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

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 | Optimización

Los problemas en los cuales se pretende encontrar la mejor solución aparecen frecuentemente en casi todos los ámbitos de la ciencia, la ingeniería, la tecnología y por supuesto, también en la vida cotidiana. Diariamente, para ir desde nuestro hogar al trabajo, tenemos que decidir si queremos ir por el camino más corto, más rápido, menos costoso, menos transitado, más lindo, etc. A menudo, a este requerimiento de obtener la "mejor solución" sobre la base de un criterio dado, le agregamos algunas restricciones adicionales que se deben cumplir también: tenemos que llegar a destino antes de una determinada hora, cruzar la menor cantidad de puentes, que nuestra ruta incluya un lugar donde tomar un desayuno.

Módulo	Versión
Matplotlib	3.9.2
SciPy	1.14.1
SymPy	1.13.3

Código disponible

En general, la optimización es un proceso por el cual queremos encontrar y seleccionar el elemento óptimo de un conjunto de candidatos posibles. Utilizando el formalismo matemático, este problema se expresa como la determinación del valor extremo de una función en un dominio dado. Tal valor extremo, o valor óptimo, puede referirse al máximo o mínimo de una función, dependiendo de la aplicación y el problema específico. En este contexto, un problema de optimización tiene dos ingredientes:

- Una **función objetivo** que debe ser maximizada o minimizada. Por ejemplo, minimizar la distancia en el recorrido de un vehículo de entrega de productos.
- Un **conjunto de restricciones** (posiblemente vacío) que se deben satisfacer. Por ejemplo, no cruzar más de dos veces un río.

Sin perder generalidad podemos considerar la optimización como un problema de minimización[1]. La forma estándar de un problema de optimización continua es:

$$\begin{aligned} \text{minimizar} \quad & f(\boldsymbol{x}) \\ \text{sujeto a} \quad & g_i(\boldsymbol{x}) \leq b_i, \quad i = 1, \ldots, m \end{aligned} \tag{1.1}$$

Aquí, el vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ es la variable de optimización del problema (también denominadas usualmente "variables de decisión"), la función $f : \mathbb{R}^n \to \mathbb{R}$ es la función objetivo,

1 Dado que es válido: $f(x_0) \geq f(x) \Leftrightarrow -f(x_0) \leq -f(x)$ es suficiente presentar el formalismo para problemas de minimización.

las funciones $g_i : \mathbb{R}^n \to \mathbb{R},\ i = 1, \cdots, m$ son las funciones de restricción, y las constantes $b_1, \cdots, b_m$ son los límites de las restricciones. Un vector $\boldsymbol{x}^*$ se denomina óptimo, o solución del problema (1.1), si tiene el menor valor objetivo entre todos los vectores que satisfacen las restricciones: para todo $\boldsymbol{z}$ con $g_1(\boldsymbol{z}) \leq b_1, \ldots, g_m(\boldsymbol{z}) \leq b_m$ tenemos que $f(\boldsymbol{x}^*) \leq f(\boldsymbol{z})$.

Las formas particulares que presentan las funciones objetivo y de restricción caracterizan familias o clases de problemas de optimización muy variados. Un abordaje matemático general es difícil de tratar y no hay métodos eficientes para resolver completamente problemas genéricos de optimización. Sin embargo, existen métodos muy eficientes para muchos casos de importancia teórica o práctica, en los cuales resulta valioso conocer todo lo posible sobre las funciones objetivo y de restricciones para resolver con éxito el problema.

Los problemas de optimización se clasifican según las propiedades de $f(\boldsymbol{x})$ y $g_i(\boldsymbol{x})$. En primer lugar, si x es un escalar ($x \in \mathbb{R}$) el problema es unidimensional o univariado, mientras que si $\boldsymbol{x}$ es un vector ($\boldsymbol{x} \in \mathbb{R}^n$), el problema es multidimensional o multivariado. Naturalmente, cuanto mayor es n, más difícil es resolver el problema de optimización, y requiere más potencia de cálculo.

Cuando la función objetivo y las funciones de restricción son lineales, es decir, satisfacen que

$$f(\alpha\boldsymbol{x} + \beta\boldsymbol{y}) = \alpha f(\boldsymbol{x}) + \beta f(\boldsymbol{y})$$

se suele llamar al problema de optimización (1.1) como **programación lineal**[2]. Si la función objetivo o las de restricción son no lineales, el problema de optimización se denomina **programación no lineal**. Cuando se requiere que el vector $\boldsymbol{x}$ solo tome valores enteros, se trata de un problema de **programación entera**, mientras que si la restricción consiste en que solo tome dos valores (por ejemplo, 0 y 1), estamos frente a un problema de **programación binaria**. En relación con las funciones de restricción g_i, los problemas de optimización pueden ser con o sin restricciones, y en este último caso dichas restricciones pueden expresarse en forma de igualdades o desigualdades.

Los problemas no lineales presentan una complejidad mayor que los lineales, debido a que tienen una variedad muy amplia de posibles comportamientos. En general, estos problemas pueden tener mínimos globales y locales, lo que hace difícil indentificar al mínimo global.

