Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

7 de noviembre de 2024

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: http://pyciencia.taniquetil.com.ar/

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

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

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

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

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

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

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

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

VERSIÓN PRELIMINAR

Índice general

VERSIÓN PRELIMINAR

Parte I
Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

Parte II
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

Parte III
Temas específicos

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

1 | Integración numérica

Muchas aplicaciones científicas y tecnológicas requieren la integración de funciones, en una gran variedad de contextos. La integración analítica de funciones es mucho más compleja que la diferenciación, y solo se puede realizar en los pocos casos en los que existe la primitiva del integrando. Por esta razón es necesario recurrir frecuentemente a la integración numérica, denominada "cuadratura" por razones históricas.

Módulo	Versión
Matplotlib	3.9.2
NumPy	1.26.4
SciPy	1.14.1
SymPy	1.13.3
mpmath	1.3.0
vegas	6.1.3

Código disponible

El cálculo de una integral definida en forma analítica consiste en evaluar

$$\int_a^b f(x)\,dx = F(b) - F(a)$$

donde

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

El problema se reduce a encontrar una función primitiva F (o "antiderivada") correspondiente a una dada $f(x)$. El intervalo de integración $[a, b]$ puede ser finito, semi infinito, (cuando $a = -\infty$ o $b = \infty$), o infinito (ambos $a = -\infty$ y $b = \infty$). Gráficamente, calcular la integral corresponde a evaluar el área entre la curva de $f(x)$ y el eje x, tal como muestra la figura 1.1, donde el área se considera positiva cuando $f(x) > 0$ (verde) y negativa si $f(x) < 0$ (roja).

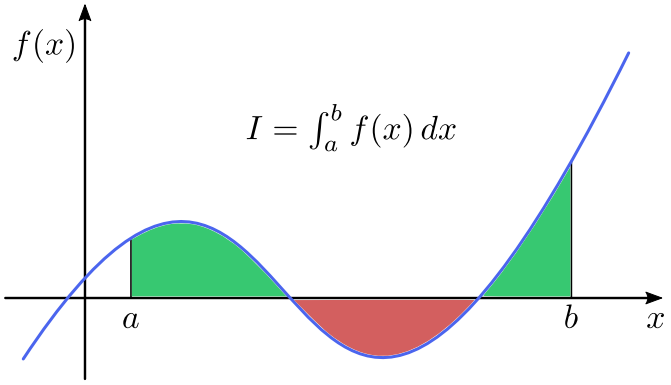

FIGURA 1.1: Interpretación gráfica de la integración.

El cálculo de las cuadraturas numéricas consiste en establecer reglas para aproximar el valor de la integral mediante el cálculo de áreas a partir de la información disponible de f, ya sea

porque se conoce su valor en un número establecido de puntos en el intervalo $[a, b]$ o porque es posible evaluarla numéricamente en ese intervalo. Veremos en la sección siguiente cómo obtener expresiones analíticas (cuando es posible) con SymPy, y las opciones que ofrece SciPy para el cálculo de cuadraturas, en la sección 1.3.

1.1. Integración simbólica

SymPy permite calcular integrales definidas e indefinidas de funciones a través del método `integrate()`, que utiliza el poderoso algoritmo Risch-Norman extendido, junto con algoritmos heurísticos y reconocimiento de patrones. De esta forma, SymPy es capaz de integrar funciones polinómicas, racionales, productos de polinomios y exponenciales, senos y cosenos. En el siguiente *jupyter-notebook* vemos algunos ejemplos:

CELL 01

```
import sympy as sym

sym.init_printing()
x = sym.Symbol('x')
```

CELL 02

```
# Función polinómica
sym.integrate(3* x**2 + x - 1, x)
```

$$x^3 + \frac{x^2}{2} - x$$

CELL 03

```
# Función racional
sym.integrate(x**2/(x**2 + 2 * x + 1), x)
```

$$x - 2\log\left(x + 1\right) - \frac{1}{x + 1}$$

CELL 04

```
# Función polinómica-exponencial
sym.integrate(x**2 * sym.exp(x) * sym.sin(x), x)
```

$$\frac{x^2 e^x \sin\left(x\right)}{2} - \frac{x^2 e^x \cos\left(x\right)}{2} + x e^x \cos\left(x\right) - \frac{e^x \sin\left(x\right)}{2} - \frac{e^x \cos\left(x\right)}{2}$$

CELL 05

```
# Integral definida
f = 4 * sym.sqrt(1 - x**2)
a, b = -1, 1
valor_simb = sym.integrate(f, (x, a, b))
valor_simb
```

$$2\pi$$

Tal como muestran las celdas 2–4, SymPy puede obtener las primitivas de algunas funciones, ahorrándonos, en algunos casos, la tediosa aplicación de la integración por partes. Incluso es capaz de resolver en forma analítica la integral definida

$$\int_{-1}^{1} 4\sqrt{1-x^2}\,dx = 2\pi \tag{1.1}$$

como se ve en la celda 5. No obstante, esta facilidad de obtener la primitiva en forma analítica es la excepción más que la regla, y en general no será posible obtener la expresión analítica en forma cerrada. En la gran mayoría de los casos debemos recurrir a cuadraturas numéricas, como las provistas por SciPy que veremos en la sección 1.3, pero es oportuno mencionar acá que es posible integrar numéricamente funciones simbólicas definidas con SymPy pasando estas funciones como argumento de `quad` de la bilbioteca `mpmath`[1], la cual permite realizar cuadraturas numéricas con precisión abritraria. Las versiones de SymPy previas a 1.0 incluian `mpmath` como un submódulo, pero actualmente es una biblioteca externa a SymPy.

Con `mpmath.quad()` podemos evaluar integrales definidas con cualquier precisión, sin la restricción de las limitaciones de la representación de punto flotante. Por supuesto, esta virtud se obtiene con el costo de ser un cálculo mucho más lento que el que ofrece SciPy, por lo que es conveniente usar `mpmath.quad()` para cuadraturas numéricas cuando se requiere evaluar pocas integrales con más precisión que las que brinda SciPy.

Por ejemplo, si queremos evaluar numéricamente la integral de la ecuación (1.1) con una precisión dada utilizando `mpmath`, podemos crear la función integrando usando `sympy.lambdify` pasando `mpmath` como tercer argumento para establecer que usaremos una función compatible con esa biblioteca, tal como vemos a continuación:

CELL 06

```
import mpmath

mpmath.mp.dps = 50
f_mpmath = sym.lambdify(x, f, 'mpmath')
```

CELL 07

```
valor_num = mpmath.quad(f_mpmath, (a, b))
sym.sympify(valor_num)
```

$$6{,}28318530717958647692528676655900576839433879875 02$$

1 `mpmath` es una biblioteca para aritmética real o compleja con punto flotante de precisión arbitraria. Recomendamos ver su documentación.

CELL 08

```
sym.N(valor_simb, mpmath.mp.dps + 1) - valor_num
```

$$-2{,}00457353256914673460540235036361140911139766007879 \cdot 10^{-51}$$

En la celda 6 importamos el módulo `mpmath`, definimos la precisión con la que vamos a trabajar, y construimos la función `f_mpmath` utilizando la función `lambdify` de SymPy, la cual permite transformar expresiones de SymPy a funciones lambda para poder evaluarlas en forma numérica muy rápido. Luego, en la celda 7 asignamos a la variable `valor_num` el resultado de la cuadratura numérica calculada con `mpmath`, y obtenemos su representación numérica con `sympify`, que convierte cualquier expresión arbitraria en un tipo de dato específico que usa internamente SymPy. Finalmente, en la celda 8 calculamos la diferencia entre el valor numérico resultante y el resultado exacto que obtuvimos en la celda 5. Vemos que esta diferencia es un orden de magnitud menor al establecido en la celda 6 (50).

1.1.1. Transformaciones integrales (Laplace y Fourier)

Existen varias clases de problemas que son difíciles de resolver en sus representaciones originales, pero que a partir de una transformación integral que mapea una ecuación en su dominio original a otro, su manipulación y solución pueden ser más convenientes que en el dominio original. Una vez resuelto el problema en el nuevo dominio, es posible volver a la representación original con la operación inversa de la transformación integral. Un problema típico es la transformación de una ecuación diferencial en una algebraica, que se puede realizar mediante una transformada de Laplace, o la transformación de un problema en el dominio temporal al dominio en frecuencia a través de una transformada de Fourier.

