CÁLCULO NUMÉRICO Y ESTADÍSTICA APLICADA. PARTE 1: INTRODUCCIÓN AL CÁLCULO NUMÉRICO.

26 de septiembre de 2025

Jaime Arturo de la Torre
Departamento de Física Fundamental, UNED

César Fernández Ramírez
Departamento de Física Interdisciplinar, UNED

Universidad Nacional de Educación a Distancia

ÍNDICE GENERAL

Introducción **13**

1. Fundamentos de programación **15**

1. Introducción 15
2. Variables 16
3. Operaciones básicas 21
4. Estructuras 26
5. Funciones 31
6. Ejercicios 32

2. Sistemas de ecuaciones **33**

1. Introducción 33
2. Método de Gauss 33
3. Factorización LU 38
   - 3.1. Método de Crout 39
   - 3.2. Descomposición de Cholesky 40
4. Métodos iterativos 41
5. Diagonalización 44
   - 5.1. Método iterativo de Jacobi para matrices reales simétricas 45
6. Ejercicios 48

3. Interpolación **51**

1. Introducción 51
2. Interpolación lineal 51
3. Interpolación polinómica 53
4. Ejercicios 56

4. Resolución de ecuaciones no lineales **59**

1. Introducción 59
2. Método de la bisección 60
3. Método de Newton 65
4. Método de la secante 69
5. Regula falsi 71
6. Ejercicios 73

5. Derivación e integración numérica **77**

1. Introducción 77
2. Derivación numérica 78
   - 2.1. Fórmulas de dos puntos 78
   - 2.2. Fórmulas de tres puntos 80
3. Integración numérica 81
4. Métodos de cuadratura 84
5. Ejercicios 89

6. Soluciones numéricas de ecuaciones diferenciales ordinarias **91**

1. Introducción 91
2. Ecuaciones diferenciales ordinarias 91

2.1. Método de Euler 91
2.2. Métodos predictor-corrector 98
2.3. Método de Runge-Kutta de cuarto orden 100
3. Ejercicios 101

7. Soluciones a los ejercicios **103**

Soluciones de los ejercicios **104**

Bibliografía **105**

INTRODUCCIÓN

Muchos problemas científicos que aparecen en Química pueden resolverse de manera exacta o aproximada haciendo un uso adecuado de las matemáticas. Ejemplos sencillos son las relaciones estequimétricas, el área bajo una curva o la trayectoria que seguirá un proyectil; más complejos son la predicción meteorológica o la modelización molecular. Algunos de los primeros se pueden resolver obteniéndose soluciones analíticas haciendo uso de álgebra y cálculo. Otros requieren el uso de métodos numéricos para la obtención de las soluciones, ya sean estas exactas o aproximadas.

Los objetivos de esta parte de la asignatura son por un lado dotar al estudiante de la comprensión de los fundamentos de los métodos numéricos de cálculo que en la actualidad son ubicuos en la ciencia y por otro conocer y saber aplicar algunos de los diferentes métodos numéricos para resolver problemas típicos como son la resolución de ecuaciones y sistemas de ecuaciones, la diagonalización, la interpolación, la estimación de derivadas, las integrales definidas y la resolución de ecuaciones diferenciales ordinarias. Como libros de referencia de métodos numéricos se recomiendan [1, 3, 4, 8, 9], para fórmulas matemáticas [10] y para programación [2, 5, 6, 7].

En el primer capítulo se desarrollaran las nociones básicas de los lenguajes de programación de alto nivel, incluyendo los tipos de datos y estructuras más usuales. El capítulo 2 cubre los conocimientos necesarios para resolver numéricamente sistemas de ecuaciones lineales, realizando descomposiciones de matrices y utilizando métodos iterativos para hallar las soluciones. El tercer capítulo explica los métodos básicos de interpolación lineal y polinómica. El capítulo 4 cubre los métodos de resolución de ecuaciones no lineales, obteniéndose de forma numérica los ceros de una función haciendo uso de distintos métodos iterativos.

El capítulo 5 se ocupa de la derivación e integración numérica y finalmente el capítulo 6 cubre la resolución de ecuaciones diferenciales ordinarias.

Tema 1

FUNDAMENTOS DE PROGRAMACIÓN

1. INTRODUCCIÓN

Para poder afrontar problemas que por su naturaleza no permiten la aplicación de métodos analíticos pero que pueden ser encarados mediante el uso de métodos numéricos hace falta hacer uso de ordenadores y eso implica ser capaz de programar. Por ello, el primer paso en necesario aprender los fundamentos básicos de los lenguajes de programación, que nos permitirán traducir instrucciones humanas sencillas en un lenguaje que entiendan los ordenadores. Para ello es necesario entender, de forma somera, cómo funcionan en general los lenguajes de programación, qué tipo de datos podemos introducir en un programa, cómo operar con estos datos y, finalmente, cómo conseguir que el ordenador nos muestre la solución que buscamos.

El número de lenguajes de programación existentes y las particularidades de cada uno de ellos es inmenso. Si bien es posible utilizar cualquiera de ellos, en este curso vamos a centrarnos en el lenguaje de programación Python y algunas veces recurriremos a C. La elección de Python radica en que es un lenguaje con una curva de aprendizaje suave y que se utiliza ampliamente en el cómputo científico.

Algunos recursos para aprender más sobre Python son:

- `https://www.codecademy.com/learn/python`
- `https://campus.datacamp.com/courses/intro-to-python-for-data-science/`
- `https://developers.google.com/edu/python`

Para aquellos lectores que deseen instalarse Python en sus respectivos ordenadores, basta saber que tanto Linux como MacOS suelen incluir, de serie, un compilador de Python que puede ejecutarse desde cualquier terminal

del sistema sin más que escribir desde la línea de comandos de un terminal 'python'. Sin embargo, se recomienda, independientemente del sistema operativo, la instalación de Anaconda, el cuál funciona en Windows, MacOS y Linux e incluye la mayor parte de paquetes que serán de utilidad en este curso:

- `https://www.anaconda.com/download/`

Incorporar un paquete nuevo en Anaconda es relativamente sencillo.

2. VARIABLES

Los lenguajes de programación permiten interaccionar con distintos tipos de elementos, también llamados *variables*. Aquí, las posibilidades de interacción son diversas. Puede por supuesto operarse con los elementos (realizar una suma o una multiplicación, calcular el resto de una división, obtener la parte entera, etc.), pero también es posible realizar acciones más diversas: concatenar palabras, "leer" el contenido de una oración, etc.

Más información:

Una de las primeras dificultades de la que un programador debe ser consciente es que, en general, los ordenadores no se equivocan. Es el programador quien comete errores que hacen que la salida de un programa no sea la esperada. Esto se refleja con frecuencia en el lema *garbage in, garbage out*. Si los datos que introducimos en un programa son erróneos, cabe esperar que la salida sea también errónea. Por ello es esencial saber, en todo momento, qué tipo de variables le introducimos a un programa, con qué tipo de variables trabaja en su interior y qué tipo de variables obtiene como salida. Reconocer la diferencia entre un tipo de variable y otra nos ahorrará futuros problemas.

Usualmente[1] para poder trabajar con variables es necesario declarar su tipo

[1] Python es una excepción.

previamente. De este modo el ordenador sabrá qué espera obtener cuando recurrimos a ella. La sintaxis más frecuente consiste en la estructura `tipo NombreVariable`, donde `tipo` especifica la clase de variable que vamos a definir y `NombreVariable` es el nombre que le damos a ella. De este modo, una vez declarada una variable llamada, por ejemplo, *foo*, basta usar ese nombre para referirnos a la misma a lo largo de todo el código.

Si bien existen multitud de tipos de variables específicas de muchos lenguajes de programación, las estructuras más básicas se presentan a continuación.

Variable booleana

Una variable de tipo *bool* es, en esencia, un interruptor. Puede tomar dos únicos valores, llamados generalmente VERDADERO y FALSO. En algunos lenguajes de programación se define el valor VERDADERO como un '1' y el valor FALSO como un '0', por analogía con la lógica binaria. Este tipo de dato será extremadamente útil cuando veamos estructuras condicionales o selectivas, ya que permitirán actuar de una u otra forma en función de si una cierta variable es verdadera o falsa.

```
1  bool MyBool;
2  MyBool=true;
3
```

Código 1.1: Tipo de dato booleano en C-99. Observamos aquí que declaramos el tipo de variable (**`bool`**) y después el nombre que toma la variable, *MyBool*. Posteriormente veremos que en este código estamos haciendo además una asignación del valor que toma la variable *MyBool*, en este caso, verdadero.

Variable carácter

Una variable de tipo carácter (en inglés, *char*), en su forma más simple, es una letra. Aquí, por "letra" entendemos cualquier pulsación del teclado que origine un símbolo: 'a', 'A', '$', '2', '.', o incluso '". Podemos considerar como carácter cualquier tipo de dato que tenga la consideración de texto. Notemos que existe una distinción entre mayúsculas y minúsculas, por lo

que en general no es lo mismo el carácter ‘b’ que el carácter ‘B’.

```
1 char MyChar;
2 MyChar='a';
3
```

Código 1.2: Tipo de dato carácter en C. Para distinguir una letra de un número, los datos de tipo carácter se encierran entre comillas (simples o dobles, dependiendo del lenguaje).

Variable cadena

Aunque un conjunto de carácteres sigue siendo un tipo de dato carácter, algunos lenguajes de programación tienen un tipo de dato propio para carácteres de longitud mayor de la unidad. Así, ‘Hola Mundo’ podría ser un dato de tipo cadena.

```
1 string MyString;
2 MyString='Hola Mundo';
3
```

Código 1.3: Tipo de dato cadena en C. Como hemos dicho anteriormente, `'Hola Mundo'` es una cadena distinta de `'hola mundo'`.

Variable entera

Una variable entera (en inglés, *integer*) es un tipo de dato numérico que, como su nombre indica, almacena el valor entero de un número. Así, 42, -128, 31250, o 2500000 son todos datos de tipo entero[2]

```
1 int MyInt;
2 MyInt=42;
3
```

[2] Un tipo de dato *integer* puede almacenar tanto el valor de un número entero (de forma exacta) como la parte entera de un número racional (de forma aproximada). Dependiendo de la implementación del lenguaje de programación usado, la aproximación puede ser por defecto, por exceso, o un mero truncamiento del número a su parte entera.

Código 1.4: Tipo de dato entero en C. Notemos que aquí no escribimos `'42'`, por ser un número y no una cadena de carácteres.

Variable coma flotante

Una variable en coma flotante (en inglés, *float*) es, en general, un número real con una cierta cantidad finita de cifras decimales. 3.1415926, 1.65, -3/4, o 0.0 son datos de tipo coma flotante. Notemos que los números enteros también forman parte del conjunto de los números reales, por lo que 42, -128, 31250 y 2500000 pueden ser también números en coma flotante

Es importante remarcar la existencia de una cantidad **finita** de cifras decimales para las variables en coma flotante. Esto tiene implicaciones importantes ya que, por ejemplo, si definimos un número en coma flotante como

```
1 float MyFloat;
2 MyFloat=1.0/6.0;
3
```

Código 1.5: Tipo de dato en coma flotante en C.

nos encontraremos con que si imprimimos el valor de *MyFloat* el resultado es 0.16666667, que no es más que una aproximación al verdadero valor del resultado de la expresión $1/6 = 0{,}166666666\ldots$. Si definimos la variable $\pi = 4\,\mathrm{atan}\,(1{,}0)$, nos encontraremos con que $\pi = 3{,}1415927$, que es de nuevo una aproximación al valor de π.

Vectores

Por analogía con el álgebra, un tipo de dato vector (*array*) no es más que un contenedor de variables. Así, podemos definir un vector de números enteros, un vector de datos tipo doble, etc. Una cadena de carácteres, de hecho, no es más que un vector de carácteres de una longitud determinada.

```
1  int MyArray[4];
2  MyArray[0]=1;
3  MyArray[1]=2;
4  MyArray[2]=3;
5  MyArray[3]=4;
6
```

Código 1.6: Definiendo un vector de 4 enteros (**int**) en C.

Notemos que del vector de cuatro componentes $(1, 2, 3, 4)$ nos referimos en general a la primera componente con el índice 0 y a la última con el índice 3. Así, el último elemento de un vector de N componentes será siempre el $N - 1$.[3]

Podemos también definir matrices como vectores bidimensionales, refiriéndonos a sus elementos por filas y columnas. Por ejemplo, la matriz

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{1.1}$$

puede definirse en C como

```
1  int MyArray[2][2];
2  MyArray[0][0]=1;
3  MyArray[0][1]=2;
4  MyArray[1][0]=3;
5  MyArray[1][1]=4;
6
```

Código 1.7: Definiendo una matriz de 4 enteros (2 filas y 2 columnas) en C.

Antes de finalizar esta sección es importante reseñar que muchos lenguages de programación, como python, octave, etc., son capaces de interpretar el tipo de dato a partir del contenido, sin necesidad de declarar previamente qué tipo de datos vamos a introducir. Esto presenta algunas comodidades de cara al programador, que no necesita preocuparse de estas cuestiones

[3] Algunos lenguajes de programación, como octave, identifican el primer elemento de un vector de tamaño N con el índice 1 y el último con N.

y puede enfocarse directamente en la programación. Así, por ejemplo, en python,

```
>>> a=2
>>> b=1.0/6.0
>>> c=[1,2,3]

```

Código 1.8: Asignando algunos elementos en Python.

se definen, respectivamente, un entero, un número en coma flotante, y un vector de tres componentes enteras. En lo sucesivo, dado que usaremos Python como lenguaje de programación paradigmático, prescindiremos de la definición del tipo de datos.

3. OPERACIONES BÁSICAS

Decíamos al inicio de la sección anterior que con las variables de un programa se pueden realizar multitud de operaciones, desde las más elementales como la suma o la resta hasta operaciones complicadas no solo de matemáticas, sino de prácticamente cualquier área de conocimiento.

La primera de las operaciones básicas, que ya hemos utilizado aunque no definido, es la asignación. Decíamos que en general antes de utilizar una variable que llamamos, por ejemplo, *foo*, es necesario declararla. Una vez realizada la declaración podemos hacer uso de ella y asignarle un valor determinado. Cuando escribimos, por ejemplo,

```
>>> foo=3
```

Código 1.9: Declarando variables en Python (`declaraciones.py`).

estamos asignándole a la variable *foo* el valor '3'. De este modo, cada vez que nos refiramos a la variable el ordenador, internamente, sabrá que nos estamos refiriendo al valor que toma la variable (en este caso, '3'). Podemos comprobar este hecho pidiéndole al código que nos imprima (**`print`**) el valor que toma la variable *foo*

```
>>> print(foo)
```

```
3 3
```

Código 1.10: Declarando variables en Python (`declaraciones.py`).

Una vez una variable está definida, nada nos impide reasignarle un nuevo valor, que quedará almacenado en memoria esperando su uso

```
4 >>> var=2
5 >>> print(foo)
6 3
7 >>> foo=4
8 >>> print(foo)
9 4
```

Código 1.11: Declarando variables en Python (`declaraciones.py`).

Una vez vista la operación básica de asignación, podemos ir más allá con otro tipo de operaciones. Supongamos dos variables (de tipo entero) que llamaremos a y b, que toman los valores $a = 3$ y $b = 4$. Esto, como ya hemos visto, puede escribirse simplemente como

```
1 >>> a=3
2 >>> b=4
```

Código 1.12: Operaciones básicas en python (`operaciones.py`).

Definamos ahora una nueva variable $c = a \times b$, escribiéndola directamente como

```
3 >>> c=a*b
4 >>> print(c)
5 12
```

Código 1.13: Operaciones básicas en python (`operaciones.py`).

donde notamos el símbolo `*` para referirnos a la multiplicación. De forma natural la suma, resta y división se denotan respectivamente mediante los símbolos `+`, `-` y `/`.

Podemos escribir variables algo más enrevesadas, como $d = a^2 + b^2$, sin más que escribir

```
>>> d=a**2+b**2
>>> print(d)
25
```

Código 1.14: Operaciones básicas en python (`operaciones.py`).

donde notamos que para representar la potenciación escribimos `**` (algunos lenguajes de programación utilizan `^`).

Una última operación básica en programación es el módulo de una división, normalmente indicado por `mod` o el símbolo `%`. El módulo de una división no es más que el resto del cociente entre dos números. Así, dados los números 4 y 2, la operación `4%2` (o `mod(4,2)`) da como resultado `0`, ya que al dividir 4 entre 2 el resto de la división es cero. Del mismo modo se puede calcular, por ejemplo, `5%3`, dando como resto 2.

Estas (y algunas más) son las operaciones más básicas de las que dispone cualquier lenguaje de programación. Otras operaciones más complicadas, como podría ser la raíz cuadrada, funciones trigonométricas, etc., pueden también realizarse. Pero, para ello, es necesario indicar al lenguaje de programación dónde se ubican las definiciones de estas operaciones. La forma de hacerlo en python (y en general en cualquier otro lenguaje) es *importando* paquetes que incluyen las operaciones que deseamos. En el caso de python, por ejemplo, casi todas las operaciones matemáticas interesantes están bajo el paquete `math`, que puede importarse usando la instrucción

```
>>> from math import *
```

Código 1.15: Operaciones básicas en python (`operaciones.py`).

que le indica a python que, del paquete `math`, debe importar todo su contenido. Podemos entonces definir una nueva función $e = \sqrt{d}$ que nos devolvería, si rescatamos las definiciones iniciales, la hipotenusa de un triángulo rectángulo de catetos $a = 3$ y $b = 4$:

```
>>> e=sqrt(d)
>>> print (e)
5.0
```

Código 1.16: Operaciones básicas en python (`operaciones.py`).

Más información:

Podemos fijarnos en que, ahora, la salida aparece como 5.0 en lugar de 5. Sin entrar en muchas consideraciones, es importante notar que hasta este momento estábamos trabajando con número enteros y que, al usar la raíz cuadrada, python ha transformado el resultado en números en coma flotante.

Ejemplo 1.1. Muestre el valor del número π con 2, 4 y 6 decimales a partir de la definición $\pi = 4\text{atan}(1)$.

Solución:

Utilizando un lenguaje como Python podemos escribir directamente

```
>>> from math import *
>>> p=4*atan(1.0)
>>> print(p)
3.141592653589793
```

Código 1.17: Cómo obtener el número π (`pi.py`).

Python permite mostrar un número de cifras decimales concreto utilizando una versión modificada del comando **`print`**, sin más que escribir

```
>>> print(f'El valor de pi es aproximadamente {p:.2f}.')
El valor de pi es aproximadamente 3.14
>>> print(f'{p:.4f}')
3.1416
>>> print(f'{p:.6f}')
3.141593
```

Código 1.18: Cómo obtener el número π (`pi.py`).

esta instrucción debe leerse como que le estamos pidiendo a python que nos muestre el valor de p en un formato específico y el tipo de formato lo indica en este caso la letra `f` (de *float*, coma flotante).

Podemos ampliar la información que muestra el programa sin más que seguir escribiendo dentro de las comillas

```
>>> print(f'PI con dos decimales es {p:1.2f}.')
PI con dos decimales es 3.14
```

Código 1.19: Cómo obtener el número π (`pi.py`).

o incluso mostrar todos los resultados en una sola línea

```
>>> print(f'PI con 4 y 6 decimales es {p:1.4f} {p:1.6f}.')
PI con 4 y 6 decimales es 3.1416 y 3.151953
```

Código 1.20: Cómo obtener el número π (`pi.py`).

Lo que se escribe después de ":." es el formato en el que se presentará el resultado. Si se escribe un entero, ese es el número de caracteres que se reserva. Por ejemplo, si se pone print (f'p:.2f.) quiere decir que se utilizarán 2 caracteres. Lo que va después del punto (".") es el número de decimales. Por ejemplo, si se escribe print (f'p:5.2f.) quiere decir que se escribirán 5 caracteres, de los cuáles 2 serán reservados para los decimales.

Ejemplo 1.2. Dados dos números $a = 3{,}4$ y $b = 12{,}812$, obtenga el valor de $c = a\sin(b/(2\pi))$ y muestre en pantalla el mensaje `Si a=$a y b=$b entonces c=$c`, donde `$a`, `$b` y `$c` son los valores que toman las variables a, b y c, respectivamente.