1.1. Optimización unidimensional

La forma más obvia de encontrar el mínimo de una función es diferenciarla y encontrar el valor de la variable independiente que hace la derivada igual a cero. Sin embargo, en muchas situaciones no es práctico encontrar la derivada directamente. El submódulo `optimize` de SciPy provee dos métodos para encontrar el mínimo de una función unidimensional: `brent` y `golden`, aunque en la documentación de SciPy sugieren no utilizar este último, incluído solo con fines académicos. Estos métodos solo utilizan evaluaciones de la función a optimizar, sin requerir del cálculo de la derivada, y se aproximan al valor óptimo mediante técnicas de *bracketing* o acotamiento. Para funciones con un único mínimo en un intervalo dado, estas metodologías de *bracketing* garantizan la convergencia al punto óptimo, pero esto no está asegurado para funciones más complejas.

Como ejemplo de optimización unidimensional consideraremos el ejemplo clásico de minimizar el área de un cilindro con un volumen V_0 dado. Las variables a considerar son el radio r y

2 Pese al nombre, no tiene que ver con programación de computadoras.

la altura h del cilindro, y la función objetivo es $f(r,h) = 2\pi r^2 + 2\pi rh$, con la condición de restricción $g(r,h) = \pi r^2 h - V_0 = 0$. El problema así planteado presenta dos variables, pero a partir de la restricción podemos reducirlo a una sola dimensión usando $h = V_0/\pi r^2$ y substituyendo en $f(r,h)$, lo que conduce a minimizar $f(r) = 2\pi r^2 + 2V_0/r$.

Comenzaremos solucionando el problema en forma analítica aprovechando las capacidades de cálculo simbólico de SymPy. En la primera celda declaramos las variables y ecuaciones que representan el problema, obteniendo el valor exacto de r que minimiza el área para un volumen V_0 dado:

CELL 01

```
from sympy import symbols, pi, solve

r, h, V_0 = symbols("r h V_0")
Area = 2 * pi * r**2 + 2 * pi * r * h
Volumen = pi * r**2 * h
h_r = solve(Volumen - V_0, h)[0] # hallamos h en función de r
Area_r = Area.subs({h: h_r})     # expresamos el área como función de r
r_sol = solve(Area_r.diff(r), r)[0] # solución de r con significado físico
r_sol
```

$$\frac{2^{\frac{2}{3}}\sqrt[3]{V_0}}{2\sqrt[3]{\pi}}$$

Podemos verificar que es un mínimo evaluando las derivadas primera y segunda de `Area_r` en el valor obtenido para `r_sol`:

CELL 02

```
print(Area_r.diff(r, 1).subs(r, r_sol), Area_r.diff(r, 2).subs(r, r_sol))
```

```
0 12*pi
```

dando un valor positivo (12π) para la derivada segunda confirmando que `r_sol` es un mínimo. El área que corresponde a esta solución es:

CELL 03

```
Area_r.subs(r, r_sol)
```

$$3\sqrt[3]{2}\sqrt[3]{\pi}V_0^{\frac{2}{3}}$$

Utilizaremos el valor exacto del mínimo obtenido para r (`r_sol`) en la celda 1 para compararlo con el que obtendremos mediante una optimización numérica. Para ello evaluamos `r_sol` para el caso en que $V_0 = 1$:

CELL 04

```
r_sol.subs({V_0: 1}).evalf()
```

$$0{,}541926070139289$$

Para problemas simples la resolución analítica es posible, pero en problemas reales es común tener que recurrir a métodos numéricos (por ejemplo cuando no es posible resolver la ecuación $f'(x) = 0$). Para abordar este problema con el submódulo `optimize` de SciPy, primero debemos definir la función `f()` que implementa la función objetivo (en este caso asumimos que $V_0 = 1$), y luego pasamos esta función como argumento de `minimize_scalar` para realizar la minimización numérica, junto con un intervalo de *bracketing* y la selección del método de optimización. El método de Brent utiliza una combinación de interpolación parabólica y el método de sección de oro. En la celda siguiente realizamos la optimizaciòn de nuestro problema con este método:

CELL 05

```
import numpy as np
from scipy.optimize import minimize_scalar

def f(r):
    return 2 * np.pi * r**2 + 2 / r

solucion = minimize_scalar(f, bracket=[0.01, 1], method='brent')
print(solucion)
print(f"r_min = {solucion.x}")
```

```
 message:
          Optimization terminated successfully;
          The returned value satisfies the termination criteria
          (using xtol = 1.48e-08 )
 success: True
     fun: 5.535810445932086
       x: 0.5419260710334868
     nit: 12
    nfev: 15
r_min = 0.5419260710334868
```

Como resultado de la optimización, el método devuelve un objeto (asignado en este caso a la variable `solucion`) que en sus atributos contiene informción acerca de cómo fue la ejecución de la optimización, tal como un mensaje que describe el motivo de la finalización, una variable booleana que indica si hubo convergencia (`success`), el valor de la función evaluada en el valor óptimo (`fun`), el valor óptimo hallado (`x`), el número de iteraciones (`nit`) y las llamadas a la función objetivo (`nfev`). Podemos ver que el resultado numérico coincide con el valor "exacto" obtenido con SymPy hasta la precisión informada. La figura 1.1 muestra la función y el valor óptimo hallado. Siempre es recomendable visualizar la función antes de aceptar el valor obtenido como válido.