En general, la transformada de una función f se puede escribir como

$$F(u) = (Tf)(u) = \int_{t_1}^{t_2} K(u,t) f(t)\, dt \tag{1.2}$$

Aquí, la función f es transformada en una función F mediante el operador T, que se define mediante la función núcleo o *kernel* de dos variables $K(u,t)$. Algunos kernels tienen asociados un kernel inverso $K^{-1}(u,t)$ que permite la transformación inversa

$$f(t) = \int_{u_1}^{u_2} F(u) K^{-1}(t,u)\, du \tag{1.3}$$

SymPy dispone de varias funciones para realizar transformaciones integrales. Veremos ejemplos de transformaciones de Laplace y de Fourier.

1.1.1.1. Tranformada de Laplace

La transformada y antitransformada de Laplace se definen como

$$L(s) = \int_0^{\infty} \exp(-st) f(t)\, dt \tag{1.4}$$

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} \exp(st) F(s)\, ds \tag{1.5}$$

donde $i = \sqrt{-1}$ y c es lo suficientemente grande como para que $F(s)$ no tenga singularidades en el semiplano $\mathrm{Re}(s) > c - \epsilon$. SymPy puede obtener la transformada de Laplace de muchas funciones elementales y combinaciones de ellas. Por ejemplo, calculemos la transformada de $f(t) = \exp(-at)$:

CELL 09

```
s = sym.symbols("s")
a, t = sym.symbols("a, t", positive=True)
f = sym.exp(-a*t)
```

CELL 10

```
sym.laplace_transform(f, t, s)
```

$$\left(\frac{1}{a+s},\ -a,\ \mathrm{True}\right)$$

En la celda 9 definimos los símbolos y la función `f`, y en la 10 obtenemos la transformada de Laplace. Como resultado obtenemos una tupla con la función transformada, y las condiciones de convergencia dado que la transformación requiere de una integración indefinida. Si lo que necesitamos es solamente la función, podemos pasar el argumento `noconds=True` y la asignamos a `F`:

CELL 11

```
F = sym.laplace_transform(f, t, s, noconds=True)
F
```

$$\frac{1}{a+s}$$

Podemos recuperar la función original a partir de la antitransformada de `F`:

CELL 12

```
sym.inverse_laplace_transform(F, s, t, noconds=True)
```

$$e^{-at}$$

En las celdas 13 y 14 podemos ver un ejemplo en el que combinamos funciones simples en `f`:

CELL 13

```
f = sym.exp(-a*t)*(1 + t)**2
F = sym.laplace_transform(f, t, s, noconds=True)
F
```

$$\frac{1}{a+s} + \frac{2}{(a+s)^2} + \frac{2}{(a+s)^3}$$

CELL 14

```
sym.inverse_laplace_transform(F, s, t, noconds=True)
```

$$t^2e^{-at} + 2te^{-at} + e^{-at}$$

1.1.1.2. Transformada de Fourier

Las transformada y antitransformada de Fourier son calculadas por SymPy de un modo análogo a las de Laplace. Las definiciones de estas transformaciones son:

$$\mathcal{F}(k) = \int_{-\infty}^{\infty} f(x) \exp(-2\pi i x k)\, dx \tag{1.6}$$

$$f(x) = \int_{-\infty}^{\infty} \mathcal{F}(k) \exp(-2\pi i x k)\, dk \tag{1.7}$$

Veamos como ejemplo la transformada de Fourier de $f(x) = \exp(-ax^2)$ (celda 8):

CELL 15

```
x, k = sym.symbols("x, k")
f = sym.exp(-a * x**2)
F = sym.fourier_transform(f, x, k)
F
```

$$\frac{\sqrt{\pi}e^{-\frac{\pi^2k^2}{a}}}{\sqrt{a}}$$

Al calcular la transformación inversa recuperamos la función f original:

CELL 16

```
sym.inverse_fourier_transform(F, k, x)
```

$$e^{-ax^2}$$

En caso que las transformadas no puedan obtenerse en forma analítica cerrada, estas funciones devuelven un objeto no evaluado.

1.2. Integración numérica en una dimensión

El abordaje central para la integración numérica (definida) de una función consiste en aproximar la integral por una suma:

$$I = \int_a^b f(x)\, dx = \sum_{i=1}^{n} \omega_i f(x_i) + r_n \tag{1.8}$$

donde x_i son las "abscisas nodales", ω_i son los "pesos" y r_n es el residuo producto de la aproximación. Es importante tener una estimación del término r_n para saber con qué precisión estamos estimando la integral, pero en general asumiremos que es una cantidad pequeña.

En la suma de la expresión (1.8) tenemos n abscisas, por lo que estamos considerando una cuadratura de n-puntos. Según cómo elijamos la ubicación de las abscisas y los pesos correspondientes tendremos diferentes esquemas de cuadratura, con diferencias en el grado de precisión alcanzado y la complejidad en el cálculo. En este aspecto, podemos distinguir dos grandes grupos:

- **Métodos de Newton-Cotes**: consisten en considerar que la integral se puede aproximar por la suma de áreas de formas elementales (rectángulos, trapecios). Por lo general, este método utiliza abscisas igualmente espaciadas que se utilizan para interpolar la función integrando, lo que resulta útil (y a veces necesario) cuando $f(x)$ ha sido evaluada en puntos de una grilla.
- **Cuadraturas gaussianas**: utilizan abscisas que no están equiespaciadas, eligiendo los valores x_i de modo que pueda obtenerse un aumento en la precisión. Como resultado se utilizan menos abscisas que en las fórmulas de Newton-Cotes y por lo tanto un número menor de evaluaciones de f, lo cual es una ventaja cuando estas evaluaciones son costosas computacionalmente.

Por otra parte, se puede considerar otra categorización teniendo en cuenta si los límites del intervalo de integracion (a y b) pertenecen al grupo de abscisas x_i en donde se evalúa la función. Los métodos que incluyen las evaluaciones $f(a)$ y $f(b)$ son "cerrados", mientras que los que los excluyen son "abiertos". Las cuadraturas gaussianas forman parte de los métodos abiertos, al igual que la regla del punto medio, mientras que las fórmulas de la regla trapezoidal y Simpson de Newton-Cotes constituyen ejemplos de métodos cerrados.

1.2.1. Métodos de Newton-Cotes

Tal como adelantamos, para explorar estos métodos asumiremos que conocemos los valores de f en un conjunto discreto de n puntos, es decir, en una tabla de n pares $(x_i, f(x_i))$.

Las diversas reglas de integración de Newton-Cotes se derivan del orden de interpolación para la función integrando. Por ejemplo, si aproximamos $f(x)$ con un polinomio de grado cero (es decir, un valor constante) utilizando el valor medio de f en el intervalo $[a, b]$, obtenemos:

$$I = \int_a^b f(x)\, dx \approx f\left(\frac{a+b}{2}\right) \int_a^b dx = (b-a) f\left(\frac{a+b}{2}\right) \tag{1.9}$$

que se conoce como la "regla del punto medio". En este caso, la integral se aproxima con el área de un rectángulo cuya base es $(b-a)$ y la altura es $f[(b-a)/2]$, y es capaz de integrar polinomios hasta primer orden (es decir, funciones lineales), y por este motivo se dice que es de grado polinomial uno. Con la misma aproximación (polinomio de grado cero) se puede utilizar uno de los extremos del intervalo dando origen de esta forma a la "regla del rectángulo". Este abordaje es muy común en las reglas de integración compuestas (de la que hablaremos más adelante).

Aproximando la función $f(x)$ por un polinomio de primer grado, evaluada en los puntos extremos del intervalo de integración, obtenemos la "regla del trapecio":

$$I = \int_a^b f(x)\, dx \approx \frac{(b-a)}{2} \left[f(a) + f(b)\right] \tag{1.10}$$

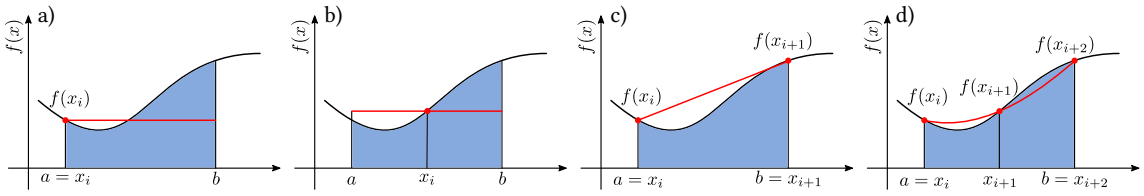

FIGURA 1.2: Esquemas de Newton-Cotes: los puntos rojos representan los valores de la función evaluada en las abscisas x_i. a) Regla del rectángulo; b) regla del punto medio, c) regla del trapecio y d) regla de Simpson.

que también es polinomial de grado uno. Si utilizamos, en cambio, un polinomio de interpolación de segundo grado, resulta la "regla de Simpson":

$$I = \int_a^b f(x)\,dx \approx \frac{(b-a)}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] \tag{1.11}$$

que utiliza la evaluación en los extremos del intervalo de integracion y en el punto medio del mismo. Este método es polinomial de segundo grado, lo que significa que podemos integrar exactamente polinomios de grado menor o igual a tres.

