

Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

6 de agosto de 2021

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-???-???-?
© Facundo Batista & Manuel Carlevaro
Primera Edición (versión preliminar)
Escrito con X₃LaTeX.

Licencia: [Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional \(CC BY-NC-SA 4.0\)](#)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2021
Web: <https://github.com/facundobatista/libro-pyciencia>

10 9 8 7 6 5 4 3 2 1

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científicos, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos" que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene dos grandes partes. La primera habla de Python, algunas bibliotecas importantes y otros temas que son fundamentales. En la segunda, mostramos cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En ambos casos este libro esquivo la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de los científicos para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Tanto los textos como el código fuente, ejemplos e imágenes son Copyright de Facundo Batista y Manuel Carlevaro y están compartidos bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) [1], salvo que se especifique puntualmente lo contrario.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

| | |
|--|-----------|
| Prefacio | 2 |
| I Herramientas fundamentales | 4 |
| II Temas específicos | 5 |
| 1. Ecuaciones en derivadas parciales | 6 |
| 1.1. Método de las diferencias finitas | 7 |
| 1.1.1. Ecuación de conducción de calor en 1D con un método explícito | 10 |
| 1.1.2. Ecuación de conducción de calor en 1D con un método implícito | 15 |
| 1.1.3. Implementación con matriz rala | 18 |
| 1.2. Método de elementos finitos | 21 |
| 1.2.1. Ecuación de Poisson | 22 |
| 1.2.2. Formulación variacional | 22 |
| 1.2.3. Implementación en FEniCS | 24 |
| 1.3. Lecturas recomendadas | 27 |
| III Apéndices | 29 |
| A. Zen de Python | 30 |

Parte I

Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos y técnicas fundamentales que serán utilizados en la Parte II al abordar temas de aplicaciones específicas. Se sugiere, a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

Parte II

Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte [I](#).

1 | Ecuaciones en derivadas parciales

La mayoría de los problemas presentados en la naturaleza, que son de interés en para físicos y matemáticos, se representan por medio de una o de un sistema de ecuaciones diferenciales. En general, los sistemas físicos involucran más de una variable independiente, por lo que en ese caso el modelo matemático contiene ecuaciones en derivadas parciales (EDP).

Las EDP juegan un papel fundamental en muchas ramas de las ciencias aplicadas y las ingenierías, por ejemplo, en transferencia de calor, elasticidad, dinámica de fluidos, electromagnetismo, óptica, mecánica cuántica, matemática financiera, etc. En particular, fenómenos como el flujo de calor en una varilla metálica u ondas en una cuerda se pueden representar por una EDP de segundo orden del tipo

$$R(x, y) \frac{\partial^2 u}{\partial x^2} + S(x, y) \frac{\partial^2 u}{\partial x \partial y} + T(x, y) \frac{\partial^2 u}{\partial y^2} + L \left(x, y, u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y} \right) = 0 \quad (1.1)$$

en la que u es la función desconocida de x y y (omitimos por simplicidad la dependencia explícita en las expresiones), las funciones R , S y T son funciones continuas de las variables x y y , mientras que L es una función continua de x , y , u , $\partial u / \partial x$ y $\partial u / \partial y$, de esta manera, la ecuación es lineal.

En lo que sigue, con el propósito de simplificar la escritura, adoptaremos la notación:


$$u_x = \frac{\partial u}{\partial x}, \quad u_{xx} = \frac{\partial^2 u}{\partial x^2}, \quad u_{xy} = \frac{\partial^2 u}{\partial x \partial y}$$

La ecuación (1.1) se clasifica en tres clases diferentes según el valor de $S^2 - 4RT$:

- a) Elíptica, si $S^2 - 4RT < 0$.
- b) Parabólica, si $S^2 - 4RT = 0$.
- c) Hiperbólica, si $S^2 - 4RT > 0$.

Esta clasificación es útil por dos razones. Por un lado, cada clase está asociada a diferentes problemas específicos, y por otro, cada grupo requiere técnicas de solución analítica especiales. La terminología utilizada para clasificar las ecuaciones surge por analogía con la utilizada en la clasificación de ecuaciones generales de segundo orden en la geometría analítica. Es importante notar que según como sea la dependencia de R , S y T de x y y , la ecuación puede estar en una clase diferente dependiendo del dominio para el cual se quiere obtener la solución.

Para determinar completamente una solución particular de una EDP es necesario definir las condiciones de borde, que son valores conocidos de la función o una combinación de sus derivadas sobre la frontera del dominio Ω del problema. Usualmente, la frontera del dominio se denota

|  | |
|---|---------|
| Módulo | Versión |
| argparse | 1.1 |
| Matplotlib | 3.3.4 |
| NumPy | 1.19.5 |
| SciPy | 1.6.0 |
| Sympy | 1.7.1 |
| Código disponible | |

como Γ o $\partial\Omega$, y por lo general las condiciones de borde pueden especificarse de forma diferente en partes distintas de la frontera. Dos condiciones de borde importantes son las de Dirichlet, en las que se especifica el valor de la función en la frontera, $u(\mathbf{x}) = h(\mathbf{x})$ para $\mathbf{x} \in \Gamma_D$, y las de Neumann, que especifica la derivada normal en la frontera:

$$\frac{\partial u(\mathbf{x})}{\partial \mathbf{n}} = g(\mathbf{x})$$

para $\mathbf{x} \in \Gamma_N$, donde \mathbf{n} es el vector normal a la frontera con sentido hacia afuera. Aquí, $h(\mathbf{x})$ y $g(\mathbf{x})$ son funciones arbitrarias.

Del mismo modo que sucede con las ecuaciones diferenciales ordinarias, solo se puede hallar soluciones analíticas de un problema representado mediante EDP en muy pocos y especiales casos [2]. En casi todos los problemas de interés es necesario buscar soluciones mediante técnicas numéricas.

Muchas de las técnicas numéricas para la solución de EDP se basan en la idea de discretizar el dominio en cada una de las variables independientes que aparecen en las ecuaciones, transformando de este modo el problema en un sistema de ecuaciones algebraicas. Este abordaje suele ser muy demandante de recursos computacionales, en parte debido a que el número de puntos requerido para discretizar una región crece exponencialmente con el número de dimensiones. Por ejemplo, si en un problema unidimensional discretizamos su dominio en 100 puntos, un problema bidimensional con resolución similar requiere $100^2 = 10^4$ puntos, y uno tridimensional necesita $100^3 = 10^6$ puntos. Dado que cada punto en el espacio discretizado corresponde a una variable desconocida, un problema en tres dimensiones puede resultar en un sistema muy grande de ecuaciones.

En este capítulo abordaremos dos de estas técnicas. En primer lugar, el método de las diferencias finitas (MDF) que consiste en aproximar las derivadas por sus expresiones en diferencias finitas, y posteriormente veremos el método de los elementos finitos (MEF), en el que la función desconocida se escribe como combinación lineal de una base de funciones simples que pueden ser diferenciadas e integradas fácilmente. La función desconocida se describe entonces por un conjunto de coeficientes en esta representación, y mediante una reformulación del problema se pueden obtener ecuaciones algebraicas para estos coeficientes.

1.1. Método de las diferencias finitas

El método de las diferencias finitas es conceptualmente simple y el más utilizado para resolver ecuaciones diferenciales con condiciones de borde. Consiste, básicamente, en discretizar el dominio y aproximar las ecuaciones diferenciales con sus expresiones en diferencias finitas, calculadas en los puntos discretos del dominio.

Estas ecuaciones en diferencias finitas proceden de las expansiones en series de Taylor de funciones multivariadas, que dan origen a diferentes esquemas. Por ejemplo, la expansión en

serie de Taylor de la función $u = u(x, y)$ es:

$$u(x + h, y) = u(x, y) + hu_x + \frac{h^2}{2!}u_{xx} + \frac{h^3}{3!}u_{xxx} + \dots$$

$$u(x - h, y) = u(x, y) - hu_x + \frac{h^2}{2!}u_{xx} - \frac{h^3}{3!}u_{xxx} + \dots$$

$$u(x + 2h, y) = u(x, y) + 2hu_x + \frac{(2h)^2}{2!}u_{xx} + \frac{(2h)^3}{3!}u_{xxx} + \dots$$

$$u(x - 2h, y) = u(x, y) - 2hu_x + \frac{(2h)^2}{2!}u_{xx} - \frac{(2h)^3}{3!}u_{xxx} + \dots$$

$$u(x, y + k) = u(x, y) + ku_y + \frac{k^2}{2!}u_{yy} + \frac{k^3}{3!}u_{yyy} + \dots$$

$$u(x, y - k) = u(x, y) - ku_y + \frac{k^2}{2!}u_{yy} - \frac{k^3}{3!}u_{yyy} + \dots$$

...

$$u(x + h, y + k) = u(x, y) + (hu_x + ku_y) + \frac{1}{2!}(h^2u_{xx} + 2hku_{xy} + k^2u_{yy}) + \dots$$

