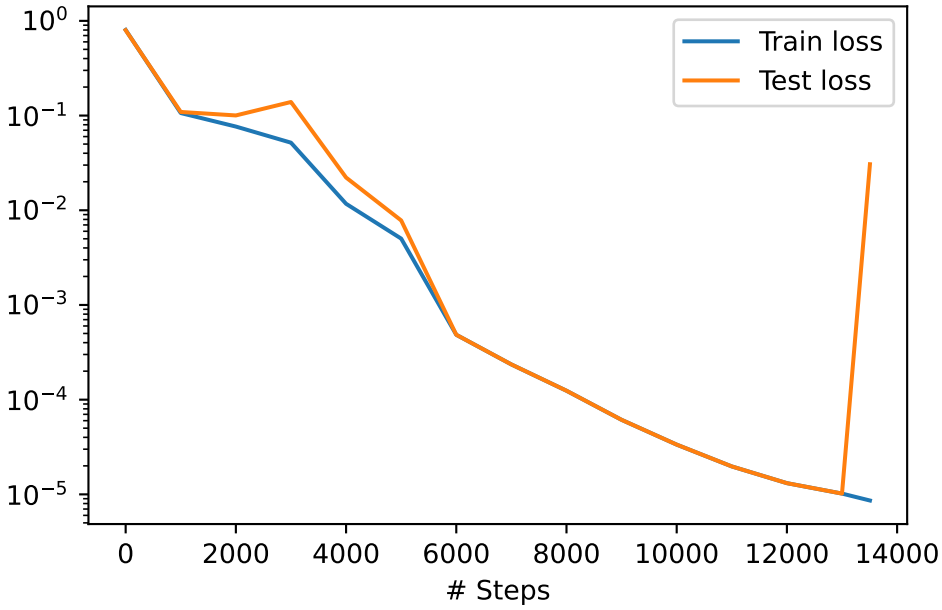
$$u(-1,t) = u(1,t) = 0, \quad u(x,0) = -\sin(\pi x)$$

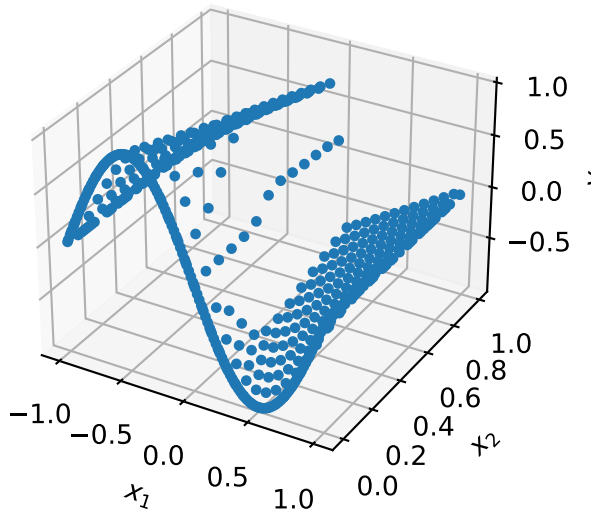
```
import deepxde as dde
import numpy as np


def gen_testdata():
    data = np.load("data/Burgers.npz")
    t, x, exact = data["t"], data["x"], data["usol"].T
    xx, tt = np.meshgrid(x, t)
    X = np.vstack((np.ravel(xx), np.ravel(tt))).T
    y = exact.flatten()[:, None]
    return X, y


def pde(x, y):
    dy_x = dde.grad.jacobian(y, x, i=0, j=0)
    dy_t = dde.grad.jacobian(y, x, i=0, j=1)
    dy_xx = dde.grad.hessian(y, x, i=0, j=0)
    return dy_t + y * dy_x - 0.01 / np.pi * dy_xx


geom = dde.geometry.Interval(-1, 1)
timedomain = dde.geometry.TimeDomain(0, 0.99)
geomtime = dde.geometry.GeometryXTime(geom, timedomain)

bc = dde.icbc.DirichletBC(geomtime, lambda x: 0, lambda _, on_boundary: on_boundary)
ic = dde.icbc.IC(
    geomtime, lambda x: -np.sin(np.pi * x[:, 0:1]), lambda _, on_initial: on_initial
)

data = dde.data.TimePDE(
    geomtime, pde, [bc, ic],
    num_domain=2540,
    num_boundary=80,
    num_initial=160,
    num_test=300
)
net = dde.nn.FNN([2] + [20] * 3 + [1], "tanh", "Glorot normal")
model = dde.Model(data, net)

model.compile("adam", lr=1e-3)
model.train(iterations=5000)
model.compile("L-BFGS")
```

```
losshistory, train_state = model.train()
dde.saveplot(losshistory, train_state, issave=False, isplot=True)

X, y_true = gen_testdata()
y_pred = model.predict(X)
f = model.predict(X, operator=pde)
print("Mean residual:", np.mean(np.absolute(f)))
print("L2 relative error:", dde.metrics.l2_relative_error(y_true, y_pred))
```

```
2025-06-25 15:55:47.861593: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda d
2025-06-25 15:55:47.908201: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda d
2025-06-25 15:55:47.909072: I tensorflow/core/platform/cpu_feature_guard.cc:182] This Tens
critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow wi
2025-06-25 15:55:48.764102: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-
TRT Warning: Could not find TensorRT
Using backend: tensorflow.compat.v1
Other supported backends: tensorflow, pytorch, jax, paddle.
paddle supports more examples now and is recommended.

WARNING:tensorflow:From /home/damian/.local/lib/python3.8/site-packages/tensorflow/python/
Instructions for updating:
non-resource variables are not supported in the long term
Compiling model...
Building feed-forward neural network...
'build' took 0.045695 s

/home/damian/.local/lib/python3.8/site-packages/deepxde/nn/tensorflow_compat_v1/fnn.py:116

`tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.ke

'compile' took 0.365102 s

Training model...

Step      Train loss                        Test loss                         Test metric
0         [2.03e-01, 5.38e-02, 5.46e-01]    [1.97e-01, 5.38e-02, 5.46e-
01]    []

2025-06-25 15:55:50.306423: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:375

1000      [4.10e-02, 1.67e-03, 6.36e-02]    [4.39e-02, 1.67e-03, 6.36e-
02]    []
```

```
2000      [3.14e-02, 1.44e-04, 4.50e-02]    [5.51e-02, 1.44e-04, 4.50e-
02]    []
3000      [2.06e-02, 7.27e-05, 3.10e-02]    [1.08e-01, 7.27e-05, 3.10e-
02]    []
4000      [6.45e-03, 8.74e-05, 5.17e-03]    [1.69e-02, 8.74e-05, 5.17e-
03]    []
5000      [3.00e-03, 2.28e-05, 2.00e-03]    [5.79e-03, 2.28e-05, 2.00e-
03]    []

Best model at step 5000:
  train loss: 5.02e-03
  test loss: 7.81e-03
  test metric: []

'train' took 36.179496 s

Compiling model...
'compile' took 0.178539 s

Training model...

Step      Train loss                        Test loss                         Test metric
5000      [3.00e-03, 2.28e-05, 2.00e-03]    [5.79e-03, 2.28e-05, 2.00e-
03]    []
6000      [2.76e-04, 1.85e-06, 2.06e-04]    [2.76e-04, 1.85e-06, 2.06e-04]
7000      [1.42e-04, 3.94e-07, 9.28e-05]    [1.42e-04, 3.94e-07, 9.28e-05]
8000      [8.45e-05, 4.89e-07, 3.93e-05]    [8.45e-05, 4.89e-07, 3.93e-05]
9000      [4.75e-05, 2.75e-07, 1.35e-05]    [4.75e-05, 2.75e-07, 1.35e-05]
10000     [2.71e-05, 1.15e-07, 6.31e-06]    [2.71e-05, 1.15e-07, 6.31e-06]
11000     [1.70e-05, 1.59e-07, 2.59e-06]    [1.70e-05, 1.59e-07, 2.59e-06]
12000     [1.13e-05, 7.90e-08, 1.77e-06]    [1.13e-05, 7.90e-08, 1.77e-06]
13000     [9.05e-06, 5.33e-08, 1.09e-06]    [9.05e-06, 5.33e-08, 1.09e-06]
INFO:tensorflow:Optimization terminated with:
  Message: CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
  Objective function value: 0.000009
  Number of iterations: 7837
  Number of functions evaluations: 8507
13507     [7.79e-06, 5.75e-08, 7.62e-07]    [3.06e-02, 5.75e-08, 7.62e-
07]    []

Best model at step 13507:
  train loss: 8.61e-06
  test loss: 3.06e-02
  test metric: []

'train' took 88.146945 s
```






```
Mean residual: 0.003893385
L2 relative error: 0.017751091603930687
```

6.2 Comparación con Redes Neuronales Tradicionales

Mientras que las redes neuronales tradicionales dependen exclusivamente de grandes volúmenes de datos etiquetados para su entrenamiento (Karniadakis et al. 2021), las PINNs integran el conocimiento físico como parte esencial de su arquitectura (Blechschmidt y Ernst

2021). Esta diferencia clave permite a las PINNs generar soluciones físicamente consistentes incluso con datos escasos, evitando el sobreajuste común en enfoques puramente basados en datos. Otra ventaja significativa de las PINNs es su naturaleza mesh-free, que contrasta con los métodos numéricos tradicionales como FEM o FDM que requieren discretización espacial. Sin embargo, el entrenamiento de PINNs puede ser más desafiante debido a la necesidad de optimizar múltiples objetivos simultáneamente (ajuste a datos y cumplimiento de leyes físicas) (Blechschmidt y Ernst 2021; Karniadakis et al. 2021).

7 DeepONet

DeepONet (Deep Operator Network) es una arquitectura de red neuronal profunda diseñada para aprender operadores no lineales que mapean funciones de entrada a funciones de salida. A diferencia de las redes convencionales que aprenden funciones escalares, DeepONet se enfoca en representar operadores completos, como soluciones de ecuaciones diferenciales, a partir de datos observados o simulaciones numérica (Lu, Jin, et al. 2021).

7.1 Arquitectura

La arquitectura de DeepONet está compuesta por dos redes principales: la red de *branch* y la red de *trunk*. La red *branch* procesa las evaluaciones discretas de la función de entrada (por ejemplo, condiciones iniciales o de frontera), mientras que la red *trunk* recibe como entrada los puntos del dominio donde se desea evaluar la función de salida. La salida final se obtiene mediante el producto punto de los vectores generados por ambas redes, lo que permite representar operadores complejos con alta generalización a nuevos datos (Lu, Jin, et al. 2021).

7.2 Ejemplo de resolución de un operador usando DeepONet

Se resolverá el operador

$$G : f \to u$$

para el problema unidemensional de Poisson:

$$u''(x) = f(x), \quad x \in [0, 1]$$

con la condición de frontera de Dirichlet

$$u(0) = u(1) = 0$$

el término f representa a una función continua arbitraria.