La figura 1.2 muestra los distintos esquemas mencionados. En todos los casos la integral (área azul) se aproxima por el área de la figura delimitada por el eje x por debajo, la línea roja por arriba y las líneas verticales en $x = a$ y $x = b$. Utilizaremos las capacidades de álgebra simbólica e integración de SymPy para determinar los valores de los pesos en la regla de Simpson.

Si consideramos la regla de Simpson como una cuadratura de tres puntos, es muy simple utilizar la condición de integración exacta de un polinomio de segundo grado para obtener los pesos ω_i de la fórmula (1.8). Siguiendo con el *jupyter-notebook* de la sección anterior, en la celda 17 definimos los símbolos que vamos a utilizar y declaramos que `f` será una función, y en la celda 18 definimos el conjunto de las abscisas en los que se evalúa la función, y también declaramos la lista `w` conteniendo los pesos respectivos:

CELL 17

```
a, b, x = sym.symbols("a, b, x")
f = sym.Function("f")
```

CELL 18

```
# Abscisas (x) y pesos (w) de la regla de Simpson
abs_x = a, (a + b)/2, b
w = [sym.symbols(f"w_{i}") for i in range(len(abs_x))]
```

Dadas las abscisas y los pesos, podemos construir la regla de Simpson en forma simbólica:

CELL 19

```
simpson = sum(w[i] * f(abs_x[i]) for i in range(len(abs_x)))
simpson
```

$$w_0 f(a) + w_1 f\left(\frac{a}{2} + \frac{b}{2}\right) + w_2 f(b)$$

Ahora haremos uso del hecho que con la regla de Simpson podemos integrar en forma exacta hasta polinomios de tercer grado. Dado que tenemos que obtener tres pesos ω_i, nos será suficiente con construir una base de polinomios hasta segundo grado, con los cuales podemos interpolar $f(x)$. Usaremos `sympy.Lambda` para crear las representaciones simbólicas de esta base:

CELL 20

```
phi = [sym.Lambda(x, x**n) for n in range(len(abs_x))]
phi
```

$$\left[(x \mapsto 1)\,,\ (x \mapsto x)\,,\ \left(x \mapsto x^2\right)\right]$$

Con esta base podemos construir un sistema de ecuaciones para determinar los pesos. Para cada una de las funciones de la base tenemos:

$$\sum_{i=0}^{2} \omega_i \phi_n(x_i) - \int_a^b \phi_n(x)\,dx = 0$$

La lista `ecs` de la celda 21 muestra cómo construimos el sistema de ecuaciones, cuya solución obtenemos en la celda 24. En la celda 25 substituimos los ω_i obtenidos en la regla de Simpson, y podemos ver que recuperamos la expresión dada por (1.11).

CELL 21

```
ecs = [simpson.subs(f, phi[n]) - sym.integrate(phi[n](x), (x, a, b)) for n in range(len(abs_x))]
ecs[0]
```

$$a - b + w_0 + w_1 + w_2$$

CELL 22

```
ecs[1]
```

$$\frac{a^2}{2} + aw_0 - \frac{b^2}{2} + bw_2 + w_1\left(\frac{a}{2} + \frac{b}{2}\right)$$

CELL 23

```
ecs[2]
```

$$\frac{a^3}{3} + a^2 w_0 - \frac{b^3}{3} + b^2 w_2 + w_1\left(\frac{a}{2} + \frac{b}{2}\right)^2$$

CELL 24

```
ecs_sol = sym.solve(ecs, w)
ecs_sol
```

$$\left\{w_0 : -\frac{a}{6} + \frac{b}{6},\ w_1 : -\frac{2a}{3} + \frac{2b}{3},\ w_2 : -\frac{a}{6} + \frac{b}{6}\right\}$$

CELL 25

```
simpson.subs(ecs_sol).simplify()
```

$$\frac{(a-b)\left(-f(a)-f(b)-4f\left(\frac{a}{2}+\frac{b}{2}\right)\right)}{6}$$

Las reglas de cuadratura de mayor orden se pueden obtener mediante el abordaje general de aproximar la función $f(x)$ mediante un polinomio de interpolación de grado n, $P_n(x)$, con $n+1$ puntos de interpolación. En efecto, para cada n natural las fórmulas de Newton-Cotes

$$\int_a^b P_n(x)\,dx = h\sum_{i=0}^{n}\omega_i f_i \tag{1.12}$$

proveen valores aproximados para la integral de $f(x)$ en $[a,b]$, siendo $f_i = f(a+ih)$ y $h = (b-a)/n$. Los pesos ω_i se encuentran tabulados[2] y son números racionales con la propiedad

$$\sum_{i=0}^{n}\omega_i = n \tag{1.13}$$

.

Para funciones $f(x)$ suficientemente suaves en el intervalo cerrado $[a,b]$, se puede demostrar que el error en la aproximación de la integral se puede expresar como:

$$\int_a^b P_n(x)\,dx - \int_a^b f(x)\,dx = Kh^{p+1}f^{(p)}(\xi), \quad \xi\in(a,b) \tag{1.14}$$

donde los valores de K y p dependen solo de n pero no del integrando f. En particular, para el caso de la regla de Simpson la estimación del error es

$$\varepsilon = \frac{h^5}{90}f^{(iv)}(\xi) \tag{1.15}$$

donde $f^{(iv)}$ denota la derivada cuarta de f. Podemos observar que esta estimación del error contiene una derivada cuarta, lo que implica que la regla de Simpson es exacta hasta polinomios de tercer grado.

Las reglas de cuadratura de mayor orden se pueden obtener aumentando el número de abscisas (y pesos) en el intervalo $[a,b]$. Sin embargo, la interpolación con polinomios de mayor orden pueden introducir efectos indeseables entre los puntos de interpolación, por lo que resulta aconsejable dividir el intervalo $[a,b]$ en subintervalos $[a,x_1]$, $[x_1,x_2]$, $\cdots$, $[x_{n-1},x_n=b]$, y usar cuadraturas de orden bajo en cada uno de ellos. Este abordaje da origen a las fórmulas compuestas de Newton-Cotes.

1.2.2. Fórmulas de Newton-Cotes compuestas

El uso de las fórmulas de Newton-Cotes es inapropiado en intervalos largos de integración. Por un lado, esto requeriría usar interpolaciones de grado superior y los pesos correspondientes

2 Algunos casos se pueden ver en la entrada de Wikipedia.

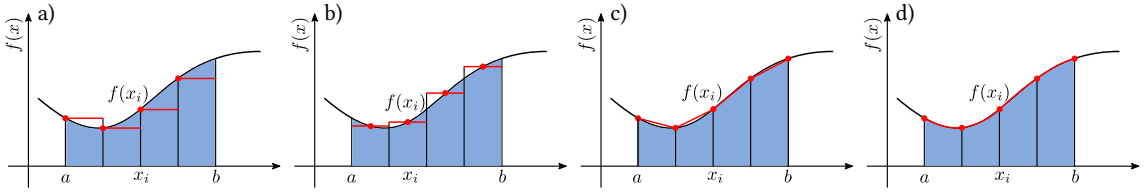

FIGURA 1.3: Fórmulas compuestas de Newton-Cotes: los puntos rojos representan los valores de la función evaluada en las abscisas x_i. a) Regla del rectángulo; b) regla del punto medio, c) regla del trapecio y d) regla de Simpson.

son engorrosos de calcular. Por otro, dado que estas fórmulas utilizan abscisas equiespaciadas, la naturaleza oscilatoria de los polinomios de interpolación de alto orden no necesariamente convergen al valor exacto de la integral.