La función `minimize_scalar` permite establecer, además, la tolerancia requerida para la convergencia o el número máximo de iteraciones. Sugerimos ver las opciones en su documentación[3].

1.2. Optimización multidimensional sin restricciones

La optimización multidimensional es más compleja que la correspondiente a la de una sola dimensión que vimos en la sección anterior. Los métodos de *bracketing* no son aplicables y en

3 https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize_scalar.html#scipy.optimize.minimize_scalar.

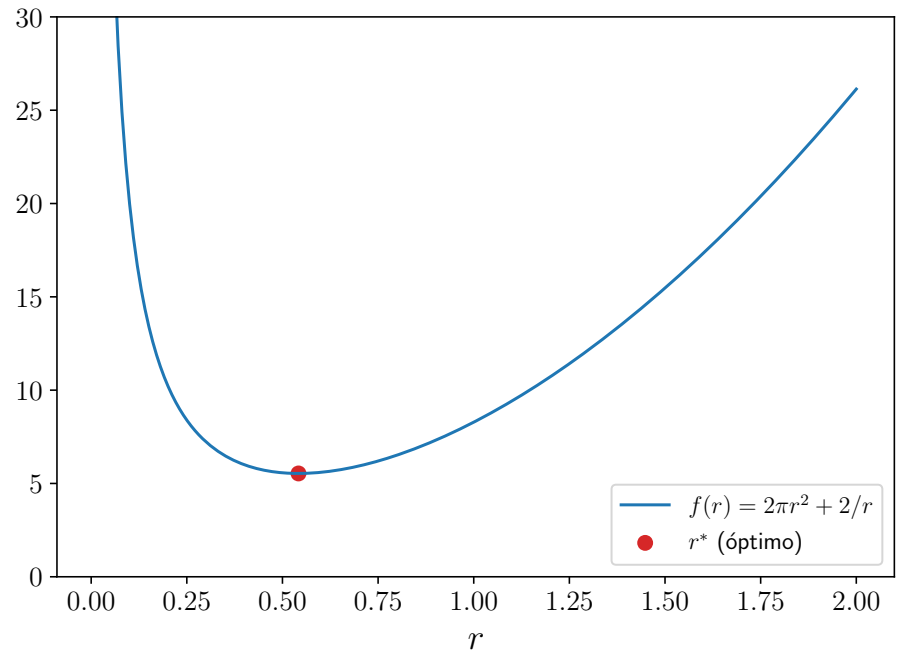


FIGURA 1.1: Optimización de la superficie de un cilindro con volumen dado.

general se deben aplicar métodos que comienzan en algún punto en el espacio de coordenadas y mediante alguna estrategia se buscan nuevos puntos que sucesivamente se aproximen al mínimo.

Como abordaremos el caso de una minimización sin restricciones, podemos expresar nuestro problema como

$$\text{Minimizar } f(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \mathbb{R}^n$$

donde $f(\boldsymbol{x})$ es una función no lineal de $\boldsymbol{x} = (x_1, \ldots, x_n)$. Estos problemas son frecuentes en muchas aplicaciones, por ejemplo en problemas de redes neuronales donde es necesario encontrar los pesos que minimizan la diferencia entre la salida de la red y la respuesta conocida a la entrada.

Iniciamos entonces asumiendo una aproximación inicial $\boldsymbol{x}_0$, y procedemos a generar una mejor aproximación utilizando una fórmula iterativa de la forma

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + s\boldsymbol{d}_k, \qquad \text{para } k = 0, 1, 2, \ldots$$

Para utilizar esta fórmula es necesario determinar el vector $\boldsymbol{d}_k$, que representa la dirección en la que buscamos el nuevo valor $\boldsymbol{x}_{k+1}$, y el escalar s que representa la distancia en que nos movemos en esa dirección. Se pueden implementar muchas estrategias para definir estas cantidades. Tal vez la elección más simple consiste en tomar la dirección $\boldsymbol{d}_k$ como el negativo del vector gradiente de f en $\boldsymbol{x}_k$. Para un paso d suficiente pequeño, esta estrategia garantiza una reducción en el valor de la función. Esto conduce al algoritmo del descenso del gradiente:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - s\nabla f(\boldsymbol{x}_k)$$

donde $\nabla f(\boldsymbol{x}) = (\partial f/\partial x_1, \partial f/\partial x_2, \cdots, \partial f/\partial x_n)$, y s es un valor constante pequeño. El mínimo se alcanza cuando el gradiente se anula. Aunque en cada paso nos movemos hacia un valor menor

de la función, utilizando valores chicos de s el algoritmo es muy lento. Por otra parte, un valor grande de s pone en riesgo la convergencia hacia el mínimo.