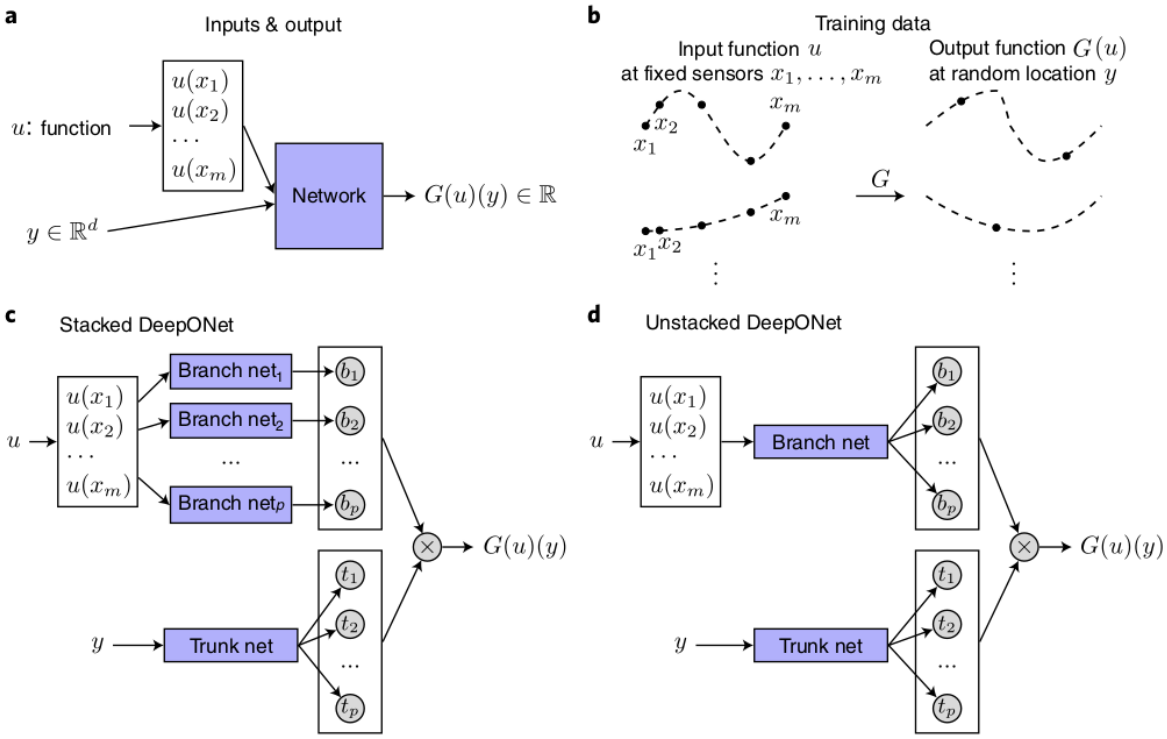


Figura 7.1: **Ilustraciones del planteamiento del problema y arquitectura DeepONet que conducen a una buena generalización. a)** Para que la red aprenda un operador $G : u \to G(u)$ se necesitan dos entradas $[u(x_1), u(x_2), ..., u(x_m)]$ e y. **b)** Ilustración de los datos de entrenamiento. Para cada función de entrada u, se requiere el mismo número de evaluaciones en los mismos sensores dispersos $x_1, x_2, ..., x_m$. Sin embargo, no se impone ninguna restricción sobre el número ni las ubicaciones para la evaluación de las funciones de salida. **c)** La DeepONet *stacked* se inspira en el **Teorema de aproximación universal para operadores** y consta de una red *Trunk* y p redes *Branch* apiladas. La red cuya construcción se inspira en el mismo teorema es una DeepONet *stacked* formada al elegir la red *Trunk* como una red de una capa de ancho p y cada red *Branch* como una red de una capa oculta de ancho n. **d)** La red DeepONet *unstacked* se inspira en el **Teorema general de aproximación universal para operadores** y consta de una red *Trunk* y una red *Branch*. Una red DeepONet *unstacked* puede considerarse como una red DeepONet *stacked*, en la que todas las redes *Branch* comparten el mismo conjunto de parámetros (Lu, Jin, et al. 2021).

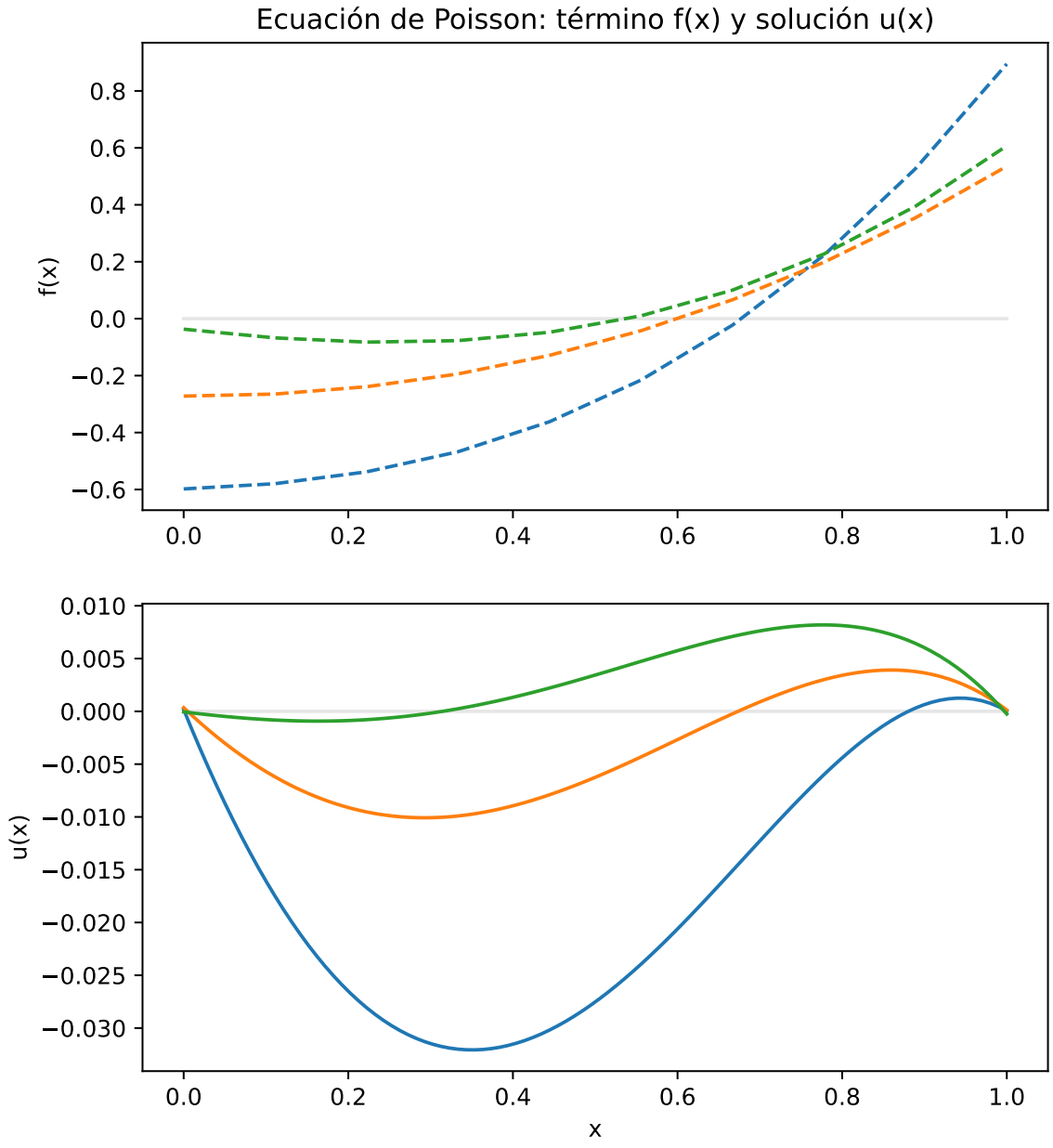
```python
import deepxde as dde
import matplotlib.pyplot as plt
import numpy as np


# Poisson equation: -u_xx = f
def equation(x, y, f):
    dy_xx = dde.grad.hessian(y, x)
    return -dy_xx - f

# Domain is interval [0, 1]
geom = dde.geometry.Interval(0, 1)

# Zero Dirichlet BC
def u_boundary(_):
    return 0

def boundary(_, on_boundary):
    return on_boundary

bc = dde.icbc.DirichletBC(geom, u_boundary, boundary)

# Define PDE
pde = dde.data.PDE(geom, equation, bc, num_domain=100, num_boundary=2)

# Function space for f(x) are polynomials
degree = 3
space = dde.data.PowerSeries(N=degree + 1)

# Choose evaluation points
num_eval_points = 10
evaluation_points = geom.uniform_points(num_eval_points, boundary=True)

# Define PDE operator
pde_op = dde.data.PDEOperatorCartesianProd(
    pde,
    space,
    evaluation_points,
    num_function=100,
    num_test=20
)

# Setup DeepONet
dim_x = 1
p = 32
```

```
net = dde.nn.DeepONetCartesianProd(
    [num_eval_points, 32, p],
    [dim_x, 32, p],
    activation="tanh",
    kernel_initializer="Glorot normal",
)

# Define and train model
model = dde.Model(pde_op, net)
dde.optimizers.set_LBFGS_options(maxiter=1000)
model.compile("L-BFGS")
model.train()

# Plot realisations of f(x)
n = 3
features = space.random(n)
fx = space.eval_batch(features, evaluation_points)

x = geom.uniform_points(100, boundary=True)
y = model.predict((fx, x))

# Setup figure
fig = plt.figure(figsize=(7, 8))
plt.subplot(2, 1, 1)
plt.title("Ecuación de Poisson: término f(x) y solución u(x)")
plt.ylabel("f(x)")
z = np.zeros_like(x)
plt.plot(x, z, "k-", alpha=0.1)

# Plot source term f(x)
for i in range(n):
    plt.plot(evaluation_points, fx[i], "--")

# Plot solution u(x)
plt.subplot(2, 1, 2)
plt.ylabel("u(x)")
plt.plot(x, z, "k-", alpha=0.1)
for i in range(n):
    plt.plot(x, y[i], "-")
plt.xlabel("x")

plt.show()
```

```
2025-06-25 15:58:00.806415: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda d
2025-06-25 15:58:00.864478: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda d
```

```
2025-06-25 15:58:00.865325: I tensorflow/core/platform/cpu_feature_guard.cc:182] This Tens
critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow wi
2025-06-25 15:58:01.689938: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-
TRT Warning: Could not find TensorRT
Using backend: tensorflow.compat.v1
Other supported backends: tensorflow, pytorch, jax, paddle.
paddle supports more examples now and is recommended.

WARNING:tensorflow:From /home/damian/.local/lib/python3.8/site-packages/tensorflow/python/
Instructions for updating:
non-resource variables are not supported in the long term
Compiling model...
Building DeepONetCartesianProd...
'build' took 0.053998 s

/home/damian/.local/lib/python3.8/site-packages/deepxde/nn/tensorflow_compat_v1/deeponet.p

`tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.ke

/home/damian/.local/lib/python3.8/site-packages/deepxde/nn/tensorflow_compat_v1/deeponet.p

`tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.ke

/home/damian/.local/lib/python3.8/site-packages/deepxde/nn/tensorflow_compat_v1/deeponet.p

`tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.ke

'compile' took 16.498563 s

2025-06-25 15:58:19.638744: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:375

Training model...

Step      Train loss              Test loss               Test metric
0         [3.90e-01, 4.61e-02]    [4.48e-01, 7.08e-02]    []
1000      [2.74e-06, 1.95e-07]    [2.79e-06, 1.76e-07]
INFO:tensorflow:Optimization terminated with:
  Message: STOP: TOTAL NO. of ITERATIONS REACHED LIMIT
  Objective function value: 0.000003
  Number of iterations: 1000
  Number of functions evaluations: 1061
1061      [2.48e-06, 1.48e-07]    [2.71e-06, 1.63e-07]    []
```

```
Best model at step 1061:
  train loss: 2.62e-06
  test loss: 2.88e-06
  test metric: []

'train' took 32.643402 s
```



7.3 Comparación con una PINN

En contraste con una red PINN convencional (Physics-Informed Neural Network), que resuelve una instancia específica de una ecuación diferencial para un conjunto dado de condiciones, DeepONet aprende el operador general que resuelve muchas instancias a la vez. Mientras que una PINN debe ser reentrenada para cada nuevo problema, DeepONet, una vez entrenado, puede predecir soluciones rápidamente para múltiples condiciones nuevas. Esto lo hace especialmente eficiente en aplicaciones donde se requiere realizar inferencias repetidas, como en control o diseño inverso (Kumar et al. 2024).