El procediemiento para obtener las fórmulas compuestas de Newton-Cotes, por ejemplo para la regla de Simpson, consiste en seleccionar un entero par n y subdividir el intervalo $[a, b]$ en n subintervalos, aplicando la regla de Simpson en cada par consecutivo de ellos. Definiendo el "paso" $h = (b - a)/n$, y $x_i = a + ih$ para cada $i = 0, 1, \ldots, n$ tenemos:

$$\begin{aligned}\int_a^b f(x)\,dx &= \sum_{i=1}^{n/2} \int_{x_{2i-2}}^{x_{2i}} f(x)\,dx \\ &= \sum_{i=1}^{n/2} \frac{h}{3}\left[f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})\right]\end{aligned}$$

Dado que para cada $i = 1, 2, \ldots, n/2 - 1$ aparece $f(x_{2i})$ en el intervalo $[x_{2i-2}, x_{2i}]$ y también en el $[x_{2i}, x_{2i+2}]$, podemos agrupar estos términos y reducir la suma:

$$\int_a^b f(x)\,dx = \frac{h}{3}\left[f(x_0) + 2\sum_{i=1}^{n/2-1} f(x_{2i}) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + f(x_n)\right] \tag{1.16}$$

Del mismo modo se pueden construir las fórmulas compuestas para la regla del rectángulo y del trapecio. La estimación del error de las fórmulas compuestas se obtiene simplemente sumando los errores en cada subintervalo. Por ejemplo, para el caso de la fórmula compuesta de la regla de Simpson, la suma de los errores dados por la expresión (1.15) para cada par de subintervalos consecutivos resulta:

$$\varepsilon_C = \frac{b-a}{180} h^4 f^{(iv)}(\xi) \quad \xi \in (a, b)$$

siempre que $f \in C^4[a, b]$, es decir, f sea continua y con todas sus derivadas hasta orden 4 continuas en $[a, b]$. Por lo tanto este método es de orden 4.

1.2.3. Cuadraturas gaussianas

En las secciones anteriores hemos visto que dado un conjunto de n abscisas x_i, podemos encontrar los pesos ω_i de modo tal que la fórmula resultante sea exacta para la integración de polinomios de grado menor o igual a $n - 1$. En algunos casos, la fórmula de Simpson de tres puntos es exacta para polinomios hasta tercer grado. En las fórmulas de Newton-Cotes tenemos

fijos los valores de las abscisas, y obtenemos los pesos para que las integrales sean exactas en el contexto mencionado.

Podemos aumentar la precisión de las aproximaciones si tenemos más parámetros que podamos utilizar aumentando de este modo los "grados de libertad" disponibles para este propósito. De esta manera, si podemos *elegir* los valores de las abscisas y de los pesos, podemos obtener expresiones que dan valores exactos para la integración de polinomios hasta de grado $2n-1$, dando origen de este modo a las "cuadraturas gaussianas".

Ilustraremos el caso de derivación de las abscisas y los pesos de la fórmula de Gauss-Legendre, de modo de poder integrar exactamente polinomios de tercer grado en el intervalo $[-1,1]$ solo con dos evaluaciones de la función integrando. Usaremos la misma metodología que para derivar la regla de Simpson.

Cell 26

```
abs_x = [sym.symbols(f"x_{i}") for i in range(2)]
w = [sym.symbols(f"w_{i}") for i in range(len(abs_x))]
```

Cell 27

```
GL = sum(w[i] * f(abs_x[i]) for i in range(len(abs_x)))
GL
```

$$w_0 f(x_0) + w_1 f(x_1)$$

En las celda 26 definimos la lista de abscisas y pesos considerando que usaremos solo dos valores para cada caso, y luego, en la celda 27, la representación de la integral según la cuadratura de Gauss-Legendre de dos puntos:

$$I = \int_{-1}^{1} f(x)\,dx \approx \omega_0 f(x_0) + \omega_1 f(x_1) \tag{1.17}$$

A continuación definimos en `phiGL` la base del espacio de los polinomios de grado menor o igual a 3:

Cell 28

```
phiGL = [sym.Lambda(x, x**n) for n in range(4)]
phiGL
```

$$\left[(x \mapsto 1)\,,\ (x \mapsto x)\,,\ \left(x \mapsto x^2\right),\ \left(x \mapsto x^3\right)\right]$$

Ahora construimos el sistema de ecuaciones que establecen la igualdad entre la integración de Gauss-Legendre y la integral exacta para cada una de las funciones base, es decir:

$$\omega_0 \phi_n^{GL}(x_0) + \omega_1 \phi_n^{GL}(x_1) - \int_{-1}^{1} \phi_n^{GL}(x)\,dx = 0$$

donde $\phi_n^{GL}(x) = x^n$ (que son las funciones base almacenadas en `phiGL`). Las ecuaciones resultantes para cada $n = 0, 1, 2, 3$ se muestran en las celdas 29–32.

CELL 29

```
ecsGL = [GL.subs(f, phiGL[n]) - sym.integrate(phiGL[n](x), (x, -1, 1)) for n in range(4)]
ecsGL[0]
```

$$w_0 + w_1 - 2$$

CELL 30

```
ecsGL[1]
```

$$w_0 x_0 + w_1 x_1$$

CELL 31

```
ecsGL[2]
```

$$w_0 x_0^2 + w_1 x_1^2 - \frac{2}{3}$$

CELL 32

```
ecsGL[3]
```

$$w_0 x_0^3 + w_1 x_1^3$$

En la celda 33 construimos la lista `y` que concatena las listas `x` y `w` para tener en un único contenedor todas las incógnitas del sistema de ecuaciones, que en la misma celda resolvemos y mostramos las soluciones obtenidas:

CELL 33

```
y = abs_x + w
ecsGL_sol = sym.solve(ecsGL, y)
ecsGL_sol
```

$$\left[\left(-\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, 1, 1\right), \left(\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, 1, 1\right)\right]$$

Estas soluciones muestran que:

$$\omega_0 = \omega_1 = 1$$
$$x_0, x_1 = \pm\frac{1}{\sqrt{3}}$$

De este modo, podemos substituir en la ecuación (1.17) que resulta en la fórmula de Gauss-Legendre de dos puntos:

$$I \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \tag{1.18}$$

con lo que llegamos al resultado interesante que con la simple suma de los valores de la función en $x = -1/\sqrt{3}$ y $x = 1/\sqrt{3}$ resulta en la estimación de una integral que es precisa hasta tercer

orden.

Se puede observar que en la ecuación (1.17) estamos integrando en el intervalo $[-1, 1]$, lo que hacemos para simplificar la matemática y hacer la formulación tan general como es posible. Si la integración se realiza sobre otro intervalo, se puede reescalar el intervalo de integración para llevarlo al $[-1, 1]$ mediante un simple cambio de variable lineal.

Es posible extender la metodología para obtener fórmulas de Gauss-Legendre de mayor orden aumentando el número de abscisas. Existen tablas[3] en donde figuran las abscisas y los pesos para cuadraturas de Gauss-Legendre con más de dos puntos. En la sección 1.3.2 mostramos cómo realizar la evaluación numérica de cuadraturas gaussianas utilizando el módulo `integrate` de SciPy.

El análisis del error en las cuadraturas gaussianas excede el alcance de este libro (invitamos a las personas interesadas en consultar la referencia [2] para ver una demostración rigurosa), pero podemos mencionar que en este caso, si $f \in C^{(2n)}[a, b]$, el error en la cuadratura es

$$\varepsilon_{CG} = K \frac{f^{(2n)}(\xi)}{(2n!)} \tag{1.19}$$

donde K es una constante y ξ algún valor en (a, b) (desconocido). Comparando con las fórmulas de Newton-Cotes, vemos que para el mismo esfuerzo computacional (es decir, cantidad de evaluaciones de f), la integración gaussiana ofrece una precisión mayor. Sin embargo, dado que las cuadraturas gaussianas requieren la evaluación de las funciones en un conjunto no equiespaciados de puntos dentro del intervalo de integración, no son apropiadas para los casos en los que no se conoce la expresión analítica de la función, como en los casos en que ésta se presenta en forma de tabla sobre una grilla de abscisas. No obstante, cuando es posible evaluar la función en puntos arbitrarios, la eficiencia de las cuadraturas gaussianas es una ventaja, que resulta particularmente útil cuando deben realizarse numerosas evaluaciones de integrales.

1.2.4. Integración de Monte Carlo

Una forma alternativa de estimar integrales en forma numérica es mediante la integración *Monte Carlo*, que utiliza números aleatorios. No es un método eficiente para evaluar integrales unidimensionales, pero lo mencionamos aquí porque su interpretación gráfica es sencilla, y el método es muy poderoso para evaluar integrales multidimensionales.

La situación se representa en la figura 1.4. La función $f(x)$ a ser integrada está contenida en un rectángulo cuya base es el intervalo de integración, y su altura supera con cierto margen el máximo de f en ese intervalo. Naturalmente, el valor de la integral es el área bajo la curva que representa a $f(x)$.