El valor de s se puede optimizar considerando que el valor que produce la máxima reducción de f es el que se obtiene de minimizar la función a lo largo de la dirección establecida por $-\nabla f(\boldsymbol{x}_k)$. Formalmente, esto significa que para cada k debemos hallar el valor de s que minimiza

$$f[\boldsymbol{x}_k - s\nabla f(\boldsymbol{x}_k)] \tag{1.2}$$

que también constituye un problema de minimización denominado "búsqueda lineal", dado que en definitiva es una minimización en una sola variable, s, a lo largo de la dirección del gradiente, ya que $\boldsymbol{x}_k$ es conocido. Claramente, el valor que minimiza $f[\boldsymbol{x}_k - s\nabla f(\boldsymbol{x}_k)]$ es tal que la derivada de $f[\boldsymbol{x}_k - s\nabla f(\boldsymbol{x}_k)]$ es cero. Entonces, diferenciando $f[\boldsymbol{x}_k - s\nabla f(\boldsymbol{x}_k)]$ respecto de s tenemos:

$$\frac{df[\boldsymbol{x}_k - s\nabla f(\boldsymbol{x}_k)]}{ds} = -\left(\nabla f(\boldsymbol{x}_{k+1})\right)^\intercal \nabla f(\boldsymbol{x}_k) = 0$$

lo que muestra que las direcciones sucesivas de búsqueda son ortogonales. Si bien este método genera una secuencia que se aproxima al mínimo de f, no presenta la mejor forma de hacerlo dado que los cambios en las direcciones de búsqueda son grandes.

Una mejora consiste en tomar una combinación de la dirección previa y de la nueva dirección para aproximarnos al valor óptimo en forma más directa, lo que da origen al método del gradiente conjugado. Utiliza el mismo paso s obtenido en (1.2), y, llamando $\boldsymbol{g}_{k+1} = \nabla f(\boldsymbol{x}_{k+1})$, la nueva dirección resulta de

$$\boldsymbol{d}_{k+1} = -\boldsymbol{g}_{k+1} + \beta \boldsymbol{d}_k$$

Esto es, la nueva dirección es una combinación del negativo del gradiente en la nueva posición mas el escalar β por la dirección previa de búsqueda. Ahora el problema es establecer el valor de β, y el criterio utilizado es que las sucesivas direcciones de búsqueda sean conjugadas, esto es, que $(\boldsymbol{d}_{k+1})^\intercal \boldsymbol{A} \boldsymbol{d}_k = 0$ para alguna matriz dada $\boldsymbol{A}$.

Esta elección posee propiedades adecuadas de convergencia para el método del gradiente conjugado. Se puede mostrar que conduce a un valor de β dado por:

$$\beta = \frac{(\boldsymbol{g}_{k+1})^\intercal \boldsymbol{g}_{k+1}}{(\boldsymbol{g}_k)^\intercal \boldsymbol{g}_k}$$

El método de Newton para optimización multidimensional se puede considerar como una modificación del descenso del gradiente que puede mejorar la convergencia. Del mismo modo que para el caso unidimensional, el método de Newton consiste en una aproximación cuadrática local de la función, que cuando se minimiza da origen a un esquema iterativo. Escribiendo una expansión en serie de Taylor de segundo orden para $f(\boldsymbol{x})$ alrededor de $\boldsymbol{x} = \boldsymbol{x}_i$:

$$f(\boldsymbol{x}) = f(\boldsymbol{x}_i) + \nabla f^\intercal(\boldsymbol{x}_i)(\boldsymbol{x} - \boldsymbol{x}_i) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_i)^\intercal H_i (\boldsymbol{x} - \boldsymbol{x}_i)$$

donde H_i es la matriz hessiana[4]. En el mínimo,

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}_j} = 0, \quad \text{para } j = 1, 2, \cdots, n$$

[4] La matriz hessiana de un campo escalar $f : \mathbb{R}^n \to \mathbb{R}$ es la matriz cuadrada de tamaño $n \times n$ que tiene como entradas las derivadas parciales de segundo orden. Ver entrada en Wikipedia.

Entonces,

$$\nabla f = \nabla f(\boldsymbol{x}_i) + H_i(\boldsymbol{x} - \boldsymbol{x}_i) = 0$$

Si H_i no es singular,

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - H_i^{-1}\nabla f(\boldsymbol{x}_i)$$

que se puede mostrar converge cuadráticamente cerca del mínimo. Este procedimiento altera, por lo general, tanto la dirección como la longitud del paso, por lo que no es estrictamente un método del descenso del gradiente, y puede no converger si el punto de partida no está suficientemente cerca del mínimo. Tal como está planteado aquí, el método de Newton requiere tanto el cálculo como el almacenamiento del gradiente y del hessiano de la función. Esto puede resultar impractible para funciones de muchas variables (particularmente para optimización de la función de pérdida durante el entrenamiento de redes neuronales), por lo que existen métodos más apropiados, como los algoritmos cuasi-Newton, siendo uno de los más populares el BFGS (Broyden - Fletcher - Goldfarb - Shanno).

El submódulo `optimize` de SciPy contiene la función `minimize`, que permite utilizar, entre otros, los métodos mencionados aquí. Vamos a mostrar un ejemplo de optimización de la función de dos variables:

$$f(x_1, x_2) = \frac{1}{2}\sum_{i=1}^{2}(x_i^4 - 16x_i^2 + 5x_i)$$

que tomamos de Styblinski y Tang [2] y tiene cuatro mínimos locales, con el mínimo global localizado en $x_i^* = -2{,}903534, i = 1, 2$, y un máximo local ubicado cerca del origen. Para utilizar el método de Newton, necesitamos calcular el gradiente de la función y también el hessiano. En el ejemplo que estamos analizando es simple realizar estos cálculos manualmente, pero aprovecharemos la oportunidad para utilizar la potencia de cálculo simbólico de SymPy, y veremos cómo utilizar las expresiones simbólicas obtenidas en el contexto numérico de NumPy.