A partir de estas expresiones es sencillo obtener las aproximaciones para las derivadas parciales de primer y segundo orden. Para las de primer se pueden obtener diferencias hacia adelante:

$$u_x = \frac{u(x + h, y) - u(x, y)}{h} + O(h)$$

$$u_y = \frac{u(x, y + k) - u(x, y)}{k} + O(k)$$

también diferencias hacia atrás:

$$u_x = \frac{u(x, y) - u(x - h, y)}{h} + O(h)$$

$$u_y = \frac{u(x, y) - u(x, y - k)}{k} + O(k)$$

y diferencias centrales:

$$u_x = \frac{u(x + h, y) - u(x - h, y)}{2h} + O(h^2)$$

$$u_y = \frac{u(x, y + k) - u(x, y - k)}{2k} + O(k^2)$$

Del mismo modo se pueden obtener expresiones en diferencias para las derivadas parciales de segundo orden. En el caso de diferencias hacia adelante resultan:

$$u_{xx} = \frac{u(x + 2h, y) - 2u(x + h, y) + u(x, y)}{h^2} + O(h)$$

$$u_{yy} = \frac{u(x, y + 2k) - 2u(x, y + k) + u(x, y)}{k^2} + O(k)$$

$$u_{xy} = \frac{u(x + h, y + k) - u(x, y + k) - u(x + h, y) + u(x, y)}{hk} + O(h, k)$$

donde la última derivada mixta se obtiene a partir de aproximaciones en diferencias finitas de las derivadas de primer orden. Para el caso de las diferencias hacia atrás las aproximaciones son

$$u_{xx} = \frac{u(x, y) - 2u(x - h, y) + u(x - 2h, y)}{h^2} + O(h)$$

$$u_{yy} = \frac{u(x, y) - 2u(x, y - k) + u(x, y - 2k)}{k^2} + O(k)$$

$$u_{xy} = \frac{u(x, y) - u(x, y - k) - u(x - h, y) + u(x - h, y - k)}{hk} + O(h, k)$$

y por último, para las diferencias centrales, se obtiene

$$\begin{aligned} u_{xx} &= \frac{u(x+h, y) - 2u(x, y) + u(x-h, y)}{h^2} + O(h^2) \\ u_{yy} &= \frac{u(x, y+k) - 2u(x, y) + u(x, y-k)}{k^2} + O(k^2) \\ u_{xy} &= \frac{u(x+h, y+k) - u(x+h, y-k) - u(x-h, y+k) + u(x-h, y-k)}{4hk} + O(hk) \end{aligned}$$

Debe notarse que las expresiones en diferencias centrales son mejores aproximaciones que las demás, dado que estamos despreciando términos que son de orden h^2 en diferencias centrales comparados con términos de orden h en las diferencias hacia adelante y hacia atrás. Por este motivo, es preferible utilizar expresiones en diferencias centrales.

Para utilizar las expresiones en diferencias finitas obtenidas en la solución de una EDP es necesario discretizar el dominio. Esto se hace introduciendo una malla particionada uniformemente. Los puntos de la malla son $x_i = ih$ y $y_j = jk$, donde $i = 0, 1, \dots, N_x$, siendo $N_x + 1$ el número de puntos en los que particionamos el dominio en la dimensión x . Si $x \in [0, L_x]$, resulta que $h = L_x/N_x$. Análogamente, $j = 0, 1, \dots, N_y$, y si $y \in [0, L_y]$ el valor de k es L_y/N_y .

Una vez discretizado el dominio, procederemos a la obtención de una aproximación a la solución de la EDP en los puntos de la malla. Consideremos la función $u(x_i, y_j) = u_{i,j}$. Las expresiones anteriores para las derivadas parciales de primer y segundo orden en el punto (x_i, y_j) se pueden establecer para cada caso. Por ejemplo, para las derivadas de primer orden en diferencias finitas hacia adelante resultan:

$$\begin{aligned} \frac{\partial u(x_i, y_j)}{\partial x} &= u_x \Big|_{i,j} = \frac{u(x_i+h, y_j) - u(x_i, y_j)}{h} + O(h) \\ &= \frac{u_{i+1,j} - u_{i,j}}{h} + O(h) \\ &= \frac{u_{i+1,j} - u_{i,j}}{h} + O(h) \end{aligned}$$

Análogamente para u_y :

$$\frac{\partial u(x_i, y_j)}{\partial y} = u_y \Big|_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{k} + O(k)$$

Del mismo modo, las expresiones para las ecuaciones en diferencias hacia atrás:

$$\begin{aligned} \frac{\partial u(x_i, y_j)}{\partial x} &= u_x \Big|_{i,j} = \frac{u_{i,j} - u_{i-1,j}}{h} + O(h) \\ \frac{\partial u(x_i, y_j)}{\partial y} &= u_y \Big|_{i,j} = \frac{u_{i,j} - u_{i,j-1}}{k} + O(k) \end{aligned}$$

y para las diferencias centrales:

$$\begin{aligned} \frac{\partial u(x_i, y_j)}{\partial x} &= u_x \Big|_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2h} + O(h^2) \\ \frac{\partial u(x_i, y_j)}{\partial y} &= u_y \Big|_{i,j} = \frac{u_{i,j+1} - u_{i,j-1}}{2k} + O(k^2) \end{aligned}$$

Las expresiones de las derivadas de segundo orden, aproximadas por ecuaciones en diferencias finitas, calculadas sobre los puntos de la malla se obtienen de manera análoga que las de primer orden. Para las ecuaciones en diferencias hacia adelante se obtienen:

$$\begin{aligned}
\frac{\partial^2 u(x_i, y_j)}{\partial x^2} &= u_{xx} \Big|_{i,j} = \frac{u(x_i + 2h, y_j) - 2u(x_i + h, y_j) + u(x_i, y_j)}{h^2} + O(h) \\
&= \frac{u(x_{i+2}, y_j) - 2u(x_{i+1}, y_j) + u(x_i, y_j)}{h^2} + O(h) \\
&= \frac{u_{i+2,j} - 2u_{i+1,j} + u_{i,j}}{h^2} + O(h)
\end{aligned}$$

Del mismo modo se obtienen:

$$\begin{aligned}
\frac{\partial^2 u(x_i, y_j)}{\partial y^2} &= u_{yy} \Big|_{i,j} = \frac{u_{i,j+2} - 2u_{i,j+1} + u_{i,j}}{k^2} + O(k) \\
\frac{\partial^2 u(x_i, y_j)}{\partial x \partial y} &= u_{xy} \Big|_{i,j} = \frac{u_{i+1,j+1} - u_{i+1,j} - u_{i,j+1} + u_{i,j}}{hk} + O(h, k)
\end{aligned}$$

Procediendo de forma análoga se obtienen las expresiones para las diferencias hacia atrás:

$$\begin{aligned}
\frac{\partial^2 u(x_i, y_j)}{\partial x^2} &= u_{xx} \Big|_{i,j} = \frac{u_{i,j} - 2u_{i-1,j} + u_{i-2,j}}{h^2} + O(h) \\
\frac{\partial^2 u(x_i, y_j)}{\partial y^2} &= u_{yy} \Big|_{i,j} = \frac{u_{i,j} - 2u_{i,j-1} + u_{i,j-2}}{k^2} + O(k) \\
\frac{\partial^2 u(x_i, y_j)}{\partial x \partial y} &= u_{xy} \Big|_{i,j} = \frac{u_{i,j} - u_{i-1,j} - u_{i,j-1} + u_{i-1,j-1}}{hk} + O(h, k)
\end{aligned}$$

y finalmente las expresiones para las diferencias centrales:

$$\begin{aligned}
\frac{\partial^2 u(x_i, y_j)}{\partial x^2} &= u_{xx} \Big|_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + O(h^2) \\
\frac{\partial^2 u(x_i, y_j)}{\partial y^2} &= u_{yy} \Big|_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} + O(k^2) \\
\frac{\partial^2 u(x_i, y_j)}{\partial x \partial y} &= u_{xy} \Big|_{i,j} = \frac{u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}}{4hk} + O(hk)
\end{aligned}$$

1.1.1. Ecuación de conducción de calor en 1D con un método explícito

Aplicaremos ahora el método de las diferencias finitas a la resolución de la EDP que representa la conducción de calor en una varilla (1D):

$$u_t = c u_{xx}, \quad x \in (0, L), \quad t \in (0, T) \quad (1.2)$$

En esta ecuación, del tipo parabólico, $u(x, t)$ es la temperatura en función de la localización en la barra y el tiempo, y $c > 0$ es la difusividad térmica del medio:

$$c = \frac{k}{c_p \rho}$$

donde k es la conductividad térmica (cuyas unidades son $\text{J m}^{-1} \text{s}^{-1} \text{K}^{-1}$), c_p es el calor específico a presión constante (en $\text{J kg}^{-1} \text{K}^{-1}$) y ρ es la densidad del material que compone la barra (en kg m^{-3}), por lo que las unidades de c resultan $\text{m}^2 \text{s}^{-1}$. Consideramos que dicha barra es unidimensional (una de sus dimensiones es mucho mayor que las otras dos) y se encuentra térmicamente aislada en su superficie lateral, intercambiando calor solo por sus extremos que se mantienen a temperatura constante.

Para obtener una solución única necesitamos condiciones de borde. Con solo una derivada temporal en el tiempo, necesitamos solamente una condición inicial, mientras que al haber una derivada de segundo orden espacial, son necesarias dos condiciones de borde. En el caso de la variable temporal, especificar la solución a $t = 0$ representa una condición inicial que podemos formalizar como $u(x, 0) = I(x)$ donde I es una función conocida. En la variable espacial las condiciones de borde se establecen en cada punto de la frontera del dominio, en los que debemos conocer el valor de u , u_x o una combinación de ambos.

Para resolver un problema simple, adoptaremos las condiciones de borde en x estableciendo el valor de u . Entonces, el sistema completo es:

$$u(x, 0) = I(x), \quad x \in [0, L] \quad (1.3)$$

$$u(0, t) = 0, \quad t > 0 \quad (1.4)$$

$$u(L, t) = 2, \quad t > 0 \quad (1.5)$$

El primer paso en el procedimiento de discretización es reemplazar el dominio $[0, L] \times [0, T]$ por un conjunto de puntos de una malla. Tomaremos los puntos equiespaciados:

$$x_i = i\Delta x, \quad i = 0, 1, \dots, N_x$$

y

$$t_j = j\Delta t, \quad j = 0, 1, \dots, N_t$$

donde tomamos $\Delta x = h = L/N_x$ y $\Delta t = k = T/N_t$. Como hicimos antes, u_{ij} denota la función en los puntos de la malla que aproxima a $u(x_i, t_j)$. Dado que la ecuación (1.2) debe cumplirse en el punto (x_i, t_j) se obtiene la ecuación:

$$u_t|_{i,j} = cu_{xx}|_{i,j} \quad (1.6)$$

El próximo paso consiste en reemplazar las derivadas por las aproximaciones en diferencias finitas. Esto da origen a diversos esquemas según la representación elegida. El método computacionalmente más simple resulta de utilizar una diferencia hacia adelante en el tiempo (esquema de Euler hacia adelante) y una diferencia central en el espacio:

$$\frac{u_{i,j+1} - u_{i,j}}{k} = c \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \quad (1.7)$$

De este modo hemos convertido una EDP en ecuaciones algebraicas (o discretas), que son fáciles de resolver. Esto se hace a partir de una ecuación de recurrencia en la que asumimos que $u_{i,j}$ es conocida, mientras que $u_{i,j+1}$ es la única incógnita en (1.7):

$$u_{i,j+1} = u_{i,j} + F(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) \quad (1.8)$$

donde hemos introducido al número de malla de Fourier $F = ck/h^2$. F es un número adimensional que agrupa el parámetro físico c del problema, junto con los parámetros de la discretización h y k . Las propiedades del método numérico dependen en forma crítica del valor de F .