Para estimar el valor del área bajo la curva, generamos a partir de una distribución uniforme una colección n_{MC} de puntos (x_i, y_i) dentro del área que contiene a la función f. La estimación de la integral corresponde, entonces, a la proporción de puntos que caen bajo la función, multiplicado por el área total de la figura. Cuanto mayor sea n_{MC}, mas precisa será la estimación de la integral.

En el código siguiente vemos un ejemplo de cómo realizar la integración de Monte Carlo de

3 Ver tablas 3.5.1-5 en https://dlmf.nist.gov/3.5#v

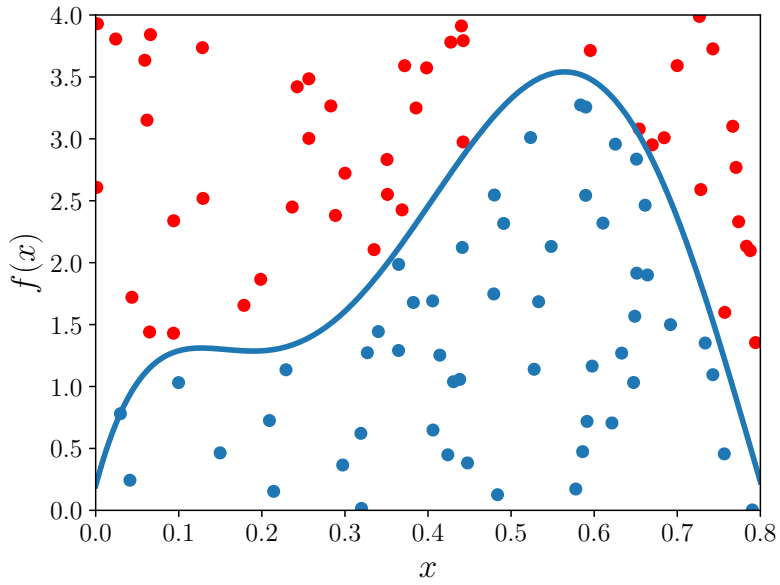

FIGURA 1.4: Estimación de la integral mediante integración de Monte Carlo con $n_{\text{MC}} = 100$.

la función polinómica:

$$f(x) = 400x^5 - 900x^4 + 675x^3 - 200x^2 + 25x + 0{,}2$$

en el intervalo $[0, 0{,}8]$, que está contenida en un rectángulo de altura $4{,}0$.

```python
#!/usr/bin/env python3

import matplotlib.pyplot as plt
import numpy as np
import sympy as sym
plt.rcParams.update({
    "text.usetex": True,
    "axes.labelsize": 18,
    "xtick.labelsize": 14,
    "ytick.labelsize":14})

def f(x):
    return 400 * x**5 - 900 * x**4 + 675 * x**3 - 200 * x**2 + 25 * x + 0.2

a, b, c = 0, 0.8, 4 # Límites de integración y escala vertical
xs = sym.Symbol('x')
fs = f(xs)
int_sym = sym.integrate(fs, (xs, a, b))
print(f'Integral sympy = {int_sym}')

n_mc = 100 # Puntos de Monte Carlo
rng = np.random.default_rng(seed=13)
x = rng.uniform(a, b, n_mc)
y = rng.uniform(a, c, n_mc)

mask = y < f(x) # Selección de puntos de MC debajo de la curva
no_mask = ~mask   # Selección de puntos de MC arriba de la curva
```

```
28 int_MC = y[mask].size / n_mc * (b - a) * c
29 print(fr'   Integral MC = {int_MC}')
30 print(f'Diferencia porcentual = {(int_MC - int_sym)/int_sym * 100:.3f} %')
31
32 x_plot = np.linspace(a, b, 200) # Coordenadas para graficar f(x)
33 plt.plot(x_plot, f(x_plot), c='tab:blue', lw=3)
34 plt.scatter(x[mask], y[mask], c='tab:blue')
35 plt.scatter(x[no_mask], y[no_mask], c='tab:red')
36
37 plt.xlabel(r'$x$')
38 plt.ylabel(r'$f(x)$')
39 plt.xlim([a, b])
40 plt.ylim([a, c])
41 plt.savefig('mc-1d.pdf', bbox_inches='tight')
```

Incialmente importamos los módulos necesarios usuales (`matplotlib.pyplot` y `numpy`) y también `sympy`, ya que usaremos su capacidad de integración para comparar con el resultado que da Monte Carlo. En las líneas 6–10 realizamos alguna configuración de aspectos gráficos de la figura producida, que se muestra en la Figura 1.4.

En la línea 15 definimos el intervalo de integración (a, b) así como la altura del rectángulo que contiene la función (`c`). En las líneas 16–17 definimos `f(x)` que utilizaremos para las evaluaciones numéricas, y luego definimos en `xs` y `fs` el argumento y la función simbólicas (líneas 18–19) para evaluar la integral con SymPy, que es almacenada en la variable `int_sym` y mostrada en pantalla en las líneas 17–18.

A continuación generamos los puntos de Monte Carlo que usaremos para la evaluación de la integral. Primero definimos en `n_mc` la cantidad de puntos a generar (línea 21), luego inicializamos el generador de números aleatorios con una semilla arbitraria[4], y finalmente creamos dos arrays a partir de dicho generador, con distribución uniforme en (a, b) para `x` y (a, c) para `y`, determinando que evaluaremos `n_mc` puntos (líneas 21–24). Estos arrays contienen las coordenadas cartesianas de los puntos en el rectángulo $(a, b) \times (a, c)$ que contiene a $f(x)$.

Ahora realizamos la estimación de la integral. Para ello creamos una máscara[5] que tiene **`True`** en los valores del array `y` que son menores que los valores de `f` evaluados para los valores de `x` correspondientes. Para poder distinguir en el gráfico los valores que están por arriba de la función, generamos la máscara `no_mask` que es la negación de la máscara `mask` (lo que convierte los **`True`** en **`False`** y viceversa). La estimación del área bajo la curva se obtiene multiplicando la proporción de los puntos bajo la curva por el área total que contiene la función (línea 28). A continuación imprimimos esta estimación y también la diferencia porcentual obtenida con SymPy. El resto del código genera la Figura 1.4.

Al ejecutar este programa obtenemos, además del archivo PDF con la figura, el cálculo numérico de la integral por ambos métodos y su comparación:

```
$ ./mc-1d.py
Integral sympy = 1.64053333333333
   Integral MC = 1.6960000000000002
Diferencia porcentual = 3.381 %
$
```

4 Para este ejemplo elegimos el número de la suerte 13, que es la suma y la diferencia de dos cuadrados consecutivos: $13 = 2^2 + 3^2 = 7^2 - 6^2$, además de ser uno de los tres números primos de Wilson conocidos.

5 Para ver como se usan las máscaras en NumPy, invitamos a visitar la sección **??**.

Se puede apreciar que con apenas 100 puntos al azar, la precisión del método (considerando como "exacto" el valor que da SymPy) es de poco más del 3 %. Si repetimos el cálculo con 1000 puntos la diferencia es menor al 1 %. En este caso se obtiene una precisión relativamente buena con pocos puntos dado que en el área de integración, la función ocupa una buena proporción de ésta. No obstante, si la función presenta un "pico" en un intervalo pequeño del dominio de integración, el muestreo con una distribución uniforme no resulta eficiente. Existen diversas técnicas que permiten hacer integraciones Monte Carlo mas "eficientes estadísticamente", entendiendo por esto que podemos alcanzar una mayor precisión para el mismo número de muestras, u obtener la misma precisión con menos muestras. Podemos mencionar, como ejemplos de métodos de reducción de varianza, el muestreo de importancia o los muestreos estratificado y condicional. La descripción de estas técnicas están fuera del alcance de este libro, por lo que sugerimos a quienes estén interesados las lecturas recomendadas al final del capítulo.

1.3. Integración numérica con SciPy

El submódulo `integrate` de SciPy ofrece varias técnicas de integración que pueden agruparse según la forma en que disponemos de la función a integrar. Si dicha función se conoce solo en un número dado de valores los métodos disponibles son `trapezoid`, `simpson` y el de Romberg `romb` (extensión recursiva de la regla del trapecio que utiliza la extrapolación de Richardson). Por otra parte, si disponemos del integrando como una función que podemos evaluar en valores arbitrarios del argumento, `integrate` dispone de `quad`, `fixed_quad`, `quadrature` y `romberg`. `integrate` también dispone de rutinas para el cálculo de integrales de varias variables, lo que veremos en la sección 1.4.