Part III

Ecuación del Bio-Calor

La ecuación del bio-calor, formulada por Pennes (1948), surgió de su estudio pionero *"Analysis of Tissue and Arterial Blood Temperatures in the Resting Human Forearm"*. Publicado en el *Journal of Applied Physiology*, este trabajo fue el primero en cuantificar la interacción entre la temperatura arterial y tisular en humanos. Pennes combinó principios termodinámicos con mediciones experimentales en el antebrazo, estableciendo un modelo matemático que relacionaba el flujo sanguíneo, la producción metabólica de calor y la conducción térmica en tejidos.

Experimento

Durante su estudio, Pennes diseñó un experimento riguroso para medir la temperatura interna del antebrazo humano. Utilizó termopares tipo "Y" insertados transversalmente en la musculatura del antebrazo mediante una aguja estéril, como se ilustra en la Figura 7.2. Esta configuración permitía capturar un perfil térmico a lo largo del eje transversal, minimizando interferencias derivadas del contacto externo o la conducción axial no deseada.

La técnica experimental buscó máxima precisión geométrica y térmica: los termopares eran fijados con tensión controlada mediante un sistema mecánico que aseguraba trayectorias rectas y repetibles dentro del tejido. La inserción se realizaba con anestesia tópica mínima y bajo condiciones ambientales estables, lo cual garantizaba que los gradientes de temperatura registrados fueran atribuibles principalmente al metabolismo local y al efecto del flujo sanguíneo arterial.

Trascendencia

El modelo de Pennes simplificó la complejidad biológica al asumir un flujo sanguíneo uniforme y una transferencia de calor proporcional a la diferencia entre la temperatura arterial y la tisular. Aunque posteriores investigaciones refinaron sus supuestos, su ecuación sigue siendo un referente en bioingeniería térmica. Su trabajo no solo sentó las bases para aplicaciones clínicas, como la hipertermia oncológica, sino que también inspiró avances en el estudio de la termorregulación humana y el diseño de dispositivos médicos.

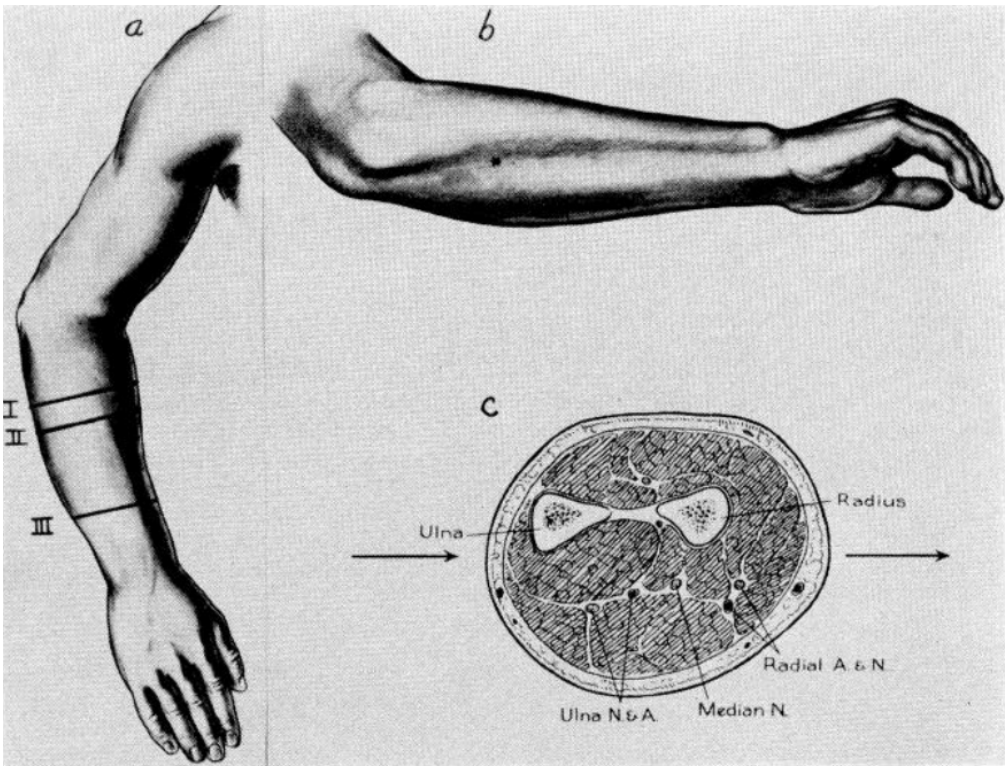


Figura 7.2: **a)** Posición del brazo derecho (vista superior). La linea horizontal II indica el nivel de la figura c). **b)** Posición del brazo derecho (vista lateral). **c)**Sección transversal anatómica del antebrazo en el nivel II (Pennes 1948).

8 Forma de la ecuación

La ecuación diferencial de bio-calor de Pennes (1948) modela la transferencia de calor en tejidos biológicos, integrando efectos de conducción, perfusión sanguínea y metabolismo. Su forma general es:

$$\rho c \frac{\partial T}{\partial t} = k_{\text{eff}} \frac{\partial^2 T}{\partial x^2} - \rho_b c_b \omega_b (T - T_a) + Q, \quad x \in \Omega,\, t \in [0, t_f] \tag{8.1}$$

Tabla 8.1: Tabla de nomenclaturas de la Ecuación 8.1.

Símbolo	Descripción	Unidades
T	Temperatura del tejido	°C
ρ	Densidad del tejido	$\frac{kg}{m^3}$
c	Calor específico del tejido	$\frac{J}{kg°C}$
k_{eff}	Conductividad térmica	$\frac{W}{m°C}$
ρ_b	Densidad de la sangre	$\frac{kg}{m^3}$
c_b	Calor específico de la sangre	$\frac{J}{kg°C}$
ω_b	Tasa de perfusión sanguínea	$1/s$
T_a	Temperatura arterial	°C
$Q = q_m + q_p$	Fuente de calor	$\frac{W}{m^3}$
q_m	Metabolismo	$\frac{W}{m^3}$
q_p	Externa	$\frac{W}{m^3}$

8.1 Versión reducida (adimensionalizada)

Mediante escalamiento:

$$T' = T - T_a \qquad \theta = \frac{T'}{T_M - T_a} \qquad X = \frac{x}{L_0} \qquad \tau = \frac{t}{t_f}$$

Tabla 8.2: Tabla de nomenclatura de las relaciones para escalamiento.

Símbolo	Descripción	Unidades
L_0	Longitud característica del dominio	m
t_f	Tiempo final de simulación	s

la Ecuación 8.1 se convierte en:

$$\partial_\tau \theta = a_1 \partial_{XX} \theta - a_2 W \theta + a_3 \tag{8.2}$$

Parámetros adimensionales:
- $a_1 = \frac{t_f}{\alpha L_0^2}$ (difusividad térmica $\alpha = \frac{k_{\text{eff}}}{\rho c}$).
- $a_2 = \frac{t_f c_b}{\rho c}$.
- $a_3 = \frac{t_f Q}{\rho c (T_M - T_a)}$.
- $W = \rho_b \omega_b$: Tasa volumétrica de perfusión ($\text{kg/m}^3 \cdot \text{s}$).

8.2 Condiciones de uso adecuadas

1. **Tejidos homogéneos**: Aproximación válida para regiones con propiedades térmicas uniformes.

2. **Perfusión sanguínea constante**: Supone flujo sanguíneo estable en el dominio.

3. **Aplicaciones clínicas**: Hipertermia, crioterapia y modelado térmico en terapias oncológicas.

9 Modelado del Bio-Calor en Hipertermia

La ecuación del bio-calor, formulada por Pennes (1948), permite modelar la distribución de temperatura en tejidos biológicos considerando el flujo sanguíneo, la conductividad térmica y fuentes internas o externas de calor. Su modelación es fundamental en la física médica para predecir el comportamiento térmico durante tratamientos como la hipertermia. Además, constituye una herramienta poderosa en el desarrollo de simulaciones computacionales aplicadas al diseño y control de terapias térmicas en tejidos vivos.

9.1 Aplicaciones recientes de la ecuación del bio-calor

Quintero et al. (2017) desarrollan un modelo basado en ecuaciones diferenciales parciales que integra la ecuación del bio-calor y la ley de Arrhenius para estimar el daño térmico en tratamientos de hipertermia superficial. Utilizan el método de líneas para resolver el sistema y plantean un problema de optimización que busca maximizar el daño al tejido tumoral minimizando el daño colateral. Su trabajo demuestra cómo la modelación matemática puede guiar estrategias terapéuticas más seguras y eficaces.

Dutta y Rangarajan (2018) presentan una solución analítica cerrada en dos dimensiones para la ecuación del bio-calor, considerando modelos de conducción tanto de tipo Fourier como no-Fourier. Mediante el uso de la transformada de Laplace, analizan la influencia de parámetros fisiológicos como la perfusión sanguínea y el tiempo de relajación térmica sobre la evolución de la temperatura. Su investigación aporta una base teórica sólida para comprender la propagación térmica en tejidos vivos durante la hipertermia terapéutica.

Yang et al. (2014) propone una estrategia numérica para resolver problemas inversos de conducción térmica en tejidos biológicos multicapa, utilizando un enfoque en diferencias finitas y el concepto de tiempo futuro. El estudio se enfoca en predecir las condiciones de frontera necesarias para generar distribuciones de temperatura deseadas. La implementación de este método permite estimar parámetros relevantes en tiempo real, lo cual resulta esencial para el control térmico preciso en procedimientos médicos no invasivos como la hipertermia localizada.

Part IV

Estudio de caso

Hipertermia como opción terapéutica complementaria en el manejo de cáncer

La Organización Mundial de la Salud (2022) en su página web define Cáncer como:

«Cáncer» es un término genérico utilizado para designar un amplio grupo de enfermedades que pueden afectar a cualquier parte del organismo; también se habla de «tumores malignos» o «neoplasias malignas». Una característica definitoria del cáncer es la multiplicación rápida de células anormales que se extienden más allá de sus límites habituales y pueden invadir partes adyacentes del cuerpo o propagarse a otros órganos, en un proceso que se denomina «metástasis». La extensión de las metástasis es la principal causa de muerte por la enfermedad.

Por su parte Instituto Nacional del Cáncer (2021) aporta lo siguiente:

Es posible que el cáncer comience en cualquier parte del cuerpo humano, formado por billones de células. En condiciones normales, las células humanas se forman y se multiplican (mediante un proceso que se llama división celular) para formar células nuevas a medida que el cuerpo las necesita. Cuando las células envejecen o se dañan, mueren y las células nuevas las reemplazan. A veces el proceso no sigue este orden y las células anormales o células dañadas se forman y se multiplican cuando no deberían. Estas células tal vez formen tumores, que son bultos de tejido. Los tumores son cancerosos (malignos) o no cancerosos (benignos).

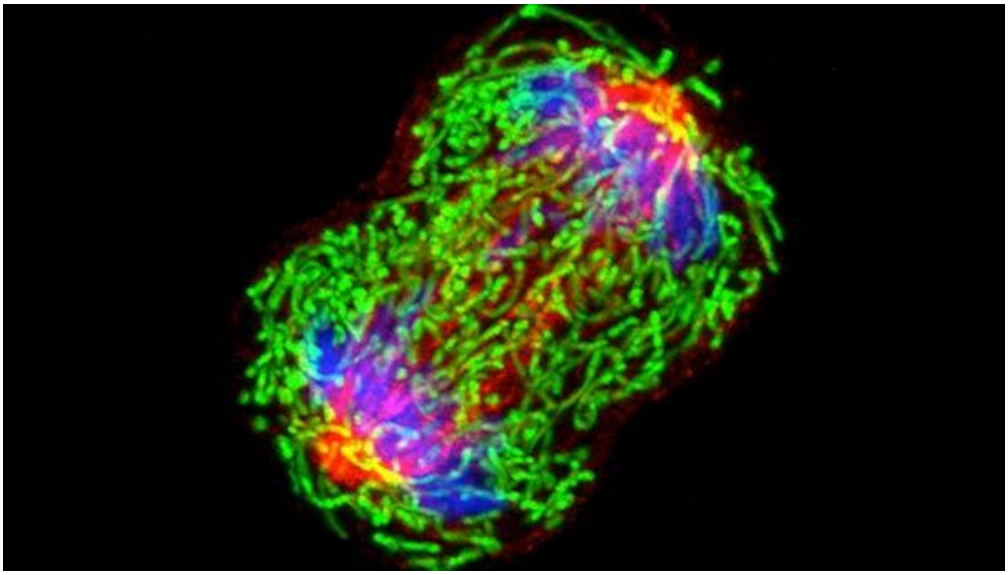


Figura 9.1: Una célula de cáncer de seno que se multiplica (Instituto Nacional del Cáncer 2021).