Podemos ver una representación gráfica de la aplicación de este método en la figura 1.1. A la ecuación de recurrencia (1.8) se la puede representar como un estencil sobre la grilla, que al ir desplazando sobre puntos en los que ya conocemos el valor aproximado de la solución, nos permite obtener los valores sobre los puntos aún no calculados de la malla.

Analizaremos ahora el código que permite resolver la ecuación (1.2) con las condiciones de borde (1.3)–(1.5). En las primeras líneas del programa importamos, como es usual, los módulos necesarios para acceder a las rutinas especializadas para graficar (líneas 7–9), numpy para utilizar arrays (línea 10) y algunos módulos de scipy para disponer de diversos métodos de álgebra lineal. A continuación configuramos algunos parámetros de los gráficos que vamos a generar (líneas 18–22). Debe notarse que este bloque inicial de código (líneas 1–22) es común al resto del código analizado en esta sección.

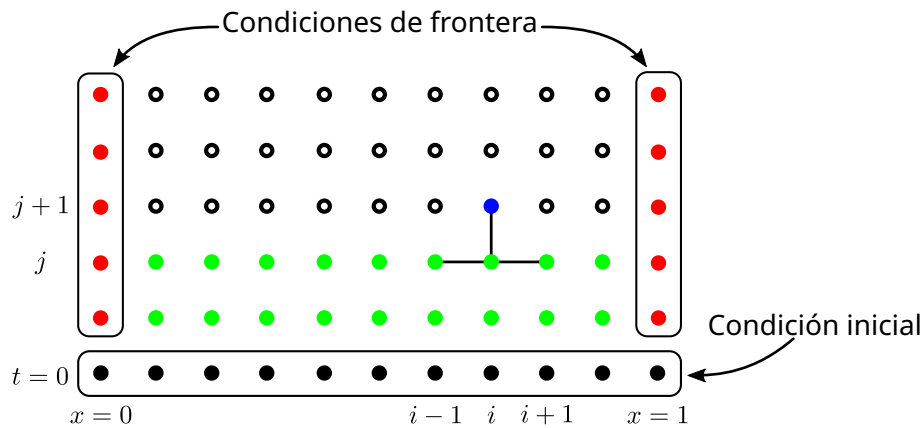


FIGURA 1.1: Esquema de solución por diferencias finitas de la ecuación de calor en 1D. Puntos negros: valores conocidos de $u(x, 0)$ (condición inicial). Puntos rojos: valores conocidos de $u(0, t)$ y $u(1, t)$ (condiciones de frontera). Puntos verdes, valores ya calculados de $u(x_i, t_j)$. Puntos blancos: valores por calcular sobre la grilla. Punto azul: valor de $u(x_i, t_{j+1})$ que se obtiene al aplicar el estencil representado por las líneas negras.



El código completo se encuentra en [calor1D-completo.py](#)

Este programa incluye los tres métodos tratados a continuación para resolver el problema en diferencias finitas, y también la visualización de los resultados.

```

1 #!/usr/bin/env python3
2 """Programa que resuelve la ecuación de calor en 1D."""
3
4 import argparse
5 from contextlib import contextmanager
6
7 import matplotlib.pyplot as plt
8 import matplotlib.cm as cmx
9 from matplotlib import colors
10 import numpy as np
11 from scipy import (
12     linalg, # Módulo de álgebra lineal, para el método implícito
13     sparse, # Paquete para matrices ralas
14 )
15 from scipy.sparse import linalg as sparse_linalg # Solvers para matrices ralas
16
17 # Configure matplotlib
18 plt.rcParams.update({
19     'text.usetex': True,
20     'font.size': 14,
21     'axes.labelsize': 'large',
22 })
23
24
25 def I(x):
26     """Representa la curva de temperatura (en función de x) inicial (para t=0)
27
28     Entonces  $u(x, 0) = I(x)$ :
29          $u(0, 0) = 0$ 
30          $u(1, 0) = 2$ 
31     """

```

```

32     return 4 * x * (3 / 2 - x)
33
34
35 # Parámetros globales del problema
36 L = 1 # Longitud de la barra
37 T = 1 # Tiempo máximo de la solución
38 c = 2 # Difusividad térmica
39
40
41 @contextmanager
42 def graficar(x, t):
43     """Muestra la temperatura en función de x y t."""
44
45     # Semántica de tonos
46     cm = plt.get_cmap('Greens_r')
47     c_norm = colors.PowerNorm(gamma=0.25, vmin=t[0], vmax=t[-1])
48     scalar_map = cmx.ScalarMappable(norm=c_norm, cmap=cm)
49
50     # La curva en sí
51     fig, ax = plt.subplots()
52
53     def agregar_curva(point, u):
54         cval = scalar_map.to_rgba(point)
55         ax.plot(x, u, color=cval)
56
57     # Pasamos la función para agregar las curvas
58     yield agregar_curva
59
60     # Dibujamos
61     fig.colorbar(cmx.ScalarMappable(norm=c_norm, cmap=cm))
62     plt.xlabel(r'$x$')
63     plt.ylabel(r'$u(x)$')
64     plt.show()

```

En la línea 25 definimos la condición inicial. Dado el array x como argumento, devuelve un nuevo array con los valores de una función que en este caso es una parábola que abre hacia abajo, manteniendo los valores de borde establecidos. Luego, en las líneas 36–38 definimos los parámetros físicos del problema: longitud de la barra, el valor máximo de tiempo sobre el que vamos a calcular la solución, y el coeficiente de difusividad térmica.

A continuación (líneas 41–64) definimos la función `graficar(x, t)` que se ocupará de visualizar las curvas que obtengamos de la solución a intervalos de tiempo regulares. Esta función trabaja como un administrador de contexto o *context manager* que nos permite utilizarla “alrededor” de las diferentes rutinas de resolución de las ecuaciones en diferencias, evitando de este modo la repetición de código para cada caso.

```

67 def resolver_explicito():
68     """Método explícito."""
69     # Puntos de resolución del problema
70     Nx = 10 # Número de puntos de la malla en x
71     Nt = 500 # Número de puntos de la malla en t
72     h = L / Nx # Paso en x
73     k = T / Nt # Paso en t
74     F = c * k / h ** 2 # Número de malla de Fourier
75     print(f'{h = } | {k = } | {F = }')
76
77     # Malla en x y t
78     x = np.linspace(0, L, Nx + 1)
79     t = np.linspace(0, T, Nt + 1)
80

```

```

81  # Valores de u(x,t)
82  u = np.zeros(Nx + 1) # u en j
83
84  # Iteración en t (j)
85  with graficar(x, t) as agregar_curva:
86      # Condición inicial (y la dibujamos)
87      up = I(x)
88      agregar_curva(0, up)
89
90      for j in range(1, Nt):
91          # estencil (sin los bordes)
92          u[1:Nx] = up[1:Nx] + F * (up[0:Nx - 1] - 2 * up[1:Nx] + up[2:Nx + 1])
93
94          # Condiciones de borde
95          u[0] = 0
96          u[Nx] = 2
97
98          # Trazamos una solución cada 10 pasos en j
99          if not j % 10:
100             agregar_curva(j * k, u)
101
102          # Cambiamos las variable antes del próximo paso
103          up = u

```

La implementación del método explícito de solución queda delimitada por la función `resolver_explicito()` entre las líneas 67 y 103. En dicha función, definimos inicialmente los parámetros del algoritmo: los números de puntos en que dividimos la malla en x (N_x) y en t (N_t), y calculamos el paso en ambas direcciones (h y k), así como el valor del número de malla de Fourier. Estos valores se imprimen en pantalla en la línea 75, especialmente para verificar la condición de estabilidad de la solución $F \leq 0,5$.

Representamos los valores de la malla en cada variable mediante los arrays x y t , definidos en las líneas 78 y 79, mientras que los valores de la función $u(x_i, t_j)$ serán almacenados en el array u para cada valor de t_j , al cual dimensionamos con $N_x + 1$ valores nulos en la línea 82.

En la línea 85 inicializamos el administrador de contexto y a continuación almacenamos en el array up los valores de la temperatura de la barra para $t = 0$, graficando este estado inicial (líneas 87 y 88). A continuación iniciamos el bucle de la línea 90 en el que iremos iterando sobre los valores de j que representan los distintos instantes de tiempo en que dividimos la malla para la variable t . En la línea 92 calculamos la solución para j conociendo los valores de la solución para $(j - 1)$ que tenemos almacenados en up con la ecuación de recurrencia (1.8). Este cálculo se realiza para los puntos interiores de la malla en x , dado que conocemos los valores de borde que se agregan al array u en las líneas 95–96. Cada 10 pasos de j agregamos la curva que representa la solución al gráfico (líneas 99–100) y finalmente actualizamos los valores en el array up con los calculados recientemente almacenados en u para avanzar en la iteración en el cálculo del siguiente valor de j .