1.3.1. Fórmulas de Newton-Cotes

La implementación de las fórmulas de Newton-Cotes comprende la del trapecio (`trapezoid`), la de Simpson (`simpson`) y la fórmula de Romberg (`romb`). En estos últimos dos casos existen requerimientos sobre la partición del intervalo de integración utilizado. Para el caso de Simpson, es necesario que el número de intervalos sobre los que se aplica la fórmula compuesta sea par, por lo que el número de abscisas debe ser impar. Para el caso del método de Romberg, el número de abscisas debe ser $n = 2^k + 1$, con k entero.

Como ejemplos de uso calculamos la integral

$$I = \int_0^5 2 \exp\left(-\frac{x^2}{2}\right) dx \tag{1.20}$$

de la cual obtenemos primero, haciendo uso de SymPy, su valor exacto y una aproximación numérica con una precisión de 20 cifras significativas (celdas 1–2), de modo de poder estimar la precisión de las cuadraturas de Newton-Cotes. Luego definimos la función en la celda 3, y la evaluamos en el array de abscisas `x` que tiene 17 valores (celda 4), cumpliendo de este modo los requisitos de Simpson y Romberg. En la celda 5 obtenemos los valores de las integraciones numéricas para cada método, notando que en el caso de Romberg es necesario indicar en `dx` el paso de integración (o longitud de cada subintervalo).

CELL 01

```
import sympy as sym

sym.init_printing()
X = sym.Symbol('X')
I_exacta = sym.integrate(2 * sym.exp(-X**2/2), (X, 0, 5))
I_exacta
```

$$\sqrt{2}\sqrt{\pi}\,\mathrm{erf}\left(\frac{5\sqrt{2}}{2}\right)$$

CELL 02

```
I_ex_num = sym.N(I_exacta, 20)
I_ex_num
```

$$2{,}50662683757313042 28$$

CELL 03

```
import numpy as np

def f(x, A, μ):
    return A * np.exp(-μ * x**2)
```

CELL 04

```
a, b = 0, 5  # Intervalo de integración
A, μ = 2.0, 0.5 # Parámetros de la función f
n = 2**4 + 1 # número de abscisas (n = 17) con k = 4
x = np.linspace(a, b, n)
y = f(x, A, μ)
```

CELL 05

```
import operator
import scipy.integrate as spi

I_trap = spi.trapezoid(y, x=x)
I_simp = spi.simpson(y, x=x)
I_romb = spi.romb(y, dx=x[1]-x[0])
I_NC = [("Trapecio", I_trap), ("Simpson", I_simp), ("Romberg", I_romb)]
I_NC_errores = [(name, value, (value-I_ex_num) / I_ex_num * 100) for (name, value) in I_NC]
for name, value, error in sorted(I_NC_errores, key=operator.itemgetter(2), reverse=True):
    print(f'{name:10s}: {value} ({error:.3e} %)')
```

```
Simpson   : 2.5066268019880233 (-1.420e-6 %)
Trapecio  : 2.5066265447230225 (-1.168e-5 %)
Romberg   : 2.5065305814372403 (-3.840e-3 %)
```

En la salida de la celda 5 mostramos la diferencia relativa con el valor "exacto" de la celda 2. Vemos que la fórmula de Simpson da el resultado más preciso, con $n = 17$, seguido por la fórmula del trapecio y luego por la de Romberg. Sin embargo, si repetimos el cálculo para $n = 257$

(duplicando el valor de k), la precisión dada por la fórmula de Romberg mejora significativamente, superando en cuatro órdenes de magnitud a la de Simpson, y en siete a la del trapecio.

En caso que querramos usar la fórmula de Simpson con un número impar de subintervalos (n par), es posible indicarle al método `simpson` que utilice una fórmula mixta usando la regla de Simpson en un número par de intervalos y la del trapecio en el restante. Esto se configura en el parámetro `even` que admite las cadenas `'first'` para el método del trapecio en el primer subintervalo, `'last'` en el último y `'avg'` haciendo un promedio entre los dos casos anteriores (siendo esta la opción por defecto).

1.3.2. Cuadraturas con funciones evaluables

Si tenemos la posibilidad de evaluar la función integrando en argumentos arbitrarios, podemos utilizar los métodos de SciPy `quad`, `fixed_quad`, `quadrature` y `romberg`.

El método `quad` utiliza la biblioteca QUADPACK[6] desarrollada en Fortran, que consiste en rutinas automáticas de integración que intentan obtener el resultado con un error (relativo o absoluto) menor al establecido como argumento. En la celda 6 vemos el cálculo de la integral (1.20) pasando la función `f`, los límites de integración `a` y `b`, y los parámetros adicionales `A` y `μ` en `args` (notar que el primer argumento de `f`, `x`, es asumido implícitamente como variable de integración). En este caso usamos el valor por defecto del error absoluto máximo `epsabs` de `1.49e-08`:

CELL 06

```
I_quad = spi.quad(f, a, b, args=(A, μ), full_output=1)
print(f'I = {I_quad[0]:.9f} ± {I_quad[1]:.3e} ( {(I_quad[0] - I_ex_num) /I_ex_num*100:.3e} % )')
print(f"Evaluaciones de f: {I_quad[2]['neval']}")
```

```
I = 2.506626838 ± 2.173e-09 ( 9.087e-15 % )
Evaluaciones de f: 21
```

En la salida de la celda 6 vemos, entre paréntesis, la diferencia porcentual con el valor obtenido a partir de la integración exacta con 20 cifras significativas (celda 2). En este caso, la estimación obtenida por `quad` se aproxima al valor "exacto" en varios órdenes comparados con las estimaciones dadas por las fórmulas de Newton-Cotes. Al pasar el argumento `full_output=1` obtenemos como tercer elemento de la tupla de salida de `quad` un diccionario con diversa información sobre la ejecución del método. En la salida de la celda 6 imprimimos el número de evaluaciones de la función `f` efectuadas durante la integración (21), lo que muestra la eficiencia de este método.

En las celdas 7–8 vemos el cálculo de la integral (1.20) utilizando cuadraturas gaussianas. En el primer caso, el método `fixed_quad` utiliza un número establecido de abscisas (en el caso mostrado usamos el valor por defecto, cinco), mientras que en el segundo `quadrature` utiliza el número de abscisas necesario para obtener el resultado con un error menor que el requerido (que por defecto es `1.49e-08`). En las salidas de ambas celdas mostramos también entre paréntesis la diferencia con el valor "exacto" (con 20 cifras significativas), mostrando que con pocos puntos se alcanza una precisión aceptable (el más preciso con apenas 11 puntos).

6 Sitio web de QUADPACK.

CELL 07

```
I_fix_gauss = spi.fixed_quad(f, a, b, args=(A, μ), n=5)[0]
print(f'I = {I_fix_gauss:.9f} ( {(I_fix_gauss - I_ex_num)/I_ex_num * 100:.3e} % )')
```

```
I = 2.508871444 ( 8.955e-2 % )
```

CELL 08

```
I_gauss = spi.quad(f, a, b, args=(A, μ))[0]
print(f'I = {I_gauss:.9f} ( {(I_gauss - I_ex_num)/I_ex_num * 100:.3e} % )')
```

```
I = 2.506626838 ( 9.087e-15 % )
```

La función `quad` permite calcular integrales impropias con límite de integración infinitos. Estos límites se representan con la representación de punto flotante de infinito `float('inf')`, que en NumPy está disponible como `np.inf`. Por ejemplo, para evaluar la integral

$$\int_{-\infty}^{\infty} e^{-x^2} dx$$

podemos hacerlo con:

CELL 09

```
def f_imp(x):
    return np.exp(-x**2)

val, err = spi.quad(f_imp, -np.inf, np.inf)
print(f'{val = } ± {err}')
```

```
val = 1.7724538509055159 ± 1.4202636780944923e-08
```

`quad` también es capaz de evaluar integrales con singularidades en el integrando, aunque en estas condiciones suele ser necesario "ayudar" al método indicando dónde no evaluar la función, ya que puede suceder que la cuadratura gaussiana intente evaluar la función en estos puntos singulares (por ejemplo, cuando la singularidad está en $x = 0$).

1.4. Integración múltiple

No es tarea sencilla evaluar una integral de varias variables. Esto se debe a dos motivos: en primer lugar, el número de evaluaciones de funciones necesarias para realizar un muestreo de un espacio N-dimensional crece como la potencia N-ésima del número de evaluaciones necesarias para una integral unidimensional. Por ejemplo, si utilizamos 50 puntos para obtener una cuadratura en una dimensión, serán necesarias 125.000 evaluaciones en un espacio tridimensional. En segundo lugar, a diferencia del caso unidimensional donde el dominio de integración queda definido por dos números (los límites del intervalo de integración), una región de integración multidimensional requiere de un contorno de integración $(N-1)$-dimensional que puede ser un objeto muy complejo.