Ésta enfermedad es la principal causa de muerte a nivel mundial, solo en 2020 arrebató casi 10 millones de vidas y según datos de Organización Mundial de la Salud (2022) los cánceres más comunes en 2020 fueron:

- De mama (2.26 millones de casos)
- De pulmón (2.21 millones de casos)
- De colon (1.93 millones de casos)
- De próstata (1.41 millones de casos)
- De piel (distinto del melanoma) (1.20 millones de casos)
- Gástrico (1.09 millones de casos)

Es ante este panorama que distintos tratamientos surgen con el objetivo de erradicar la enfermedad siempre que se tenga una detección oportuna. Uno de dichos tratamientos es la hipertermia, según en el National Cancer Institute (2021) es un método que consiste en calentar el tejido corporal hasta los 39-45 °C para ayudar a erradicar células cancerígenas con pequeñas o nulas lesiones en el tejido sano. La hipertermia también es llamada terapia térmica o termoterapia.

Uno de los principales retos de este tratamiento es la creación de un modelo óptimo que se adecue al comportamiento de la transferencia de calor que se hace a los tejidos con el fin de dañar únicamente el área en el que se encuentran las célular cancerígenas, es por ello que los modelos de integencia artificial y más precisamente las PINN's (aqui irá una cita) surgen como posible solución a este reto.

El presente estudio utilizó como punto de partida el trabajo realizado por Alessio Borgi (2023) para modelar el calentamiento del tejido corporal usando la ecuación del Bio-Calor en dos dimensiones.

10 Metodología

En esta sección se describe el enfoque metodológico utilizado para evaluar la efectividad de una PINN utilizando una arquitectura DeepONet con el objetivo de resolver la ecuacion del Bio-Calor. El proceso metodológico se divide en las siguientes etapas:

10.1 Aportaciones del modelo

Ya que se parte del trabajo de Alessio Borgi (2023), se examinó que dos de los puntos a mejorar de la red neuronal que plantearon son:

1. Desarrollar nuevas arquitecturas para la red neuronal y explorar nuevas configuraciones
2. Combinar las fortalezas de los algoritmos de optimización Adam y L-BFGS para mejorar la velocidad de convergencia y la precisión

Tenindo los anteriores puntos en cuenta, se procedió a abordarlos e implementarlos dentro del diseño del modelo.

10.2 Diseño del modelo

El lenguaje seleccionado fué Python, a su vez el código se basa enteramente en la librería Deepxde creada por Lu, Meng, et al. (2021) la cual está directamente enfocada a resolver ecuaciones diferenciales, se usó además como backend *tensorflow_compat_v1* siendo su elección debida únicamente a la familiarización previa que se tenía con ella. Finalmente el entorno donde se programó y optimizó el código fué en *Google Colab* ya que la potencia de cómputo ofrecida por la plataforma era necesaria para ejecutar el modelo.

10.3 Implementación del modelo

La implementación del modelo se llevó a cabo en dos etapas clave: **(1) el desarrollo del código base para resolver la ecuación del Bio-Calor mediante DeepXDE**, y **(2) la optimización sistemática de los hiperparámetros**. Para esta última, se siguieron las recomendaciones del estudio de Alessio Borgi (2023), adaptadas a las particularidades del problema. Se ajustaron parámetros críticos como el número de épocas de entrenamiento (*iterations*), el ratio de aprendizaje (*learning rate*) así como un decaimiento en el mismo dependiente de la itearación actual (*decay*), la función de activación (*elu*) y el esquema de

inicialización de pesos (*Glorot normal*). Estos ajustes se realizaron mediante un proceso iterativo que buscaba minimizar la función de pérdida mientras se mantenía un tiempo de entrenamiento computacionalmente viable.

10.4 Evaluación del modelo

Para validar el desempeño del modelo propuesto, se realizó una evaluación exhaustiva utilizando un *conjunto de datos independiente*, el cual no fue empleado durante las fases de entrenamiento o ajuste de hiperparámetros. Este enfoque garantiza una medición objetiva de la capacidad de generalización del modelo ante datos no vistos.

Las predicciones generadas por el modelo fueron analizadas mediante visualizaciones espaciotemporales, las cuales permiten comparar cualitativamente el comportamiento de las soluciones pronosticadas frente a los rangos físicos y temporales definidos en el problema. En particular, se generaron gráficas de superficies 3D que muestran la evolución de las variables de interés a lo largo del dominio espacial y temporal bajo estudio. Adicionalmente, se incluyeron representaciones de cortes transversales y series temporales en puntos estratégicos para facilitar la interpretación de los resultados.

Cabe destacar que este análisis preliminar se centró en examinar la coherencia física y la estabilidad numérica de las predicciones. Para la evaluación cuantitativa del modelo, se implementó una comparación directa con las soluciones obtenidas mediante el método numérico de **Crank-Nicolson**, resuelto en *Julia* utilizando la librería *DifferentialEquations.jl*.

10.5 Comparación de resultados

Para evaluar el desempeño predictivo del modelo propuesto, se realizaron dos tipos de comparaciones:

1. Una evaluación cualitativa basada en visualizaciones.
2. Un análisis cuantitativo mediante métricas de error estandarizadas.

En primer lugar, se llevó a cabo una comparación visual con los resultados reportados en el trabajo de Alessio Borgi (2023), dado que dicho estudio no incluye datos numéricos tabulados, sino únicamente representaciones gráficas de las soluciones. Esta comparación permitió identificar coincidencias y discrepancias en el comportamiento espaciotemporal de las variables de interés, destacando las fortalezas del modelo propuesto en términos de estabilidad numérica.

En segundo lugar, para una evaluación cuantitativa rigurosa, se compararon las predicciones del modelo con soluciones de referencia generadas mediante el método de Crank-Nicolson, implementado en Julia utilizando la librería DifferentialEquations.jl. La comparación se realizó sobre una malla uniforme de 26×26 puntos en el cuadrado de $[0, 1] \times [0, 1]$, calculando para cada instante de tiempo relevante las siguientes métricas:

- Error Absoluto Medio (MAE)
- Error Absoluto Máximo (MaxAE)
- Error L2 (norma euclidiana)

Estos criterios permitieron cuantificar no solo la precisión global del modelo, sino también sus desviaciones locales más significativas, particularmente en regiones con alta variabilidad espacial. Los resultados detallados de este análisis, junto con una discusión sobre la eficiencia computacional relativa entre ambos métodos, se presentan en la Tabla 13.1.

10.6 Análisis y conclusión

Finalmente, se realizó un análisis detallado de los resultados obtenidos para extraer conclusiones significativas. Se proporcionaron recomendaciones basadas en los hallazgos del estudio, lo que permitió establecer un marco para interpretaciones analíticas profundas y recomendaciones bien fundamentadas en la sección de conclusiones del estudio.

Este enfoque metodológico proporcionó una base sólida para los resultados obtenidos, asegurando la integridad y la calidad del análisis realizado en el estudio.

11 Predicciones del método numérico

Para validar los resultados del modelo propuesto, se implementó el método de Crank-Nicolson en Julia utilizando la librería *DifferentialEquations.jl*. El método se resolvió sobre una malla refinada de 51×51 puntos para garantizar alta precisión en la solución numérica, calculando las predicciones en los tiempos clave: t = [0.0, 0.25, 0.50, 0.75, 1.0]. En la Figura 11.1 se muestran las cinco gráficas generadas, las cuales ilustran la evolución temporal de la solución en el dominio de estudio.

```
using DifferentialEquations, LinearAlgebra
using DataFrames, CSV

# --- PARÁMETROS FÍSICOS Y DIMENSIONALES ----------------------------------
 , c   = 1050.0, 3639.0                          # densidad, calor específico
k_eff = 5.0                                      # conductividad
t_f   = 1800.0                                   # tiempo final
L     = 0.05                                     # longitud del dominio
c_b   = 3825.0                                   # coef. perfusión
Q     = 0.0                                      # fuente térmica
T_M, T_a = 45.0, 37.0                            # temp máxima, temp ambiente

# --- COEFICIENTES ADIMENSIONALES -----------------------------------------
  =   * c / k_eff
a  = t_f / (  * L^2)
a  = t_f * c_b / (  * c)
a  = (t_f * Q) / (  * c * (T_M - T_a))

# --- MALLA ESPACIAL ------------------------------------------------------
Nx, Ny = 51, 51
dx, dy = 1.0 / (Nx - 1), 1.0 / (Ny - 1)
x = range(0, 1, length=Nx)
y = range(0, 1, length=Ny)
N = Nx * Ny  # total de puntos

# --- CONDICIÓN INICIAL ---------------------------------------------------
u0 = zeros(N)

# --- SISTEMA DE EDOs DEL PDE ---------------------------------------------
function f!(du, u, _,  )
    U = reshape(u, Nx, Ny)
```

```
    D = similar(U)
    @inbounds for i in 1:Nx, j in 1:Ny
        # Derivadas segunda en x
        d2x = if i == 1
            (U[2, j] - 0) / dx^2
        elseif i == Nx
            U_ghost = U[Nx, j] +   * dx
            (U_ghost - 2U[Nx, j] + U[Nx-1, j]) / dx^2
        else
            (U[i+1, j] - 2U[i, j] + U[i-1, j]) / dx^2
        end

        # Derivadas segunda en y
        d2y = if j == 1
            (U[i, 2] - U[i, 1]) / dy^2
        elseif j == Ny
            (U[i, Ny-1] - U[i, Ny]) / dy^2
        else
            (U[i, j+1] - 2U[i, j] + U[i, j-1]) / dy^2
        end

        D[i, j] = (d2x + d2y - a  * U[i, j] + a ) / a
    end
    du .= vec(D)
end

# --- RESOLVER PDE ----------------------------------------------------
 span = (0.0, 1.0)
prob = ODEProblem(f!, u0,  span)
taus = [0.0, 0.25, 0.5, 0.75, 1.0]
sol = solve(prob, Trapezoid(), dt=8e-4, saveat=taus)

# --- PROCESAR SOLUCIÓN EN GRILLA REDUCIDA ----------------------------
idxs = 1:2:Nx  # índices para submuestreo
npts = length(idxs)^2 * length(taus)

# Preasignar vectores para crear el DataFrame
times = Float64[]
Xs = Float64[]
Ys = Float64[]
Thetas = Float64[]

for (k,  ) in enumerate(taus)
    θ = reshape(sol( ), Nx, Ny)
    for j in idxs, i in idxs
```

```
        push!(times,  )
        push!(Xs, x[i])
        push!(Ys, y[j])
        push!(Thetas, θ[i, j])
    end
end

df = DataFrame(time=times, X=Xs, Y=Ys, Theta=Thetas)

# --- GUARDAR CSV ------------------------------------------------------
ruta = "data"
CSV.write(joinpath(ruta, "crank_nick.csv"), df)
```

```
using DataFrames, CSV, Plots, Statistics
pyplot()

# --- OBTENER VALORES ÚNICOS Y ORDENADOS DE X, Y, TIME ----------------------
x_vals = sort(unique(df.X))
y_vals = sort(unique(df.Y))
times = sort(unique(df.time))  # tiempos

Nx, Ny = length(x_vals), length(y_vals)

# --- RECONSTRUIR MATRICES 2D DE THETA PARA CADA TIEMPO ---------------------
solutions = []

for t in times
    dft = filter(:time => ==(t), df)

    # Crear matriz vacía
    θ = fill(NaN, Nx, Ny)

    # Llenar la matriz con los valores correspondientes
    for row in eachrow(dft)
        ix = findfirst(==(row.X), x_vals)
        iy = findfirst(==(row.Y), y_vals)
        θ[ix, iy] = row.Theta
    end

    push!(solutions, θ)
end

# --- DETERMINAR ESCALA GLOBAL DE COLORES -----------------------------------
zmin = minimum([minimum(u) for u in solutions])
zmax = maximum([maximum(u) for u in solutions])
```

```
# --- GRAFICAR EN LAYOUT 3x2 ------------------------------------------
p = plot(layout = (3, 2), size = (800, 900))

for (i, (t, θ)) in enumerate(zip(times, solutions))
    surface!(
        p, y_vals, x_vals, θ;
        camera = (45,30),
        xlabel = "Y",
        ylabel = "X",
        zlabel = "T   ",
        title = "t = $(t)",
        subplot = i,
        c = :thermal,
        clim = (zmin, zmax),
        legend = false
    )
end

# Eliminar ejes y contenido del subplot 6
plot!(p[6], framestyle = :none,
            grid = false,
            xticks = false,
            yticks = false)
close("all")

display(p)
```