Como salida del programa generamos un gráfico que representa el valor de $u(x)$ para distintos valores de t , y representamos la evolución temporal con una secuencia de tonos, desde el más oscuro para $t = 0$ hasta el más claro para $t = T$. Esta semántica de tonos está definida en las líneas 46–48 del código. Primero elegimos un mapa de color ('Greens_r') entre los disponibles en Matplotlib. Luego, en la línea 47, escalamos los valores del array t en el intervalo $[0, 1]$ en forma exponencial, ya que la dinámica es rápida al inicio y luego se hace muy lenta a medida que se aproxima a la solución de equilibrio, y esta forma de escalar permite visualizar ese cambio rápido. El resultado obtenido puede verse en la figura 1.2.

En esta solución en particular, con la malla establecida obtenemos valores de $h = 0,1$ y $k = \Delta t = 0,002$, lo cual da un valor de $F \approx 0,4 < 0,5$. Las soluciones obtenidas son suaves, a

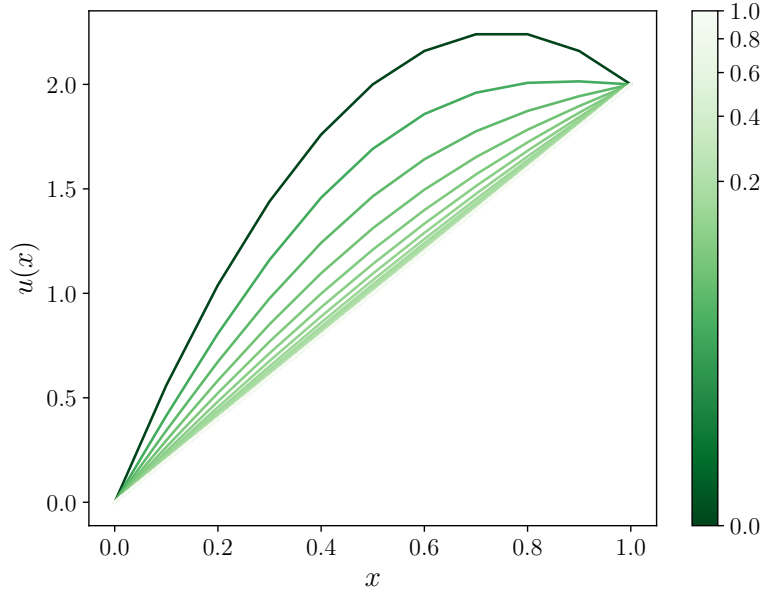


FIGURA 1.2: Solución obtenida para el problema de transmisión de calor en 1D. La escala de colores indica la evolución temporal de la solución.

costa de usar un paso temporal pequeño de modo de obtener acotado el valor de F . Invitamos al lector a modificar el código del programa para verificar que con valores de F mayores a 0,5 se obtienen soluciones que presentan inestabilidades no físicas.

1.1.2. Ecuación de conducción de calor en 1D con un método implícito

La solución mediante el esquema de Euler hacia adelante impone la restricción $F \leq 0,5$ para obtener soluciones numéricamente estables. Si fijamos la separación espacial h de la malla, esto significa que debemos tomar un paso temporal de integración $k = \delta t = h^2/(2c)$. En procesos representados por la ecuación (1.2) (conducción de calor, difusión, etc.) la solución varía rápido al inicio, pero a medida que transcurre el tiempo el proceso se hace más lento a medida que se acerca al equilibrio (en condición estacionaria obtenemos que $u_t = 0$) y en esta situación tener un paso temporal pequeño puede resultar inconveniente.

Los métodos implícitos dan origen a un sistema acoplado de ecuaciones que se deben resolver en cada paso de tiempo, en los cuales es posible utilizar cualquier paso δt , sin obtener inestabilidades numéricas, aunque naturalmente la precisión disminuye al aumentar δt . El método implícito más simple de implementar es el esquema de Euler hacia atrás, que consiste en utilizar nuevamente una diferencia central como aproximación a la derivada parcial espacial, pero una diferencia hacia atrás en el tiempo:

$$\frac{u_{i,j} - u_{i,j-1}}{k} = c \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \quad (1.9)$$

En este caso asumimos que conocemos la solución $u_{i,j-1}$ en el tiempo $j-1$, pero todas las cantidades en el tiempo j son ahora desconocidas, con lo cual no es posible resolver con respecto de $u_{i,j}$ porque este valor está acoplado a los vecinos en la malla $u_{i-1,j}$ y $u_{i+1,j}$, que también son desconocidos. Esta expresión da origen a un sistema de $(N_x - 1) \times (N_t - 1)$ ecuaciones algebraicas acopladas.

Multiplicando ambos términos de la ecuación (1.9) por k y reacomodando obtenemos:

$$-Fu_{i-1,j} + (1 + 2F)u_{i,j} - Fu_{i+1,j} = u_{i,j-1} \quad (1.10)$$

para $i = 1, \dots, N_x - 1$, ya que por las condiciones de borde (1.4) y (1.5) conocemos los valores en $i = 0$ y $i = N_x$. Entonces, dado un valor de j , tenemos $N_x - 1$ ecuaciones dadas por (1.10) sobre los puntos internos de la malla en x . Una vez obtenida la solución para este sistema, es necesario iterar sobre j para obtener las sucesivas soluciones que representan la evolución temporal del sistema.

El sistema de ecuaciones acopladas dadas por la ecuación (1.10), junto con las condiciones de borde (1.4) y (1.5), pueden representarse en forma matricial:

$$\mathbb{A} \cdot \mathbf{u} = \mathbf{b} \quad (1.11)$$

donde $\mathbf{u} = (u_{1,j}, u_{1,j}, \dots, u_{N_x-1,j})$ y la matriz \mathbb{A} tiene $N_x - 1$ filas y $N_x - 1$ columnas con la siguiente estructura tridiagonal:

$$\mathbb{A} = \begin{pmatrix} (1+2F) & -F & 0 & 0 & \dots & \dots & 0 & 0 \\ -F & (1+2F) & -F & 0 & \dots & \dots & 0 & 0 \\ 0 & -F & (1+2F) & -F & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \dots & \vdots & \vdots \\ 0 & 0 & \ddots & 0 & -F & (1+2F) & -F & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 & -F & (1+2F) & -F \\ 0 & \dots & \dots & \dots & 0 & 0 & -F & (1+2F) \end{pmatrix}$$

mientras que el vector \mathbf{b} es:

$$\mathbf{b} = \begin{pmatrix} u_{1,j-1} + F u_{0,j} \\ u_{2,j-1} \\ \vdots \\ u_{N_x-2,j-1} \\ u_{N_x-1,j-1} + F u_{N_x,j} \end{pmatrix}$$

La forma de calcular los valores de la matriz \mathbb{A} puede representarse desplazando el estencil mostrado en la figura 1.3 a lo largo de la fila j , y resolver este sistema nos permite obtener la solución para el tiempo correspondiente a ese valor de j . Una vez resuelto el sistema, se construye uno nuevo para la fila $j + 1$ a partir de los valores recién calculados, avanzando de este modo la solución temporal.

Debe notarse que la matriz \mathbb{A} no posee elementos que cambien con el tiempo, por lo que una vez construida es la misma para todos los soluciones de la progresión en j , mientras que el vector \mathbf{b} depende de los valores de u calculados en $j - 1$. Por lo tanto, en cada iteración que resuelva el sistema en su evolución temporal, debemos actualizar los valores de \mathbf{b} .

La implementación del algoritmo para resolver la ecuación de calor en una dimensión usando el esquema implícito se encuentra integrada al código que presentamos en la subsección anterior (??), esta vez en la definición de la función `resolver_implicito()` en la línea 106. Inicialmente definimos los parámetros del algoritmo: cantidad de puntos en que dividimos la malla en cada dimensión x y t (N_x y N_t , los pasos en cada malla (h y k) y el número de malla de Fourier (F), tal como se puede ver en las líneas 109–113.

```
106 def resolver_implicito():
107     """Método implícito."""
108     # Puntos de resolución del problema
109     Nx = 100 # Número de puntos de la malla en x
```