Solución:

```
    from math import *
    p = 4.*atan(1.)
    a = 3.4
    b = 12.812
    c = a*sin(b/(2.*p))
    print ("si a=",a,"y b=",b,"entonces c=",c)

```

Código 1.21: Solución al Ejemplo 1.2.

4. ESTRUCTURAS

Una vez conocidos los principales tipos de variables y las operaciones más básicas, podemos trabajar con ellos a través de estructuras. Estas se dividen en tres categorías esenciales.

Estructuras secuenciales

Una estructura secuencial es una sucesión de instrucciones que realizará nuestro programa siguiendo el orden lógico de aparición. Esto es lo que hemos estado haciendo hasta ahora por ejemplo en el cód. 1.17, donde ejecutábamos, una a una, las instrucciones de importar una librería de matemáticas, definir una variable y asignarle un valor, e imprimir el valor de la variable calculada. Las estructuras secuenciales son el pilar básico de la programación, pues describe cómo se van a ir ejecutando las instrucciones que aparecen en un código.

Estructuras selectivas

Una estructura selectiva (también llamada condicional) es aquella que se ejecuta si se cumplen unas determinadas condiciones. Podemos plantear el ejemplo de estructura secuencial en la que se escoge un número al azar y se establece que, si es menor que un determinado valor, se ejecutará una acción determinada. Estas estructuras son básicas en prácticamente todos los lenguajes de programación y se articulan, en su forma más básica, en un esquema *if-then-else*. Supongamos que vamos a definir una variable a y que queremos determinar si es un número par o impar. Sabemos que si el número es par entonces al dividirlo entre dos el resto de la división es cero, por lo que si `a%2=0` entonces a es un número par. En cualquier otro caso podemos afirmar que el número es impar[4].

[4] En adelante prescindiremos del símbolo `>>>` para representar la consola de Python. El lector puede utilizar su procesador de texto plano favorito, escribir el código, y ejecutarlo en Python usando simplemente la instrucción `python micodigo.py`

```
1       a=5
2       if a%2==0:
3         print ("a es par")
4       else:
5         print ("a es impar")
6
```

Código 1.22: Cómo determinar si un número es par.

El cód. 1.22 muestra cómo escribir en python una estructura selectiva. La condición a cumplir se declara con la instrucción **`if`**, que admite como argumento un tipo de dato booleano (verdadero o falso). El interior del paréntesis establece una comparación, con el símbolo `==` que representa igualdad. Este paréntesis se pregunta si `a%2` es exactamente igual a `0`. Si esto es cierto, entonces el paréntesis es verdadero y se cumplen las condiciones para ejecutar el código selectivo que aparece tras el símbolo `:`. En este caso el bloque de código es una única instrucción que imprime la cadena de carácteres "a es par".

Si la comparación del módulo entre a y 2 y cero es falsa, entonces el código *salta* el bloque verdadero y ejecuta el bloque tras la instrucción **`else`**:, en este caso, imprimiendo el mensaje "a es impar".

En general una estructura *if-then-else* tiene la siguiente forma

```
if (CONDICIÓN1) then
  INSTRUCCIONES A EJECUTAR SI CONDICIÓN1 ES VERDADERA
else if (CONDICIÓN 2) then
  INSTRUCCIONES A EJECUTAR SI CONDICIÓN1 ES FALSA Y
  CONDICIÓN2 ES VERDADERA
else
  INSTRUCCIONES A EJECUTAR SI CONDICIÓN1 ES FALSA Y
   CONDICIÓN2 ES FALSA
```

donde los bloques **`else if`** y **`else`** son en general opcionales, por lo pueden aparecer o no en una determinada estructura condicional.

Ejemplo 1.3. Plantee un código que imprima en pantalla, dado un número a, si es divisible por 2 o por 3.

Solución:

```
1     a=33
2     if a%2==0:
3       print ("a es divisible entre 2")
4     elif a%3==0:
5       print ("a es divisible entre 3")
6     else:
7       print ("a no es divisible ni entre 2 ni entre 3")
8
```

Código 1.23: Cómo comprobar si un número es divisible entre 2 o entre 3. La instrucción *elif* permite hacer comparaciones sucesivas.

Este código, en su línea 2, pregunta (**if**) si el número a es divisible entre 2. Si lo es, imprimirá el mensaje adecuado. Si no lo es, (línea 4) se preguntará (**elif**) si no cumpliendo la condición anterior sí cumple que a es divisible entre 3. Si lo es imprimirá el mensaje. Finalmente, si no cumple ninguna de las dos condiciones entonces ejecutará el bloque **else**.

Hasta ahora hemos planteado estructura condicionales sencillas pero podemos, por supuesto, aumentar la complejidad de las mismos. Supongamos que queremos saber si un número es divisible entre 2 y, además, que sea menor que 10. Esto puede plantearse con una condición doble dada por el operador comparativo **and**, denotado a veces con el símbolo `&&`. Así, la pregunta que nos hacemos en este caso sería `if ((a%2==0) and (a<10))` donde, notamos, es importante el uso de paréntesis para indicar cuáles son las dos comparaciones que se están realizando.

Otros operadores de comparación son el **or**, denotado a veces con `||` y el operador **not**, escrito como `!`, que es el operador de negación. El funcionamiento de estos tres operadores condicionales se describe en las llamadas tablas de verdad tab. 1.1, tab. 1.2 y tab. 1.3. Una tabla de verdad se divide en dos zonas separadas. A la izquierda se muestran las entradas de la condi-

ción que se va a imponer. Estas entradas pueden tomar el valor `VERDADERO` (1) o `FALSO` (0). A la derecha se muestra el resultado de la operación correspondiente, también en forma de `VERDADERO` o `FALSO`.

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

Tabla 1.1. Tabla de verdad para el operador **`and`**. Solo si las dos entradas son verdaderas el resultado es verdadero.

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

Tabla 1.2. Tabla de verdad para el operador **`or`**. Bata que una de las dos condiciones sea veradera para que el resultado sea verdadero.

A	not A
0	1
1	0

Tabla 1.3. Tabla de verdad para el operador **`not`**. Si A es verdadera, **`not`** A será falsa. Si A es falsa, **`not`** A será verdadera.

Estructuras iterativas

Las estructuras iterativas son aquellas que se repiten indefinidamente mientras se cumpla una determinada condición. Supongamos que queremos saber cuáles de los primeros diez números naturales son divisibles entre dos. Podríamos hacer un programa que repitiera el cód. 1.22 para los números

1, 2, 3, 4, 5, 6, 7, 8, 9, y 10. O podríamos hacer una estructura iterativa que simplificara la complejidad del código.

```
1 for x in range(1,11):
2     if (x%2==0):
3         print(f'{x:2d} es divisible entre 2.')
4     else:
5         print(f'{x:2d} no es divisible entre 2.')
```

Código 1.24: Estructura iterativa para ver si un número es divisible entre 2 (`iteraciones1.py`). Observamos que para referirnos a números enteros no usamos el formato `:.f` sino `:d`.

El cód. 1.24 muestra una estructura iterativa basada en un bucle **`for`**. La instrucción **`range`** se expande a todos los elementos que vayan desde 1 hasta 11 (sin incluir el 11), de forma que el bucle **`for`** hace que la variable x tome cada uno de los valores de la expansión y, para cada uno de ellos, se ejecute la estructura selectiva para ver si es divisible o no entre 2.

Esta estructura iterativa tiene la contrapartida de ser poco flexible, pues considera para la variable iterada únicamente los miembros de la lista definida a priori. Una estructura más flexible es el bucle **`while`**, que ejecuta una serie de instrucciones mientras se verifique una condición, analizando en cada iteración la validez (o no) de la condición impuesta. Supongamos que queremos calcular el mayor exponente x tal que $2^x < 10^6$. El código cód. 1.25 muestra, en apenas 7 líneas, cómo hacerlo.

```
1 x=2
2 i=0
3 while x<1000000:
4     i+=1
5     x=x*2
6     print( f'2^({i:d}) = {int(x/2):d} < 10^6')
7 print('Encontrado el mayor exponente x tal que 2^x < 10^6')
```

Código 1.25: Estructura iterativa para encontrar exponentes (`iteraciones2.py`). Notamos en el código (línea 4) la estructura `i+=1`, que es una forma compacta de decir `i=i+1`. Esta operación, que será muy frecuente a lo largo de estas próximas lecciones, nos indica que el valor de la variable `i` se incrementa en una unidad cada vez que se ejecuta.

5. FUNCIONES

En matemáticas sabemos que una función $f(x)$ es aquella aplicación que devuelve, para un valor de x_i dado, un valor $y_i = f(x_i)$. Si $f(x) = x^2$, entonces para $x = 3{,}5$ tenemos $y(x) = 12{,}25$. Este concepto puede ampliarse a lenguajes de programación, donde podemos definir funciones que devuelvan (o no) valores. El potencial de las funciones es que podemos crearlas de forma que devuelvan no solo números, sino cualquier tipo de datos que queramos. Pensemos en el cód. 1.22, donde devolvíamos, para un valor de a determinado, si este era par o impar. Podemos generalizar este código y definirlo como una función de modo que podamos usarla para cualquier valor de a.

```
1 def ParImpar(x):
2     if (x%2==0):
3         return "%d es par"    % (x)
4     else:
5         return "%d es impar" % (x)
6
7 for x in range(5,10):
8     print (ParImpar(x))
```

Código 1.26: Definiendo funciones en python (`funciones.py`). Las línea 7 y 8 harán que se muestre, para todos los números entre 5 y 9 (ambos incluidos) si estos son par o impar.

El cód. 1.26 es un ejemplo de cómo se define una función en python. Comenzamos con la instrucción **`def`** que nos permite identificar qué tipo de dato (en este caso, una función) estamos definiendo. Esta función puede tener ninguno, uno o varios argumentos de entrada (en este caso, solo un argumento de entrada dado por la variable x). Tras el símbolo ':' se inicia el contenido de la función, donde podemos utilizar las mismas estructuras (secuenciales, selectivas, iterativas) que hemos visto anteriormente. Al final, la función devuelve (con **`return`**) un tipo de dato también (en este caso, una cadena) que podremos imprimir posteriormente en nuestro código.

```
1 from math import *
2
3 def f(k,x):
```

```
4       return cos(k*x)
5
6   def dfdx(k,x):
7       return -k*sin(k*x)
8
9   L=10.0
10  k0=2*pi/L
11  x=0.0
12
13  while (x <= 2*L):
14      print (f'{f(k0,x):1.4f}  \t {dfdx(k0,x):1.4f}')
15      x+=0.1
```

Código 1.27: Un ejemplo más avanzado de funciones (`funciones2.py`).

El cód. 1.27 da una estructura algo más compleja de la definición de funciones. Definimos aquí dos funciones $f(x) = \cos(kx)$ y su derivada $f'(x) = -k\sin(kx)$. Dado que ambas funciones utilizan funciones trigonométricas necesitamos importar anteriormente la librería matemática de python. Una vez están hechas las definiciones, damos algunos valores a los parámetros que vamos a utilizar (L, k y un valor inicial para x). Con esta información iniciamos un bucle desde $x = 0$ hasta $x = 2L$, donde imprimimos, en forma de tabla, los valores que toman tanto la función como su derivada.

6. EJERCICIOS

Ej. 1.1 — Si definimos, tras la ejecución del cód. 1.11, `foo=var`, ¿qué mostrará la instrucción **`print`** (`foo`)?

Ej. 1.2 — El código mostrado en el ejemplo 1.3 es claramente incompleto. ¿Qué ocurriría para $a = 24$? Escriba un nuevo código que permita discernir si un número es divisible a la vez entre dos y entre tres. Pista: dentro de las "instrucciones a ejecutar si condición es verdadera" nada le impide introducir (*anidar*) una nueva estructura selectiva.

Tema 2

SISTEMAS DE ECUACIONES

1. INTRODUCCIÓN

Un sistema de ecuaciones lineales puede escribirse como

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1\,, \tag{2.1}$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2\,, \tag{2.2}$$
$$\vdots \tag{2.3}$$
$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n\,, \tag{2.4}$$

o, en forma matricial,

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}, \tag{2.5}$$

que escribimos en forma compacta como

$$\mathbf{A}\mathbf{x} = \mathbf{b}\,. \tag{2.6}$$

La solución de esta ecuación es, siempre que $\mathbf{A}$ sea invertible,

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \tag{2.7}$$

En este tema veremos cómo encontrar los valores de $\mathbf{x}$ a partir del conocimiento de $\mathbf{A}$ y $\mathbf{b}$.

2. MÉTODO DE GAUSS

Consideremos cada una de las ecuaciones E_i:

$$E_i : a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n = b_i\,. \tag{2.8}$$

Para simplificar un sistema de ecuaciones $\{E_i\}$ podemos hacer uso de las siguientes operaciones

1. Multiplicar una ecuación E_i por una constante $\alpha \neq 0$, denotado por $(\alpha E_i) \to (E_i)$, manteniendo su posición[1].
2. Multiplicar una ecuación E_i por una constante y sumar otra ecuación E_j, denotado por $(\alpha E_i + E_j) \to (E_i)$.
3. Intercambiar las posiciones que ocupan las ecuaciones E_i y E_j, denotado por $(E_i) \leftrightarrow (E_j)$

Notemos que cualquiera de estas operaciones no modifica los valores de las incógnitas x_i, pues solo reescalamos, sumamos y restamos sobre la ecuación completa. En notación matricial, hacer operaciones sobre la ecuación i es equivalente a hacer operaciones sobre la fila i. De nuevo, ninguna de estas operaciones sobre las filas altera el valor de las incógnitas $x_1, x_2, \ldots, x_N$.

Construyamos una ecuación matricial aumentada

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & a_{1(n+1)} \\ a_{21} & a_{22} & \ldots & a_{2n} & a_{2(n+1)} \\ \vdots & & & & \\ a_{n1} & a_{n2} & \ldots & a_{nn} & a_{n(n+1)} \end{array}\right) \tag{2.9}$$

donde en esta matriz $n \times (n+1)$ hemos definido los elementos $a_{i(n+1)} \triangleq b_i$.

El **método de eliminación gaussiana con sustitución hacia atrás** consiste en realizar operaciones sobre la matriz de coeficientes aumentada de forma que $\mathbf{A}$ se transforme en una matriz triangular superior. Si conseguimos una matriz $\mathbf{A}'$ de la forma

$$\left(\begin{array}{ccccc} a'_{11} & a'_{12} & a'_{13} & \ldots & a'_{1n} \\ 0 & a'_{22} & a'_{23} & \ldots & a'_{2n} \\ 0 & 0 & a'_{33} & \ldots & a'_{2n} \\ \vdots & & & & \\ 0 & 0 & 0 & \ldots & a'_{nn} \end{array}\right) \tag{2.10}$$

[1] Formalmente esta operación consiste en: multiplicar la ecuación E_i (que ocupa la posición i) por una constante α y reemplazar la ecuación original E_i por la nueva ecuación multiplicada por α, manteniendo la posición i original. De ahí la notación $(\alpha E_i) \to (E_i)$.

donde obviamente el vector $\mathbf{b}$ se transformará igualmente en *otro* vector $\mathbf{b}'$, el sistema de ecuaciones $\mathbf{A}'\mathbf{x} = \mathbf{b}'$ tendrá, en su última fila, la ecuación

$$a'_{nn}x_n = b'_n, \tag{2.11}$$

de donde obtenemos el valor de x_n como

$$x_n = \frac{1}{a'_{nn}}b'_n. \tag{2.12}$$

Con este resultado podemos **sustituir hacia atrás** en la fila anterior, de forma que

$$a'_{(n-1)(n-1)}x_{n-1} + a'_{(n-1)n}x_n = b'_{n-1}\,, \tag{2.13}$$

o sea,

$$x_{n-1} = \frac{1}{a'_{(n-1)(n-1)}}\left(b'_{n-1} - a'_{(n-1)n}x_n\right). \tag{2.14}$$

Para la fila anterior tendremos

$$a'_{(n-2)(n-2)}x_{n-2} + a'_{(n-2)(n-1)}x_{n-1} + a'_{(n-2)n}x_n = b'_{n-2}\,, \tag{2.15}$$

o sea,

$$x_{n-2} = \frac{1}{a'_{(n-2)(n-2)}}\left(b'_{n-2} - a'_{(n-2)(n-1)}x_{n-1} - a'_{(n-2)n}x_n\right). \tag{2.16}$$

Este procedimiento permite así obtener secuencialmente todos los valores de las incógnitas x_i a partir de los posteriores:

$$x_i = \frac{1}{a'_{ii}}\left(b'_i - \sum_{j=i+1}^{N} a'_{ij}x_j\right). \tag{2.17}$$

El problema se reduce por tanto a encontrar una matriz $\mathbf{A}'$ triangular superior a partir de la matriz original $\mathbf{A}$. Para ello basta darse cuenta de que

tenemos que eliminar las componentes por debajo de la diagonal. Fijémonos por ejemplo en las ecuaciones (E_1), (E_2) y (E_3):

$$E_1 : a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + \ldots = b_1\,, \tag{2.18}$$
$$E_2 : a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + \ldots = b_2\,, \tag{2.19}$$
$$E_3 : a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + \ldots = b_3\,. \tag{2.20}$$

Si efectuamos la operación $(E_2 - a_{21}E_1/a_{11} \to E_2')$ llegamos a

$$E_2 : \left(a_{22} - \frac{a_{21}a_{12}}{a_{11}}\right)x_2 + \left(a_{23} + \frac{a_{21}a_{13}}{a_{11}}\right)x_3 + \ldots = b_2 - \frac{a_{21}}{a_{11}}b_1 \tag{2.21}$$
$$E_2' : a_{22}'x_2 + a_{23}'x_3 + a_{24}'x_4 + \ldots = b_2'\,. \tag{2.22}$$

Ejemplo 2.1. Realice las operaciones adecuadas sobre (E_3) para obtener una ecuación (E_3'') cuyos elementos a_{31}'' y a_{32}'' sean cero.

Solución:
Para la ecuación (E_3) podemos efectuar la operación $(E_3 - a_{31}E_1/a_{11} \to E_3')$ para eliminar su componente a_{31}, de forma que

$$E_3 : \left(a_{32} - \frac{a_{31}a_{12}}{a_{11}}\right)x_2 + \left(a_{23} + \frac{a_{31}a_{13}}{a_{11}}\right)x_3 + \ldots = b_3 - \frac{a_{31}}{a_{11}}b_1 \tag{2.23}$$
$$E_3' : a_{32}'x_2 + a_{33}'x_3 + a_{34}'x_4 + \ldots = b_3'\,. \tag{2.24}$$

Si ahora efectuamos la operación $(E_3' - a_{32}'E_2'/a_{22}' \to E_3'')$ conseguimos eliminar también la segunda componente a_{32}

Siguiendo el desarrollo del ej. 1 se observa que el procedimiento a seguir para eliminar la primera componente de la j-ésima ecuación es

$$\left(E_j - \frac{a_{j1}}{a_{11}}E_1 \to E_j'\right). \tag{2.25}$$

Una vez eliminada la primera componente de todas las $j > 1$ ecuaciones, la

segunda componente de (E_j) (para $j > 2$) se elimina con la operación

$$\left(E'_j - \frac{a'_{j2}}{a'_{22}} E'_2 \to E''_j\right). \tag{2.26}$$

De forma que la i-ésima componente de la j-ésima ecuación se eliminará a través de la operación

$$\left(E_j - \frac{a_{ji}}{a_{ii}} E_i \to E'_j\right). \tag{2.27}$$

Así, el procedimiento para encontrar las $\{x_i\}$ soluciones del sistema de ecuaciones pasa por

1. Escribir el sistema de n ecuaciones en forma matricial, $\mathbf{Ax} = \mathbf{b}$.
2. Realizar las operaciones ec. (2.25) sobre la matriz aumentada $(\mathbf{A} \,|\, \mathbf{b})$ para eliminar la columna $i = 1$ de las E_j (con $j > 1$) ecuaciones.
3. Realizar las operaciones ec. (2.26) sobre la matriz aumentada para eliminar la columna $i = 2$ de las E_j (con $j > 2$) ecuaciones.
4. Realizar las operaciones ec. (2.27) para eliminar las restantes columnas $i = 3, \ldots, n$ de las E_j (con $j = i + 1, \ldots, n$) ecuaciones
5. Una vez conseguida una matriz triangular superior $\mathbf{A}'$ del tipo de ec. (2.10), utilizar la ec. (2.17) para encontrar, con sustitución hacia atrás, las incógnitas x_n, x_{n-1}, x_{n-2}, $\ldots$, x_1.

```
import numpy as np
A=np.loadtxt('gauss.dat')
NRows=len(A)
NCols=len(A)+1
for i in range(0,NRows):
    for j in range(i+1,NRows):
        mji = A[j,i]/A[i,i]
        for k in range(0,NCols):
            A[j,k] -= mji*A[i,k]
print(np.matrix(A))
```

```
x=np.zeros(NRows)
x[NRows-1] = A[NRows-1,NCols-1]/A[NRows-1,NRows-1]
for i in range(NRows-2,-1,-1):
    x[i] = A[i,NRows]
    for j in range(i+1,NRows):
        x[i] -= A[i,j]*x[j]
    x[i] = x[i]/A[i,i]

print(np.matrix(x))
```

Código 2.1: Método de Gauss (`gauss.py`). Este código necesita un archivo de texto `gauss.dat` que contiene la matriz sobre la que se operará.