En algunos casos es posible reducir la dimensionalidad de la integración, por ejemplo, aprovechando alguna simetría de la función respecto de sus variables. No obstante, estos suelen ser casos especiales y no la generalidad del problema.

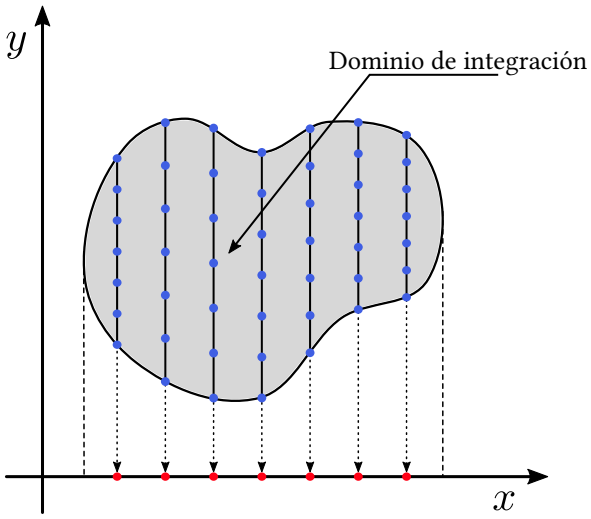

FIGURA 1.5: Evaluaciones de la función $f(x, y)$ en una integración bidimensional.

Existen dos abordajes básicos para la integración multidimensional. Si la función a integrar es suficientemente suave y el contorno de integración es simple, es posible descomponer la integración múltiple en sucesivas integraciones unidimensionales utilizando algún esquema de Newton-Cotes o gaussiano. También este enfoque es necesario si se necesita obtener el valor de la integral con alta precisión.

El segundo abordaje, cuando el dominio de integración es complejo o no se require de un alto grado de precisión, es utilizando el método de Monte Carlo. Veremos a continuación algunos ejemplos utilizando estas técnicas.

1.4.1. Integración en dos, tres y n dimensiones con SciPy

SciPy dispone de los métodos `dblquad`, `tplquad` y `nquad` para la integración de funciones en dos, tres y n dimensiones, respectivamente. Todas utilizan el método de integración unidimensional `quad` descripto en la sección 1.3, que se invoca sucesivamente en cada dimensión de la integral. Por ejemplo, en el caso de la integral bidimensional

$$I = \int_a^b \int_{g(x)}^{h(x)} f(x, y)\, dx\, dy \tag{1.21}$$

el esquema de integración se muestra en la figura 1.5. En esta integral doble sobre el dominio sombreado en gris, la rutina externa de integración establece los valores de x (puntos rojos), para los cuales se selecciona un conjunto de valores de y que están delimitados por $g(x)$ y $h(x)$ (puntos azules) y que constituyen la rutina interna de evaluación. La rutina interna evalúa la integral sobre y en valores fijos de x, y finalmente la rutina externa obtiene el valor final de la integral sobre los valores de x.

Como un ejemplo simple de uso de las rutinas de SciPy, evaluaremos la integral

$$I = \int_{-1}^{1} \int_{-1}^{1} \exp\left(-x^2 - y^2\right)\, dx\, dy \tag{1.22}$$

En la celda 10 definimos la función integrando `f2`, y en la 11 definimos los límites de integración para x (`a` y `b`), así como las funciones $g(x)$ y $h(x)$, y finalmente invocamos el método `dblquad` de `scipy.integrate`, cuidando que el orden en el que aparecen los argumentos debe respetar el orden de los argumentos de `f2`. El resultado de `dblquad` muestra una tupla con el valor de la integral y una estimación del error.

CELL 10

```
def f2(x, y):
    return np.exp(-x**2 - y**2)
```

CELL 11

```
def g(x):
    return -1

def h(x):
    return 1

a, b = -1, 1
spi.dblquad(f2, a, b, g, h)
```

$$\left(2{,}23098514140413,\ 2{,}47689107158348 \cdot 10^{-14}\right)$$

Podemos extender el ejemplo a tres dimensiones fácilmente. En este caso definimos la función `f3` agregando el argumento `z`, así como los límites de integración para esta variable que en este caso deben ser funciones de las variables de las rutinas "mas internas" que integran en `x` y `y`. Nuevamente definimos funciones asignando los límites en `z` a `q` y `r`:

CELL 12

```
def f3(x, y, z):
    return np.exp(-x**2 - y**2 - z**2)

def q(x, y):
    return -1

def r(x, y):
    return 1

spi.tplquad(f3, a, b, g, h, q, r)
```

$$\left(3{,}3323070870931,\ 3{,}69960405321235 \cdot 10^{-14}\right)$$

Para integrales de dimensiones mayores, SciPy ofrece el método `nquad`. Este método recibe como argumentos la función a ser integrada y los rangos de integración para cada variable como una secuencia de dos números. Naturalmente, `nquad` devuelve los mismos resultados que `dblquad` y `tplquad` para dos y tres dimensiones, respectivamente:

Cell 13

```
spi.nquad(f3, [(-1, 1), (-1, 1), (-1, 1)])
```

$$\left(3{,}3323070870931,\ 3{,}69960405321235 \cdot 10^{-14}\right)$$

Tal como anticipamos al comienzo de esta sección, la complejidad computacional crece rápidamente al incrementar las dimensiones de la integral. Para cuantificar esta tendencia usaremos `nquad` con una función generalizada del integrando de los ejemplos anteriores que acepta un número arbitrario de argumentos:

Cell 14

```
def fn(*args):
    """
    f(x1, x2, ..., x2) = exp(-x1^2 - x2^2 ... -xn^2)
    """
    return np.exp(-np.sum(np.array(args)**2))
```

Evaluaremos entonces la integral con un número creciente de dimensiones (desde 1 hasta 5) y estimaremos el tiempo de ejecución utilizando el comando "mágico" `%time`[7]:

Cell 15

```
%time spi.nquad(fn, [(-1, 1)] * 1) # Dimensión 1
```

```
CPU times: user 557 µs, sys: 0 ns, total: 557 µs
Wall time: 566 µs
```

$$\left(1{,}49364826562485,\ 1{,}65828269518814 \cdot 10^{-14}\right)$$

Cell 16

```
%time spi.nquad(fn, [(-1, 1)] * 2) # Dimensión 2
```

```
CPU times: user 1.64 ms, sys: 0 ns, total: 1.64 ms
Wall time: 1.66 ms
```

$$\left(2{,}23098514140413,\ 2{,}47689107158348 \cdot 10^{-14}\right)$$

Cell 17

```
%time spi.nquad(fn, [(-1, 1)] * 3) # Dimensión 3
```

```
CPU times: user 47.6 ms, sys: 2.79 ms, total: 50.4 ms
Wall time: 48.8 ms
```

$$\left(3{,}3323070870931,\ 3{,}69960405321235 \cdot 10^{-14}\right)$$

7 Estos cálculos se realizaron en un procesador Intel i7 1355U.

CELL 18

```
%time spi.nquad(fn, [(-1, 1)] * 4) # Dimensión 4
```

```
CPU times: user 649 ms, sys: 0 ns, total: 649 ms
Wall time: 648 ms
```

$$\left(4{,}97729470116603,\ 5{,}52590717757931 \cdot 10^{-14}\right)$$

CELL 19

```
%time spi.nquad(fn, [(-1, 1)] * 5) # Dimensión 5
```

```
CPU times: user 13.1 s, sys: 0 ns, total: 13.1 s
Wall time: 13.1 s
```

$$\left(7{,}43432759790041,\ 8{,}25376167179527 \cdot 10^{-14}\right)$$

Podemos ver que los tiempos de ejecución aumentan desde algunas centenas de microsegundos para una dimensión, hasta varios segundos para 5. Es decir, aumentando el número de dimensiones 5 veces, el tiempo aumenta cuatro órdenes de magnitud.

1.4.2. Integración Monte Carlo

En la sección anterior vimos que el tiempo de evaluación de integrales múltiples crece rápidamente con el número de dimensiones, por lo que utilizar cuadraturas unidimensionales "anidadas" puede resultar impracticable incluso con un número bajo de dimensiones. Si no se requiere una gran precisión en el resultado, la integración Monte Carlo ofrece una alternativa poderosa y simple que consiste en la evaluación del integrando en puntos del dominio seleccionados aleatoriamente.