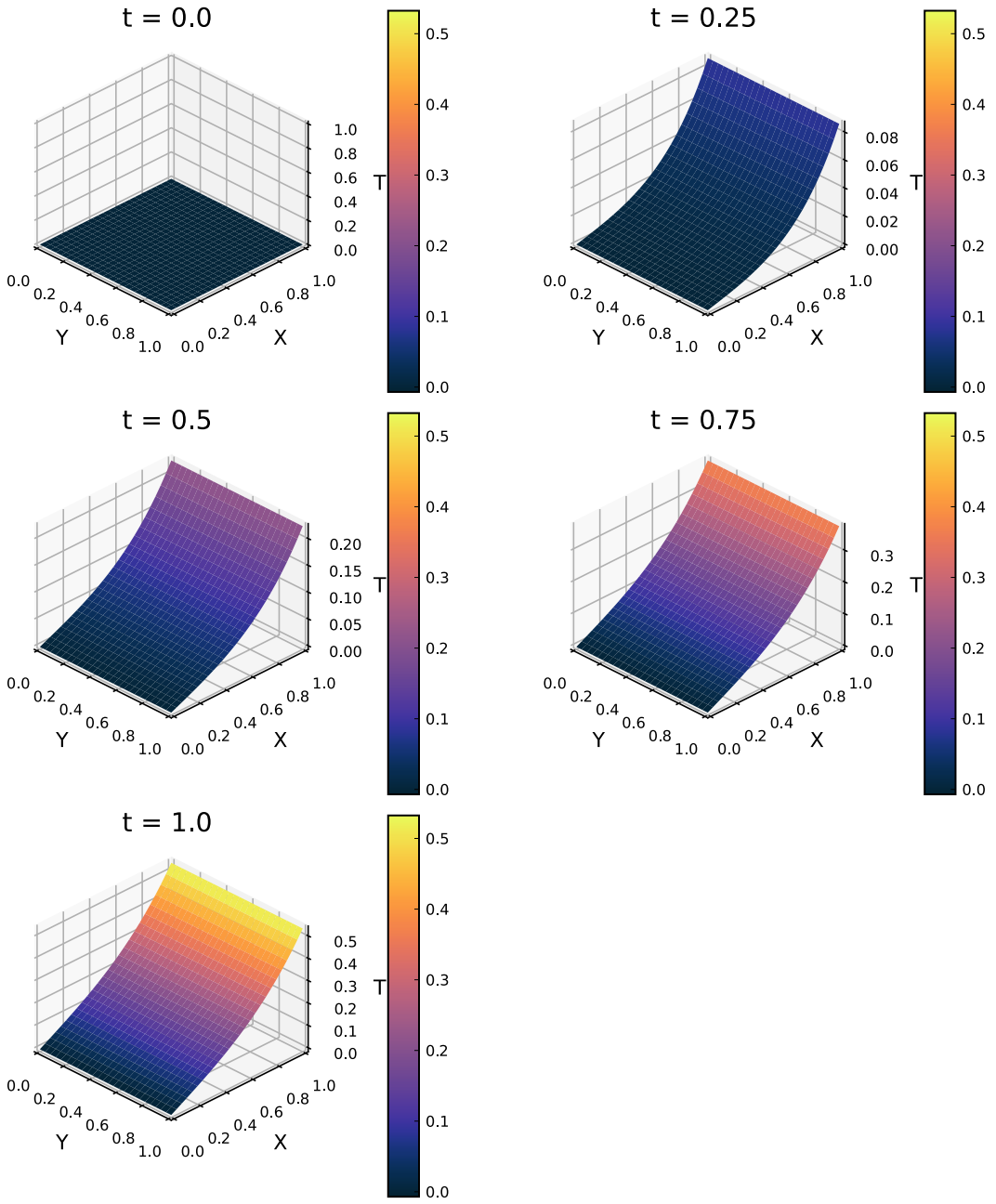


Figura 11.1: Resultados obtenidos mediante el método numérico de Crank Nickolson.

11.1 Guardado de datos

Con el objetivo de optimizar el proceso de comparación cuantitativa con el modelo de redes neuronales, se exportó un subconjunto representativo de los resultados. Aunque la simulación original utilizó una malla de 51×51 puntos, se almacenaron únicamente los valores

correspondientes a una grilla de 26×26 puntos. Esta decisión se basó en:

1. Suficiencia estadística: La densidad de puntos conserva los patrones espaciales críticos.
2. Eficiencia computacional: Reduce el tamaño del archivo sin perder información relevante.

Los datos se guardaron en un archivo CSV estructurado con las siguientes columnas:

- Coordenadas espacio-temporales (t, x, y) para cada punto de la grilla 26×26.
- Valores de la solución en los tiempos de interés.

Este archivo permitió calcular de manera estandarizada las métricas de error (MAE, MaxAE, error L2) en la sección de comparación de resultados.

12 Métricas del modelo

Conforme se ha referido previamente, el desarrollo del modelo predictivo se realizó utilizando el framework DeepXDE (versión 1.10.1) con backend de TensorFlow 1.x (configurado mediante tensorflow.compat.v1). Para asegurar reproducibilidad, se fijó la semilla aleatoria en 123 a nivel de DeepXDE, TensorFlow y NumPy. La red neuronal se implementó como un DeepONetCartesianProd con la siguiente estructura especializada:

- Rama (*branch*)
  - Capa de entrada: $(\text{num\_sensors} + 1)^2 = 25$ neuronas.
  - 3 capas ocultas de 20 neuronas cada una.
- Tronco (*trunk*)
  - Capa de entrada: 3 neuronas (coordenadas espaciotemporales x, y, t).
  - Misma configuración de capas ocultas que la rama.
- Hiperparámetros clave
  - Función de activación: ELU (Exponential Linear Unit).
  - Inicialización de pesos: Glorot normal.
  - Optimizador: Adam con tasa de aprendizaje inicial de $2{\times}10^{-3}$ y decaimiento exponencial (decay_rate=0.05 cada 1000 pasos).

```
import deepxde as dde
import numpy as np
import tensorflow as tf

# ------------------------------------------------------------------------------
# Constants and Parameters
# ------------------------------------------------------------------------------

# Backend and seed
dde.backend.set_default_backend("tensorflow.compat.v1")
dde.config.set_random_seed(123)

# Physical parameters
p = 1050
c = 3639
keff = 5
tf = 1800
```

```
L0 = 0.05
cb = 3825
Q = 0
TM = 45
Ta = 37
alpha = p * c / keff

# Dimensionless coefficients
a1 = tf / (alpha * L0**2)
a2 = tf * cb / (p * c)
a3 = (tf * Q) / (p * c * (TM - Ta))

# Domain boundaries
x_initial, x_boundary = 0.0, 1.0
y_initial, y_boundary = 0.0, 1.0
t_initial, t_final = 0.0, 1.0

# Dataset configuration
pts_dom = 10
pts_bc = 25
pts_ic = 55
num_test = 25

# Sensor grid and function space
num_sensors = 4
size_cov_matrix = 40

# Network architecture
width_net = 20
len_net = 3
AF = "elu"
k_initializer = "Glorot normal"

# Training parameters
num_iterations = 5000
learning_rate = 2e-3
decay_rate = 0.05
decay_steps = 1000

# ------------------------------------------------------------------------------
# Geometry and Time Domain
# ------------------------------------------------------------------------------

spatial_domain = dde.geometry.Rectangle([x_initial, y_initial],
                                        [x_boundary, y_boundary])
```

```
time_domain = dde.geometry.TimeDomain(t_initial, t_final)
geomtime = dde.geometry.GeometryXTime(spatial_domain, time_domain)

# ------------------------------------------------------------------------------
# PDE and Conditions
# ------------------------------------------------------------------------------

def initial_condition(X):
    return 0

def heat_equation(func, u, coords):
    u_t = dde.grad.jacobian(u, func, i=0, j=2)
    u_xx = dde.grad.hessian(u, func, i=0, j=0)
    u_yy = dde.grad.hessian(u, func, i=1, j=1)
    return a1 * u_t - (u_xx + u_yy) + a2 * u

def zero_value(X):
    return 0

def time_value(X):
    return X[:, 2]

def is_on_vertex(x):
    vertices = np.array([[x_initial, y_initial],
                         [x_boundary, y_initial],
                         [x_initial, y_boundary],
                         [x_boundary, y_boundary]])
    return any(np.allclose(x, v) for v in vertices)

def is_initial(X, on_initial):
    return on_initial and np.isclose(X[2], t_initial)

def left_boundary(X, on_boundary):
    spatial = X[0:2]
    t = X[2]
    return (
        on_boundary
        and np.isclose(spatial[0], x_initial)
        and not np.isclose(t, t_initial)
        and not is_on_vertex(spatial)
    )

def right_boundary(X, on_boundary):
    spatial = X[0:2]
    t = X[2]
```

```
    return (
        on_boundary
        and np.isclose(spatial[0], x_boundary)
        and not np.isclose(t, t_initial)
        and not is_on_vertex(spatial)
    )

def up_low_boundary(X, on_boundary):
    spatial = X[0:2]
    t = X[2]
    return (on_boundary
    and (np.isclose(spatial[1], y_initial)
    or np.isclose(spatial[1], y_boundary))
    and not np.isclose(t, t_initial)
    and not is_on_vertex(spatial)
    )

# Initial and boundary conditions
ic = dde.icbc.IC(geomtime, initial_condition, is_initial)
left_bc = dde.icbc.DirichletBC(geomtime,
                               zero_value, left_boundary)
right_bc = dde.icbc.NeumannBC(geomtime,
                               time_value, right_boundary)
up_low_bc = dde.icbc.NeumannBC(geomtime,
                               zero_value, up_low_boundary)

# ------------------------------------------------------------------------------
# Dataset Construction
# ------------------------------------------------------------------------------

pde_data = dde.data.TimePDE(
    geomtime,
    heat_equation,
    [ic, left_bc, right_bc, up_low_bc],
    num_domain=pts_dom,
    num_boundary=pts_bc,
    num_initial=pts_ic
)

# ------------------------------------------------------------------------------
# Sensor Points and Function Space
# ------------------------------------------------------------------------------

side = np.linspace(x_initial, x_boundary, num_sensors + 1)
x, y = np.meshgrid(side, side, indexing='xy')
```

```
sensor_pts = np.stack([x.ravel(), y.ravel()], axis=1)

fs = dde.data.function_spaces.GRF2D(N=size_cov_matrix,
                                    interp="linear")

data = dde.data.PDEOperatorCartesianProd(
    pde_data,
    fs,
    sensor_pts,
    num_function=(num_sensors + 1)**2,
    function_variables=[0, 1],
    num_test=num_test
)

# ------------------------------------------------------------------------------
# Network Definition
# ------------------------------------------------------------------------------

branch_layers = [(num_sensors + 1)**2] + len_net * [width_net]
trunk_layers = [3] + len_net * [width_net]

net = dde.nn.DeepONetCartesianProd(
    branch_layers,
    trunk_layers,
    activation=AF,
    kernel_initializer=k_initializer
)

# ------------------------------------------------------------------------------
# Model Compilation and Training
# ------------------------------------------------------------------------------

model = dde.Model(data, net)
model.compile("adam", lr=learning_rate, decay=("inverse time", decay_steps, decay_rate))
losshistory, train_state = model.train(iterations=num_iterations)

# Fine-tuning with LBFGS optimizer
model.compile("L-BFGS")
losshistory, train_state = model.train()
```

```
2025-06-25 15:59:06.992646: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda d
2025-06-25 15:59:07.045297: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda d
2025-06-25 15:59:07.046419: I tensorflow/core/platform/cpu_feature_guard.cc:182] This Tens
critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow wi
```

```
2025-06-25 15:59:07.911547: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-
TRT Warning: Could not find TensorRT
Using backend: tensorflow.compat.v1
Other supported backends: tensorflow, pytorch, jax, paddle.
paddle supports more examples now and is recommended.

WARNING:tensorflow:From /home/damian/.local/lib/python3.8/site-packages/tensorflow/python/
Instructions for updating:
non-resource variables are not supported in the long term
Setting the default backend to "tensorflow.compat.v1". You can change it in the ~/.deepxde
Compiling model...
Building DeepONetCartesianProd...
'build' took 0.088748 s

/home/damian/.local/lib/python3.8/site-packages/deepxde/nn/tensorflow_compat_v1/deeponet.p

`tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.ke

/home/damian/.local/lib/python3.8/site-packages/deepxde/nn/tensorflow_compat_v1/deeponet.p

`tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.ke

/home/damian/.local/lib/python3.8/site-packages/deepxde/nn/tensorflow_compat_v1/deeponet.p

`tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.ke

'compile' took 14.967762 s

2025-06-25 15:59:24.600128: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:375

Training model...

Step      Train loss                                              Test loss
0         [2.13e+00, 2.30e-01, 2.85e-01, 6.87e-01, 6.91e-01]    [3.30e+00, 2.78e-
01, 4.39e-01, 1.03e+00, 6.38e-01]    []
1000      [3.11e-03, 2.58e-03, 5.22e-04, 6.89e-02, 1.74e-04]    [1.34e-
02, 2.74e-03, 2.06e-03, 7.10e-02, 3.39e-04]    []
2000      [3.09e-03, 1.09e-03, 1.89e-04, 6.85e-02, 5.77e-05]    [1.14e-
02, 1.18e-03, 9.29e-04, 6.94e-02, 1.10e-04]    []
3000      [9.30e-04, 5.82e-04, 1.27e-05, 6.83e-02, 2.02e-05]    [7.56e-
03, 7.20e-04, 5.66e-04, 6.89e-02, 8.23e-05]    []
4000      [3.17e-03, 4.59e-04, 2.08e-04, 6.82e-02, 1.82e-05]    [6.66e-
03, 6.02e-04, 5.99e-04, 6.86e-02, 5.63e-05]    []
```

```
5000      [3.66e-04, 3.11e-04, 3.68e-05, 6.82e-02, 7.74e-06]    [6.75e-
03, 3.82e-04, 6.64e-04, 6.86e-02, 1.75e-05]    []

Best model at step 5000:
  train loss: 6.90e-02
  test loss: 7.65e-02
  test metric: []

'train' took 44.766442 s

Compiling model...
'compile' took 33.439356 s

Training model...

Step      Train loss                                              Test loss
5000      [3.66e-04, 3.11e-04, 3.68e-05, 6.82e-02, 7.74e-06]    [6.75e-
03, 3.82e-04, 6.64e-04, 6.86e-02, 1.75e-05]    []
INFO:tensorflow:Optimization terminated with:
  Message: CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
  Objective function value: 0.068749
  Number of iterations: 8
  Number of functions evaluations: 44
5044      [1.97e-04, 3.33e-04, 1.37e-05, 6.82e-02, 7.14e-06]    [5.67e-
03, 3.98e-04, 5.57e-04, 6.86e-02, 1.81e-05]    []

Best model at step 5044:
  train loss: 6.87e-02
  test loss: 7.52e-02
  test metric: []

'train' took 10.900959 s
```