El cód. 2.1 muestra cómo seguir los distintos pasos para encontrar la solución de un sistema de ecuaciones lineales

3. FACTORIZACIÓN LU

La ec. (2.10) muestra una matriz triangular superior. Todos los elementos por debajo de la diagonal son nulos. Es frecuente que una matriz cuadrada $\mathbf{A}$ pueda descomponerse como el producto de dos matrices $\mathbf{L}$ y $\mathbf{U}$ tales que $\mathbf{A} = \mathbf{LU}$, donde $\mathbf{L}$ es una matriz triangular inferior (*lower*) y $\mathbf{U}$ es una matriz triangular superior (*upper*). Por ejemplo, para una matriz 3×3

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}. \tag{2.28}$$

De esta forma, si queremos resolver el sistema de ecuaciones ec. (2.6), tenemos,

$$\mathbf{Ax} = \mathbf{b}\,, \tag{2.29}$$

$$\mathbf{LUx} = \mathbf{b}\,, \tag{2.30}$$

$$\mathbf{Ly} = \mathbf{b}\,, \tag{2.31}$$

donde hemos definido $\mathbf{y} \triangleq \mathbf{Ux}$. Si conocemos las formas explícitas de las matrices triangular superior y triangular inferior la ventaja de este método

es evidente. Basta sustituir hacia atrás para encontrar los valores de **y** y, después, sustituir hacia adelante para encontrar los valores de **x**.

3.1. Método de Crout

El **método de Crout** es una forma particular de factorización LU en la que los elementos de la diagonal de la matriz triangular superior U son todos igual a la unidad, esto es

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{pmatrix} . \tag{2.32}$$

Los elementos l_{ij} y u_{ij} pueden obtenerse de forma secuencial multiplicando las distintas filas y columnas.

Ejemplo 2.2. Obtener los valores que toman las matrices **U** y **L** de la ec. (2.32).

Solución:

- Los elementos de la primera columna de **L** se obtienen multiplicando las filas de **L** por la primera columna de **U**,

$$a_{11} = l_{11} , \tag{2.33}$$
$$a_{21} = l_{21} , \tag{2.34}$$
$$a_{31} = l_{31} . \tag{2.35}$$

- Los elementos de la primera fila de **U** se obtienen multiplicando la primera fila de **L** por las distintas columnas de **U**

$$a_{12} = l_{11}u_{12} \rightarrow u_{12} = \frac{a_{12}}{l_{11}} , \tag{2.36}$$
$$a_{13} = l_{11}u_{13} \rightarrow u_{13} = \frac{a_{13}}{l_{11}} . \tag{2.37}$$

- Los elementos de la segunda columna de **L** se obtienen multiplicando las filas de **L** por la segunda columna de **U**

$$a_{22} = l_{21}u_{12} + l_{22} \rightarrow l_{22} = a_{22} - l_{21}u_{12}\,, \tag{2.38}$$
$$a_{32} = l_{31}u_{13} + l_{32} \rightarrow l_{32} = a_{32} - l_{31}u_{13}\,. \tag{2.39}$$

- Los elementos de la segunda fila de **U** se obtienen multiplicando la segunda fila de **L** por las distintas columnas de **U**

$$a_{23} = l_{21}u_{13} + l_{22}u_{23} \rightarrow u_{23} = \frac{1}{l_{22}}\left(a_{23} - l_{21}u_{13}\right)\,. \tag{2.40}$$

- Los elementos de la tercera columna de **L** se obtienen multiplicando las filas de **L** por la tercera columna de **U**

$$a_{33} = l_{31}u_{13} + l_{32}u_{23} + l_{33} \rightarrow l_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23}\,. \tag{2.41}$$

El ej. 2 permite generalizar la forma que toman los elementos l_{ij} y u_{ij}, dado por

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\,, \tag{2.42}$$
$$u_{ij} = \frac{1}{l_{ii}}\left(a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}\right)\,. \tag{2.43}$$

3.2. Descomposición de Cholesky

Si la matriz **A** es real y definida positiva[2], entonces admite una factorización LU particular, dada por

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T\,, \tag{2.44}$$

[2] Una matriz **A** es definida positiva si todos sus autovalores son positivos.

donde $\mathbf{L}$ es una matriz triangular inferior y $\mathbf{L}^T$ su transpuesta. La ventaja de este método es que permite, a partir de los elementos l_{ij} obtener los elementos de la matriz triangular superior $u_{ij} = l_{ji}$. Para una matriz 3×3, por ejemplo, la descomposición de Cholesky es

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & u_{23} \\ 0 & 0 & l_{33} \end{pmatrix}, \tag{2.45}$$

donde cada uno de los elementos l_{ij} puede obtenerse a partir de la multiplicación directa de las matrices $\mathbf{L}\mathbf{L}^T$

$$\begin{array}{lcl|lcl|lcl} a_{11} & = & l_{11}^2 & a_{12} & = & l_{11}l_{21} & a_{13} & = & l_{11}l_{31} \\ a_{21} & = & l_{21}l_{11} & a_{22} & = & l_{21}^2 + l_{22}^2 & a_{23} & = & l_{21}l_{31} + l_{22}l_{32} \\ a_{31} & = & l_{31}l_{11} & a_{32} & = & l_{31}l_{21} + l_{32}l_{22} & a_{33} & = & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{array} \tag{2.46}$$

A partir de estos resultados es fácil generalizar y obtener los distintos elementos l_{ij}

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}, \tag{2.47}$$

$$l_{ji} = \frac{1}{l_{ii}} \left(a_{ji} - \sum_{k=1}^{i-1} l_{jk} l_{ik} \right). \tag{2.48}$$

4. MÉTODOS ITERATIVOS

Usando un método iterativo es posible resolver el sistema de ecuaciones $\mathbf{A}\mathbf{x} = \mathbf{b}$. La idea es dar una aproximación inicial para los valores de $\mathbf{x}$ y escribir el sistema de ecuaciones equivalente

$$\mathbf{x} = \mathbf{T}\mathbf{x} + \mathbf{c}. \tag{2.49}$$

A partir de la condición inicial $\mathbf{x}^{(0)}$ se construye la solución

$$\mathbf{x}^{(1)} = \mathbf{T}\mathbf{x}^{(0)} + \mathbf{c}\,, \tag{2.50}$$

$$\mathbf{x}^{(2)} = \mathbf{T}\mathbf{x}^{(1)} + \mathbf{c}\,, \tag{2.51}$$

$$\vdots \tag{2.52}$$

$$\mathbf{x}^{(n)} = \mathbf{T}\mathbf{x}^{(n-1)} + \mathbf{c}\,, \tag{2.53}$$

donde establecemos una tolerancia a la iteración dada, por ejemplo, por[3]

1. $\left\|\mathbf{x}^{(n)} - \mathbf{x}^{(n-1)}\right\|_\infty < \epsilon$,
2. $\frac{\left\|\mathbf{x}^{(n)} - \mathbf{x}^{(n-1)}\right\|_\infty}{\left\|\mathbf{x}^{(n)}\right\|_\infty} < \epsilon$.

El procedimiento general consiste en escribir la matriz $\mathbf{A}$ como

$$\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}, \tag{2.54}$$

donde $\mathbf{D}$ es una matriz diagonal cuyos elementos no nulos son los mismos que los de $\mathbf{A}$, y $\mathbf{L}$ y $\mathbf{U}$ son dos matrices triangulares estrictas[4], inferior y superior respectivamente, cuyos elementos coinciden con los de $-\mathbf{A}$. Por ejemplo, para una matriz 3×3

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{2.55}$$

$$= \underbrace{\begin{pmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{pmatrix}}_{\mathbf{D}} - \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ -a_{21} & 0 & 0 \\ -a_{31} & -a_{32} & 0 \end{pmatrix}}_{\mathbf{L}} - \underbrace{\begin{pmatrix} 0 & -a_{12} & -a_{13} \\ 0 & 0 & -a_{23} \\ 0 & 0 & 0 \end{pmatrix}}_{\mathbf{U}}. \tag{2.56}$$

Con esta definición tenemos que

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{2.57}$$

$$(\mathbf{D} - \mathbf{L} - \mathbf{U})\,\mathbf{x} = \mathbf{b}, \tag{2.58}$$

donde, reordenando los términos llegamos a

$$\mathbf{D}\mathbf{x} = (\mathbf{L} + \mathbf{U})\,\mathbf{x} + \mathbf{b}, \tag{2.59}$$

---

[3] Dado un vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ se define su norma infinita $\|\mathbf{x}\|_\infty \triangleq \max\{|x_1|, |x_2|, \ldots, |x_n|\}$.

[4] Se dice que una matriz triangular (superior o inferior) es estricta si todos los elementos de la diagonal principal son nulos.

y multiplicando por $\mathbf{D}^{-1}$

$$\mathbf{x} = \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\,\mathbf{x} - \mathbf{D}^{-1}\mathbf{b} \tag{2.60}$$
$$= \mathbf{T}\mathbf{x} + \mathbf{c}\,, \tag{2.61}$$

donde hemos definido

$$\mathbf{T} \triangleq \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\,, \tag{2.62}$$
$$\mathbf{c} \triangleq \mathbf{D}^{-1}\mathbf{b}\,. \tag{2.63}$$

```
1  import numpy as np
2  A=np.loadtxt('jacobi.dat')
3  NRows=len(A)
4  NCols=len(A[0])
5
6  TOL   = 1e-4
7  error = 1
8
9  x0 = np.zeros(NRows)
10 x  = np.zeros(NRows)
11
12 while(error > TOL):
13     for i in range(0,NRows):
14         m=0
15         for j in range(0,NRows):
16             if (j != i):
17                 m += A[i,j]*x0[j]
18         x[i] = (A[i,NCols-1] - m)/A[i,i]
19     error=np.sqrt(np.sum((x-x0)*(x-x0)))
20     x0[:]=x
21 print(np.matrix(x))
```

Código 2.2: Método iterativo de Jacobi (`jacobi.py`). Este código necesita un archivo de texto `jacobi.dat` que contiene la matriz sobre la que se operará. Los métodos iterativos son muy sensibles a los valores iniciales. Si estos no son apropiados, es normal que el métodos no converja.

El cód. 2.2 muestra los pasos necesarios para encontrar las soluciones usando

el **método de Jacobi**. La tolerancia se ha definido en este caso como

$$\epsilon = \sqrt{\sum_{i=1}^{N}(x_i^k - x_i^{k-1})^2}\,. \tag{2.64}$$

5. DIAGONALIZACIÓN

La diagonalización de matrices y el cálculo de autovalores (*eigenvalores*) y autovectores (*eigenvectores*) es común en química cuántica para la obtención de los niveles energéticos en átomos polielectrónicos o en moléculas complejas. Además, dichos problemas suelen ser numéricamente exigentes y requieren de técnicas de cómputo avanzadas. En este sección vamos a estudiar el caso de diagonalizar matrices reales simétricas.

Más información:

Recordamos que una matriz real simétrica es aquella en la que todos los elementos son reales y, además, su traspuesta es la misma matriz, esto es:

$$\mathbf{A}^T = \mathbf{A}.$$

Supongamos la matriz $\mathbf{A}$ de tamaño $N \times N$. Se dice que $\mathbf{A}$ tiene un autovector $\mathbf{x}$ y un autovalor correspondiente λ si se cumple:

$$\mathbf{A}\mathbf{x}_\lambda = \lambda\mathbf{x}_\lambda \tag{2.65}$$

A la resolución de esta ecuación obteniendo los autovalores y sus correspondientes autovectores se le denomina diagonalización. Obviamente, cualquier múltiplo del autovector $\mathbf{x}_\lambda$ es también solución de la ecuación (2.65). El caso trivial $\mathbf{x}_\lambda = \mathbf{0}$ no se considera un autovector. La ecuación (2.65) se cumple solo si:

$$\det|\mathbf{A} - \lambda\mathbf{1}| = 0 \tag{2.66}$$

La solución de esta ecuación existe para N autovalores λ, los cuales pueden ser repetidos.

Nótese que si se añade un vector $\alpha \mathbf{x}_\lambda$, donde α es una constante, a ambos lados de la ecuación (2.65) los autovalores se modifican de acuerdo a $\lambda \rightarrow \lambda + \alpha$, mientras que los autovectores no cambian, $\mathbf{x}_\lambda = \mathbf{x}_{\lambda+\alpha}$. En consecuencia, si $\lambda = 0$, esto no tiene ninguna significación especial ya que cualquier autovalor puede ser convertido a cero, y cualquier autovalor que fuera cero puede ser convertido a un valor distinto de cero.

En la actualidad existen muchos paquetes de cómputo numérico que permiten resolver el problema de diagonalización para una gran variedad de matrices **A** diagonalización. Ejemplos son la clásica librería LAPACK implementada en FORTRAN y C, o la más moderna librería scipy.linalg.eig en Python que se apoya en LAPACK.

En este capítulo aprenderemos a calcular dichos autovalores λ junto con sus autovectores asociados $\mathbf{x}_\lambda$ mediante el método iterativo de Jacobi.

5.1. Método iterativo de Jacobi para matrices reales simétricas

El método de Jacobi se basa en una serie de transformaciones denominadas *rotaciones de Jacobi* especialmente diseñadas para eliminar elementos de **A** que no pertenecen a la diagonal. Al aplicar dicha transformación algunos elementos de fuera de la diagonal que fuesen cero pueden cambiar, aunque dicho cambio es pequeño. El método se aplica iterativamente hasta que los elementos que no pertenecen a la diagonal son menores que la precisión del ordenador, obteniéndose un matriz diagonal. El producto de las transformaciones proporciona los autovectores y los elementos de la diagonal los autovalores. Para poder definir el algoritmo, primero hay que introducir la matriz rotación de Jacobi. Dicha matriz tiene la forma:

$$\mathbf{P}_{mn} = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & \cdots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \cdots & c & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{2.67}$$

En esta matriz todos los elementos de la diagonal son 1 excepto los dos elementos c en filas y columnas m y n. Todos los elementos no diagonales son cero excepto s y $-s$. Los números c y s vienen dados por el coseno y el seno de un ángulo de rotación θ tal que $s^2 + c^2 = 1$.

Aplicando la rotación de Jacobi sobre $\mathbf{A}$, podemos calcular una nueva matriz $\mathbf{A}'$

$$\mathbf{A}' = \mathbf{P}_{mn}^T \mathbf{A} \mathbf{P}_{mn} \tag{2.68}$$

$\mathbf{P}_{mn}^T \mathbf{A}$ cambia las filas m y n de $\mathbf{A}$, mientras que $\mathbf{A}\mathbf{P}_{mn}$ cambia las columnas m y n de $\mathbf{A}$. De esta forma $\mathbf{A}'$ queda dada por:

$$\mathbf{A}' = \begin{bmatrix} & & A'_{0,m} & & A'_{0,n} & & \\ & & \vdots & & \vdots & & \\ A'_{m,0} & \cdots & A'_{m,m} & \cdots & A'_{m,n} & \cdots & A'_{m,N-1} \\ & & \vdots & & \vdots & & \\ A'_{n,0} & \cdots & A'_{n,m} & \cdots & A'_{n,n} & \cdots & A'_{n,N-1} \\ & & \vdots & & \vdots & & \\ & & A'_{N-1,m} & & A'_{N-1,n} & & \end{bmatrix} \tag{2.69}$$

De hecho, tenemos las siguientes fórmular explícitas:

$$\left.\begin{aligned} A'_{rm} &= c\, A_{rm} + s\, A_{rn} \\ A'_{rn} &= c\, A_{rm} - s\, A_{rn} \end{aligned}\right\} r \neq m, r \neq n \tag{2.70}$$

$$A'_{mm} = c^2\, A_{mm} + s^2\, A_{nn} - 2\, c\, s A_{mn} \tag{2.71}$$

$$A'_{nn} = s^2\, A_{mm} + c^2\, A_{nn} + 2\, c\, s A_{mn} \tag{2.72}$$

$$A'_{mn} = \left(c^2 - s^2\right) A_{mn} + s\, c\, \left(A_{mm} - A_{nn}\right) \tag{2.73}$$

El objetivo es hacer que los elementos que no pertenecen a la diagonal se hagan cero. Para ello, un primer paso es conseguir que $A'_{mn} = 0$ en la ecuación (2.73), lo que da la siguiente relación para el ángulo θ:

$$\phi \equiv \cot 2\theta = \frac{c^2 - s^2}{2sc} = \frac{A_{nn} - A_{mm}}{2A_{mn}} \tag{2.74}$$

Teniendo en cuenta que

$$t = \tan\theta = \frac{\sin\theta}{\cos\theta} = \frac{s}{c} \tag{2.75}$$

podemos escribir

$$t^2 - 2t\phi - 1 = 0 \tag{2.76}$$

Que tiene dos soluciones. Podemos tomar la raíz menor a fin de tener la menor transformación en la matriz, lo que genera iteraciones más estables. Esta es:

$$t = \frac{\operatorname{sgn}|\phi|}{|\phi| + \sqrt{\phi^2 + 1}} \tag{2.77}$$

y por lo tanto:

$$c = \frac{1}{\sqrt{t^2 + 1}} \tag{2.78}$$

$$s = t\,c \tag{2.79}$$

Si hacemos uso de $A'_{mn} = 0$ y del cálculo de t podemos eliminar A_{nn} de la ecuación (2.71), obteniéndose:

$$A'_{mm} = A_{mm} - t\,A_{mn} \tag{2.80}$$

y también en el resto de ecuaciones:

$$A'_{nn} = A_{nn} + t\,A_{mn} \tag{2.81}$$

$$A'_{rm} = A_{rm} - s\left(A_{rn} + \frac{s}{1+c}A_{rm}\right) \tag{2.82}$$

$$A'_{rn} = A_{rn} + s\left(A_{rm} - \frac{s}{1+c}A_{rn}\right) \tag{2.83}$$

La convergencia del método se puede caracterizar a partir de la suma de los cuadrados de los elementos de matriz no diagonales

$$R^2 = \sum_{r \neq s} |A_{rs}|^2 \tag{2.84}$$

El resultado del algoritmo iterativo es una matriz diagonal $\mathbf{D}$ cuyos elementos de la diagonal son los autovalores λ.

Lo mostrado hasta ahora muestra cómo se produce una iteración del método de Jacobi. Lo que queda para poder obtener los autovalores es establecer una estrategia para ir variando los m y n a fin de ir eliminando los elementos de matriz no diagonales iterativamente. La estrategia más sencilla es el denominado *método cíclico de Jacobi* en el que se procede secuencialmente fila a fila, es decir: $\mathbf{P}_{0,1}$, $\mathbf{P}_{0,2}$, $\mathbf{P}_{0,3}$, ..., $\mathbf{P}_{0,N-1}$; luego, $\mathbf{P}_{1,1}$, $\mathbf{P}_{1,2}$, $\mathbf{P}_{1,3}$, ..., $\mathbf{P}_{1,N-1}$; hasta llegar a $\mathbf{P}_{N-1,1}$, $\mathbf{P}_{N-1,2}$, $\mathbf{P}_{N-1,3}$, ..., $\mathbf{P}_{N-1,N-1}$; y luego repetir la secuencia hasta obtener la convergencia deseada.