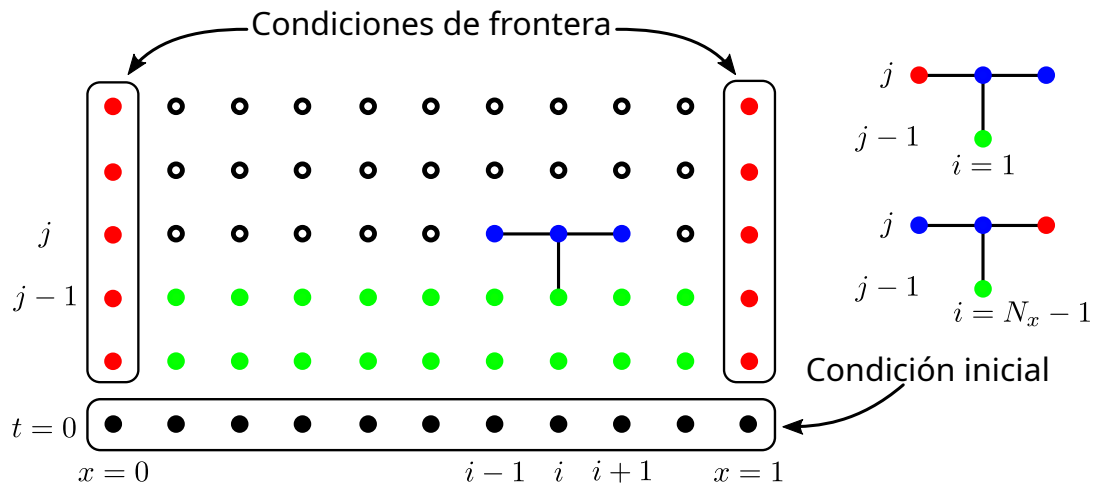


FIGURA 1.3: Esquema de solución por diferencias finitas de la ecuación de calor en 1D usando un esquema implícito. Puntos negros: valores conocidos de $u(x, 0)$ (condición inicial). Puntos rojos: valores conocidos de $u(0, t)$ y $u(1, t)$ (condiciones de frontera). Puntos verdes, valores ya calculados de $u(x_i, t_j)$. Puntos blancos: valores por calcular sobre la grilla. Puntos azules: valor de $u(x_i, t_{j+1})$ que se calculan al aplicar el estencil representado por las líneas negras. A la derecha se esquematizan las modificaciones en el estencil al evaluar las condiciones de borde.

```

110  Nt = 1000 # Número de puntos de la malla en t
111  h = L / Nx # Paso en x
112  k = T / Nt # Paso en t
113  F = c * k / h ** 2 # Número de malla de Fourier
114  print(f'{h = } | {k = } | {F = }')
115
116  # Malla en x y t
117  x = np.linspace(0, L, Nx + 1)
118  t = np.linspace(0, T, Nt + 1)
119
120  # Valores de u(x,t)
121  u = np.zeros(Nx + 1) # u en j
122
123  # Estructura de datos para el sistema lineal
124  A = np.zeros((Nx - 1, Nx - 1))
125  b = np.zeros(Nx - 1)
126  for i in range(1, Nx - 2):
127      A[i, i - 1] = -F
128      A[i, i] = 1 + 2 * F
129      A[i, i + 1] = -F
130  A[0, 0] = A[Nx - 2, Nx - 2] = 1 + 2 * F
131  A[0, 1] = A[Nx - 2, Nx - 3] = -F
132
133  with graficar(x, t) as agregar_curva:
134      # Condición inicial (y la dibujamos)
135      up = I(x)
136      agregar_curva(0, up)
137
138      # Iteración en t (j)
139      for j in range(1, Nt):
140          # Calculamos b
141          b = up[1:Nx]
142          b[Nx - 2] += F * 2 # b[0] += F * 0 (condición de borde)
143
144          # Resolvemos el sistema
145          u[1:Nx] = linalg.solve(A, b)
146          u[Nx] = 2
147

```

```

148     # Trazamos una solución cada 10 pasos en j
149     if not j % 10:
150         agregar_curva(j * k, u)
151
152     # Actualizamos los valores de up antes del próximo paso
153     up = u

```

En las líneas siguientes (117–131) generamos las estructuras en arrays para representar la malla en las dimensiones x y t , la solución $u(x_i, t_j)$, la matriz \mathbb{A} y el vector \mathbf{b} .

En la línea 133 inicializamos el administrador de contexto y a continuación generamos el valor inicial de la solución, que almacenamos en el array `up`, trazando la solución inicial en la línea 136.

En el bucle `for` de la línea 139 iteramos sobre los valores de j que representan la progresión temporal de la solución, calculando en cada paso de la iteración el valor del vector \mathbf{b} . La solución se obtiene en la línea 145 con el método `linalg.solve` que importamos previamente del módulo `scipy`, y establecemos el valor conocido en el borde en la línea 146. Completamos el bloque del `for` trazando la curva de la solución cada 10 pasos de iteración y finalmente actualizamos el valor del array `up` antes de dar el próximo paso.

Tal como hicimos al final de la sección anterior, proponemos ahora al lector resolver el sistema utilizando pasos temporales mayores (es decir con menos divisiones de la malla en la dimensión temporal) para observar que, a diferencia del esquema implícito, no se generan inestabilidades numéricas.

1.1.3. Implementación con matriz rara

Hemos visto que la matriz \mathbb{A} de la sección anterior es tridiagonal, aunque no explotamos este hecho en el código de resolución con el abordaje implícito, ya que almacenamos en el array `A` una representación completa (o *densa*) que incluye muchos elementos que sabemos desde el inicio que son nulos, y el algoritmo implementado en `solve` resuelve el sistema con todos los ceros. Considerando que tenemos $N_x - 1$ incógnitas, el método `solve` realiza operaciones del orden de $(N_x - 1)^3$ y requiere el almacenamiento de $(N_x - 1)^2$ elementos.

Por otra parte, utilizando el hecho que \mathbb{A} es tridiagonal y utilizando las herramientas de *software* correspondientes, las demanda de cálculo y almacenamiento son proporcionales solamente a N_x . Esto conduce a una mejora significativa en el rendimiento del código, tanto por el uso de memoria como por la velocidad de ejecución. Veremos ahora cómo resolver la ecuación de calor en 1 dimensión con un esquema implícito tomando ventaja de que \mathbb{A} es tridiagonal.

La idea es utilizar por un lado una representación de \mathbb{A} como matriz rara (o *sparse*, como se usa en inglés), y para ello usaremos el paquete `scipy.sparse`, de modo de almacenar solamente los elementos no nulos de \mathbb{A} . Por otro lado, utilizaremos algoritmos de solución que operan sobre estructuras de matrices ralas, contenidos también en `scipy.sparse`. Esta implementación está contenida en la función `resolver_ralas` del mismo código común a toda esta sección:

```

156 def resolver_ralas():
157     """Método implícito utilizando matrices ralas."""
158     # Puntos de resolución del problema
159     Nx = 100 # Número de puntos de la malla en x
160     Nt = 1000 # Número de puntos de la malla en t
161     h = L / Nx # Paso en x
162     k = T / Nt # Paso en t
163     F = c * k / h ** 2 # Número de malla de Fourier
164     print(f'{h = } | {k = } | {F = }')
165

```

```

166 # Malla en x y t
167 x = np.linspace(0, L, Nx + 1)
168 t = np.linspace(0, T, Nt + 1)
169
170 # Valores de u(x,t)
171 u = np.zeros(Nx + 1) # u en j
172
173 # Calculamos la matriz rala
174 diagonal = 1 + 2 * F
175 superior = -F
176 inferior = -F
177
178 A = sparse.diags(
179     diagonals=[diagonal, superior, inferior],
180     offsets=[0, 1, -1],
181     shape=(Nx - 1, Nx - 1),
182     format='csr')
183 print(A.todense())
184
185 with graficar(x, t) as agregar_curva:
186     # Condición inicial (y la dibujamos)
187     up = I(x)
188     agregar_curva(0, up)
189
190     # Iteración en t (j)
191     for j in range(1, Nt):
192         # Calculamos b
193         b = up[1:Nx]
194         b[Nx - 2] += F * 2 # b[0] += F * 0 (condición de borde)
195
196         # Resolvemos el sistema
197         u[1:Nx] = sparse.linalg.spsolve(A, b)
198         u[Nx] = 2
199
200         # Trazamos una solución cada 10 pasos en j
201         if not j % 10:
202             agregar_curva(j * k, u)
203
204         # Actualizamos los valores de up antes del próximo paso
205         up = u

```

Al igual que en los casos de las subsecciones anteriores, las primeras líneas de esta función determinan los parámetros del algoritmo y los arrays que almacenan la grilla y la solución (ver líneas 159–171).

En las líneas 174–176 calculamos los elementos no nulos de \mathbb{A} que componen la diagonal principal, superior e inferior de \mathbb{A} , almacenándolos en las variables `diagonal`, `superior` e `inferior`, respectivamente. Con estos valores construimos la matriz rala `A` en la línea 178. Este método requiere varios argumentos: `diagonals` es una lista conteniendo los valores que definen la matriz; `offsets` establece la ubicación en la matriz (0 es la diagonal principal, 1 es la primer diagonal superior, -1 es la primer diagonal inferior); `shape` establece la cantidad de elementos en la matriz y finalmente `format` da cuenta de cómo se almacenan internamente los elementos de `A`. En particular, el método `'csr'` (formato *Compressed Sparse Row*) es el apropiado para realizar las operaciones de manipulación y multiplicación requeridas para resolver el sistema lineal de ecuaciones.¹

En la línea 183 imprimimos en pantalla la representación densa de `A`, para verificar que fue construida correctamente. Cuando establecemos el valor `Nx = 10` en la línea 159 la salida en pantalla es:

¹Ver [entrada en Wikipedia](#).

```
[[ 1.8 -0.4 0. 0. 0. 0. 0. 0. 0. ]
 [-0.4 1.8 -0.4 0. 0. 0. 0. 0. 0. ]
 [ 0. -0.4 1.8 -0.4 0. 0. 0. 0. 0. ]
 [ 0. 0. -0.4 1.8 -0.4 0. 0. 0. 0. ]
 [ 0. 0. 0. -0.4 1.8 -0.4 0. 0. 0. ]
 [ 0. 0. 0. 0. -0.4 1.8 -0.4 0. 0. ]
 [ 0. 0. 0. 0. 0. -0.4 1.8 -0.4 0. ]
 [ 0. 0. 0. 0. 0. 0. -0.4 1.8 -0.4 ]
 [ 0. 0. 0. 0. 0. 0. 0. -0.4 1.8]]
```

Del mismo modo que hicimos en los métodos explícito e implícito, en la línea 185 iniciamos el administrador de contexto, seguido del cálculo del valor inicial que almacenamos en `up` y graficamos en la línea 188. El bucle `for` de la línea 191 es casi idéntico al mostrado en la subsección anterior (1.1.2), con la única diferencia de que utilizamos el método `spsolve` del submódulo `linalg` de `scipy.sparse` (ver línea 15) al que nos referimos con el nombre `sparse_linalg` para diferenciarlo del método `linalg` de `scipy`, que resuelve el sistema lineal de ecuaciones, especializado en la utilización de matrices ralas.

Ejecutando el código con una malla fina en x (usando $N_x = 100$ en la línea 159) obtenemos la figura 1.4, en la que se puede apreciar que con una malla tan fina la curva se aprecia muy suave, a diferencia de la figura 1.2. También en este caso hemos incrementado la resolución de la malla temporal, usando $N_t = 1000$ en la línea 160.