Comenzamos entonces definiendo los símbolos a utilizar , la función f (simbólica) y sus derivadas parciales, tal como se muestra en la celda 6:

CELL 06

```
x1, x2, x, y = symbols("x_1 x_2 x y")
f_sim = (x1**4 - 16 * x1**2 + 5 * x1) / 2 + (x2**4 - 16 * x2**2 + 5 * x2) / 2
f_d1 = [f_sim.diff(x_) for x_ in (x1, x2)]
```

Podemos ver el gradiente representando en forma matricial (aunque estrictamente es un vector columna en este caso) utilizando la función `Matrix` de SymPy:

CELL 07

```
from sympy import Matrix

# Gradiente:
Matrix(f_d1)
```

$$\begin{bmatrix} 2x_1^3 - 16x_1 + \frac{5}{2} \\ 2x_2^3 - 16x_2 + \frac{5}{2} \end{bmatrix}$$

Análogamente podemos obtener la matriz hessiana:

CELL 08

```
f_hess = [[f_sim.diff(x, y) for x in (x1, x2)] for y in (x1, x2)]
```

CELL 09

```
# Hessiano:
Matrix(f_hess)
```

$$\begin{bmatrix} 2\left(3x_1^2 - 8\right) & 0 \\ 0 & 2\left(3x_2^2 - 8\right) \end{bmatrix}$$

Habiendo realizado estos cálculos, necesitamos ahora obtener la representación vectorizada de estas funciones, de modo de que puedan ser manipuladas por NumPy y SciPy. Para ello invocamos la función `lambdify` de SciPy:

CELL 10

```
from sympy import lambdify

f_num = lambdify((x1, x2), f_sim, 'numpy')
f_d1_num = lambdify((x1, x2), f_d1, 'numpy')
f_hess_num = lambdify((x1, x2), f_hess, 'numpy')
```

con lo que ahora `f_num`, `f_d1_num` y `f_hess_num` representan a f, su gradiente y su matriz hessiana, respectivamente. Sin embargo, las funciones generadas por `lambdify` toman un argumento por cada variable en la correspondiente expresión, mientras que las funciones de optimización de SciPy esperan funciones vectorizadas en las que todos los argumentos se encuentran empaquetados en un array. Por lo tanto, para obtener funciones que sean compatibles con las rutinas de optimización de SciPy, debemos "envolver" las funciones generadas por `lambdify` con funciones de Python que reacomoden los argumentos, tal como mostramos en la celda 11:

CELL 11

```
def func_XY_a_X_Y(f):
    return lambda X: np.array(f(X[0], X[1]))

f = func_XY_a_X_Y(f_num)
f1 = func_XY_a_X_Y(f_d1_num)
fh = func_XY_a_X_Y(f_hess_num)
```

Finalmente tenemos todo listo para proceder con la optimización. Esto lo hacemos con la función `minimze` de `scipy.optimize`, pasándole como argumentos la función a minimizar (`f`), su gradiente (`f1`, argumento `jac`) y la matriz hessiana (`fh`, argumento `hess`), así como el punto de partida de la búsqueda: `(1, 1)`:

CELL 12

```
from scipy.optimize import minimize

solucion = minimize(f, (1, 1), method='Newton-CG', jac=f1, hess=fh)
solucion
```

```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: -50.058893310567896
       x: [ 2.747e+00  2.747e+00]
     nit: 4
     jac: [ 2.301e-04  2.301e-04]
    nfev: 7
    njev: 7
    nhev: 4
```

La función encuentra un mínimo en $(x_1, x_2) = (2{,}747, 2{,}747)$, y muestra información sobre cómo ha resultado la ejecución del algoritmo, incluyendo el número de iteraciones, el valor de la función en el mínimo, el valor del gradiente (`jac`) y la cantidad de evaluaciones de la función, del gradiente y del hessiano. Podemos verificar el valor de la función en el mínimo hallado:

CELL 13

```
f(solucion.x)
```

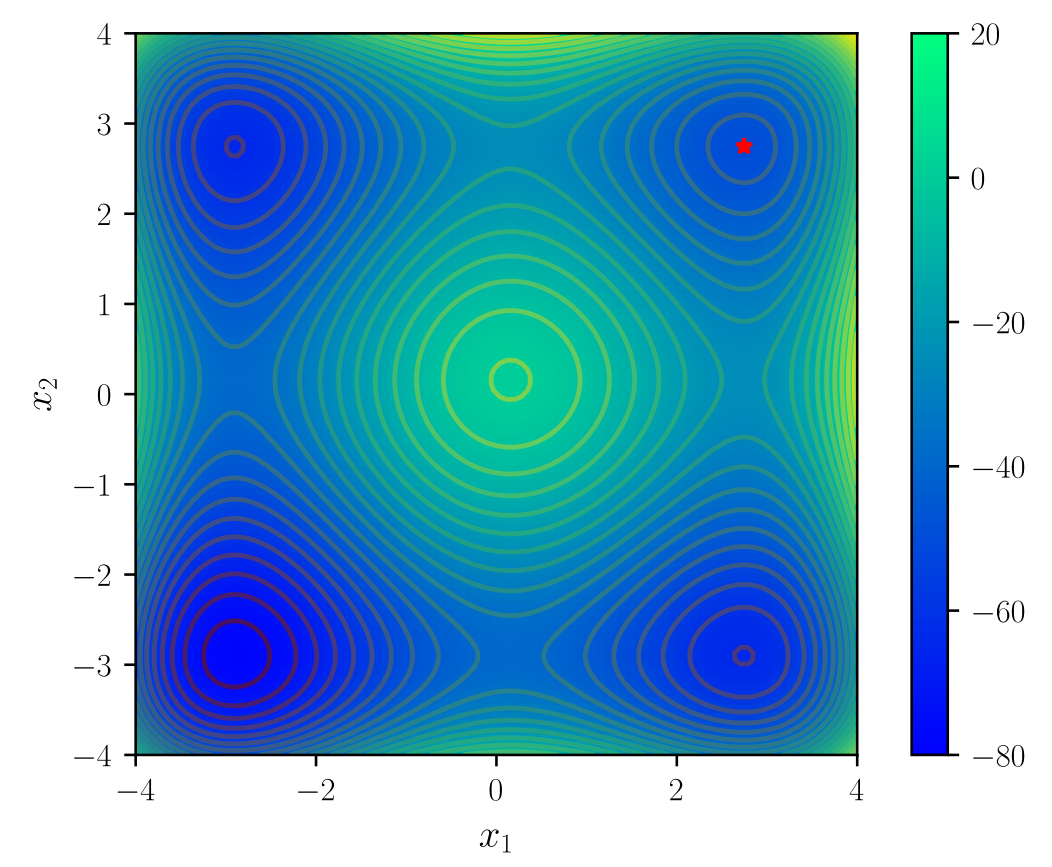
```
array(-50.05889331)
```

Como es usual en estos casos, resulta útil visualizar el problema. Para ello realizamos un mapa de calor donde el color representa el valor de la función y marcamos con una estrella roja el óptimo hallado:

CELL 14

```
import matplotlib.pyplot as plt
import matplotlib.colors as colors

fig, ax = plt.subplots(figsize=(6, 4))
xLim = yLim = 4
x = y = np.linspace(-xLim, xLim, 100)
X, Y = np.meshgrid(x, y)
x_min = solucion.x
palette = plt.cm.winter
im=ax.imshow(f_num(X, Y), extent=[-xLim, xLim, -yLim, yLim],
          cmap=palette, norm=colors.Normalize(vmin=-80.0, vmax=20),
          origin='lower')
c = ax.contour(X, Y, f_num(X, Y), 30)
ax.plot(x_min[0], x_min[1], 'r*', markersize=5)
ax.set_xlabel(r"$x_1$", fontsize=12)
ax.set_ylabel(r"$x_2$", fontsize=12)
plt.colorbar(im, ax=ax)
plt.show()
```



En la práctica no siempre suele ser factible calcular el gradiente o el hessiano de la función, por lo que pueden utilizarse otros métodos que estiman estas funciones numéricamente. Un método popular mencionado anteriormente es el BFGS. Para utilizar este método podemos invocar la función `optimize` con el argumento `method='BFGS'`:

Cell 15

```
sol_bfgs = minimize(f, (1, 1), method='BFGS')
sol_bfgs
```

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: -50.058893310566276
        x: [ 2.747e+00  2.747e+00]
      nit: 6
      jac: [-6.676e-06 -6.676e-06]
 hess_inv: [[ 4.741e-01 -4.802e-01]
            [-4.802e-01  5.582e-01]]
     nfev: 33
     njev: 11
```

Vemos que en este caso no es necesario suministrar el gradiente ni la matriz hessiana, y tambien podemos notar que, en consecuencia, el número de evaluaciones de la función aumenta significativamente (30) respecto del método de Newton (7 evaluaciones).

Otro método del que dispone SciPy es el del gradiente conjugado. En este caso podemos usar el gradiente de la función, mientras que el hessiano no es requerido:

Cell 16

```
sol_cg = minimize(f, (1, 1), method='CG', jac=f1)
sol_cg
```

```
   message: Optimization terminated successfully.
   success: True
    status: 0
       fun: -50.05889331056788
         x: [ 2.747e+00  2.747e+00]
       nit: 5
       jac: [-2.170e-09 -2.170e-09]
      nfev: 17
      njev: 16
```

Si no se conocen ni el gradiente ni el hessiano, el método BFGS suele ser un buen punto de partida para la búsqueda del óptimo. En caso de poder calcular el gradiente, el método del gradiente conjugado resulta más eficiente (en nuestro caso solo necesita 17 evaluaciones de la función para hallar el mínimo). Naturalmente, cuanta más información le podamos aportar al método, más eficiente será el cálculo, tal como se muestra al utilizar el método de Newton, que provee la convergencia más rápida. En cualquier caso es necesario también evaluar la complejidad de las funciones para el cálculo del gradiente o del hessiano, debido a que puede resultar costosa la evaluación numérica de ellas, haciendo más conveniente el uso del método BFGS.

En general, los métodos que hemos visto hasta aquí convergen a mínimos locales. En los problemas con muchos mínimos locales, es muy posible que estos métodos converjan a uno de estos mínimos locales (dependiendo, naturalmente, del valor inicial de la iteración), y que nos resulte difícil localizar el mínimo global. En estos casos es recomendable realizar una primera búsqueda utilizando un método de fuerza bruta, que consiste simplemente en discretizar el dominio en una grilla y evaluar la función en los nodos de esta discretización, y utilizar el resultado de esta

búsqueda como punto incial de un método iterativo. SimPy provee la función `brute` de `optimize` para realizar esta búsqueda inicial, a la que le pasamos como argumento la función `f`, y una tupla de objetos `slice`, con un elemento por cada dimensión del espacio de parámetros (valor inicial, valor final y paso para cada coordenada), que especifica la grilla sobre la que se evaluará la función. Como tercer argumento especificamos `finish=None` para evitar que la función `brute` refine la búsqueda (dejaremos esa tarea a uno de nuestros algoritmos, posteriormente):

Cell 17