Una vez obtenida la matriz diagonal $\mathbf{D}$ solo queda obtener los autovectores. Teniendo presente $\mathbf{D}$ y $\mathbf{A}$ se relacionan mediante:

$$\mathbf{D} = \mathbf{V}^T \mathbf{A} \mathbf{V} \tag{2.85}$$

donde $\mathbf{V}$ es una matriz cuyas columnas son los autovectores ya que

$$\mathbf{V}\mathbf{D} = \mathbf{A}\mathbf{V} \tag{2.86}$$

y teniendo presente $\mathbf{V}$ es el producto de las rotaciones de Jacobi

$$\mathbf{V} = \mathbf{P_1}\mathbf{P_2}\mathbf{P_3}\mathbf{P_4}\mathbf{P_5}\cdots \tag{2.87}$$

Podemos calcular el $\mathbf{V}$ final iterativamente utilizando la relación

$$\mathbf{V}' = \mathbf{V}\mathbf{P_i} \tag{2.88}$$

donde inicialmente $\mathbf{V}$ es la matriz identidad. Obteniéndose las relaciones de recurrencia:

$$v'_{rs} = v_{rs} \quad (s \neq m, s \neq n) \tag{2.89}$$

$$v'_{rm} = c\, v_{rm} - s\, v_{rn} \tag{2.90}$$

$$v'_{rm} = s\, v_{rm} + c\, v_{rn} \tag{2.91}$$

6. EJERCICIOS

Ej. 2.1 — Encontrar, usando el método de Gauss, la solución al sistema de ecuaciones:

$$\begin{array}{rcrcrcrcr} x_1 & + & x_2 & & & + & 3x_4 & = & 4\,, \\ 2x_1 & + & x_2 & - & x_3 & + & x_4 & = & 1\,, \\ 3x_1 & - & x_2 & - & x_3 & + & 2x_4 & = & -3\,, \\ -x_1 & + & 2x_2 & + & 3x_3 & - & x_4 & = & 4\,. \end{array}$$

Ej. 2.2 — Si intenta aplicar directamente el método de Gauss (cód. 2.1) al sistema de ecuaciones:

$$\begin{array}{rcrcrcrcr} x_1 & - & x_2 & + & 2x_3 & - & x_4 & = & -8\,, \\ 2x_1 & - & 2x_2 & - & 3x_3 & - & 3x_4 & = & -20\,, \\ x_1 & + & x_2 & - & x_3 & & & = & -2\,, \\ x_1 & - & x_2 & + & 4x_3 & + & 3x_4 & = & 4\,, \end{array}$$

observará que no es posible resolver el sistema de ecuaciones. De las operaciones a), b) y c) descritas, ¿cuál de ellas puede aplicar para resolver adecuadamente el sistema de ecuaciones con el método de Gauss? Modifique el cód. 2.1 (o realice su propio código) que tenga en cuenta esta eventualidad y que permita resolver este sistema de ecuaciones.

Ej. 2.3 — Obtenga la descomposición LU, según el método de Crout, de la matriz:

$$\begin{pmatrix} 1 & 1 & 0 & 3 \\ 2 & 1 & -1 & 1 \\ 3 & -1 & -1 & 2 \\ -1 & 2 & 3 & -1 \end{pmatrix}. \tag{2.92}$$

Ej. 2.4 — Diseñe un código de programación que permita obtener secuencialmente todos los elementos l_{ij} para la descomposición de Cholesky.

Ej. 2.5 — Encuentre la descomposición de Cholesky de la matriz:

$$\begin{pmatrix} 4 & -1 & 1 \\ -1 & 4{,}25 & 2{,}75 \\ 1 & 2{,}75 & 3{,}5 \end{pmatrix}. \tag{2.93}$$

Ej. 2.6 — Encuentre las soluciones, haciendo uso de un método iterativo, del sistema de ecuaciones lineales:

$$\begin{array}{rcrcrcrcr} x_1 & + & x_2 & & & + & 3x_4 & = & 4\,, \\ 2x_1 & + & x_2 & - & x_3 & + & x_4 & = & 1\,, \\ 3x_1 & - & x_2 & - & x_3 & + & 2x_4 & = & -3\,, \\ -x_1 & + & 2x_2 & + & 3x_3 & - & x_4 & = & 4\,, \end{array}$$

Nota. Utilice como criterio de convergencia

$$\frac{\|\mathbf{x}^{(n)} - \mathbf{x}^{(n-1)}\|_\infty}{\|\mathbf{x}^{(n)}\|_\infty}\,. \tag{2.94}$$

Tema 3

INTERPOLACIÓN

1. INTRODUCCIÓN

A veces queremos conocer el valor de una función $f(x)$ en un conjunto de puntos $x_0, x_1, \ldots, x_N$ pero no tenemos la forma analítica de la función o esta es demasiado complicada de calcular.

Hay ocasiones en las que realizamos un un conjunto de medidas de una cierta magnitud experimental, llamémosla f en una serie de puntos $x_0, x_1, \ldots, x_N$ obteniendo $x_0, x_1, \ldots, x_N$ y nos preguntamos cuál sería el valor que esperamos que tome dicha magnitud en un punto intermedio y pero no tenemos la forma analítica de la función o ésta es demasiado complicada de calcular. La idea para estimar el valor de $f(y)$ sería dibujar una curva suave que pasara por los puntos $(x_i, f(x_i))$ y comprobar para y cuál sería el valor de $f(y)$. A este problema se le denomina *interpolación* y en este capítulo vamos a presentar algunos procedimientos sencillos para realizarlo. En concreto:

1. Interpolación lineal
2. Interpolación polinómica

2. INTERPOLACIÓN LINEAL

La interpolación lineal es la más simple que se puede hacer. Supongamos que conocemos el valor de la función en dos puntos x_0 y x_1, y queremos estimar el valor que tendría en y dado que $x_0 < x < x_1$. La interpolación se basa en construir la ecuación de una recta $f(x) = m\,x + n$ que pasa por $(x_0, f(x_0))$ y $(x_1, f(x_1))$, obteniendo la pendiente m y la ordenada en el origen n y calcular en dicha recta calcular $m\,x + n$, obteniendo así la estimación de $f(x) = m\,x + n$.

Los parámetros de la recta m y n vienen dados por:

$$m = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$
$$n = f(x_0) - m\, x_0$$

Las siguiente función en Python implementa la interpoalción lineal:

```
1 def interpolacion_lineal(xa,ya,x,dy):
2     m = (ya[1] - ya[0])/(xa[1]-xa[0]);
3     n = ya[0] - m*xa[0]
4     return m*x + n;
```

Código 3.1: Función de interpolación lineal (`lineal.py`).

Ejemplo 3.1. Dados los pares de puntos $(x_0 = 2, f(x_0) = 2)$ y $(x_1 = 4, f(x_1) = 5)$ obtener una estimación del valor de la función f en $x = 3$: $f(x = 3)$.

Solución:
Aplicamos las ecuaciones para obtener m y n

$$m = \frac{5-2}{4-2} = \frac{3}{2} = 1{,}5$$
$$n = 2 - 2\, m = 2 - 2\,\frac{3}{2} = -1$$

luego la función de interpolación lineal es:

$$f(x) = \frac{3}{2}\, x - 1$$

Por tanto, para $x = 3$, la estimación de f es:

$$f(x = 3) = \frac{3}{2}\, 3 - 1 = \frac{9}{2} - 1 = \frac{7}{2} = 3{,}5$$

3. INTERPOLACIÓN POLINÓMICA

Como hemos visto, dados dos puntos es posible construir una función lineal única de interpolación. Si tenemos tres puntos, es posible construir una función cuadrática única de interpolación, etc. En general, dados M puntos, es posible construir un polinomio de interpolador único de grado $(M-1)$. De finiendo $y_0 = f(x_0), y_1 = f(x_1), \ldots, y_{M-2} = f(x_{M-2}), y_{M-1} = f(x_{M-1})$ Su cálculo se puede realizar explícitamente haciendo uso de la fórmula de Lagrange:

$$\begin{aligned} P(x) =& \frac{(x-x_1)(x-x_2)\ldots(x-x_{M-1})}{(x_0-x_1)(x_0-x_2)\ldots(x_0-x_{M-1})} y_0 \\ &+ \frac{(x-x_0)(x-x_2)\ldots(x-x_{M-1})}{(x_0-x_1)(x_0-x_2)\ldots(x_0-x_{M-1})} y_1 + \ldots \\ &+ \frac{(x-x_0)(x-x_1)\ldots(x-x_{M-2})}{(x_{M-1}-x_0)(x_{M-1}-x_1)\ldots(x_{M-1}-x_{M-2})} y_{M-1} \end{aligned}$$

Esta fórmula contiene M términos, cada uno de un polinomio de grado $M-1$ y por construcción cada término es cero a todos los x_i salvo uno en el que proporciona y_i. Es correcto implementar esta fórmula para obtener el polinomio interpolador, sin embargo existe una forma más eficiente utilizando el denominado algoritmo de Neville. Supongamos los valores y_i renombrados como P_i. Estos serían los polinomios de grado cero que pasan por cada uno de los puntos que queremos interpolar. A continuación interpolamos linealmente los pares de puntos $[y_0 = f(x_0), y_1 = f(x_1)]$, $[y_1 = f(x_1), y_2 = f(x_2)]$, $\ldots$,$[y_{M-2} = f(x_{M-2}), y_{M-1} = f(x_{M-1})]$ obteniendo los polinomios de primer grado P_{01},P_{12}, $\ldots$, $P_{(M-2)(M-1)}$, después construimos los polinomios interpoladores de grado dos P_{012}, P_{123}, P_{234}, $\ldots$, $P_{(M-3)(M-2)(M-1)}$ y así se van construyendo todos los polinomios intepoladores. Para el caso de $M = 5$ quedaría la siguiente tabla resumen de los polinomios interpoladores:

$$
\begin{array}{lllll}
P_0 = y_0 & & & & \\
 & P_{01} & & & \\
P_1 = y_1 & & P_{012} & & \\
 & P_{12} & & P_{0123} & \\
P_2 = y_2 & & P_{123} & & P_{01234} \\
 & P_{23} & & P_{1234} & \\
P_3 = y_3 & & P_{234} & & \\
 & P_{34} & & & \\
P_4 = y_4 & & & &
\end{array}
$$

El algoritmo de Neville es un algoritmo recursivo que permite rellenar esta tabla. La fórmula de recurrencia es:

$$
P_{i(i+1)\ldots(i+m)} = \frac{(x - x_{i+m})P_{i(i+1)\ldots(i+m-1)} + (x_i - x)P_{(i+1)(i+2)\ldots(i+m)}}{x_i - x_{i+m}}
$$

que funciona dado que dos polinomios adyacentes en la misma columna de la tabla coinciden en sus valores $x_{i+1}, \ldots, x_{i+m-1}$. A continuación se muestra el código en Python que implementa el algoritmo de Neville mediante la fórmula de recurrencia:

```
1  import numpy as np
2  def interpolacion_polinomica_Neville(xa,ya,x):
3    n = len(xa)
4    p = n*[0]
5    for k in range(n):
6      for i in range(n-k):
7        if k == 0:
8          p[i] = ya[i]
9        else:
10         p[i] = ((x-xa[i+k])*p[i]+ \
11             (xa[i]-x)*p[i+1])/(xa[i]-xa[i+k])
12   return p[0]
```

Código 3.2: Función de interpolación polinómica haciendo uso del algoritmo de Neville (`polinomica_neville.py`).

Ejemplo 3.2. Dados los pares de puntos: $(x_0 = 2, f(x_0) = 2)$, $(x_1 = 4, f(x_1) = 5)$ $(x_2 = 6, f(x_2) = 4)$ y $(x_3 = 8, f(x_3) = 7)$ obtener una estimación del valor de la función f en $x = 3$ utilizando un polinomio interpolador de grado 3 mediante el algoritmo de Neville.

Solución:

Aplicamos las ecuaciones del algoritmos de Neville para reconstruir la tabla de P. Ya que tenemos 4 puntos, la primera columna de la tabla serán 4 valores $P_i = f(x_i)$ y el resto de columnas vienen dadas por la fórmula de recursión donde $x = 3$. Vamos a rellenar las dos primeras columnas:

$$
\begin{array}{lll}
P_0 = f(x_0 = 2) = 2 & & \\
 & P_{01} = \frac{(x-x_1)P_0+(x_0-x)P_1}{x_0-x_1} = \frac{(3-4)2+(2-3)5}{-2} = \frac{7}{2} & \\
P_1 = f(x_1 = 4) = 5 & & P_{012} \\
 & P_{12} = \frac{(x-x_2)P_1+(x_1-x)P_2}{x_1-x_2} = \frac{(3-6)5+(4-3)4}{-2} = \frac{11}{2} & P_{0123} \\
P_2 = f(x_2 = 6) = 4 & & P_{123} \\
 & P_{23} = \frac{(x-x_3)P_2+(x_2-x)P_3}{x_2-x_3} = \frac{(3-8)4+(6-3)7}{-2} = \frac{11}{2} & \\
P_3 = f(x_3 = 8) = 7 & &
\end{array}
$$

La tercera columna se calcularía:

$$
\begin{array}{lll}
P_0 = 2 & & \\
 & P_{01} = \frac{7}{2} & \\
P_1 = 5 & P_{012} = \frac{(x-x_2)P_{01}+(x_0-x)P_{12}}{x_0-x_2} = \frac{(3-6)\frac{7}{2}+(2-3)\frac{11}{2}}{2-6} = 4 & P_{0123} \\
 & P_{12} = \frac{11}{2} & \\
P_2 = 4 & P_{123} = \frac{(x-x_3)P_{12}+(x_1-x)P_{23}}{x_1-x_3} = \frac{(3-8)\frac{11}{2}+(4-3)\frac{-1}{2}}{4-8} = 7 & \\
 & P_{23} = \frac{11}{2} & \\
P_3 = 7 & &
\end{array}
$$

Y finalmente la última columna que nos proporciona el resultado buscado:

$$
\begin{array}{lll}
P_0 = 2 & & \\
 & P_{01} = \frac{7}{2} & \\
P_1 = 5 & & P_{012} = 4 \\
 & P_{12} = \frac{11}{2} & \\
P_2 = 4 & & P_{123} = 7 \\
 & P_{23} = \frac{11}{2} & \\
P_3 = 7 & &
\end{array}
\qquad P_{0123} = \frac{(x-x_4)P_{012}+(x_0-x)P_{123}}{x_0-x_4} = \frac{(3-8)4+(2-3)7}{-6} = \frac{27}{6}
$$

Luego el resultado buscado es $f(x = 3) = \frac{27}{6} = \frac{9}{2} = 4{,}5$. Nótese que podríamos dejado sin substituir x por el valor de 3 obteniendo por este procedimiento la expresión explícita del polinomio interpolador. En concreto este sería el resultado:

$$f(x) = \frac{1}{6}\,x^3 - \frac{5}{2}\,x^2 + \frac{71}{6}\,x - 13$$

que se muestra en la figura 3.1.

4. EJERCICIOS

Ej. 3.1 — Encontrar, haciendo uso del algoritmo de Neville, el polinomio interpolador de los siguientes pares de valores:

$$
\begin{array}{ll}
x_0 = 1, & f(x_0) = 3 \\
x_1 = 2, & f(x_1) = 8 \\
x_2 = 3, & f(x_2) = 4 \\
x_3 = 4, & f(x_3) = 3
\end{array}
$$

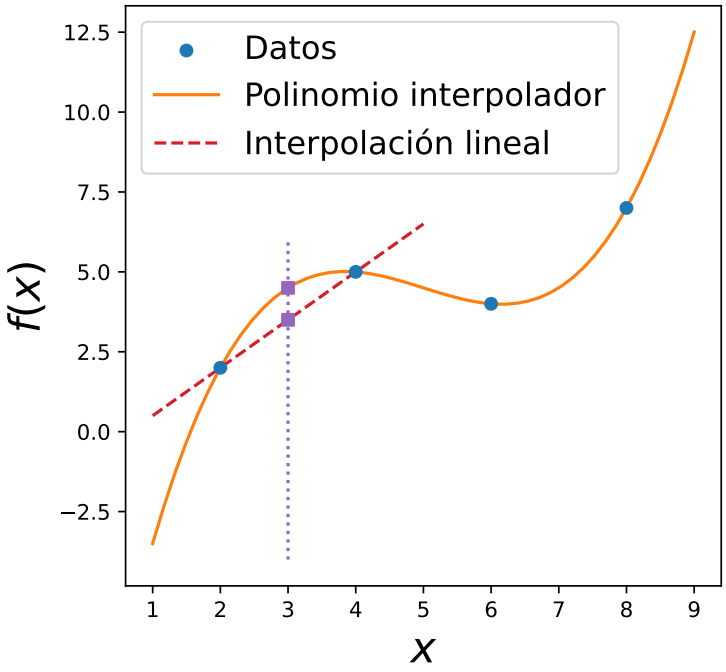

Figura 3.1. Datos e interpolaciones lineal (línea roja discontinua) y polinómica (curva naranja discontinua) correspondientes a los Ejemplos 2 y 2. Se resaltan con dos cuadrados morados los valores de las funciones interpoladores en $x = 3$. Nótese la diferencia que hay entre la estimación del valor de la función interpolando linealmente los dos primeros puntos frente a la interpolación polinómica de los cuatro puntos.

Tema 4

RESOLUCIÓN DE ECUACIONES NO LINEALES

1. INTRODUCCIÓN

Un problema recurrente en matemáticas es la obtención de las raíces de una función. Se llaman raíces de $f(x)$ a aquellos valores de x tales que $f(x) = 0$. Obtener las raíces de ecuaciones de primer y segundo grado es inmediato, pero en el momento en que tratamos con polinomios de orden superior o con funciones trascendentes la obtención de las raíces se vuelve más complicado. Es frecuente también que nos pregunten no por los ceros de una función, sino por aquel valor de x tal que $f(x) = y$. Nótese que este problema es equivalente a preguntarse por los ceros de la función $f(x) - y$.

En este tema abordaremos distintos métodos numéricos (iterativos) que permiten obtener de forma eficiente las raíces de una función. Compararemos además el orden de convergencia de los mismos en aras de establecer cuáles son las ventajas de utilizar un método frente a otro.

En todo este tema trabajaremos con un ejemplo concreto. Supongamos que la temperatura en una ciudad en un determinado día viene dada por la siguiente expresión

$$T(t) = 3{,}2^\circ\,\mathrm{C} + 5{,}1^\circ\,\mathrm{C}\sin(2\pi(t - 13\,\mathrm{h})/24\,\mathrm{h}),$$

donde t es el tiempo, en horas, transcurrido desde la medianoche. La fig. 4.1 muestra la evolución de la temperatura en función del tiempo. Observamos el mínimo de la temperatura alrededor de las siete de la mañana y el máximo alrededor de las siete de la tarde. Nuestro objetivo, en este tema, es saber en qué momento la temperatura será de 0,0°C.

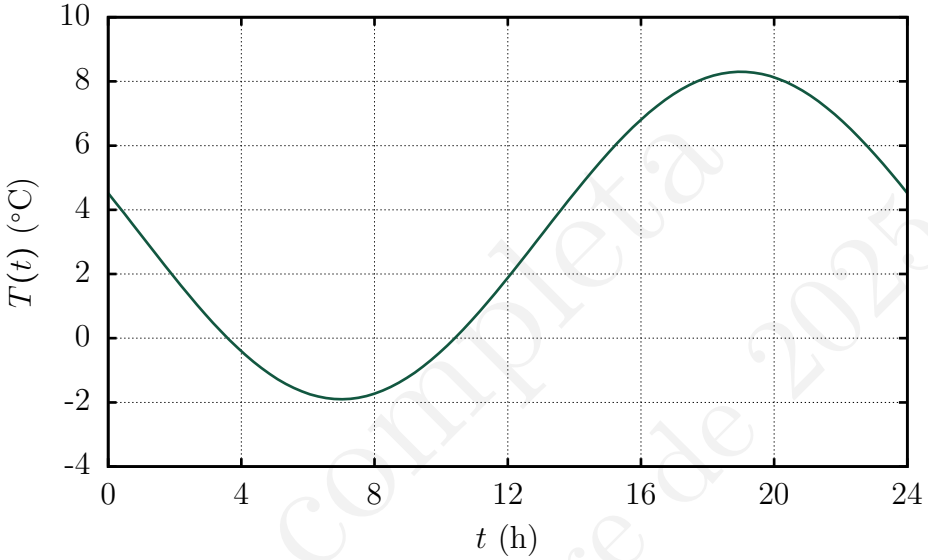

Figura 4.1. Temperatura, desde las doce de la noche, en una ciudad a lo largo del día.