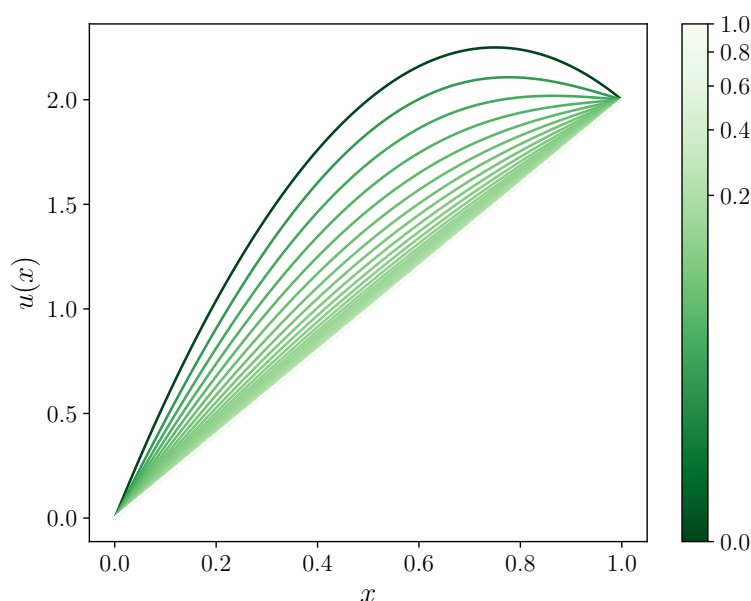


FIGURA 1.4: Solución obtenida para el problema de transmisión de calor en 1D con el método implícito y usando matrices ralas. La escala de colores indica la evolución temporal de la solución.

Podemos comparar la eficiencia de usar matrices ralas, al menos en cuanto a su velocidad de ejecución y uso de memoria, utilizando el comando

```
/usr/bin/time -v [programa ejecutable]
```

y comparando la salida al ejecutar ambos códigos en condiciones similares. Para ello comentamos las salidas a pantalla con `print` y todas las líneas relativas a la generación de gráficos. Para realizar la comparación, en ambos casos usamos una grilla temporal con $N_t = 1000$ y variamos el tamaño de la grilla espacial (N_x). En la Tabla 1.1 podemos ver la salida de `time` extrayendo solo los valores de `System time` (tiempo de ejecución del sistema) y `Maximum resident set size` que es el máximo

| Nx | Tiempo (s) | | | Memoria (kb) | | |
|------|------------|-------|------|--------------|-------|-------|
| | Densas | Ralas | % | Densas | Ralas | % |
| 500 | 20.44 | 0.43 | 2.10 | 82548 | 66144 | 80.13 |
| 1000 | 88.15 | 0.52 | 0.59 | 97212 | 66260 | 68.16 |
| 2000 | 343.49 | 0.59 | 0.17 | 145908 | 67692 | 46.39 |
| 4000 | 651.98 | 0.82 | 0.13 | 342296 | 68332 | 19.96 |

TABLA 1.1: Rendimientos con diferentes tamaños de malla. Se comparan los tiempos de ejecución y el máximo uso de memoria. Los porcentajes se calculan como fracción de los valores correspondientes a matrices densas.

tamaño que ocupa la ejecución del programa en memoria. Los valores corresponden a la ejecución de ambos códigos en un procesador Intel® Core™ i7-2600.

Vemos que el uso de matrices ralas reduce dramáticamente el tiempo de ejecución del programa, debido a la eficiencia en el manejo de los pocos valores que contienen las estructuras de datos de las matrices ralas en comparación con las densas. Por otra parte, la diferencia en memoria no es muy grande para sistemas pequeños, pero la demanda de almacenamiento se reduce a casi el 20 % para la malla más grande analizada.

1.2. Método de elementos finitos

El método de diferencias finitas descrito en la sección anterior tiene más interés académico que práctico, solo problemas con geometrías simples pueden ser abordados eficientemente de esta forma. Para problemas reales, los métodos de elementos finitos (MEF) o volúmenes finitos (MVF) son los que se usan comúnmente en ciencias e ingenierías.

Aunque ambos métodos (MEF y MVF) son aplicables a cualquier sistema de ecuaciones en derivadas parciales multi-física, el uso de MEF está más consolidado en problemas de análisis estructural, elasticidad y transferencia de calor, mientras que el MVF utiliza el teorema de Gauss para simplificar ecuaciones integrales siempre que sea posible, y tiene amplia difusión en problemas de mecánica de fluidos para resolver la ecuación de Navier-Stokes. Para problemas vinculados al electromagnetismo (ecuaciones de Maxwell), campos gravitacionales, etc., ambos métodos presentan formulaciones útiles. Una descripción detallada de estos métodos excede el alcance de este libro, en esta sección daremos solo un abordaje introductorio al MEF, resolviendo un problema simple con esta técnica y utilizando un *software* específico que implementa los algoritmos necesarios.

Existen numerosos entornos de trabajo, o *frameworks*, para resolver EDP mediante elementos finitos. Entre ellos podemos mencionar scikit-fem², SfePy³, polyfem⁴, fempy⁵ y FEniCS⁶. Estos *frameworks* incorporan diversas herramientas para resolver una variedad de problemas expresados mediante EDP, especialmente métodos eficientes para resolver los grandes sistemas de ecuaciones usando matrices ralas. El ejemplo que desarrollaremos a continuación lo implementaremos con FEniCS, ya que provee una práctica interfaz en Python a un poderoso código que implementa el MEF (utilizaremos el tutorial de FEniCS[3] como guía en esta sección).

²<https://scikit-fem.readthedocs.io/en/latest/index.html>

³<http://sfepy.org/doc-devel/index.html>

⁴<https://polyfem.github.io/python/>

⁵<https://pypi.org/project/fempy/>

⁶<https://fenicsproject.org/>

1.2.1. Ecuación de Poisson

Tal vez el problema más simple expresado en EDP sea resolver la ecuación de Poisson, que aparece en muchos problemas físicos como la conducción de calor, la difusión de sustancias, electrostática o flujos de fluidos inmiscibles, entre otros. Dicha ecuación se expresa:

$$-\nabla^2 u = f \quad \text{en } \Omega \quad (1.12)$$

$$u = u_D \quad \text{en } \partial\Omega \quad (1.13)$$

donde $u = u(\mathbf{x})$ es la función desconocida mientras que $f = f(\mathbf{x})$ está determinada. ∇^2 es el operador de Laplace (denotado a menudo Δ), Ω es el dominio espacial y $\partial\Omega$ su frontera. No tenemos en esta ecuación una derivada temporal, por lo que estamos representando una condición estacionaria, a diferencia del caso abordado en la sección anterior.

En un espacio cartesiano bidimensional con coordenadas x y y , la ecuación (1.12) se escribe:

$$-\frac{\partial^2 u}{\partial^2 x} - \frac{\partial^2 u}{\partial^2 y} = f(x, y) \quad (1.14)$$

en la cual ahora u es función de dos variables, $u(x, y)$, definida sobre el dominio bidimensional Ω .

Resolver un problema con condiciones de borde como la ecuación de Poisson (1.14) con las condiciones de borde dadas por la ecuación (1.13) consiste en transitar las siguientes etapas:

1. Identificar el dominio computacional Ω , la EDP, sus condiciones de frontera y los términos fuente f .
2. Reformular la EDP como un problema variacional de elementos finitos.
3. Escribir un código en Python que defina el dominio computacional, el problema variacional, las condiciones de contorno.
4. Utilizar las rutinas de FEniCS para resolver el problema con condiciones de contorno y, opcionalmente, extender el programa para calcular cantidades derivadas como flujos y promedios, y visualizar los resultados.

1.2.2. Formulación variacional

La idea central del método de elementos finitos consiste en representar el dominio sobre el que está definido la EDP con un conjunto finito de regiones discretas, o *elementos*, y aproximar la función desconocida por medio de una combinación lineal de funciones base con soporte local sobre estos elementos (o sobre un grupo de elementos vecinos).

El núcleo del procedimiento para convertir una EDP en un problema variacional es el de multiplicar la EDP por una función v , integrar esa función sobre el dominio Ω mediante integración por partes de los términos con derivadas de segundo orden. A la función v utilizada para multiplicar la EDP se la denomina función de prueba o *test*. Por otra parte, la función desconocida u que se va a aproximar se denomina función ensayo o *trial*. Mantendremos en adelante los nombres en inglés debido a que es como se las llama en el código Python de FEniCS.

Multiplicamos entonces la ecuación de Poisson por la función *test* v e integramos:

$$-\int_{\Omega} (\nabla^2 u) v \, dx = \int_{\Omega} f v \, dx \quad (1.15)$$

Luego aplicamos integración por partes al integrando con derivadas de segundo orden:

$$-\int_{\Omega} (\nabla^2 u) v \, dx = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds \quad (1.16)$$

donde $\partial u / \partial n$ es la derivada de u en la dirección normal a la frontera con sentido hacia afuera del dominio. Un requerimiento que debe cumplir la función v es que se anule en las partes de la frontera donde se conoce u , lo que para este problema implica que $v = 0$ en toda la frontera $\partial\Omega$. Esta condición anula el segundo término del lado derecho de la ecuación (1.16). De las ecuaciones (1.15) y (1.16) obtenemos:

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx \quad (1.17)$$

Esta ecuación es válida para toda v en algún espacio de funciones \hat{V} , mientras que la función *trial* u pertenece al (posiblemente diferente) espacio de funciones V . La ecuación (1.17) representa la *formulación débil* del problema con condiciones de borde original (o *formulación fuerte*) dado por las ecuaciones (1.12) – (1.13).