12.1 Gráficas de pérdida del modelo

El proceso de entrenamiento del modelo se monitoreó mediante el seguimiento detallado de cinco componentes de pérdida, cada una asociada a restricciones físicas y matemáticas específicas del problema:

1. Pérdida residual de la EDP
    - Función: Mide el cumplimiento de la ecuación de Bio-Calor en el dominio interior.
    - Importancia: Garantiza que la solución aprendida satisfaga la física subyacente.
    - Comportamiento esperado: Debe converger a valores cercanos a cero (típicamente $<$ 1e-3).

2. Pérdida de condición inicial
    - Función: Controla la precisión en t=0.
    - Importancia: Asegura coherencia con el estado inicial del sistema.
    - Patrón típico: Suele ser la primera en converger por su carácter puntual.
3. Pérdida de frontera izquierda (Dirichlet)
    - Función: Evalúa el cumplimiento de condiciones de valor prescrito.
    - Relevancia: Mantiene valores fijos en bordes específicos.
    - Convergencia: Normalmente rápida por ser restrictiva.
4. Pérdida de frontera derecha (Neumann)
    - Función: Verifica gradientes normales en esta frontera
    - Dificultad característica: Puede mostrar oscilaciones iniciales
5. Pérdida de fronteras superior/inferior (Neumann)
    - Función: Controla condiciones de flujo en estos bordes
    - Complejidad: En problemas 2D/3D suele ser la última en estabilizarse

12.1.1 Perdida para el conjunto de entrenamiento

```
import plotly.graph_objects as go

# Nombres de las componentes del loss
loss_labels = [
    "PDE residual loss",
    "Initial-condition loss",
    "Left-boundary (Dirichlet) loss",
    "Right-boundary (Neumann) loss",
    "Top/Bottom-boundary (Neumann) loss"
]

# Extraer pasos y pérdida de entrenamiento
steps = losshistory.steps
train_loss = np.array(losshistory.loss_train)

# Crear figura
fig_train = go.Figure()

for i in range(train_loss.shape[1]):
    fig_train.add_trace(go.Scatter(
        x=steps,
        y=train_loss[:, i],
        mode='lines',
```

```
        name=loss_labels[i]
    ))

fig_train.update_layout(
    title="Training Loss history",
    xaxis=dict(title="Iteration", tickformat=".1e"),
    yaxis=dict(title="Loss", type="log", tickformat=".1e"),
    template="plotly_white",
    legend=dict(x=0.99, y=0.99),
    font=dict(size=14)
)
```

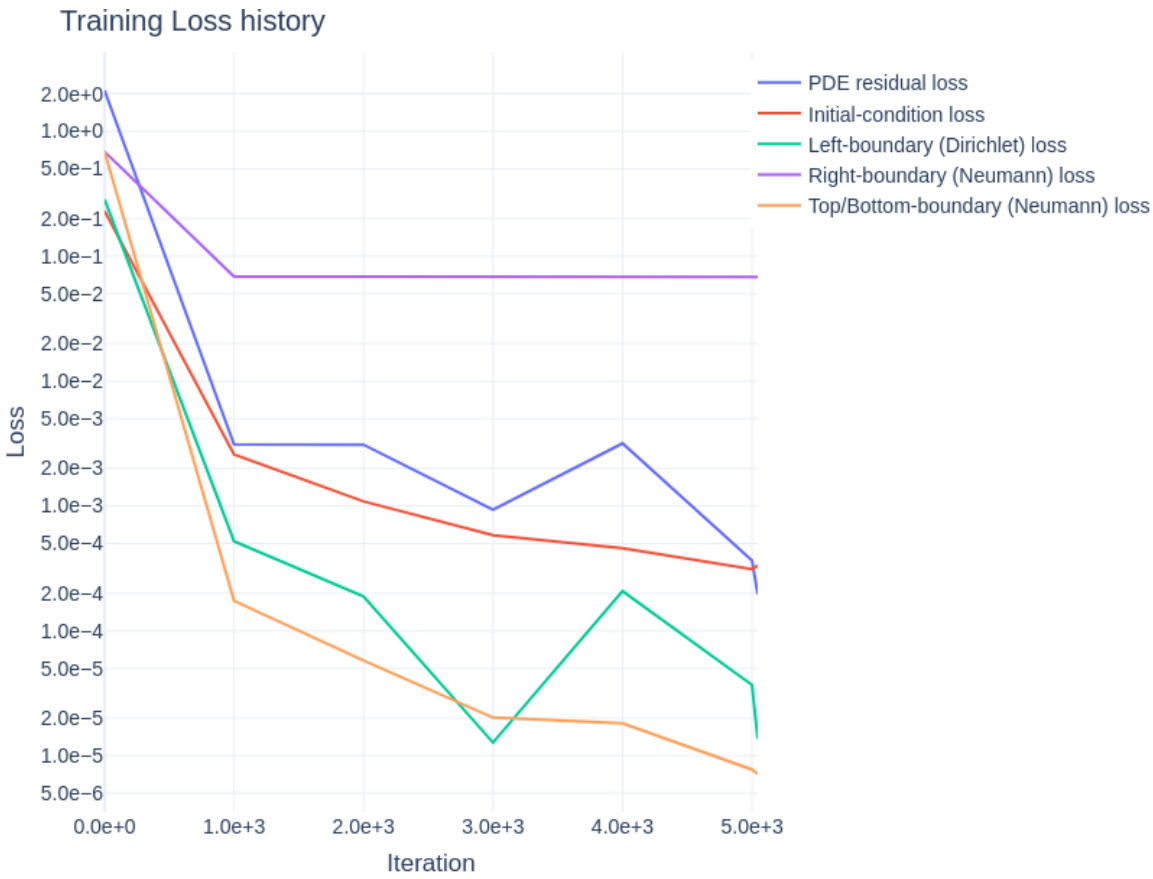


Figura 12.1: Gráfica de la perdida en el entrenamiento.