```
from scipy.optimize import brute

sol_ini = brute(f, (slice(-5, 5, 0.5), slice(-5, 5, 0.5)), finish=None)
sol_ini
```

```
array([-3., -3.])
```

Cell 18

```
f(sol_ini)
```

```
array(-78.)
```

Vemos que el punto de la grilla $(-3, 3)$ ofrece el menor valor de `f` (-78). Ahora usamos ese punto como valor inicial del método BFGS:

Cell 19

```
sol_bfgs_g = minimize(f, sol_ini, method='BFGS')
sol_bfgs_g
```

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: -78.33233140754265
        x: [-2.904e+00 -2.904e+00]
      nit: 3
      jac: [-1.907e-06 -1.907e-06]
 hess_inv: [[ 5.145e-01 -4.855e-01]
            [-4.855e-01  5.145e-01]]
     nfev: 15
     njev: 5
```

que converge al valor del mínimo global que anticipamos al inicio. Dado que nuestro punto de partida es bastante cercano al óptimo, el número de evaluaciones requeridas por el algoritmo BFGS es la mitad de la requerida para el punto inicial $(1, 1)$, que converge a un mínimo local.

1.3. Optimización con restricciones

La incorporación de restricciones a los problemas de optimización incrementa el nivel de complejidad de los métodos para obtener las soluciones posibles. No existe una única forma de categorizar estos problemas, en este capítulo abordaremos tres formas de considerar restricciones: límites en las variables de decisión, programación lineal y programación no lineal.

1.3.1. Restricciones en las variables de decisión

Tal vez la manera más simple de incluir restricciones en un problema de optimización consiste en limitar el dominio de búsqueda de los valores óptimos de las variables de decisión. Por ejemplo, encontrar el mínimo de $f(x, y)$ sujeto a $a \leq x \leq b$ y $c \leq y \leq d$. La restricción $a \leq x \leq b$ es simple dado que solo restringe los valores de la variable x independientemente de las otras variables del problema.

SciPy puede resolver estos problemas de optimización con restricciones simples por medio del algoritmo L-BFGS-B, que es una variante del método BFGS que utilizamos en la sección 1.2. Veamos cómo aplicamos este método para obtener el mínimo de la función de Rosembrock[5]:

$$f(x_1, x_2) = (a - x_1)^2 + b(x_2 - x_1^2)^2$$

donde usaremos $a = 1$ y $b = 1$. Esta función tiene un mínimo global en (a, a^2) en donde la función se anula. Intentaremos localizar los mínimos en el problema sin restricciones, y restringiendo el dominio de las variables de decisión al rectángulo $2 \leq x_1 \leq 3$ y $0 \leq x_2 \leq 2$:

CELL 20

```
def f_ros(X):
    x1, x2 = X
    a, b = 1, 1
    return (a - x1)**2 + b * (x2 - x1**2)**2

x0 = [0, 0]
res_opt = minimize(f_ros, x0, method='BFGS')
bnd_x1, bnd_x2 = (2, 3), (0, 2)
res_opt_bnd = minimize(f_ros, x0, method='L-BFGS-B',
                       bounds=[bnd_x1, bnd_x2])
print(f"Sin restricción: f = {res_opt.fun} en x = {res_opt.x}")
print(f"Con restricción: f = {res_opt_bnd.fun} en x = {res_opt_bnd.x}")
```

```
Sin restricción: f = 2.9986375725184793e-15 en x = [0.99999995 0.99999991]
Con restricción: f = 5.0 en x = [2. 2.]
```