El principio básico de la integración Monte Carlo consiste en aproximar la integral de una función f que depende de d variables:

$$I = \int_\Omega f(\boldsymbol{x})\, dV \tag{1.23}$$

donde $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)$, $dV = dx_1 dx_2 \cdots dx_d$ y Ω es un subconjunto de $\mathbb{R}^d$ con volumen V

$$V = \int_\Omega dV$$

Si hacemos un muestreo de N puntos aleatorios uniformemente distribuidos sobre Ω, que denotamos $\boldsymbol{x}_n$ con $n = 1, 2, \cdots, N$, la estimación Monte Carlo de I es

$$I_N = \langle f \rangle V \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{1.24}$$

Aquí, $\langle \cdot \rangle$ representa la media aritmética en los N puntos de la muestra:

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f(\boldsymbol{x}_i) \quad \langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f^2(\boldsymbol{x}_i) \tag{1.25}$$

La ley de los grandes números asegura que la estimación Monte Carlo converge al valor exacto de la integral:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} f(\boldsymbol{x}_i) = I$$

El segundo término de (1.24) es una estimación del error con una desviación estándar, no una cota rigurosa. El teorema del límite central indica que el error en la integración Monte Carlo escala como $1/\sqrt{N}$, independiente de la dimensión d. Este hecho hace que esta técnica de integración resulte muy eficiente para el cálculo de integrales multidimensionales, aunque generalmente la convergencia es lenta.

Estimaremos mediante Monte Carlo la versión generalizada de la integral (1.21) (`fn` de la celda 13), en cinco dimensiones, utilizando para ello el módulo **vegas** para la evaluación de integrales multidimensionales utilizando un algoritmo de Monte Carlo adaptativo [3, 4]. Implementar computacionalmente un algoritmo propio de integración Monte Carlo no es demasiado complicado (como vimos en la sección 1.2.4 para el caso unidimensional). Sin embargo, este módulo especializado tiene rutinas que permiten un muestreo más eficiente, uso de múltiples procesadores y otras características que están fuera del alcance de este libro, y que para el ejemplo de un cálculo simple es muy conveniente ya que en pocas líneas tenemos el resultado buscado.

En la celda 20 importamos el módulo **vegas**, y a continuación creamos un objeto `Integrator` que almacenamos en `integ` y donde especificamos el volumen de integración (una lista con cinco elementos `[-1, 1]`). A continuación aplicamos el objeto `integ` a nuestro integrando estableciendo que realice `nitn=10` iteraciones con un máximo de `neval=1000` puntos en cada una de ellas. Cada iteración produce una estimación independiente de la integral, pero adaptando el muestreo de los puntos dentro del volumen a partir de la precisión obtenida en iteraciones previas. El resultado de la integración se almacena en `result`, y la ejecución la hacemos con el comando `%time` para obtener el tiempo requerido de la misma. El resultado de cada una de las iteraciones se muestra imprimiendo `result.summary()`, donde se puede ver que en cada nueva iteración se alcanza una mejor precisión debido a que el nuevo muestreo "aprende" de los resultados obtenidos en las iteraciones anteriores. El promedio pesado que se muestra en la tercer columna se realiza mediante la inversa de la varianza, por lo que le asigna menos importancia a las primeras iteraciones. Las cuarta y quinta columna muestran los valores de la estimación estadística de los errores (según una distribución χ^2 y el correspondiente p-valor o Q). Finalmente mostramos el resultado de la estimación por Monte Carlo, con su correspondiente incerteza entre paréntesis.

CELL 20

```
import vegas

integ = vegas.Integrator([[-1, 1]] * 5)
%time result = integ(fn, nitn=10, neval=1000)
print(result.summary())
print(f'I_MC = {result}')
```

```
CPU times: user 64.6 ms, sys: 0 ns, total: 64.6 ms
Wall time: 65.2 ms
itn   integral        wgt average     chi2/dof        Q
-------------------------------------------------------
  1   7.54(14)        7.54(14)            0.00     1.00
  2   7.41(10)        7.455(82)           0.63     0.43
  3   7.354(86)       7.407(59)           0.68     0.51
  4   7.439(63)       7.421(43)           0.50     0.69
  5   7.415(55)       7.419(34)           0.37     0.83
  6   7.501(43)       7.451(27)           0.75     0.59
  7   7.374(36)       7.424(21)           1.12     0.35
  8   7.488(28)       7.448(17)           1.44     0.18
  9   7.436(23)       7.443(14)           1.28     0.25
 10   7.428(19)       7.438(11)           1.19     0.30

I_MC = 7.438(11)
```

Lo primero que podemos notar es que el tiempo requerido para evaluar la integral (en la misma computadora que en los casos anteriores) es de poco menos de 65 ms, esto es, unas 200 veces más rápido que utilizando `nquad` de `scipy.integrate` para la misma dimensionalidad del dominio de integración. Como contrapartida, la precisión obtenida es de apenas el $0{,}16\,\%$ frente al $1{,}1 \times 10^{-12}\,\%$ de `nquad`. Podemos mejorar la precisión obtenida con el método de Monte Carlo incrementando el número de puntos muestreados `neval`, o la cantidad de iteraciones `nitn`, aunque es generalmente mejor aumentar `neval`. Si utilizamos un orden de magnitud mayor (`neval=1e5`), la precisión pude disminuir un poco sin casi incrementar el tiempo de ejecución. Para más detalles sobre el paquete **vegas** y métodos avanzados de integración, recomendamos leer su documentación[8].

1.5. Lecturas recomendadas

Fórmulas de Newton-Cotes y cuadraturas gaussianas:

- R.K. Gupta. *Numerical Methods. Fundamentals and Applications*. Cambridge University Press, 2019.
- Alex Gezerlis. *Numerical Methods in Physics with Python*. Cambridge University Press, ago. de 2020. DOI: 10.1017/9781108772310. URL: https://doi.org/10.1017/9781108772310.
- Richard L. Burden, J. Douglas Faires y Annette M. Burden. *Numerical Analysis*. 9th ed. Cengage Learning, 2016.

8 https://vegas.readthedocs.io/

- R. Bulirsch J. Stoer. *Introduction to Numerical Analysis*. 3rd ed. Texts in Applied Mathematics 12. Springer New York, 2002.

Integración Monte Carlo:

- Alex Gezerlis. *Numerical Methods in Physics with Python*. Cambridge University Press, ago. de 2020. DOI: 10.1017/9781108772310. URL: https://doi.org/10.1017/9781108772310.
- S. Weinzierl. «Introduction to Monte Carlo methods». En: *arXiv: High Energy Physics - Phenomenology* (2000). URL: https://arxiv.org/abs/hep-ph/0006269.
- Malvin H. Kalos y Paula A. Whitlock. *Monte Carlo Methods*. Weinheim: John Wiley & Sons, 2009.
- Dirk Kroese, Thomas Taimre y Zdravko I. Botev. *Handbook of Monte Carlo Methods*. Hoboken, New Jersey: John Wiley & Sons, 2011.

Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [11].

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

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] R. Bulirsch J. Stoer. *Introduction to Numerical Analysis*. 3rd ed. Texts in Applied Mathematics 12. Springer New York, 2002.

[3] G Peter Lepage. «A new algorithm for adaptive multidimensional integration». En: *Journal of Computational Physics* 27.2 (1978), págs. 192-203. DOI: https://doi.org/10.1016/0021-9991(78)90004-9. URL: https://www.sciencedirect.com/science/article/pii/0021999178900049.

[4] G. Peter Lepage. «Adaptive multidimensional integration: vegas enhanced». En: *Journal of Computational Physics* 439 (2021), pág. 110386. DOI: https://doi.org/10.1016/j.jcp.2021.110386. URL: https://www.sciencedirect.com/science/article/pii/S0021999121002813.

[5] R.K. Gupta. *Numerical Methods. Fundamentals and Applications*. Cambridge University Press, 2019.

[6] Alex Gezerlis. *Numerical Methods in Physics with Python*. Cambridge University Press, ago. de 2020. DOI: 10.1017/9781108772310. URL: https://doi.org/10.1017/9781108772310.

[7] Richard L. Burden, J. Douglas Faires y Annette M. Burden. *Numerical Analysis*. 9th ed. Cengage Learning, 2016.

[8] S. Weinzierl. «Introduction to Monte Carlo methods». En: *arXiv: High Energy Physics - Phenomenology* (2000). URL: https://arxiv.org/abs/hep-ph/0006269.

[9] Malvin H. Kalos y Paula A. Whitlock. *Monte Carlo Methods*. Weinheim: John Wiley & Sons, 2009.

[10] Dirk Kroese, Thomas Taimre y Zdravko I. Botev. *Handbook of Monte Carlo Methods*. Hoboken, New Jersey: John Wiley & Sons, 2011.

[11] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.