2. MÉTODO DE LA BISECCIÓN

Obviemos por un momento la fig. 4.1. Sabemos que, en general, la mínima temperatura que se registra a lo largo de un día ocurre alrededor de las 7 am. A esa hora, usando la ecuación anterior, tenemos que

$$T(7\,\mathrm{h}) = -1{,}9^\circ\,\mathrm{C}.$$

Sabemos también que en general la temperatura máxima de un día se da alrededor de las 3 pm, de forma que, en nuestro ejemplo,

$$T(15\,\mathrm{h}) = 5{,}7^\circ\,\mathrm{C}.$$

Obsérvese que, de la fig. 4.1, la temperatura máxima se registra alrededor de las 19 h. Considerar este valor o el obtenido en la expresión anterior no es importante en este momento. De hecho, es común encontrarse con

funciones cuya representación gráfica desconocemos. Es por tanto necesario hacer una *apuesta inicial* como la que hemos hecho sobre los valores iniciales o extremos alrededor de los cuales queremos encontrar el cero de nuestra función.

Parece lógico suponer que, si a las 7am la temperatura estaba por debajo de 0°C, y a las 3pm la temperatura estaba por encima de los 0°, en algún punto intermedio la temperatura debió ser de exactamente 0°C. Formalmente, el **teorema de Bolzano** nos dice que, si una función $f(x)$ es continua en un intervalo cerrado $[x_a, x_b]$, donde $f(x_a)$ y $f(x_b)$ tienen signos opuestos, existe entonces al menos un punto x_c dentro del intervalo (x_a, x_b) tal que $f(x_c) = 0$. Este teorema, que es un caso particular del **teorema del valor intermedio**, nos dice si una función (real) continua es positiva en un punto y negativa en otro punto, necesariamente ha de pasar por cero en algún punto intermedio. Esta idea da lugar a un poderoso método para obtener, de forma iterativa, los ceros de cualquier función $f(x)$. Lo único importante que debemos comprobar para poder aplicar este método es:

1. que la función $f(x)$ sea continua en el intervalo $[x_a, x_b]$ de estudio,
2. que la función tome signos contrarios en los extremos del intervalo, esto es, $f(x_a)f(x_b) < 0$.

Si ambas condiciones se cumplen, el método de la bisección converge, siempre, hacia la solución exacta.

El procedimiento iterativo de este método consiste en evaluar las condiciones a) y b) para los puntos x_a y x_b. Si se verifican, tomamos el punto medio $x_c = (x_a + x_b)/2$ y volvemos a evaluar las condiciones. Si $f(x_c)$ tiene el mismo signo que $f(x_a)$, entonces el intervalo de interés será (x_c, x_b). Si $f(x_c)$ tiene el mismo signo que $f(x_b)$, entonces deberemos seleccionar el intervalo (x_a, x_c). El procedimiento se repite indefinidamente dividiendo en dos mitades cada intervalo, de forma que eventualmente alcancemos, con una tolerancia determinada, el cero solicitado.

```
DAR:

```

```
3    x_a positivo
4    x_b negativo
5    f(x)
6
7  HACER UN BUCLE:
8
9    TOMAR x_c = (x_a+x_b)/2
10   SI f(x_c) = 0 DETENER
11   SI NO:
12     SI f(x_a)f(x_c) < 0 ENTONCES x_b = x_c
13                         SI NO    x_a = x_c
```

Código 4.1: Pseudocódigo para el método de la bisección.

El cód. 4.1 muestra un procedimiento iterativo para obtener el punto x_c en el que la función $f(x)$ toma valor $f(x_c) = 0$. Observamos que toma, para cada iteración, el punto medio del intervalo. Si tenemos que $f(x_c)$ no es igual a cero, determinamos el signo de $f(x_c)$ y sustituimos el intervalo de forma que siempre haya un cambio de signo entre los extremos. De este modo, nos aseguramos las condiciones de aplicabilidad del teorema de Bolzano y, por tanto, que alcanzaremos finalmente un cero de la función.

Más información:

Este pseudocódigo, si bien es sencillo de implementar, presenta un potencial inconveniente. Cabe la posibilidad de que no encontremos, desde el punto de vista numérico, un punto x_c en el que la función $f(x)$ sea **exactamente** cero. Esto ocurre, por ejemplo, si el cero de la función es un número irracional. Aunque reduzcamos el intervalo infinitamente, siempre tendremos un valor distinto de cero para la evaluación de la función en sus sucesivas aproximaciones. De este modo, nunca se cumpliría la condición $f(x_c) = 0$ y, por tanto, nunca saldríamos del bucle.

El cód. 4.2 muestra un ejemplo en python que define la función problema (líneas 5 y 6) y plantea los parámetros iniciales (líneas 8 a 11). Observemos que se incluye un parámetro de tolerancia (`TOL`) que indicará cuándo detener

el bucle. El criterio de convergencia aquí utilizado (líneas 15 y 16) es tal que la diferencia, en valor absoluto, de $f(x)$ respecto de cero sea menor que la tolerancia `TOL`. Notemos que podríamos haber usado otros criterios de convergencia, como por ejemplo la diferencia entre una iteración $f(x_n)$ y la anterior $f(x_{n-1})$.

```
1  from math import *
2
3  def f(x):
4      return 3.2+5.1*sin(2*pi*(x-13)/24)
5
6  xa       =  7              # Limite inferior del intervalo (f(xa) <
7  xb       = 15              # Limite superior del intervalo (f(xb) >
8  tol      = 1E-6            # Tolerancia para la solucion
9  xc       = (xa+xb)/2       # Valor inicial para xc
10
11 print(f' {xc} \t \t {f(xc)}' )
12
13 # Criterio de convergencia: |f(xc)| < tol
14 while abs(f(xc)) > tol:
15     xc   = (xa+xb)/2.0
16     print(f'  {xc:.10f} \t  {f(xc):.10f} ')
17     if (f(xa)*f(xc)<0):
18         xb = xc
19     else:
20         xa = xc
```

Código 4.2: Método de la bisección (`biseccion.py`).

La tab. 4.1 muestra una ejecución del cód. 4.2, donde se observa cómo las sucesivas iteraciones añaden cifras significativas al cero de la función. La fig. 4.2 muestra gráficamente el proceso de búsqueda del cero, partiendo de un punto $x_c = 11\,\mathrm{h}$ hasta alcanzar después de 13 iteraciones el valor deseado. Nótese que, para este sencillo ejemplo, podemos obtener analíticamente el valor que toman las raíces de la temperatura a lo largo del día, dadas por

$$t_0 = 13{,}0\,\mathrm{h} + \frac{24\,\mathrm{h}}{2\pi}\sin^{-1}\left(-\frac{3{,}2}{5{,}1}\right),$$

Iteración	x_c	$f(x_c)$
1	11	0,65
2	9	−1,2167295593
3	10	−0,4062445841
4	10,5	0,0953167121
5	10,25	−0,1626636570
6	10,375	−0,0354057492
7	10,4375	0,0295310758
8	10,40625	−0,0030445301
9	10,421875	0,0132166105
10	10,4140625	0,0050793576
11	10,41015625	0,0010157410
12	10,408203125	−0,0010148130
13	10,4091796875	0,0000003594

Tabla 4.1. Iteraciones para encontrar un cero de la temperatura. Tras 13 iteraciones, $f(x_c)$ es igual a cero con 7 cifras significativas.

lo que nos permite decir que el cero encontrado tiene un error relativo del $4 \times 10^{-6}\,\%$.

Más información:

Obsérvese que hemos obtenido **uno** de los ceros de la función. Obtener **todos** los ceros de la función requeriría una elección más cuidadosa de los intervalos.

Antes de finalizar esta sección es interesante realizar un estudio de la convergencia del método de la bisección. Cualitativamente, dado que en cada iteración reducimos a la mitad la longitud del intervalo, podemos decir que este método tiene una convergencia lineal: tras la primera iteración el error se reduce a la mitad, con la segunda iteración a un cuarto, la tercera a un octavo, etc. Puede probarse que, para la n-ésima iteración, el error será a lo sumo proporcional a

$$\epsilon_n \propto 2^{-n}\,.$$

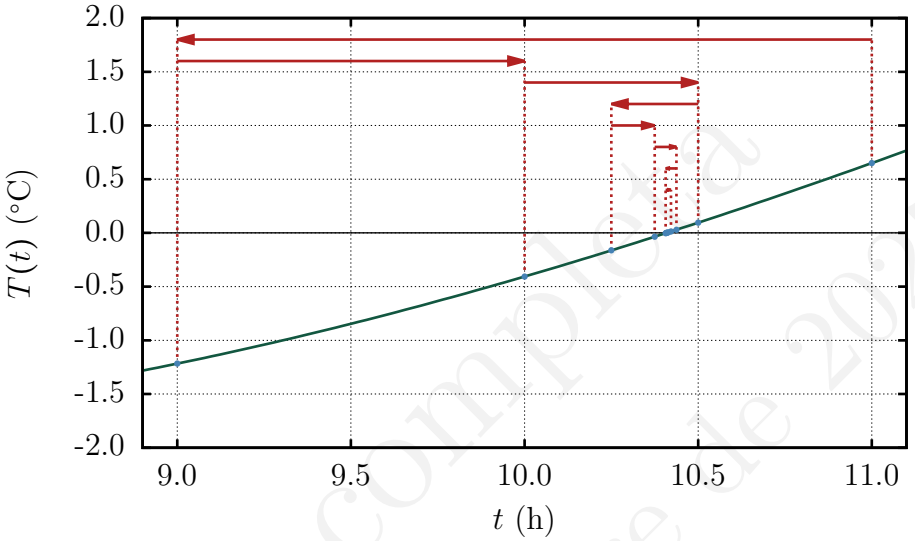

Figura 4.2. Método de la bisección aplicado a la temperatura a lo largo del día entre las 7 am y las 3 pm. Los sucesivos puntos vienen dados en la Tab. 4.1.

La fig. 4.3 muestra, numéricamente, el error respecto de cero para sucesivas iteraciones (hasta 50), donde se observa un comportamiento de tipo $\epsilon_n = ab^{-n}$. Un ajuste por mínimos cuadrados de estos 50 puntos da como resultado $b = (1{,}99 \pm 0{,}02)$, lo que verifica el cálculo realizado anteriormente.

3. MÉTODO DE NEWTON

El método de la bisección tiene la ventaja de que siempre converge a la solución correcta, si bien el tiempo de convergencia puede ser en ocasiones demasiado largo al no usar un mecanismo óptimo. Recordemos que en cada iteración dividimos el intervalo en estudio en dos, por lo que si estamos muy lejos de la solución es posible que requiramos demasiadas iteraciones para obtener la solución que buscamos.

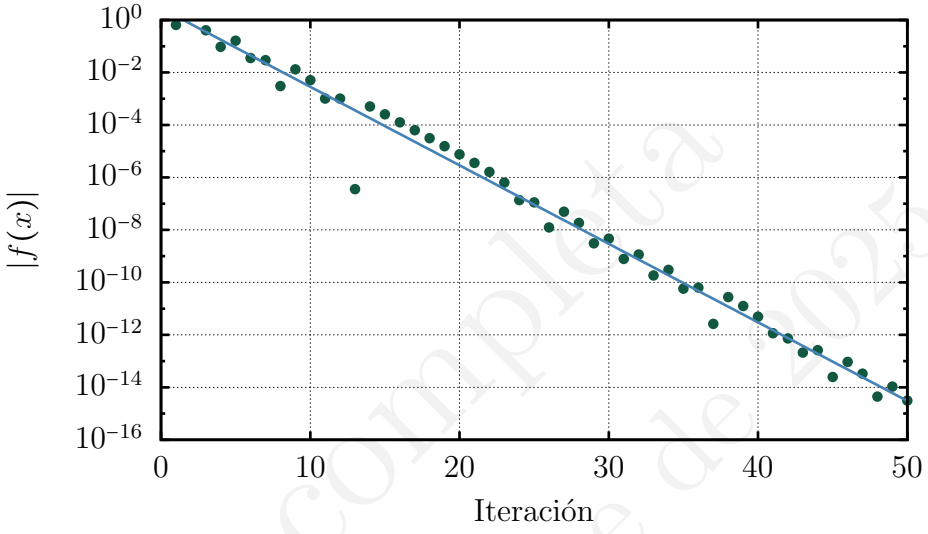

Figura 4.3. Comprobación del error cometido tras sucesivas iteraciones del método de la bisección. La recta de mejor ajuste tiene por expresión $\epsilon_n = |f(x_n)| = ab^{-n}$, con $a = (1{,}0 \pm 0{,}8)$ y $b = (1{,}99 \pm 0{,}02)$.

Ya sabemos que, en el entorno de un punto x_0, una función $f(x)$ puede ser escrita por su desarrollo en Taylor

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 + \dots$$

Supongamos que queremos encontrar el valor de x tal que $f(x) = 0$. Si x_0 es lo suficientemente cercano a x, podemos truncar el desarrollo en el primer orden de forma que

$$0 = f(x_0) + f'(x_0)(x - x_0)\,,$$

o sea

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}\,.$$

Obviamente, dado que se trata de una aproximación, el valor de x obtenido será igualmente aproximado. Lo cierto es que podemos realizar un procedimiento iterativo que nos permita obtener, paso a paso, una aproximación más cercana al verdadero cero de la función $f(x)$. Esto es, dado un valor inicial x_0, podemos obtener las sucesivas aproximaciones a la raíz de $f(x)$ mediante la serie de recurrencia

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}\,.$$

Observemos que, al contrario que con el método de la bisección, el método de Newton solo necesita de un único valor inicial x_0. Pero hace necesario el cálculo no solo del valor que toma la función en cada punto, sino también de su derivada primera.

Recuperemos el ejemplo que dimos al inicio del tema, donde la temperatura venía dada por

$$T(t) = 3{,}2^\circ\,\mathrm{C} + 5{,}1^\circ\,\mathrm{C}\sin(2\pi(t - 13\,\mathrm{h})/24\,\mathrm{h}).$$

El cód. 4.3 permite encontrar, sencillamente, el cero de la función de temperatura a partir de un valor inicial (en este caso $t = 15\,\mathrm{h}$).

```
from math import *

def f(x):
    return 3.2+5.1*sin(2*pi*(x-13)/24)

def dfdx(x):
    return 5.1*2*pi*cos(2*pi*(x-13)/24)/24

def newton(x):
    return x - f(x)/dfdx(x)

x0      = 15              # Punto inicial
tol     = 1E-6            # Tolerancia para la soluciin

print (f' {x}\t   {f(x)}'  )

```

```
17 # Criterio de convergencia: |f(x)| < tol
18 while abs(f(x0)) > tol:
19     x0 = newton(x0)
20     print (f'  {x0:.10f} \t {f(x0):.10f} ')
```

Código 4.3: Método de Newton (`newton.py`).

Después de cuatro iteraciones, la solución proporcionada (ver tab. 4.2 y fig. 4.4) es distinta de cero en menos de una millonésima parte.

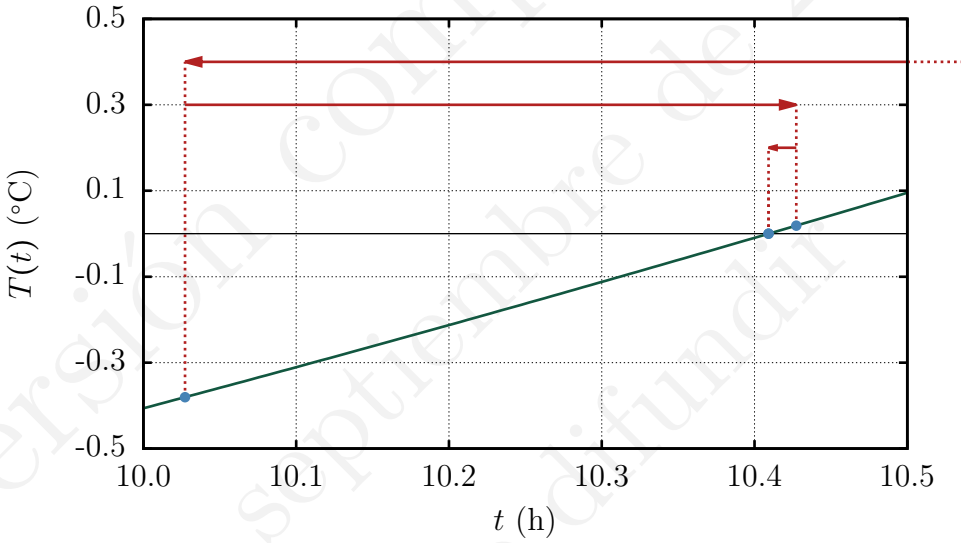

Figura 4.4. Iteraciones del método de Newton para obtener el cero de la función de temperatura a lo largo del día. Nótese que la cuarta iteración es inapreciablemente distinta a la tercera.

Sin entrar en muchos detalles, por completitud del tema, es interesante hacer notar que el método de Newton, al haber truncado el desarrollo en Taylor en segundo orden, tiene un orden de convergencia (en general) cuadrático en las sucesivas iteraciones. Basta notar, por ejemplo, cómo en la tab. 4.2 el número de dígitos iguales a cero aumenta al doble con cada iteración. Es por tanto un método que permite obtener, con menos iteraciones, un

Iteración	x	$f(x)$
1	10.0272295707	-0.3804453894
2	10.4273535117	0.0189308091
3	10.4092139126	0.0000359414
4	10.4091793419	0.0000000001

Tabla 4.2. Cuatro iteraciones del método de Newton obtenidas mediante el código newton.py.

resultado óptimo. Como contrapartida, nos obliga por un lado a evaluar la derivada primera de la función en cada punto y, además, presenta una gran sensibilidad al punto de partida. Nótese que el desarrollo del método ha supuesto que nos encontramos en las cercanías del cero de la función, lo que no siempre es así.

4. MÉTODO DE LA SECANTE

Hemos comprobado que el método de Newton permite, con un número menor de iteraciones que el método de la bisección, encontrar la solución a una ecuación tipo $f(x) = 0$. Este método tiene la contrapartida de necesitar la evaluación no solo de la función $f(x)$ en cada punto de iteración, sino también de su derivada primera $f'(x)$. En ocasiones evaluar el valor de la derivada puede ser complicado, por lo que es común reemplazar la derivada infinitesimal por una expresión en diferencias finitas

$$f'(x_n) \simeq \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}\,.$$

De este modo el método iterativo toma ahora la forma

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}\,,$$

que, observamos, requiere dos puntos anteriores a cada iteración efectuada. El cód. 4.4 muestra las modificaciones que hay que realizar respecto al método de Newton para obtener un algoritmo iterativo de obtención de raíces por el método de la secante.

```
from math import *

def f(x):
    return 3.2+5.1*sin(2*pi*(x-13)/24)

def secante(x0,x1):
    return x1 - f(x1)*(x1-x0)/(f(x1)-f(x0))

x0       =  7              # Primer  punto inicial
x1       = 15              # Segundo punto inicial
tol      = 1E-6            # Tolerancia para la solucion

print(f' {x} \t  \t {f(x)} ')

# Criterio de convergencia: |f(x)| < tol
while abs(f(x0)) > tol:
    x2 = secante(x0,x1)
    x0 = x1
    x1 = x2
    print (f'  {x2:.10f} \t  \t {f(x2):.10f}')
```

Código 4.4: Método de la secante (`secante.py`).