Para resolver numéricamente la ecuación de Poisson necesitamos transformar el problema variacional continuo (1.17) en un problema variacional discreto. Esto se consigue introduciendo espacios de dimensión finita para las funciones *test* y *trial*, que denotaremos como $\hat{V}_h \subset \hat{V}$ y $V_h \subset V$, respectivamente. El problema variacional discreto puede enunciarse entonces como: encontrar $u_h \in V_h \subset V$ tal que:

$$\int_{\Omega} \nabla u_h \cdot \nabla v \, dx = \int_{\Omega} f v \, dx \quad \forall v \in \hat{V}_h \subset \hat{V} \quad (1.18)$$

La elección de V_h y \hat{V}_h dependen del tipo de elementos finitos que queremos usar en nuestro problema, y para la mayoría de los casos de interés estos espacios son bien conocidos. Por ejemplo, si elegimos elementos lineales triangulares con tres nodos, V_h y \hat{V}_h son los espacios de todas las funciones lineales por tramos sobre una malla de triángulos, en las que las funciones de \hat{V}_h son cero en la frontera y las de V_h son iguales a u_D . Para ilustrar estas funciones en una dimensión, la figura 1.5 muestra al intervalo $[0, 6]$ discretizado en cinco puntos interiores, y una función continua (línea negra) se aproxima como una función lineal a tramos (en líneas de puntos verde) mediante una combinación lineal de funciones triangulares base (en líneas azules).

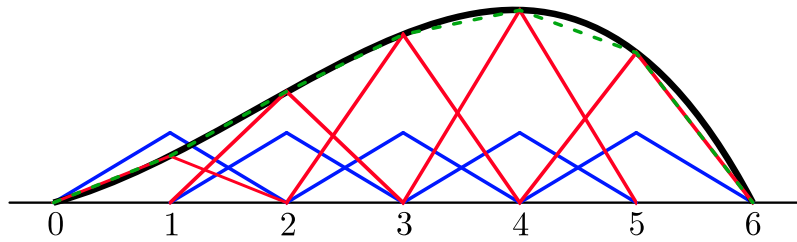


FIGURA 1.5: Ejemplo de aproximación de una función (línea negra) mediante funciones base lineales por tramos (líneas azules).

Es usual en la literatura matemática denotar a u_h como la solución del problema discreto y u la del problema continuo. Sin embargo utilizaremos u para representar la solución discreta para mantener la nomenclatura de FEniCS. La secuencia es, entonces, representar con u a la función desconocida de la EDP, derivar la formulación variacional con $u \in V$ y $v \in \hat{V}$, y luego discretizar el problema eligiendo espacios de dimensión finita para V y \hat{V} , con lo cual u se convierte en una función de elemento finito discreto. En la práctica esto ocurre a partir de crear una malla para discretizar el dominio Ω y la elección del tipo de elemento, y hacer entonces corresponder V a esta malla y elemento elegido. Dependiendo de si V es de dimensión finita o infinita, u será la solución aproximada o la exacta.

Resulta conveniente aquí introducir una notación unificada para representar la formulación débil del problema (1.18):

$$a(u, v) = L(v) \quad (1.19)$$

Para la solución de la ecuación de Poisson, las expresiones explícitas en la expresión anterior son

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, dx \quad (1.20)$$

$$L(v) = \int_{\Omega} f v \, dx \quad (1.21)$$

En la literatura matemática, $a(u, v)$ se conoce como una *forma bilineal* ($V \times \hat{V} \mapsto \mathbb{R}$), mientras que $L(v)$ es una *forma lineal* ($\hat{V} \mapsto \mathbb{R}$). De este modo, para cada problema lineal que queremos resolver tenemos que identificar los términos con la función incógnita u y agruparlos en $a(u, v)$, y del mismo modo agrupar los términos que dependen solo de las funciones conocidas en $L(v)$. Estas fórmulas para a y L se codifican directamente en el programa.

En síntesis, antes de escribir el código de Python para que FEniCS resuelva una EDP, debemos realizar dos pasos:

1. Convertir la EDP en un problema variacional discreto, esto es, encontrar $u \in V$ tal que:

$$a(u, v) = L(v) \quad \forall v \in \hat{V} \quad (1.22)$$

2. Elegir los espacios V y \hat{V} , lo que significa especificar la malla y el tipo de elementos finitos.

1.2.3. Implementación en FEniCS

Veamos ahora cómo resolvemos la ecuación de Poisson en un problema concreto. Queremos determinar la condición estacionaria de la temperatura de una placa (es decir, en dos dimensiones) cuyos bordes se mantienen a temperaturas fijas. Elegiremos un problema sencillo, del cual conocemos la solución exacta, para poder verificar la precisión de la solución. Para ello, vamos a elegir la solución exacta:

$$u_e(x, y) = 1 + x + 2y^2 \quad (1.23)$$

en un dominio en dos dimensiones. Si reemplazamos u_e en la ecuación de Poisson (1.14), vemos que $u_e(x, y)$ es solución si:

$$f(x, y) = -4, \quad u_D(x, y) = u_e(x, y) = 1 + x + 2y^2 \quad (1.24)$$

independientemente de la forma del dominio. Elegiremos por simplicidad un dominio cuadrado $\Omega = [0, 1] \times [0, 1]$. El MEF, sobre un dominio rectangular uniformemente particionado en elementos triangulares lineales, reproduce exactamente un polinomio de segundo grado en los vértices de las celdas, independientemente del tamaño de los elementos. De este modo, la solución aproximada obtenida deberá ser igual a la exacta (con la precisión de la representación de punto flotante) en los nodos.

El código para resolver la ecuación de Poisson en 2D usando FEniCS, dadas las elecciones de u_D , f y Ω es el siguiente:

```
1 #!/usr/bin/env python3
2 '''
3 Solución de la ecuación de Poisson en 2D con condiciones de Dirichlet
4 para resolver un problema cuya solución es conocida.
```

```

5
6     -Δ(u) = f     en un cuadrado unitario
7         θ = u_D   en la frontera
8
9     u_D = 1 + x + 2 y^2
10    f = -4
11    ...
12
13 import fenics as fe
14 import numpy as np
15
16 # Creamos la malla y definimos el espacio de funciones
17 mesh = fe.UnitSquareMesh(10, 10)
18 V = fe.FunctionSpace(mesh, 'P', 1)
19
20 # Definimos las condiciones de borde
21 u_D = fe.Expression("1 + x[0] + 2 * x[1]*x[1]", degree=2)
22
23 def boundary(x, on_boundary):
24     return on_boundary
25
26 bc = fe.DirichletBC(V, u_D, boundary)
27
28 # Definimos el problema variacional
29 u = fe.TrialFunction(V)
30 v = fe.TestFunction(V)
31 f = fe.Constant(-4.0)
32 a = fe.dot(fe.grad(u), fe.grad(v)) * fe.dx
33 L = f * v * fe.dx
34
35 # Calculamos la solución
36 u = fe.Function(V)
37 fe.solve( a == L, u, bc )
38
39 # Guardamos la solución en un archivo con formato VTK
40 vtk_file = fe.File('poisson.pvd')
41 vtk_file << u
42
43 # Calculamos el error en la norma L2
44 error_L2 = fe.errornorm(u_D, u, 'L2')
45
46 # Calculamos los errores en los vértices
47 vertex_values_u_D = u_D.compute_vertex_values(mesh)
48 vertex_values_u = u.compute_vertex_values(mesh)
49 error_max = np.max(np.abs(vertex_values_u_D - vertex_values_u))
50
51 print(f'    Error L2 = {error_L2}')
52 print(f'Error máximo = {error_max}')

```

Iniciamos el programa importando todo lo que está definido en el módulo `fenics`. En particular nos permite usar las funciones `UnitSquareMesh`, `FunctionSpace`, `Expression` y demás definiciones. También importamos `numpy` para realizar alguna operación sobre arrays.

Lo primero que hacemos es crear una malla que almacenamos en la variable `mesh`, en la línea 17. Para eso usamos `UnitSquareMesh` que define una malla uniforme sobre el cuadrado unitario $[0, 1] \times [0, 1]$. Esta malla está compuesta de celdas que en 2D son triángulos. Los parámetros 10 y 10 especifican que el cuadrado debe dividirse primero en 10 divisiones en cada dimensión (lo que genera 100 cuadrados de lado 0,1 que cubren todo el cuadrado unitario). Cada uno de estos cuadrados interiores se divide en un par de triángulos (incluso se puede definir en qué dirección trazar la diagonal que divide el cuadrado en dos triángulos). En total se generan $2 \times 10 \times 10 = 200$ triángulos, con un total de $(10+1)(10+1) = 121$ vértices o nodos. FEniCS dispone de rutinas que

realizan mallados sobre geometrías simples en una, dos y tres dimensiones, pero en problemas con geometrías complejas es necesario utilizar programas especializados en esta tarea (podemos destacar Gmsh⁷ como un excelente generador de mallas). En estos casos, las mallas generadas por programas externos pueden importarse en FEniCS para ser usadas en los cálculos de MEF.

A continuación creamos el espacio V de funciones de elemento finito, almacenado en la variable V de la línea 18. Este espacio se crea sobre $mesh$, que es el primer argumento. El segundo, `'P'`, declara la utilización de la familia estándar Lagrange de elementos (también se puede usar `'Lagrange'` para esta familia). FEniCS soporta todos los elementos de la Tabla Periódica de los Elementos Finitos⁸ [4] con su correspondiente notación. El tercer argumento (1) especifica el grado del elemento finito. En este caso, el elemento lineal de Lagrange estándar P_1 es un triángulo con nodos en los tres vértices (nombrado en algunas referencias como “triángulo lineal”).

Luego tenemos que definir las condiciones de borde $u = u_D$ en $\partial\Omega$. Esto se hace en la línea 26 usando la función `DirichletBC`. En los argumentos de esta función debe especificarse el espacio de funciones (V), la expresión matemática que define a u_D y la frontera donde ésta función aplica.

La definición de la función u_D se hace en la línea 21. La cadena que es el primer argumento de esta función debe ser escrita con la sintaxis de C++, ya que la evaluación de las expresiones se realiza por medio de código especializado en ese lenguaje (todas las funciones que se pueden incluir se encuentran definidas en el archivo de cabecera `cmath` de C++, como `sin`, `exp`, `log`, etc.), y en este caso depende de las variables `x[0]` y `x[1]` correspondientes a las coordenadas x y y . Como tercer argumento usamos un 2, lo que significa que u_D puede representar la solución cuadrática exacta de nuestro problema.