12.1.2 Pérdida para el conjunto de prueba

```
import plotly.graph_objects as go
```

```
# Nombres de las componentes del loss
loss_labels = [
    "PDE residual loss",
    "Initial-condition loss",
    "Left-boundary (Dirichlet) loss",
    "Right-boundary (Neumann) loss",
    "Top/Bottom-boundary (Neumann) loss"
]

# Extraer pasos y pérdida de entrenamiento
steps = losshistory.steps
test_loss = np.array(losshistory.loss_test)

# Crear figura
fig_test = go.Figure()

for i in range(test_loss.shape[1]):
    fig_test.add_trace(go.Scatter(
        x=steps,
        y=test_loss[:, i],
        mode='lines',
        name=loss_labels[i]
    ))

fig_test.update_layout(
    title="Test Loss history",
    xaxis=dict(title="Iteration", tickformat=".1e"),
    yaxis=dict(title="Loss", type="log", tickformat=".1e"),
    template="plotly_white",
    legend=dict(x=0.99, y=0.99),
    font=dict(size=14)
)
```

12.2 Guardado de datos

Para permitir la comparación cuantitativa con el método de Crank-Nicolson y facilitar la generación de visualizaciones consistentes, se exportaron las predicciones del modelo neuronal en formato CSV. El proceso consistió en:

1. Generación de la malla de evaluación:
    - Dominio espacial: Cuadrado unitario $[0,1] \times [0,1]$.
    - Discretización: 26 segmentos equiespaciados en cada eje (x, y).
    - Puntos totales: 676 (26×26)

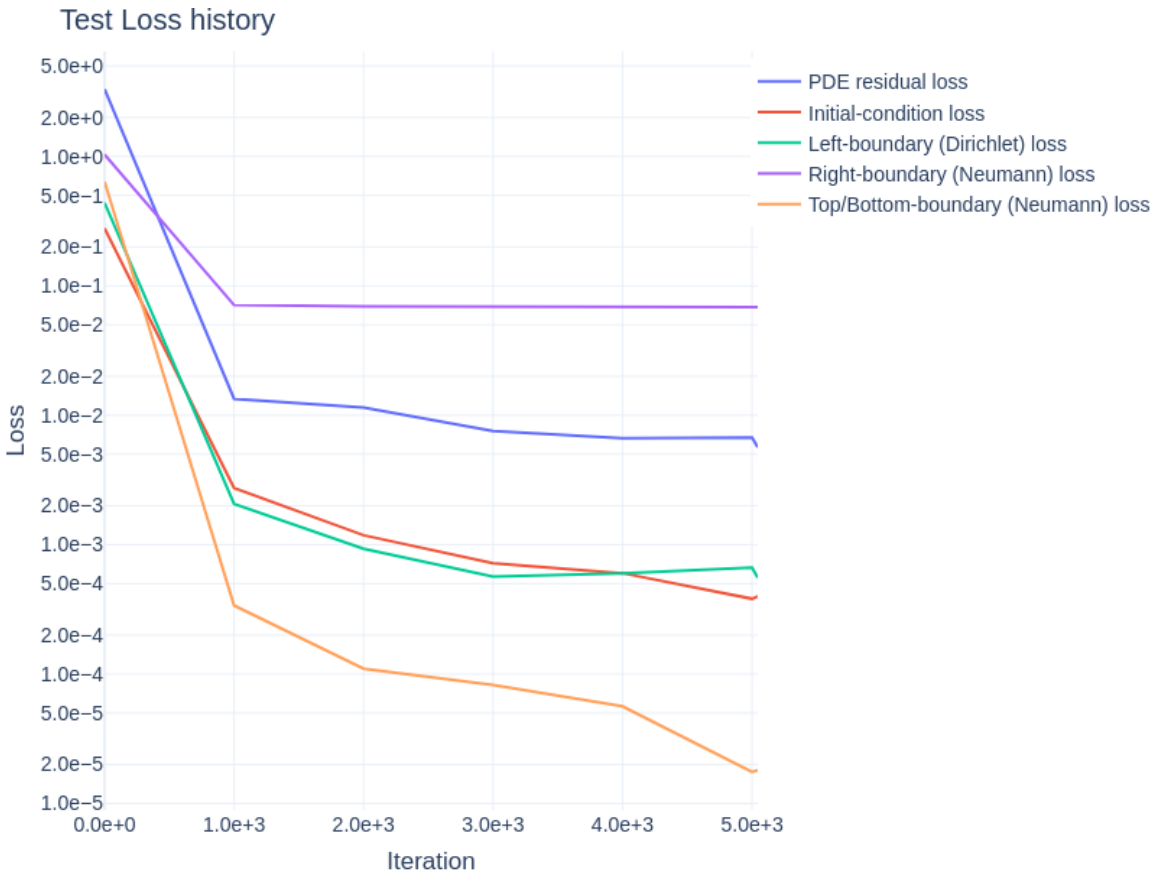


Figura 12.2: Gráfica de la perdida en el conjunto de prueba.

- Tiempos evaluados: t = [0.0, 0.25, 0.50, 0.75, 1.0].

2. Estructura del archivo:
   - Coordenadas espacio-temporales (t, x, y) para cada punto de la grilla 26×26.
   - Valores de la solución en los tiempos de interés.

```
import pandas as pd
# Lista de tiempos
times = [0.0, 0.25, 0.5, 0.75, 1.0]

# Crear la malla (x, y)
num_points = 26
x = np.linspace(0, 1, num_points)
y = np.linspace(0, 1, num_points)
X, Y = np.meshgrid(x, y)

# Lista para almacenar resultados
results = []

for t_val in times:
    # Crear entrada trunk: (num_points^2, 3)
    points = np.vstack((X.flatten(), Y.flatten(), t_val * np.ones_like(X.flatten()))).T

    # Crear entrada branch: condición inicial constante cero
    branch_input = np.zeros((1, sensor_pts.shape[0]))

    # Predecir
    predicted = model.predict((branch_input, points)).flatten()

    # Agregar los datos al resultado
    for xi, yi, thetai in zip(points[:, 0], points[:, 1], predicted):
        results.append([t_val, xi, yi, thetai])

# Crear el DataFrame
df = pd.DataFrame(results, columns=["time", "X", "Y", "Theta"])

# Obtener la ruta del script actual y guardar el archivo CSV
ruta = r"data/model_DoN.csv"
df.to_csv(ruta, index=False)
```

13 Comparación de resultados

13.1 Comparativa visual de las predicciones

Esta sección presenta un análisis cualitativo de los resultados mediante la comparación directa entre las predicciones del modelo, las soluciones reportadas en el estudio de Alessio Borgi (2023) y las obtenidas mediante el método de Crank Nickolson. La visualización paralela permite evaluar:

- Dominio espacial: Cuadrado unitario [0,1] × [0,1] con malla 26×26.
- Escala de colores: Mapa térmico viridis (consistente en ambas columnas).

13.1.1 Modelo contra resultados de Alessio Borgi (2023)

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.gridspec as gridspec
import pandas as pd
import numpy as np

# Lista de tiempos
times = [0.0, 0.25, 0.5, 0.75, 1.0]

# Cargar el dataframe
df = pd.read_csv(r'data/model_DoN.csv')

# Crear figura con subplots 3D en 1 fila y 5 columnas
fig, axes = plt.subplots(nrows=1, ncols=len(times),
                         figsize=(22, 6),
                         subplot_kw={'projection': '3d'})

# Asumimos que el grid es regular
num_points = int(np.sqrt(df[df["time"] == times[0]].shape[0]))

# Lista para almacenar los objetos surface
surf_list = []

# Reordenar para graficar
```

```
for i, (t_val, ax) in enumerate(zip(times, axes)):
    # Filtrar por tiempo actual
    df_t = df[df["time"] == t_val]

    # Obtener los valores de X, Y, Theta
    X_vals = df_t["X"].values.reshape((num_points, num_points))
    Y_vals = df_t["Y"].values.reshape((num_points, num_points))
    Z_vals = df_t["Theta"].values.reshape((num_points, num_points))

    # Dibujar la superficie
    surf = ax.plot_surface(
        Y_vals, X_vals, Z_vals,
        rstride=1, cstride=1,
        cmap="YlGnBu",
        edgecolor="none",
        antialiased=True
    )
    surf_list.append(surf)

    ax.set_title(f"Time = {t_val:.2f} s", pad=10)
    ax.set_xlabel("Y", labelpad=10)
    ax.set_ylabel("X", labelpad=10)
    ax.set_zlabel("T [K]", labelpad=10, rotation=90)


# Añadir barra de color común
cbar = fig.colorbar(surf_list[-1], ax=axes, shrink=0.9, aspect=90, pad=0.1, orientation='h
cbar.set_label('Temperatura [K]')

plt.show()
```

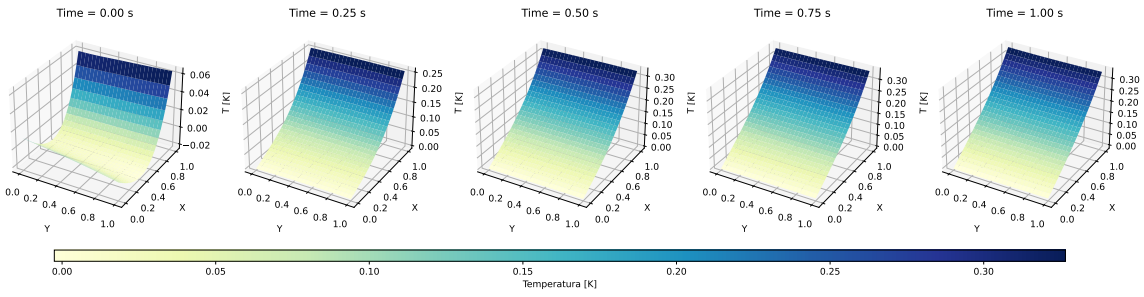


Figura 13.1: Predicciones de la red neuronal a distintos tiempos.

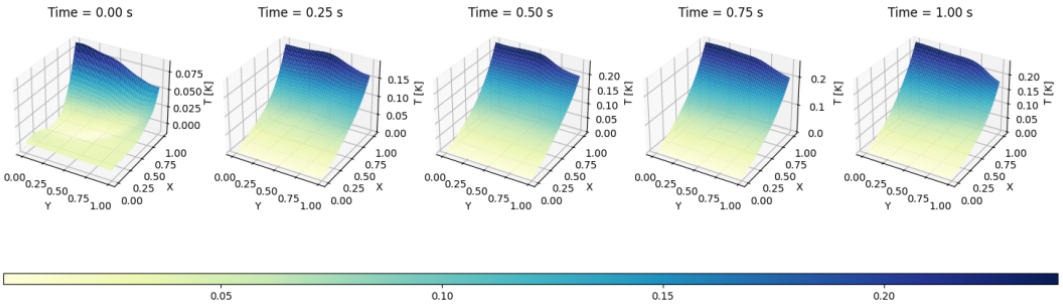


Figura 13.2: Resultados reportados por Alessio Borgi (2023) en el caso 2D.

13.1.2 Modelo contra método numérico

```
crank_nick_data = pd.read_csv(r'data/crank_nick.csv')
model_don_data = pd.read_csv(r'data/model_DoN.csv')

# Determinar los límites comunes para el colorbar
min_temp = min(model_don_data['Theta'].min(), crank_nick_data['Theta'].min())
max_temp = max(model_don_data['Theta'].max(), crank_nick_data['Theta'].max())

# Crear figura con subplots 3D en 2 filas y 5 columnas
fig = plt.figure(figsize=(22, 12))
axes = []

# Crear los subplots
for i in range(2):  # 2 filas
    for j in range(5):  # 5 columnas
        axes.append(fig.add_subplot(2, 5, i*5 + j + 1, projection='3d'))

axes = np.array(axes).reshape(2, 5)  # Convertir a matriz 2x5 para acceder fácilmente

# Añadir títulos generales para cada fila
fig.text(0.5, 0.92, "Predicciones modelo DON", ha='center', va='center', fontsize=14, font
fig.text(0.5, 0.56, "Predicciones método numérico", ha='center', va='center', fontsize=14,

# Función para graficar un dataframe en una fila específica
def plot_dataframe(df, row, num_points, cmap="viridis"):
    surf_list = []
    for col, t_val in enumerate(times):
        ax = axes[row, col]

        # Filtrar por tiempo actual
        df_t = df[df["time"] == t_val]
```

```
        # Obtener los valores de X, Y, Theta
        X_vals = df_t["X"].values.reshape((num_points, num_points))
        Y_vals = df_t["Y"].values.reshape((num_points, num_points))
        Z_vals = df_t["Theta"].values.reshape((num_points, num_points))

        # Dibujar la superficie con límites comunes
        surf = ax.plot_surface(
            Y_vals, X_vals, Z_vals,
            rstride=1, cstride=1,
            cmap=cmap,
            edgecolor="none",
            antialiased=True,
            vmin=min_temp,
            vmax=max_temp
        )
        surf_list.append(surf)

        ax.set_title(f"Time = {t_val:.2f} s", pad=10)
        ax.set_xlabel("Y", labelpad=10)
        ax.set_ylabel("X", labelpad=10)
        ax.set_zlabel("T [K]", labelpad=10, rotation=90)

    return surf_list

# Asumimos que el grid es regular para ambos dataframes
num_points = int(np.sqrt(model_don_data[model_don_data["time"] == times[0]].shape[0]))