Si ejecutamos el código, la tab. 4.3 muestra las sucesivas iteraciones efectuadas. Vemos que en esta ocasión hemos necesitado 7 iteraciones para obtener la convergencia solicitada, frente a las 4 iteraciones que fueron necesarias con el método de Newton.

Puede probarse que el orden de convergencia del método de la secante tiende al número aureo $\phi \simeq 1{,}6$. Podemos preguntarnos entonces qué ventajas tiene el método de la secante frente al método de Newton. Como hemos indicado, la clave está en que no tenemos que evaluar la derivada de la función en cada iteración. En ocasiones, obtendremos que el tiempo de cómputo de la derivada es mayor que la evaluación de la propia función en un punto. En esos casos el método de la secante puede tener un comportamiento computacionalmente más óptimo que el método de Newton.

Iteración	x	$f(x)$
1	8.9869281046	-1.2254303252
2	10.0432931322	-0.3651402360
3	10.4916551211	0.0864846731
4	10.4057952864	-0.0035169551
5	10.4091503945	-0.0000300948
6	10.4091793521	0.0000000108
7	10.4091793418	-0.0000000000

Tabla 4.3. Iteraciones para encontrar los ceros de la función problema. Tras 7 iteraciones, $f(x)$ se diferencia de cero en menos de 10^{-10}.

5. REGULA FALSI

Los dos métodos anteriores (el método de Newton y su aproximación a la secante) muestran una mejor convergencia que el método de la bisección. Pero puede darse la circunstancia de que no converjan hacia la solución correcta. Por ello, existe una solución intermedia dada por el método de la falsa regla (o *regula falsi*) que mejora la convergencia del método de la bisección y asegura la convergencia.

Este método obtiene, dado un par de valores extremos, el punto que intersecta a la recta $y = 0$ usando la expresión del método de la secante. Sustituye uno de los extremos por este nuevo punto asegurándose las condiciones de aplicabilidad del teorema de Bolzano y repite el procedimiento hasta alcanzar la solución pedida. De este modo, aglutina las ventajas de los métodos vistos hasta ahora.

El cód. 4.5 ejemplifica cómo se puede programar una rutina que obtenga las sucesivas iteraciones para el método de *regula falsi*.

```
1 from math import *
2
3 def f(x):
4     return 3.2+5.1*sin(2*pi*(x-13)/24)
5
6 def secante(x0,x1):
7     return x1 - f(x1)*(x1-x0)/(f(x1)-f(x0))
```

```
8
9  xa       =  7              # Limite inferior del intervalo (f(xa)
10 xb       = 15              # Limite superior del intervalo (f(xb)
11 tol      = 1E-15            # Tolerancia para la soluciin
12 xc       = secante(xa,xb) # Valor inicial para xc
13
14 print(f' {xc} \t\t {f(xc) }')
15
16 # Criterio de convergencia: |f(xc)| < tol
17 while abs(f(xc)) > tol:
18     xc = secante(xa,xb) # Valor inicial para xc
19     print(f' {xc:.10f}  \t  {f(xc):.10f} '  )
20     if (f(xa)*f(xc)<0):
21         xb = xc
22     else:
23         xa = xc
```

Código 4.5: *Regula falsi* (`regula-falsy.py`).

Iteración	x	$f(x)$
1	8.9869281046	-1.2254303252
2	10.0432931322	-0.3651402360
3	10.3392623243	-0.0721486621
4	10.3970186626	-0.0126265336
5	10.4071042890	-0.0021568430
6	10.4088264526	-0.0003668654
7	10.4091193630	-0.0000623562
8	10.4091691485	-0.0000105974
9	10.4091776095	-0.0000018010
10	10.4091790474	-0.0000003061

Tabla 4.4. Iteraciones del método de *regula falsi* para encontrar el cero de la función problema.

En la tab. 4.4 y la fig. 4.5 pueden verse los resultados de ejecutar el cód. 4.5. Observamos que el número de iteracines necesarias para alcanzar el cero es mayor que para el método de la secante y menor que para el método de la bisección. En la fig. 4.6 se puede ver el orden de convergencia del método,

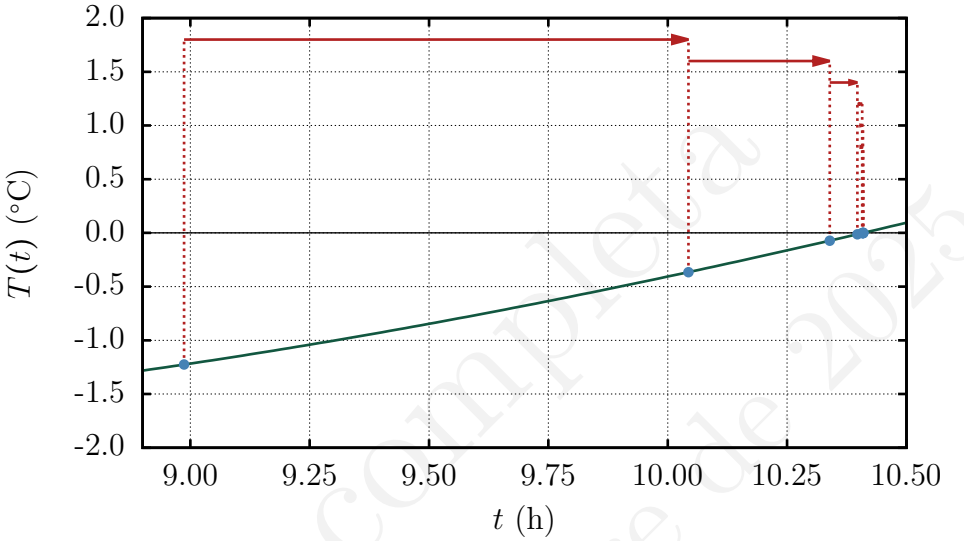

Figura 4.5. Iteraciones del método de *regula falsi* para obtener el cero de la función de temperatura a lo largo del día.

tras 25 iteraciones el error es del orden de 10^{-16}. La curva de ajuste a una función $y = ab^{-x}$ da como resultado $a = (11 \pm 2)$ y $b = (5{,}7 \pm 0{,}1)$. Observamos que el orden de convergencia es mayor que el que obteníamos con el método de la bisección.

6. EJERCICIOS

Ej. 4.1 — Utilice como criterio de convergencia $|x_n - x_{n-1}| < \epsilon$ y compare, para la misma tolerancia ϵ, el número de iteraciones necesarias respecto del criterio $|f(x_n)| < \epsilon$.

Ej. 4.2 — Considere la función $f(x) = (x-1)^4/(3x-3) - x^3 + 0{,}25$. Realice un estudio de convergencia análogo al que se observa en la fig. 4.3, para un criterio de convergencia $|x_n - x_{n-1}| < \epsilon$. Obtenga el orden de convergencia

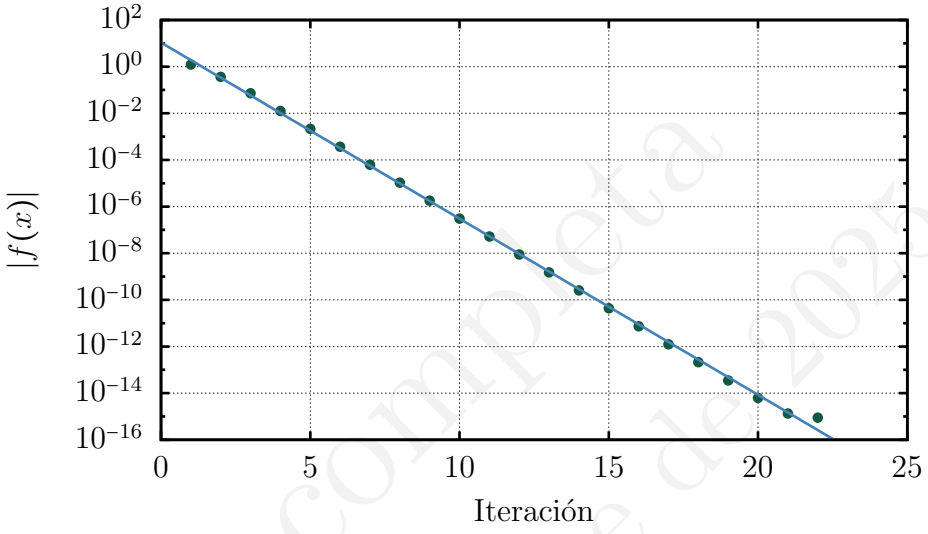

Figura 4.6. Iteraciones del método de la *regula-falsi* para obtener el cero de la función de temperatura a lo largo del día.

para este caso y discuta el resultado.

Ej. 4.3 — Sea $f(x) = \sigma/4 - \exp\left\{\frac{-x^2}{2\sigma^2}\right\}$, con $\sigma = 2{,}5$. Utilizando el método de la secante, haga un estudio de la convergencia del método con tres criterios de tolerancia distintos:

1. $|f(x_n)| < \epsilon$
2. $|x_n - x_{n-1}| < \epsilon$
3. $|x_n - x_{n-1}|/x_n < \epsilon$

Para el mismo valor de ϵ ¿con cuál de estos tres criterios se alcanza antes la convergencia? ¿Por qué?

Ej. 4.4 — Considere la función $f(x) = (x-2)^4 \cos(x) 2{,}5^x - 1$. Obtenga

todos los ceros de la función en el intervalo $[0, 5]$ usando dos métodos distintos. Realice una comparación del orden de convergencia de ambos métodos a partir del número de iteraciones.

Ej. 4.5 — La función $f(x) = x^4 e^{-\lambda x}$ (con $\lambda = 1/3$) tiene dos puntos en los que toma el valor $f(x) = 100$. Diseñe un método iterativo para encontrar dichos puntos. Antes, evalúe cuál de los métodos propuestos en este tema es el más adecuado.

Ej. 4.6 — Sea $f(x) = (x-1)^4/(3x-3) - x^3 + 0{,}25$. Obtenga analíticamente los ceros y escoja, de entre los métodos que conozca, aquel que requiera menos de 5 pasos en el intervalo $(-1, 1)$ para encontrar alguno de los ceros con un error de menos del 0,01 %.

Tema 5

DERIVACIÓN E INTEGRACIÓN NUMÉRICA

1. INTRODUCCIÓN

La derivada de una función $f(x)$ en el punto x se define como el límite, si existe, del cociente

$$\frac{df(x)}{dx} \triangleq \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}\,.$$

Respecto a la integral, definíamos la integral de Riemann como el límite, si existe, de la suma

$$\int_{x_0}^{x_f} dx f(x) \triangleq \lim_{n\to\infty} \frac{x_f - x_0}{n} \sum_{i=1}^{n} f\left(x_0 + i\frac{x_f - x_0}{n}\right).$$

Aprendimos en su momento que si $F(x)$ es una primitiva de $f(x)$, esto es, si

$$F(x) = \int dx f(x)\,,$$

entonces se tiene que

$$f(x) = \frac{dF(x)}{dx}\,,$$

que constituye **teorema fundamental del cálculo**, esto es, derivación e integración son operaciones inversas. Gracias a este teorema encontramos muchas expresiones analíticas para obtener integrales. No obstante, son muchos los casos en los que las integrales no son analíticamente resolubles, o en los que la obtención de una expresión explícita para una integral se vuelve tediosa. En este tema nos centraremos en la obtención de métodos numéricos para encontrar valores (aproximados o exactos) de cálculos de derivadas e integrales.

2. DERIVACIÓN NUMÉRICA

Recuperemos por un momento la definición de la derivada de una función $f(x)$ en un punto x_0

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}.$$

Esta expresión, que es exacta en el límite en que Δx tiende a cero, se vuelve aproximada para incrementos finitos. A priori, cuanto menor sea el tamaño del incremento Δx, mejor será la aproximación al valor real.

2.1. Fórmulas de dos puntos

Lo primero que nos preguntamos es cómo plantear la ecuación para la derivada numérica. Formalmente tenemos tres opciones a seguir:

1. Diferencia adelantada (*forward difference*). Define la derivada en un punto a partir de la derivada en un punto **posterior**, esto es

$$f'(x_0) \simeq \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}.$$

2. Diferencia atrasada (*backward difference*). Define la derivada en un punto a partir de la derivada en un punto **anterior**.

$$f'(x_0) \simeq \frac{f(x_0) - f(x_0 - \Delta x)}{\Delta x}.$$

3. Diferencia centrada (*central difference*). Define la derivada en un punto centrada en el punto x_0

$$f'(x_0) \simeq \frac{f(x_0 + \Delta x/2) - f(x_0 - \Delta x/2)}{\Delta x}.$$

Pongamos alguna de estas definiciones en su contexto. Supongamos la función $f(x) = \log(x^2)$, donde log hace referencia al logaritmo natural. Queremos evaluar su derivada en $x = e = 2{,}718\ldots$ Sabemos, analíticamente,

resolver el valor de su derivada en cualquier punto, dada por

$$f'(x) = \frac{2}{x}\,.$$

Si seguimos el método de la diferencia centrada obtenemos los datos que se reflejan en la tab. 5.1. Al comparar con el valor exacto $2/e = 0{,}73575888\ldots$ observamos que al reducir al tamaño del incremento Δx la aproximación se acerca más al valor buscado.

Δx	$f'(x)$
1	0.74
0.5	0.738
0.25	0.736
0.125	0.7359
0.0625	0.73579
0.03125	0.73577
0.015625	0.73576
0.0078125	0.735759

Tabla 5.1. Valores obtenidos para la derivada de la función $\log(x^2)$ en $x = e$, usando el método de diferencias centradas. El valor real de la derivada en ese punto es $f'(x) = 0{,}73575888\ldots$

La fig. 5.1 muestra un análisis de convergencia del método de diferencias centradas. Se observa un ajuste perfecto a una ecuación potencial $y(x) = ax^b$, con $a = -4{,}785 \pm 0{,}002$ y $b = 2{,}0009 \pm 0{,}0004$. Esto es, el orden de convergencia del método de diferencias centradas es cuadrático.

Más información:

Aunque no entraremos en detalles, para conseguir la convergencia cuadrática con diferencias centradas es necesario que la función $f(x)$ sea doblemente diferenciable. Este método, aunque presenta como ventaja una mejor convergencia frente a las diferencias adelantadas o atrasadas (que son linealmente convergentes) puede ser difícilmente aplicable en casos de funciones oscilatorias.

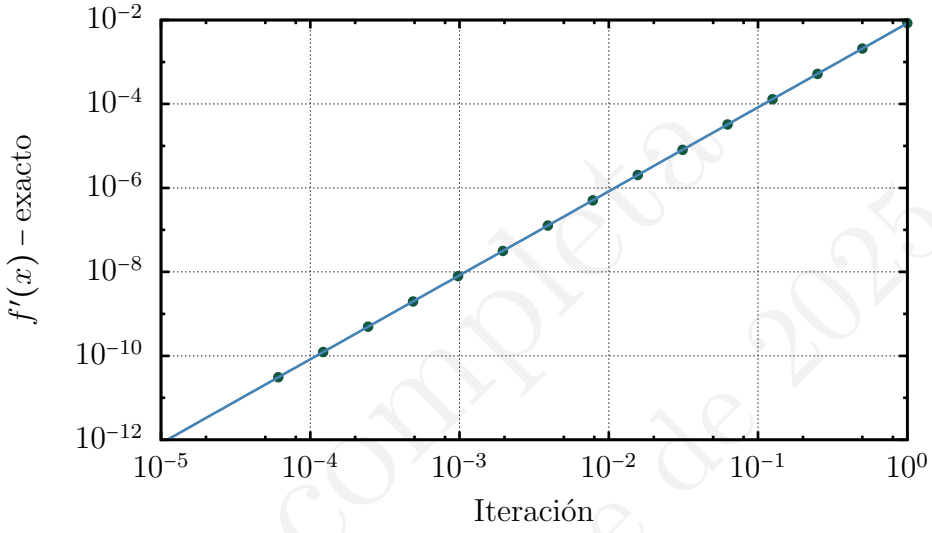

Figura 5.1. Aproximación a la derivada en función del tamaño del incremento, en escala doblemente logarítmica. Se observa un orden de convergencia cuadrático al comparar el valor obtenido con el valor exacto $2/e$.

2.2. Fórmulas de tres puntos

Tanto la fórmula de diferencias atrasadas como la fórmula de diferencias adelantadas se caracterizaban por tener una convergencia lineal. Esto es, el error que se comete al aproximar la derivada analítica por la función aproximada es del orden del intervalo de diferenciación Δx. Si queremos aumentar el orden de convergencia de la derivada en diferencias finitas podemos usar las llamadas **fórmulas de tres puntos**

$$f'(x) \simeq \frac{1}{2\Delta x}\left[-3f(x) + 4f(x+\Delta x) - f(x+2\Delta x)\right] ,$$
$$f'(x) \simeq \frac{1}{2\Delta x}\left[f(x+\Delta x) - f(x-\Delta x)\right] .$$

Estas fórmulas[1] se caracterizan por tener un orden de convergencia cuadrático, esto es, el error en la aproximación escala con el cuadrado del incremento Δx.

Más información:

Por completitud, mostramos aquí una aproximación a la derivada de una función a cinco puntos:

$$f'(x) \simeq \frac{1}{12\Delta x}\left[f(x-2\Delta x) - 8f(x-\Delta x) + 8f(x+\Delta x) - f(x+2\Delta x)\right] .$$

Esta fórmula, de diferencias centradas, tiene un error del orden de $(\Delta x)^4$. Si bien el resultado en la aproximación tiene un error mucho menor que el de, por ejemplo, una diferencia de dos puntos adelantada, requiere evaluar la función en cuatro puntos. Este procedimiento puede generalizarse para obtener fórmulas de $(n+1)$ puntos, que requieren la evaluación de, al menos, n puntos.

3. INTEGRACIÓN NUMÉRICA

La integral de Riemann se define como el límite, si existe, de la suma de rectángulos infinitamente finos que se correspondían al área encerrada bajo una curva determinada. El procedimiento, sencillo, consistía en aproximar el área por rectángulos de una anchura determinada y sumar la superficie (base por altura) de cada uno de ellos para obtener el área total. Decíamos que en el límite en el que rectángulos son infinitamente finos se obtiene *exactamente* el área bajo la curva.

El tema 2 detallaba cómo obtener, de forma analítica, el valor de la integral para un conjunto de funciones que llamábamos *integrables*. En este tema estamos interesados en métodos numéricos que nos permitan obtener, de forma aproximada, el valor de una integral sin necesidad de recurrir al cálculo de su primitiva. Así, podemos evaluar nuestra función continua $f(x)$

[1] Obsérvese la analogía con la fórmula de diferencias centradas.

en puntos concretos de forma que la integral en un intervalo (x_0, x_N) pueda aproximarse por

$$\int_{x_0}^{x_N} dx f(x) \simeq \sum_i f(x_i)\Delta x_i \,,$$

donde hemos dividido el intervalo continuo (x_0, x_N) en segmentos (x_0, x_1), (x_1, x_2), $\ldots$,(x_{N-1}, x_N). Cada uno de estos segmentos tiene una anchura $\Delta x_i = x_{i+1} - x_i$ (el procedimiento sería análogo si considerásemos $x_i - x_{i-1}$). En forma más general podemos escribir

$$\int_{x_0}^{x_N} dx f(x) \simeq \sum_{i=0}^{N} w_i f(x_i) \,,$$

lo que recibe el nombre de **método de cuadratura** para la aproximación de la integral. Estudiaremos, a lo largo de la próxima sección, distintos métodos para obtener el valor de los coeficientes w_i y, por tanto, distintas formas de obtener una aproximación por cuadratura a la integral.

Veamos un ejemplo para fijar ideas. Supongamos que queremos calcular el valor de la integral de la función $f(x) = -x^3 + 2x + 1$ en el intervalo $[0, 1{,}6]$. Mediante un lenguaje de programación (ver cód. 5.1) podemos construir un procedimiento iterativo elemental para calcular la integral de de la función solicitada.

```
1  def f(x):
2      return 1+2*x-x**3
3
4  x0       = 0.0
5  xf       = 1.6
6  dx       = 0.1
7  integral = 0.0
8  x        = x0
9
10 print (f' {x} \ {f(x)}    ')
11
12 while x < xf:
13     print( f'  {x} \t  {f(x)} '   )
14     integral += f(x)*dx
15     x += dx
```

```
16
17 print ( f'  \n Integral value: \t {integral}  '  )
```

Código 5.1: Cálculo de una integral en python (`integrate.py`).