Para finalizar la definición de `bc` debemos indicar en el tercer argumento de la línea 26 la región $\partial\Omega$ que define el borde del dominio. Para eso definimos la función `boundary` en la línea 23, que devuelve `True` si x está en el borde del dominio. El argumento `on_boundary` es `True` si x está en el borde físico del dominio, por lo que simplemente podemos devolver ese valor en la función `boundary`. Esta función será llamada para cada punto discreto de la malla, lo que significa que podemos definir bordes donde u es conocida incluso dentro de la malla, si es necesario.

Con estas definiciones previas podemos empezar a resolver el problema. En las líneas 29 y 30 declaramos las variables u y v como funciones *trial* y *test* definidas sobre el espacio V . En la matemática del método distinguimos los espacios V y \hat{V} , pero en este problema la única diferencia ocurre en las condiciones de borde. FEniCS no especifica estas condiciones como parte de un espacio de funciones, por lo que es suficiente trabajar con un espacio común V para las funciones *test* y *trial*.

Antes de definir las formas bilineal y lineal del problema es necesario definir la función f . Podemos hacerlo del mismo modo que la variable u_D de la línea 21, pero como en este caso la función es constante, se representa en forma más eficiente usando la función `Constant`. Con todas las definiciones en su lugar, podemos ahora especificar $a(u, v)$ y $L(v)$ tal como lo hacemos en las líneas 32 y 33 del programa. Podemos destacar aquí la similitud con las contrapartes matemáticas dadas por las ecuaciones (1.20) y (1.21). Esta característica de poder expresar la formulación variacional con un código Python muy similar constituye una de las fortalezas de FEniCS, ya que hace simple representar problemas en EDP más complejos que el de este ejemplo.

En las líneas 36 y 37 se resuelve el problema. Notemos que primero definimos la variable u como `TrialFunction` y la usamos para representar la función desconocida de la forma bilineal a . Ahora la redefinimos como un objeto `Function` que representa la solución. Si bien son objetos diferentes, ambos representan el mismo objeto matemático por lo que es natural usar la misma variable.

⁷<https://gmsh.info/https://gmsh.info/>

⁸<https://www-users.math.umn.edu/~arnold/femtable/>

Utilizaremos en este ejemplo un programa externo para visualizar la solución: ParaView⁹. Esta es una herramienta poderosa para la representación gráfica de campos escalares y vectoriales, incluyendo los calculados por FEniCS. Para ello es necesario guardar la información de estos campos, contenida en la variable u , en el popular formato VTK¹⁰, lo que hacemos en las líneas 40 (definiendo el nombre del archivo que escribiremos en disco) y 41 (escribiendo en el archivo el contenido de u). Debe notarse que la forma de esta operación de escritura no es muy *pythonica*, ya que hace uso del operador de inserción “<<” propio de C++.

Finalmente, dado que empezamos este ejemplo a partir de la solución exacta de la ecuación de Poisson, podemos calcular el error obtenido con el MEF. Este cálculo lo realizaremos de dos maneras: mediante la norma L^2 y el error en los vértices de la malla.

Para el primer caso, almacenamos en la variable `error_L2` el resultado de la función `errornorm`, que calcula la norma L^2 definida como:

$$E = \sqrt{\int_{\Omega} (u_D - u)^2 \, dx} \quad (1.25)$$

Dado que la solución exacta es cuadrática, mientras que la solución por MEF es lineal por tramos, el error será distinto de cero.

Para el segundo caso, debemos recordar que el error en todos los vértices de la malla debería ser nulo con precisión de punto flotante de la máquina. Para calcularlo evaluamos la función exacta y la obtenida por FEniCS en todos los vértices de la malla (usando el método `compute_vertex`, obtenemos el valor absoluto de la diferencia en cada punto, y luego buscamos el máximo de estos valores. Estas operaciones se realizan en las líneas 47 – 49.

Ejecutando el programa obtenemos:

```
> ./fem.py
Solving linear variational problem.
Error L2 = 0.003651483716703487
Error máximo = 6.661338147750939e-15
```

Podemos ver que, efectivamente, el máximo error en los nodos es del orden del error de máquina.

En la figura 1.6 podemos ver la solución obtenida por MEF usando FEniCS. Si la ecuación de Poisson representa un problema térmico, los valores de $u(x, y)$ muestran la distribución de temperaturas en la placa cuadrada. En la vista tridimensional puede apreciarse que se cumplen las condiciones de borde, al presentar una variación lineal de la temperatura en los lados $(x, 0)$ y $(x, 1)$, y una variación cuadrática en los lados $(y, 0)$ y $(y, 1)$.

El ejemplo que mostramos aquí es solo una introducción básica a la resolución de EDP usando el método de elementos finitos con el *framework* FEniCS. Queremos destacar que este software ofrece la posibilidad de representar un problema enunciado en términos de EDP de una forma muy cercana a la formulación matemática. Por supuesto que los problemas reales son significativamente más complejos que el que mostramos aquí, incluyendo geometrías complejas, multifísica, diferentes condiciones de borde, etc. Sugerimos al lector interesado las lecturas de la sección siguiente.

1.3. Lecturas recomendadas

- La categorización de las ecuaciones en derivadas parciales y técnicas de solución analíticas se puede ver en A.N. Tjonov y A.A. Samarsky. *Ecuaciones de la Física Matemática*. Editorial

⁹<https://www.paraview.org/>

¹⁰<https://vtk.org/>

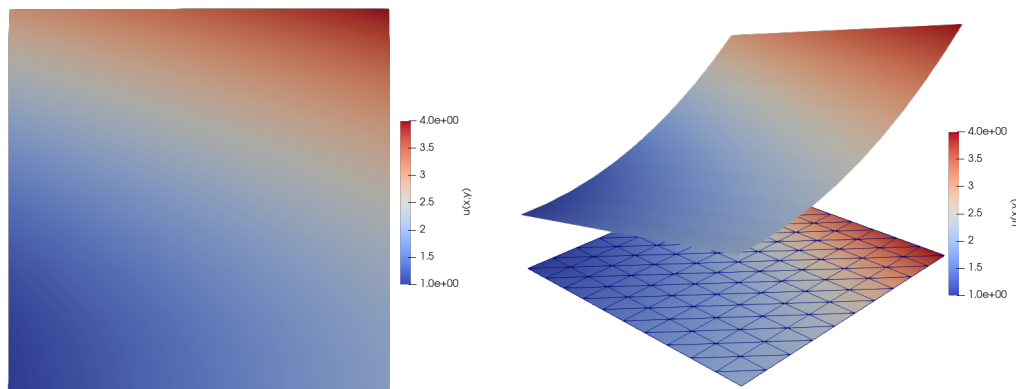


FIGURA 1.6: Visualización de la solución del problema de Poisson con condiciones de borde. A la izquierda una vista superior de la placa con la distribución de valores de $u(x, y)$. A la derecha una representación tridimensional de la solución, con la visualización de la malla generada por FEniCS.

Mir, Moscú, 1983.

- Para ver un análisis detallado de las inestabilidades numéricas del esquema de Euler hacia adelante se puede ver la sección 3.3 de Hans Petter Langtangen y Svein Linge. *Finite Difference Computing with PDEs*. Springer International Publishing, 2017. DOI: [10.1007/978-3-319-55456-3](https://doi.org/10.1007/978-3-319-55456-3). URL: <https://doi.org/10.1007/978-3-319-55456-3>. Además, este libro presenta el abordaje en diferencias finitas para una amplia variedad de problemas en EDP.
- Una buena introducción al método de elementos finitos puede leerse en M.G. Larson y F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Texts in Computational Science and Engineering. Springer Berlin Heidelberg, 2013
- El tutorial de FEniCS permite una introducción completa al uso del *framework* con muchos ejemplos de aplicación: Hans Petter Langtangen y Anders Logg. *Solving PDEs in Python*. Springer, 2017. DOI: [10.1007/978-3-319-52462-7](https://doi.org/10.1007/978-3-319-52462-7). Este tutorial acompaña a Anders Logg, Kent-Andre Mardal, Garth N. Wells y col. *Automated Solution of Differential Equations by the Finite Element Method*. Ed. por Anders Logg, Kent-Andre Mardal y Garth N. Wells. Springer, 2012. DOI: [10.1007/978-3-642-23099-8](https://doi.org/10.1007/978-3-642-23099-8) que incluye otros aspectos del diseño del *software* y la matemática necesaria.

Parte III
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [8].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tenemos más de esos!

Bibliografía

- [1] URL: <https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>.
- [2] A.N. Tijonov y A.A. Samarsky. *Ecuaciones de la Física Matemática*. Editorial Mir, Moscú, 1983.
- [3] Hans Petter Langtangen y Anders Logg. *Solving PDEs in Python*. Springer, 2017. DOI: [10.1007/978-3-319-52462-7](https://doi.org/10.1007/978-3-319-52462-7).
- [4] Douglas N Arnold y Anders Logg. «Periodic table of the finite elements». En: *SIAM News* 47.9 (2014), pág. 212.
- [5] Hans Petter Langtangen y Svein Linge. *Finite Difference Computing with PDEs*. Springer International Publishing, 2017. DOI: [10.1007/978-3-319-55456-3](https://doi.org/10.1007/978-3-319-55456-3). URL: <https://doi.org/10.1007/978-3-319-55456-3>.
- [6] M.G. Larson y F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Texts in Computational Science and Engineering. Springer Berlin Heidelberg, 2013.
- [7] Anders Logg, Kent-Andre Mardal, Garth N. Wells y col. *Automated Solution of Differential Equations by the Finite Element Method*. Ed. por Anders Logg, Kent-Andre Mardal y Garth N. Wells. Springer, 2012. DOI: [10.1007/978-3-642-23099-8](https://doi.org/10.1007/978-3-642-23099-8).
- [8] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: <https://www.python.org/dev/peps/pep-0020/>.