# Graficar el primer dataframe en la fila superior
surf_model_don = plot_dataframe(model_don_data, 0, num_points)

# Graficar el segundo dataframe en la fila inferior
surf_crank_nick = plot_dataframe(crank_nick_data, 1, num_points)

# Añadir barra de color común en la parte inferior
cbar = fig.colorbar(surf_crank_nick[-1], ax=axes.ravel().tolist(),
                    use_gridspec=True, orientation='horizontal',
                    pad=0.05, aspect=90, shrink=0.9)
cbar.set_label('Temperatura [K]', labelpad=10)


plt.show()
```

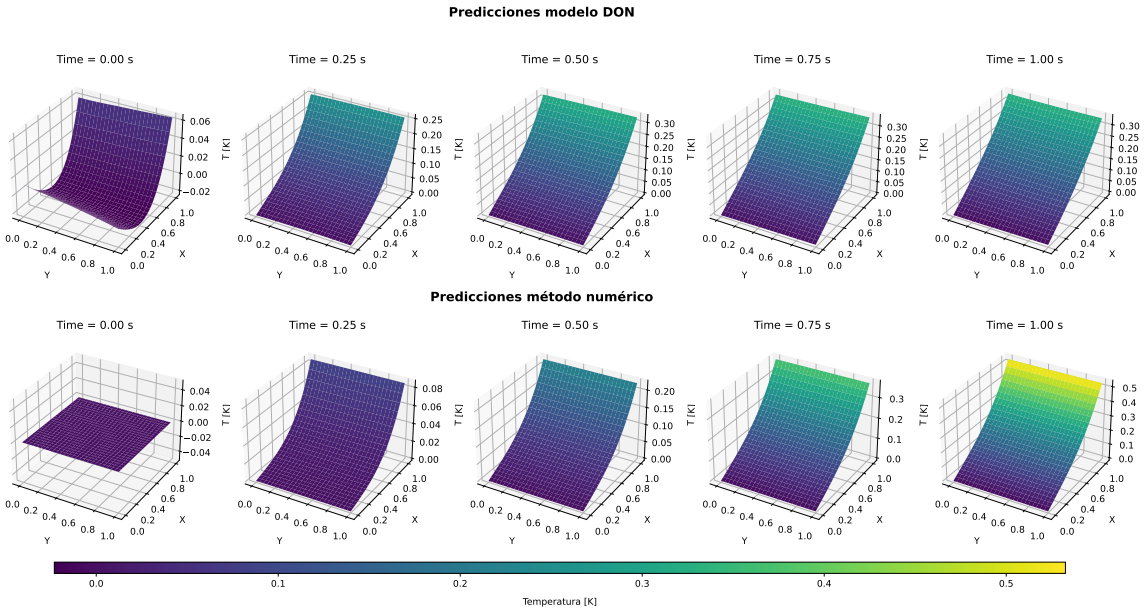


Figura 13.3: Contraste de las predicciones entre el método de Crank Nikolson y el modelo para cada tiempo.

13.2 Validación Cuantitativa frente al Método de Crank-Nicolson

Para evaluar numéricamente la precisión del modelo DeepONet, se realizó una comparación sistemática con soluciones de referencia generadas mediante el método de Crank-Nicolson. Este enfoque proporciona una métrica objetiva de la exactitud del modelo, más allá de la inspección visual.

```
# Función para calcular errores
def calculate_errors(true_data, pred_data, times):
    results = []

    for time in times:
        # Filtrar datos por tiempo
        true_subset = true_data[true_data['time'] == time]
        pred_subset = pred_data[pred_data['time'] == time]

        if len(true_subset) == 0 or len(pred_subset) == 0:
            print(f"Advertencia: No hay datos para tiempo t={time}")
            continue

        # Verificar que las dimensiones coincidan
        if len(true_subset) != len(pred_subset):
            print(f"Advertencia: Número de puntos no coincide para t={time}")
```

Tabla 13.1: Error del modelo DeepONet respecto a Crank-Nicolson.

Tiempo	MAE	MaxAE	Error L2
0.000	0.017	0.064	0.560
0.250	0.068	0.172	2.202
0.500	0.053	0.102	1.581
0.750	0.011	0.039	0.343
1.000	0.067	0.200	2.305

```
            min_len = min(len(true_subset), len(pred_subset))
            true_subset = true_subset.iloc[:min_len]
            pred_subset = pred_subset.iloc[:min_len]

        # Calcular errores para Theta
        theta_true = true_subset['Theta'].values
        theta_pred = pred_subset['Theta'].values

        absolute_error = np.abs(theta_true - theta_pred)
        l2_error = np.sqrt(np.sum((theta_true - theta_pred)**2))

        results.append({
            'time': time,
            'mean_absolute_error': np.mean(absolute_error),
            'max_absolute_error': np.max(absolute_error),
            'l2_error': l2_error
        })

    return pd.DataFrame(results)

# Calcular errores
error_results = calculate_errors(crank_nick_data, model_don_data, times)

# Guardar resultados
error_results.to_csv("data/error_comparison.csv", index=False)
```

13.2.1 Gráficas de error absoluto

```
# Calcular el error absoluto entre los dos dataframes
error_data = model_don_data.copy()
error_data['error'] = np.abs(crank_nick_data['Theta'] - model_don_data['Theta'])
```

```
# Crear figura con 3 filas y 2 columnas
fig, axes = plt.subplots(nrows=1, ncols=5, figsize=(22, 6))
axes = axes.ravel()  # Convertir a array 1D para fácil acceso

# Asumir que el grid es regular
num_points = int(np.sqrt(error_data[error_data["time"] == times[0]].shape[0]))

# Configuración común para los mapas de calor
plot_kwargs = {
    'cmap': 'hot_r',
    'shading': 'auto',
    'vmin': error_data['error'].min(),
    'vmax': error_data['error'].max()
}
# Lista para guardar los gráficos
abs_errors_pc = []

# Crear los subplots
for i, t_val in enumerate(times):
    ax = axes[i]

    # Filtrar por tiempo actual
    df_t = error_data[error_data["time"] == t_val]

    # Obtener valores y reshape
    X_vals = df_t["X"].values.reshape((num_points, num_points))
    Y_vals = df_t["Y"].values.reshape((num_points, num_points))
    error_vals = df_t["error"].values.reshape((num_points, num_points))

    # Crear mapa de calor
    pc = ax.pcolormesh(X_vals, Y_vals, error_vals, **plot_kwargs)

    # Configuración de ejes
    ax.set_title(f"Tiempo = {t_val:.2f} s", pad=10)
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_aspect('equal')

    abs_errors_pc.append(pc)

cbar = fig.colorbar(abs_errors_pc[-1], ax=axes, use_gridspec=True, shrink=0.9, aspect=90,
cbar.set_label('Error absoluto [K]')

# Mostrar el gráfico
plt.show()
```

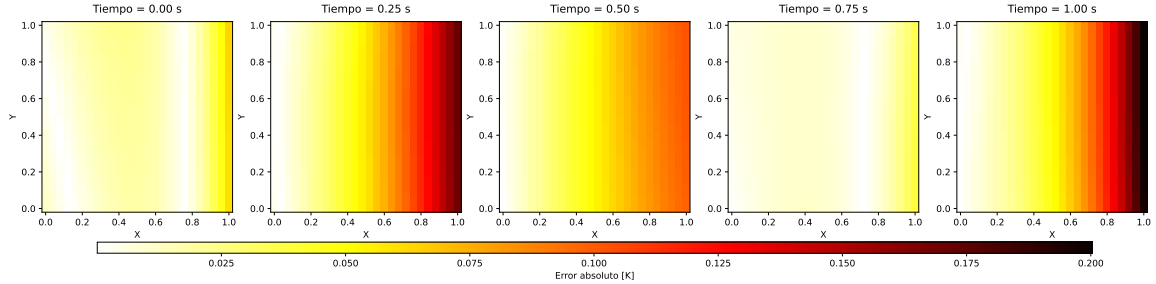


Figura 13.4: Errores absolutos entre el método de Crank Nikolson y el modelo para cada tiempo.

Referencias

Alessio Borgi, Alessandro De Luca, Eugenio Bugli. 2023. «BioHeat PINNs: Temperature Estimation with Bio-Heat Equation using Physics-Informed Neural Networks». https://github.com/alessioborgi/BioHeat_PINNs/tree/main?tab=readme-ov-file#bioheat-pinns-temperature-estimation-with-bio-heat-equation-using-physics-informed-neural-networks.

Blechschmidt, Jan, y Oliver G. Ernst. 2021. «Three ways to solve partial differential equations with neural networks—A review». *GAMM-Mitteilungen* 44 (2): e202100006. https://doi.org/10.1002/gamm.202100006.

Burden, Richard L., y J. Douglas Faires. 2010. «Numerical Analysis». En *Numerical Analysis*, 9.ª ed., 259-64. Boston, USA: Brooks Cole.

Dutta, Abhijit, y Gopal Rangarajan. 2018. «Diffusion in pharmaceutical systems: modelling and applications». *Journal of Pharmacy and Pharmacology* 70 (5): 581-98. https://doi.org/10.1111/jphp.12885.

Instituto Nacional del Cáncer. 2021. «¿Qué es el cáncer?» https://www.cancer.gov/espanol/cancer/naturaleza/que-es.

Karniadakis, George Em, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, y Liu Yang. 2021. «Physics-informed machine learning». *Nature Reviews Physics* 3 (6): 422-40. https://doi.org/10.1038/s42254-021-00314-5.

Kumar, Varun, Somdatta Goswami, Katiana Kontolati, Michael D. Shields, y George Em Karniadakis. 2024. «Synergistic Learning with Multi-Task DeepONet for Efficient PDE Problem Solving». *arXiv preprint arXiv:2408.02198*. https://arxiv.org/abs/2408.02198.

Lu, Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, y George Em Karniadakis. 2021. «Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators». *Nature Machine Intelligence* 3 (3): 218-29. https://doi.org/10.1038/s42256-021-00302-5.

Lu, Lu, Xuhui Meng, Zhiping Mao, y George Em Karniadakis. 2021. «DeepXDE: A deep learning library for solving differential equations». *SIAM Review* 63 (1): 208-28. https://doi.org/10.1137/19M1274067.

National Cancer Institute. 2021. «Hyperthermia to Treat Cancer». https://www.cancer.gov/about-cancer/treatment/types/hyperthermia.

Organización Mundial de la Salud. 2022. «Cáncer». https://www.who.int/es/news-room/fact-sheets/detail/cancer.

Pennes, H. H. 1948. «Analysis of Tissue and Arterial Blood Temperatures in the Resting Human Forearm». *Journal of Applied Physiology* 1 (2): 93-122. https://doi.org/10.1152/jappl.1948.1.2.93.

Quintero, Luis A., Mauricio Peñuela, Armando Zambrano, y Edwin Rodríguez. 2017. «Optimización del proceso de preparación de soluciones madre de antibióticos en un

servicio farmacéutico hospitalario». *Revista Cubana de Farmacia* 50 (2): 448-65. https://www.medigraphic.com/cgi-bin/new/resumen.cgi?IDARTICULO=75483.

Yang, Lihong, Xin Wu, Qian Wan, Jian Kong, Rui Liu, y Xiaoxi Liu. 2014. «Pharmaceutical preparation of antibiotics: a review on formulation and technique». *Asian Journal of Pharmaceutical Sciences* 9 (3): 145-53. https://doi.org/10.1016/j.ajps.2014.04.001.

Zill, Dennis G., y Michael R. Cullen. 2008. «Differential Equations with Boundary-Value Problems». En, 7.ª ed., 433-42. Belmont, CA: Cengage Learning.