La fig. 5.2 muestra, mediante rectángulos, el valor aproximado de la función en el intervalo solicitado, donde hemos subdividido el intervalo original en segmentos de anchura $\Delta x = 0{,}1$. En trazo continuo mostramos también la función $f(x)$. Observamos que hay diferencias entre ambas representaciones. Hasta mitad del intervalo el área de los rectángulos es inferior al área bajo la curva. A partir de ahí, ocurre que el área bajo la curva es sensiblemente inferior al área que encierran los rectángulos.

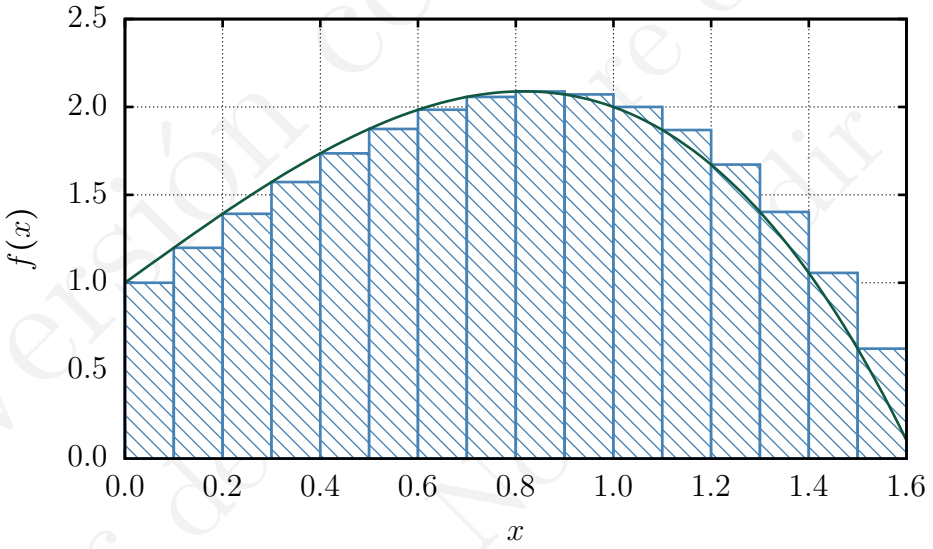

Figura 5.2. Función $f(x) = 1 + 2x - x^3$ (en trazo continuo) y aproximación mediante rectángulos de anchura $\Delta x = 0{,}1$. Obsérvese que los rectángulos ajustan, de forma somera, a la función continua.

Es interesante comentar *cuán aceptable* es la aproximación realizada. En este sencillo ejemplo podemos comparar, cuantitativamente, el valor analítico de la integral de $f(x)$ con la aproximación por rectángulos. En efecto, el cálculo

integral nos da un valor

$$\int_0^{1,6} dx(1+2x-x^3) = \left[x + x^2 - \frac{x^4}{4}\right]_0^{1,6} = 2{,}5216\,,$$

en tanto que área de los rectángulos es

$$\sum_i f(x_i)\Delta x_i = 2{,}5600\,,$$

lo que nos da un error relativo del 1,5 % en el cálculo aproximado. A priori, si deseáramos obtener una aproximación con un error menor podríamos disminuir el tamaño del segmento Δx (aumentando por tanto el número de subintervalos) de forma que, por ejemplo, si duplicamos el número de intervalos el error relativo será del 0,8 %. ¿Y si volvemos a duplicar el número de intervalos? La fig. 5.3 muestra la distancia euclídea entre la solución analítica (teó) y la solución numérica (num) al aumentar el número de subintervalos

$$d(\text{teó}, \text{num}) = \sqrt{(\text{teó} - \text{num})^2}\,.$$

Observamos que, si queremos disminuir la distancia entre la solución teórica y la numérica en un orden de magnitud, debemos aumentar en un orden de magnitud el número de intervalos considerado. Veremos, a continuación, algunos métodos de cuadratura que mejoran la convergencia del método aproximado.

4. MÉTODOS DE CUADRATURA

La fig. 5.4 muestra, en el intervalo $(0{,}4, 0{,}7)$, la curva $f(x) = 1 + 2x - x^3$ y su aproximación por cuadratura siguiendo el método explicado en la sección anterior. Fijémonos en el rectángulo correspondiente al intervalo $(0{,}5, 0{,}6)$. El área de dicho rectángulo es

$$\begin{aligned} A_i^{\text{rec}} &= f(x_i)(x_{i+1} - x_i) \\ A_5^{\text{rec}} &= f(0{,}5)(0{,}6 - 0{,}5) = 0{,}1875\,. \end{aligned}$$

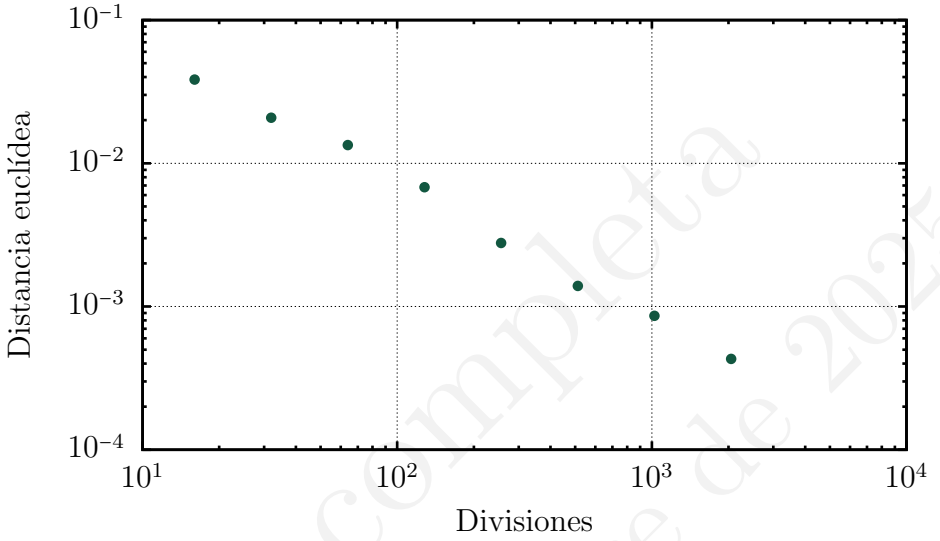

Figura 5.3. Distancia euclídea entre la solución analítica y la solución numérica para el método de cuadratura con $\Delta x_i = x_{i+1} - x_i$.

Si hacemos el cálculo de la integral de $f(x)$ en ese mismo intervalo tenemos

$$\int_{0,5}^{0,6} dx f(x) = 0{,}1932\,.$$

El hecho de que el área bajo la curva sea mayor que el área del rectángulo es algo que ya se observa en la fig. 5.4. Una forma de obtener una mejor aproximación al resultado correcto sería, en lugar de considerar un rectángulo con el segmento AB, considerar un trapecio con el segmento AC. En este caso el área del trapecio es

$$A_i^{\text{tra}} = \frac{f(x_i) + f(x_{i+1})}{2}(x_{i+1} - x_i)$$
$$A_5^{\text{tra}} = \frac{f(0{,}5) + f(0{,}6)}{2}(0{,}6 - 0{,}5) = 0{,}1930\,,$$

que se aproxima mucho mejor el resultado exacto de la integral.

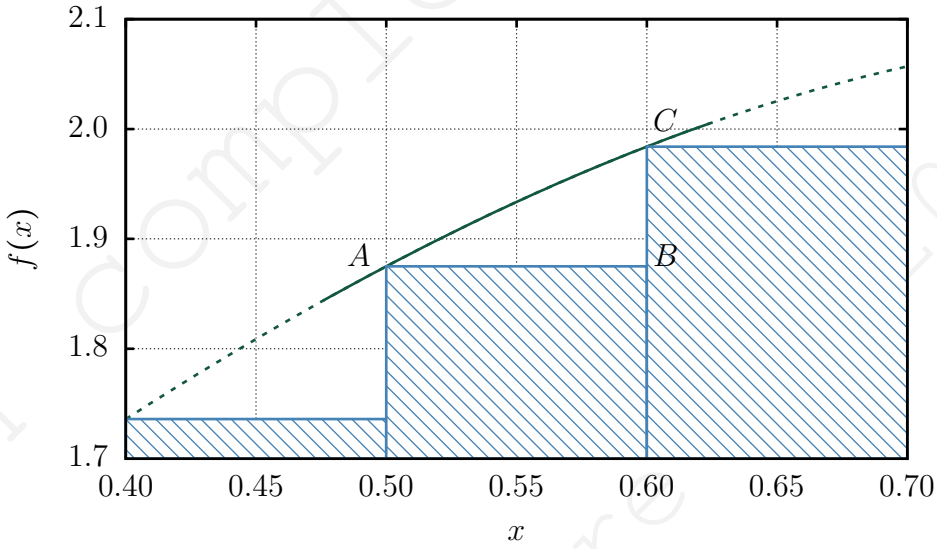

Figura 5.4. Aproximación del área bajo una curva $f(x)$ por rectángulos. Obsérvese que el área del rectángulo es menor al área bajo la curva.

Este método de cuadratura, llamado **método de los trapecios**, formalmente realiza una aproximación de la función $f(x)$ a una función lineal a trozos. La ventaja de este método frente a los rectángulos utilizados anteriormente es que puede probarse que para polinomios de grado 1 o inferior el método de los trapecios da el resultado exacto.

El cód. 5.2 muestra cómo se puede realizar el cálculo de la integral de $f(x)$ por el método de los trapecios. Si lo comparamos con el cód. 5.1 vemos que la única diferencia radica en la línea 15.

```
1 def f(x):
2     return 1+2*x-x**3
3
4 x0       = 0.0
5 xf       = 1.6
6 dx       = 0.1
7 integral = 0.0
```

```
8  x         = x0
9
10 print( f'  {x} \t  {f(x)} '   )
11 while x < xf:
12     print( f'  {x} \t  {f(x)} '   )
13     integral += 0.5*(f(x+dx)+f(x))*dx
14     x        += dx
15
16 print ( f'  \n Integral value: \t {integral}  '  )
```

Código 5.2: Método de los trapecios para obtener una integral (`integrate-trapezoid.py`).

La fig. 5.5 muestra la aproximación por trapecios al área bajo la curva. El cód. 5.2 nos muestra, para 16 subdivisiones, un valor de la integral $A^{\text{tra}} = 2{,}5152$, lo que nos da un error relativo del 0,3 %. La distancia euclídea, para esas pocas 16 subdivisiones, es $d = 6{,}4 \times 10^{-3}$, equivalente a haber utilizado 128 divisiones usando el método de los rectángulos.

Acabamos de ver que una aproximación lineal a trozos mejora el cálculo integral sustancialmente. Parece lógico que, si en lugar de usar una aproximación lineal usamos una aproximación en polinomios de mayor grado, el ajuste con respecto al resultado exacto será mejor. En concreto, el **método de Simpson** utiliza un polinomio de tercer grado para aproximar la función $f(x)$, de forma que el resultado obtenido será exacto para funciones polinómicas de este grado (o menor).

El método de Simpson define, para un intervalo (x_A, x_B), un punto intermedio $x_C = (x_B - x_A)/2$ de forma que

$$\int_{x_A}^{x_B} dx f(x) = \frac{\Delta x}{6} \left(f(x_A) + 4f(x_C) + f(x_B) \right) .$$

Para nuestra función de ejemplo $f(x) = 1 + 2x - x^3$ en el intervalo $(0{,}5, 0{,}6)$ tenemos

$$A_5^{Sim} = \frac{0{,}6 - 0{,}5}{6} \left(f(0{,}5) + 4f(0{,}55) + f(0{,}6) \right) = 0{,}1932 ,$$

que es el resultado exacto dado por la integral.

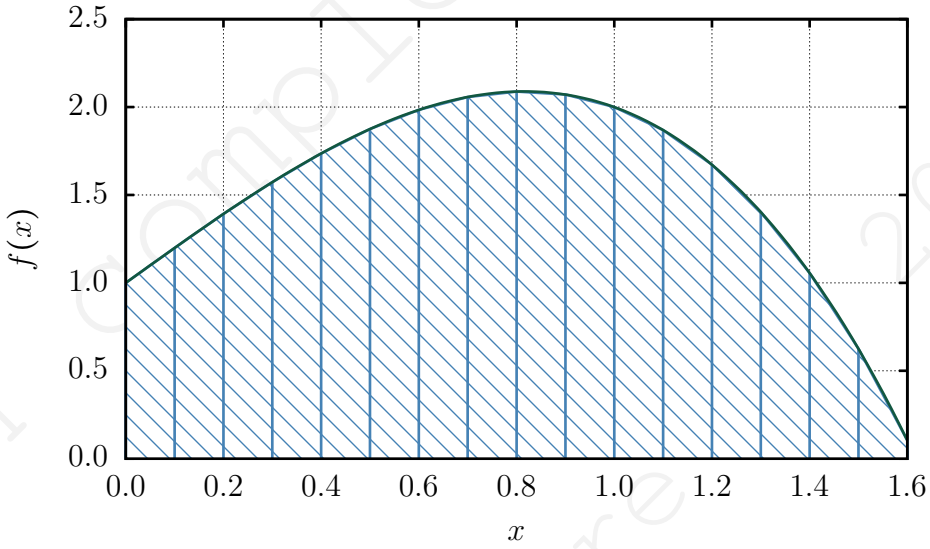

Figura 5.5. Aproximación a la integral según el método de los trapecios.

Si aplicamos el método de Simpson en un código podemos ver que, en efecto, el resultado de integrar numéricamente la función $f(x)$ coincide con el valor exacto calculado analíticamente.

```
1  def f(x):
2      return 1+2*x-x**3
3
4  x0       = 0.0
5  xf       = 1.6
6  dx       = 0.1
7  integral = 0.0
8  x        = x0
9
10 print( f'  {x} \t  {f(x)} '   )
11 while x < xf:
12     print( f'  {x} \t  {f(x)} '   )
13     integral += (f(x)+4*f(x+dx/2.0)+f(x+dx))*dx/6.0
```

```
14      x          += dx
15
16  print ( f'  \n Integral value: \t {integral}  '  )
```

Código 5.3: Método de Simpson (`integrate-simpson.py`).

5. EJERCICIOS

Ej. 5.1 — Considere la función $f(x) = \log(x+1)\sin(x)$. Obtenga, tanto adelantada como atrasada, la derivada aproximada de la función en $x = 0$, $x = \pi/4$, $x = \pi/2$ y $x = 3\pi/4$. Utilice en todos los casos un incremento Δx tal que el error relativo cometido en el cálculo sea menor del 0,1 %

Ej. 5.2 — Compare dos métodos de cuadratura para la obtención de la integral

$$\int_{2\pi}^{12\pi} dx\, 5\frac{\sin(x)}{x} + 5$$

Analice la convergencia de ambos métodos utilizando como estimador del error $\epsilon_n = |\mathrm{Int}(\Delta_n) - \mathrm{Int}(\Delta_{n-1})|$, donde $\mathrm{Int}(\Delta_n)$ es el valor de la integral numérica usando un incremento Δ_n tal que $\Delta_n = \frac{1}{2}\Delta_{n-1}$.

Ej. 5.3 — Supongamos que tenemos una función polinómica de grado 3. ¿Qué método de integración usaría para obtener un resultado exacto? Proponga un ejemplo y obtenga el resultado, comparándolo con el valor exacto analítico y con el resultado que se obtendría, para el mismo incremento Δx, con el método de los trapecios.

Tema 6

SOLUCIONES NUMÉRICAS DE ECUACIONES DIFERENCIALES ORDINARIAS

1. INTRODUCCIÓN

En el tema 6 estudiamos la resolución de ecuaciones diferenciales ordinarias (EDOs). Recordemos que una EDO de primer orden puede escribirse como

$$y'(t) = f(t, y(t)), \tag{6.1}$$

donde, así escrita, t juega el papel de variable independiente e $y(t)$ es la variable dependiente. Aquí, $f(t, y(t))$ es una función genérica que suponemos conocida y que, por el momento, dejaremos escrita de forma implícita.

Ya hemos visto cómo resolver, analíticamente y para algunas formas funcionales explícitas, EDOs del tipo de la ec. (6.1). Lo que veremos en este tema es cómo encontrar una expresión cerrada (recurrente) para $y(t)$ que satisfaga, precisamente, la ec. (6.1), sin necesidad de recurrir a los procedimientos vistos en el tema 6.

También veremos, en este tema, cómo resolver ecuaciones diferenciales en derivadas parciales, del tipo que se introdujo en el tema 8, también de forma numérica.

2. ECUACIONES DIFERENCIALES ORDINARIAS

2.1. Método de Euler

Supongamos conocido, para un instante $t = t_0$, el valor que toma $y(t)$ en ese instante. Esto es, supongamos conocido $y(t_0)$. Si consideramos un variación infinitesimal de tiempo Δt, sabemos que $y(t_0 + \Delta t)$ puede escribirse a partir

de su expansión en Taylor como

$$y(t_0 + \Delta t) = y(t_0) + y'(t_0)\Delta t + \frac{1}{2}y''(t_0)(\Delta t)^2 + \dots ,$$

donde $y'(t_0)$ representa la evaluación de la derivada primera de $y(t)$ en el instante $t = t_0$.

Para valores de Δt suficientemente pequeños podemos quedarnos con el primer orden en la aproximación de forma que

$$y(t_0 + \Delta t) \simeq y(t_0) + y'(t_0)\Delta t\,,$$

que, sustituyendo $y'(t_0)$ por su valor dado en la ecuación (6.1) nos deja

$$y_1 = y(t_0) + f(t_0, y(t_0))\Delta t\,,$$

donde $y_1 \triangleq \tilde{y}(t_0+\Delta t)$ es la aproximación al valor que toma $y(t)$ en el instante $t_1 = t_0 + \Delta t$.

Más información:

Nótese que este desarrollo tiene un **error de truncamiento** (local) $\mathcal{O}(\Delta t)$[a]. Cuanto mayor sea el valor de Δt (cuanto más grande sea nuestro paso de tiempo), peor será la aproximación realizada y, por tanto, cabe esperar que las diferencias con una predicción teórica sean mayores.

[a] La notación $\mathcal{O}(\Delta t)$ (notación "O grande") indica una cota máxima del error en el método de Euler, en este caso, el error viene determinado por el tamaño del paso de tiempo Δt.

Si llamamos $t_1 = t_0 + \Delta t$, tenemos entonces

$$y(t_1) = y(t_0) + f(t_0, y(t_0))\Delta t\,.$$

El razonamiento aquí seguido puede usarse para, partiendo de un valor conocido $y(t_1)$, encontrar el valor $y(t_2)$ en $t_2 = t_1 + \Delta t$ como

$$y(t_2) = y(t_1) + f(t_1, y(t_1))\Delta t\,,$$

o, en términos más genéricos, conocido el valor $y(t_n)$, su valor un instante posterior t_{n+1} será

$$y_{n+1} = y_n + f(t_n, y_n)\Delta t\,, \tag{6.2}$$

donde hemos usado la notación $y_n \triangleq \tilde{y}(t_n) = \tilde{y}(t_0 + n\Delta t)$, con $\tilde{y}(t)$ la aproximación a la solución $y(t)$ en el instante t.

La ecuación (6.2) es el llamado **método de Euler** para la solución de ecuaciones diferenciales. Conocida la forma funcional de $y'(t)$ según la ec. (6.1) y el valor $y(t_0)$ en un instante inicial t_0, el método de Euler nos permite obtener la evolución temporal $y(t)$ para cualquier instante posterior a t_0. Dado un valor $y(t_0)$ podemos obtener, mediante un proceso iterativo, los sucesivos valores $y(t_1)$, $y(t_2)$, $y(t_3)$, etc. sin más que sustituir para cada valor anterior en la ecuación (6.2).

El cód. 6.1 implementa el método de Euler para un conjunto de N pasos de tiempo, suponiendo dados los valores iniciales t_0 e y_0, y permite obtener los nuevos valores de y_n y t_n para cada iteración del bucle.

```
1  // Parametros del programa
2  Dar y0                          // Valor inicial de y
3  Dar t0                          // Tiempo inicial
4  Dar dt                          // Paso de tiempo
5  i  =  0                         // Contador de pasos
6  N  =  Numero_Total_de_pasos
7
8  // Derivada de y respecto del tiempo
9  Escribir funcion f(t,y)
10
11 // Bucle
12 Mientras i < N; hacer
13   y  = y0 - f(t0,y0)*dt
14   t  = t0 + dt
15   y0 = y
16   t0 = t
17   Sumar 1 a i
18   Imprimir valores: t, y
19   Repetir
```

Código 6.1: Pseudocódigo de Euler.

Planteemos el siguiente ejemplo para fijar ideas. Supongamos una manzana que se encuentra en reposo, colgada de un árbol a una altura h_0 del suelo. En un momento dado, $t_0 = 0\,\mathrm{s}$, la manzana se desprende del árbol descendiendo con una aceleración constante g. El movimiento que realiza la manzana es un Movimiento Rectilíneo Uniformemente Acelerado (MRUA), cuyas ecuaciones, según las leyes de Newton, vienen dadas por

$$\begin{aligned} y(t) &= h_0 - \frac{1}{2}gt^2\,, \\ v(t) &= y'(t) = -gt\,, \\ a(t) &= y''(t) = -g = \mathrm{cte}\,, \end{aligned}$$

donde hemos tomado como origen de coordenadas el suelo de forma que $y(t)$ decrece con el tiempo.

Formalmente, tenemos una ecuación diferencial de orden 2 que describe el movimiento, dada por

$$y''(t) = -g\,.$$

Esta ecuación diferencial puede escribirse como un conjunto de dos ecuaciones diferenciales de orden 1

$$\begin{aligned} v'(t) &= -g\,, \\ y'(t) &= v(t)\,. \end{aligned}$$

Para simplificar el problema, consideremos la conocida solución para la velocidad $v(t) = -gt$, de forma que podemos considerar este problema como descrito con una única ecuación diferencial de primer orden

$$y'(t) = -gt\,.$$

Así, con el método de Euler podemos obtener, para cada instante de tiempo t_n, el valor aproximado que tomará la altura de la manzana, $y_n = \tilde{y}(t = t_n)$, partiendo de la altura inicial $y(t_0) = h_0$:

$$\begin{aligned} y_1 &= y_0 + v_0\Delta t\,, \\ y_2 &= y_1 + v_1\Delta t\,, \\ y_3 &= y_2 + v_2\Delta t\,, \\ &\vdots \\ y_{n+1} &= y_n + v_n\Delta t\,. \end{aligned}$$

Particularizado a este caso tenemos

$$\begin{aligned} y_1 &= h_0\,, \\ y_2 &= y_1 - gt_1\Delta t\,, \\ y_3 &= y_2 - gt_2\Delta t\,, \\ &\vdots \\ y_{n+1} &= y_n - gt_n\Delta t\,. \end{aligned}$$

Nótese que podemos obtener directamente el valor de la altura y_2 en función del instante inicial sin más que sustituir y_1 por su valor

$$y_2 = h_0 - g(\Delta t)^2\,,$$

donde hemos usado que $t_1 = t_0 + \Delta t$ y tomamos $t_0 = 0\,\mathrm{s}$. Análogamente

$$\begin{aligned} y_3 &= (h_0 - g(\Delta t)^2) - 2g(\Delta t)^2 \\ &= h_0 - 3g(\Delta t)^2\,, \end{aligned}$$

donde, esta vez, $t_2 = t_1 + \Delta t = 2\Delta t$. Si continuamos la serie obtendremos los valores de y_n para cada iteración

$$\begin{aligned} y_4 &= h_0 - 6g(\Delta t)^2\,, \\ y_5 &= h_0 - 10g(\Delta t)^2\,, \\ y_6 &= h_0 - 15g(\Delta t)^2\,. \end{aligned}$$

Si nos fijamos en los coeficientes que acompañan al factor $g(\Delta t)^2$ veremos que se cumple la siguiente relación para cada valor de y_i

i	1	2	3	4	5	6	...
coeficiente	0	1	3	6	10	15	...

tabla que puede escribirse como

i	1	2	3	4	...
coeficiente	0	$0+1$	$0+1+2$	$0+1+2+3$	

De forma que para cualquier y_{n+1} tendremos

$$\begin{aligned} y_{n+1} &= h_0 - g(\Delta t)^2 \sum_{i=0}^{n} i \\ &= h_0 - \frac{1}{2} gn(n+1)(\Delta t)^2 \\ &= h_0 - \frac{1}{2} g(n\Delta t)^2 - \frac{1}{2} gn(\Delta t)^2 \,. \end{aligned}$$

Dado que este problema es analíticamente resoluble, podemos comparar la solución dada por el método de Euler con la solución exacta

$$y(t) = h_0 - \frac{1}{2} gt^2 \,,$$

de forma que podemos realizar una estimación del error que cometemos al utilizar la solución aproximada, dado por

$$\begin{aligned} \epsilon &= y(t) - y(t_n) \\ &= -\frac{1}{2} gn(\Delta t)^2 \,, \end{aligned}$$

Más información:

Si observamos detenidamente esta expresión, veremos que el error cometido al aproximar la solución exacta por la solución dada por el método de Euler escala linealmente con el número de pasos de tiempo, n. De esta forma, cuanto mayor sea su valor, mayor será la diferencia (a tiempos largos) entre la solución exacta y la aproximada. esto es, el error será tanto mayor cuanto mayor sea el paso de tiempo utilizado, y cuanto mayor sea el número de pasos de tiempo.

El cód. 6.2 permite obtener, mediante un proceso iterativo, el valor de la altura en función del tiempo.

```
g=9.81

def dydt(t,y):
    return -g*t

y = 1.4     # Altura inicial
t = 0.0     # Tiempo inicial
dt= 0.025   # Paso de tiempo

while y >= 0:
    y += dydt(t,y)*dt
    t += dt
    print (f'  {t:.6f} \t {y:.6f}  ')
```

Código 6.2: Método de Euler (`euler.py`).

La fig. 6.1 muestra, en trazo continuo, la solución exacta para la posición en función del tiempo; en puntos, los resultados al aplicar el método de Euler para $\Delta t = 0{,}1\,\mathrm{s}$, $\Delta t = 0{,}05\,\mathrm{s}$ y $\Delta t = 0{,}025\,\mathrm{s}$. Un análisis pormenorizado nos muestra que, en efecto, al incrementar el número de pasos de tiempo **todas** las soluciones numéricas se alejan cada vez más de la solución exacta. Por otro lado, cuanto menor es el paso de tiempo, menor es la diferencia entre la solución exacta y la aproximación dada por el método de Euler.

Si nos fijamos (o realizamos el cálculo analítico) en un tiempo $t \simeq 0{,}54\,\mathrm{s}$ la manzana ha alcanzado el suelo. De este modo, considerar en el método de Euler un paso de tiempo $t = 0{,}1\,\mathrm{s}$ es a priori una apuesta arriesgada, habida cuenta de que en tan solo 5 pasos hemos realizado la trayectoria completa. En general, y como primera aproximación, una medida cualitativa de cuán grande o pequeño debe ser el paso de tiempo lo da el número de iteraciones a realizar. Cuanto menor sea el paso de tiempo mayor será el número de realizaciones pero, a cambio, el resultado será más preciso para tiempos mayores.

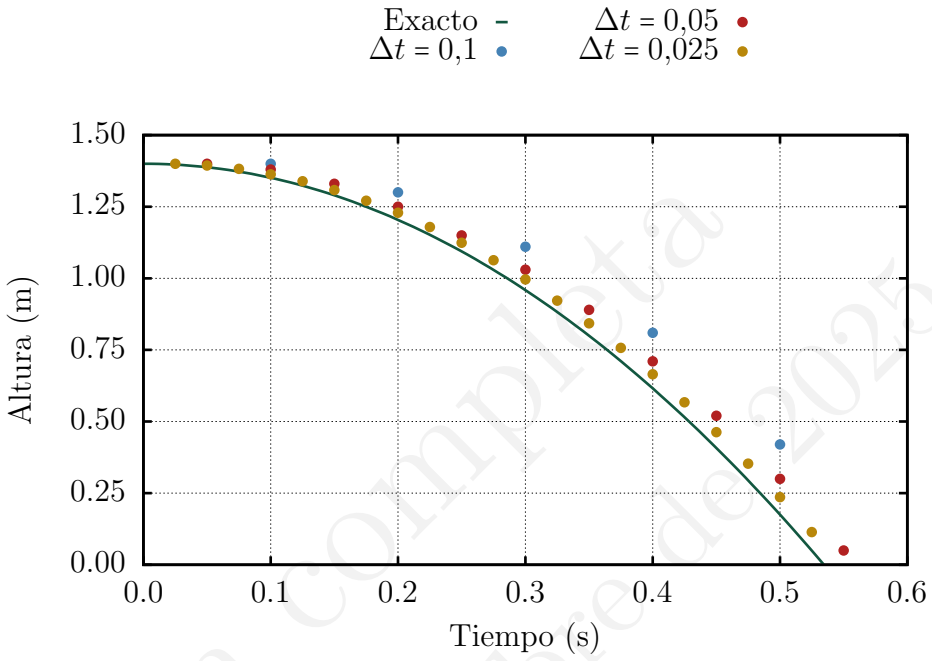

Figura 6.1. Movimiento de caída libre, partiendo del reposo, para un cuerpo que desciende desde una altura inicial $h_0 = 1{,}4\,\text{m}$. En línea continua se representa la solución exacta para el movimiento. Con puntos, de arriba a abajo, las aproximaciones numéricas para $\Delta t = 0{,}1\,\text{s}$, $\Delta t = 0{,}05\,\text{s}$ y $\Delta t = 0{,}025\,\text{s}$.

2.2. Métodos predictor-corrector

Hemos comentado que el método de Euler tiene un error de truncamiento del orden de $\mathcal{O}(\Delta t)$. Por ello, su uso no está muy extendido y es recomendable utilizar métodos alternativos que presentan errores del orden de $\mathcal{O}(\Delta t^2)$ (métodos de Euler modificado o de Heun). Los **métodos predictor-corrector** se sustentan en el uso de dos pasos diferenciados para cada iteración del algoritmo:

1. Paso de predicción. Suele tratarse de un método explícito que *anticipa* el valor que va a tomar la función en el siguiente paso de tiempo.

2. Paso de corrección. El valor previamente calculado es introducido en un método implícito que refina la aproximación inicial, haciendo uso no solo del valor intermedio calculado sino también del valor inicial.

El **método de Euler modificado**, por ejemplo, consiste en utilizar el predictor

$$\tilde{y}_{n+1} = y_n + f(t_n, y_n)\Delta t$$

(nótese que este paso es el método de Euler original) y el corrector

$$y_{n+1} = y_n + \frac{1}{2}\left(f(t_n, y_n) + f(t_{n+1}, \tilde{y}_{n+1})\right)\Delta t$$

Puede probarse que este método tiene un error de truncamiento del orden de $\mathcal{O}(\Delta t^2)$. Notemos que este es un método simétrico, que da el mismo peso al paso del predictor y al paso del corrector.

El cód. 6.3 muestra cómo se podría implementar un método predictor-corrector de Euler. La salida del código se muestra en la fig. 6.2, donde además se compara con la solución dada por el cód. 6.2. Observamos que, para el mismo paso de tiempo $\Delta t = 0{,}5$, el Predictor-Corrector da un resultado óptimo.

```
g=9.81

def dydt(t,y):
    return -g*t

y = 1.4     # Altura inicial
t = 0.0     # Tiempo inicial
dt= 0.05    # Paso de tiempo

while y >= 0:
    f      = dydt(t,y)
    ytilde = y + f*dt
    y     += 0.5*(f + dydt(t+dt,ytilde))*dt
    t += dt
    print(f'  {t:.6f} \t  {y:.6f} ')
```

Código 6.3: Predictor-corrector de Euler (`predictor-corrector.py`).

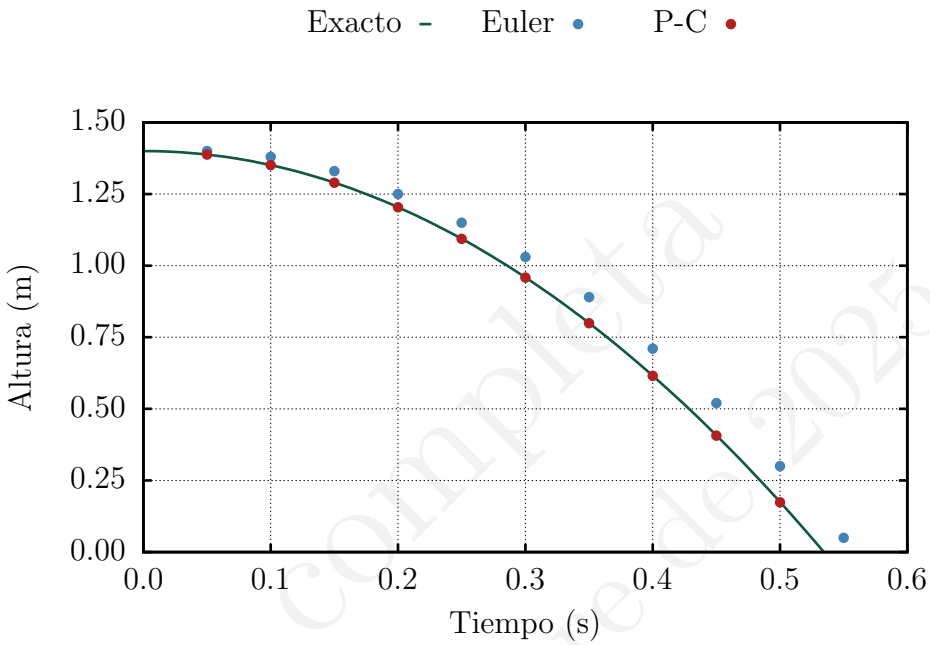

Figura 6.2. Comparación entre el método de Euler (en azul) y el método Predictor-Corrector (en rojo) para el movimiento de caída libre, partiendo del reposo, para un cuerpo que desciende desde una altura inicial $h_0 = 1{,}4\,\text{m}$. En línea continua se representa la solución exacta para el movimiento. El paso de tiempo utilizado es, para ambos métodos, $\Delta t = 0{,}5$.

2.3. Método de Runge-Kutta de cuarto orden

El método predictor-corrector tiene un error de truncamiento cuadrático, lo que supone una ventaja frente al método de Euler, con error lineal. Podemos no obstante ir más allá y plantear métodos de orden superior. Probablemente el método de Runge-Kutta de orden cuatro (RK4) es de los más conocidos e implementados en la bibliografía. Como su nombre indica, su error de truncamiento es cuártico. Este método subdivide cada iteración en

4 etapas

$$
\begin{aligned}
k_1 &= f(t_n, y_n)\Delta t \\
k_2 &= f(t_n + \Delta t/2, y_n + k_1/2)\Delta t \\
k_3 &= f(t_n + \Delta t/2, y_n + k_2/2)\Delta t \\
k_4 &= f(t_n + \Delta t, y_n + k_3)\Delta t
\end{aligned}
$$

de forma que, conocida la posición y_n en un tiempo t_n, una nueva iteración vendrá dada por

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

3. EJERCICIOS

Ej. 6.1 — Supongamos que la altura inicial de la manzana es $h_0 = 1{,}4\,\mathrm{m}$. Para $g = 9{,}8\,\mathrm{m\,s^{-2}}$ y un instante inicial $t_0 = 0\,\mathrm{s}$, comparar la solución exacta con la aproximación dada por el método de Euler para $\Delta t = 0{,}1\,\mathrm{s}$, $\Delta t = 0{,}05\,\mathrm{s}$ y $\Delta t = 0{,}025\,\mathrm{s}$.

Ej. 6.2 — La ecuación que describe un oscilador armónico unidimensional es $\ddot{x}(t) = -\frac{k}{m}x$, donde $\ddot{x} \triangleq \frac{d^2x}{dt^2}$, k es la constante de restauración (constante elástica) del oscilador y m su masa. Recordando que una ecuación diferencial ordinaria de segundo orden puede escribirse como un conjunto de dos ecuaciones diferenciales ordinarias de primer orden, utilice el método de Euler para obtener la evolución temporal de la posición del oscilador suponiendo que, en el instante incial, $x(0) = 0{,}1A$, siendo $A = 1{,}0\,\mathrm{cm}$ la amplitud de oscilación. Utilice un paso de tiempo tal que el error después de un tiempo $t = 2T$ (siendo T el periodo de oscilación) sea menor del 1 %.

Ej. 6.3 — Para el ejemplo de una manzana cae verticalmente por efecto de la gravedad desde una altura $h = 1{,}4\,\mathrm{m}$, utilice el método de Runge-Kutta de orden 4 para obtener la posición de la manzana en función del tiempo hasta que llegue al suelo. Compare esta solución con la solución exacta del

problema, y estudie el error cometido, para un paso de tiempo $\Delta t = 0{,}1\,\mathrm{s}$, cuando la manzana llega al suelo. Haga la comparación entre el método de Runge-Kutta, el método predictor-corrector y el método de Euler.

Tema 7

SOLUCIONES A LOS EJERCICIOS

1.1. 2.

1.2.

```
a = 24
if a%2==0:
    if a%3==0:
        print("a es divisible entre 2 y 3")
    else:
        print("a es divisible entre 2 ")
elif a%3==0:
    print("a es divisible entre 3")
else:
    print("a no es divisible ni entre 2 ni entre 3")


```

Código 7.1: Solución al ejercicio 1.2

2.1. $x_1 = -1$; $x_2 = 2$; $x_3 = 0$; $x_4 = 1$.

2.3.

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 \\ 3 & -4 & 3 & 0 \\ -1 & 3 & 0 & -13 \end{pmatrix} \qquad \mathbf{U} = \begin{pmatrix} 1 & 1 & 0 & 3 \\ 0 & 1 & 1 & 5 \\ 0 & 0 & 1 & 4{,}333 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.5.

$$\mathbf{L} = \begin{pmatrix} 2 & 0 & 0 \\ -0{,}5 & 2 & 0 \\ 0{,}5 & 1{,}5 & 1 \end{pmatrix}$$

2.6. $x_1 = -1$; $x_2 = 2$; $x_3 = 0$; $x_4 = 1$.

3.1 $f(x) = 2\,x^3 - \frac{33}{2}\,x^2 + \frac{91}{2}\,x - 23$.

4.4. $x_1 = 1{,}06158$; $x_2 = 4{,}71264$.

4.5. $x_1 = 4{,}66465$; $x_2 = -2{,}55568$.

4.6. $x_1 = -2{,}20574$; $x_2 = 0{,}0923963$; $x_3 = 0{,}613341$.

BIBLIOGRAFÍA

[1] Burden, R.; Burden, A. y Faires, J. *Numerical Analysis*. Brooks/Cole, 2015.

[2] Chazallet, S. *Python 3 - Los fundamentos del lenguaje*. Eni, 2020.

[3] Isaacson, E. y Keller, H. B. *Analysis of Numerical Methods*. Dover books, 1994.

[4] Moreno González, C. *Introducción al cálculo numérico*. UNED, 2021.

[5] Moreno Muñoz, A. y Córcoles Córcoles, S. *Aprende Python en un fin de semana*, 2018.

[6] —— *Curso de Programación Python*. ANAYA Multimedia, 2021.

[7] —— *Python avanzado en un fin de semana*. ANAYA Multimedia, 2021.

[8] Press, William H.; Teukolsky, Saul A.; Vetterling, William T. y Flannery, Brian P. *Numerical Recipes in C*. Cambridge University Press, 1992.

[9] Sesé, L. M. *Cálculo Numérico y Estadística Aplicada*. UNED, 2011.

[10] Spiegel, M. R.; Lipschutz, S. y Liu, J. *Fórmulas y Tablas de Matemática Aplicada*. McGraw-Hill, México, 2014.