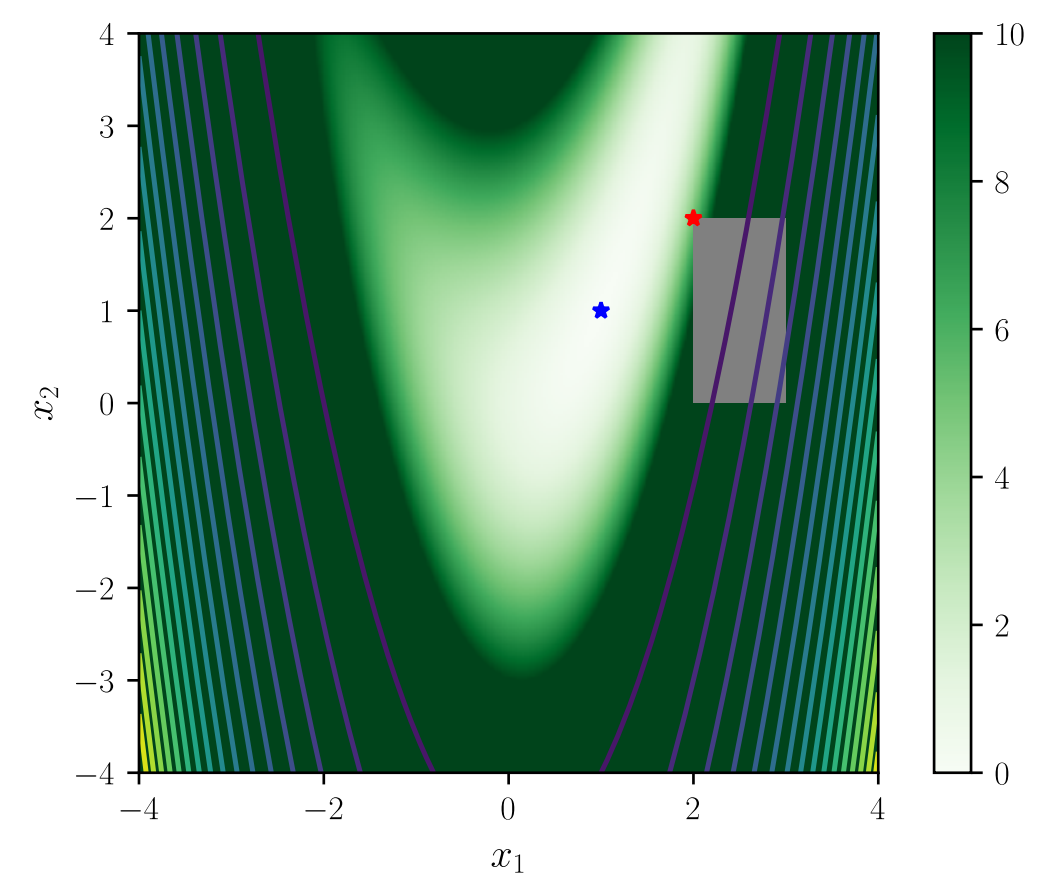
Aquí podemos ver que cuando optimizamos sin restringir los valores de x_1 y x_2 localizamos el mínimo global de la función de Rosembrock, pero al establecer la optimización limitada a los intervalos especificados para cada variable, el valor óptimo se establece en $(2, 2)$ en donde $f = 5$. Dado que este ejemplo presenta una optimización de dos variables, podemos realizar un gráfico para visualizar cómo se ubican los valores óptimos en cada caso. En la celda siguiente construimos esta visualización, realizando previamente una definición de la función de Rosembrock para que acepte como parámetros de entrada dos valores en vez de una tupla, ya que de este modo necesitamos ingresar los valores de la grilla de visualización. Marcamos también en gris el rectángulo que comprenden los intervalos de la restricción. La estrella azul representa los valores que minimizan la función sin restricciones, mientras que la roja lo hace al establecer las restricciones.

5 Ver https://es.wikipedia.org/wiki/Función_de_Rosenbrock.

CELL 21

```
def f_ros_p(x, y):
    return f_ros((x, y))

fig, ax = plt.subplots(figsize=(6, 4))
xLim = yLim = 4
x = y = np.linspace(-xLim, xLim, 200)
X, Y = np.meshgrid(x, y)
palette = plt.cm.Greens
im=ax.imshow(f_ros_p(X, Y), extent=[-xLim, xLim, -yLim, yLim],
          cmap=palette, norm=colors.Normalize(vmin=-0.0, vmax=10),
          origin='lower')
c = ax.contour(X, Y, f_ros_p(X, Y), 20)
ax.plot(res_opt.x[0], res_opt.x[1], 'b*', markersize=5)
ax.plot(res_opt_bnd.x[0], res_opt_bnd.x[1], 'r*', markersize=5)
bound_zone = plt.Rectangle((bnd_x1[0], bnd_x2[0]),
                           bnd_x1[1] - bnd_x1[0],
                           bnd_x2[1] - bnd_x2[0],
                           facecolor='gray')
ax.add_patch(bound_zone)
ax.set_xlabel(r"$x_1$", fontsize=12)
ax.set_ylabel(r"$x_2$", fontsize=12)
plt.colorbar(im, ax=ax)
plt.show()
```



1.3.2. Programación lineal

La programación lineal aborda el problema de optimizar una función objetivo lineal sujeta a restricciones lineales en forma de igualdades y desigualdades sobre las variables de decisión. Estos problemas se pueden expresar en una variedad de formas equivalentes. Veremos en un ejemplo simple cómo abordar un problema de programación lineal.

Supongamos que disponemos de tres tipos de alimentos, A_1, A_2 y A_3, que contienen tres tipos de nutrientes (carbohidratos, proteínas y vitaminas) en las siguientes cantidades:

Alimento	Carbohidratos	Proteínas	Vitaminas	Costo ($/kg)
A_1	1	4	3	0.82
A_2	7	2	2	0.50
A_3	2	2	0	0.40

El requerimiento diario de carbohidratos, proteínas y vitaminas es 10, 15 y 6, respectivamente. El problema es determinar cuánto es necesario consumir de cada alimento para obtener las cantidades requeridas de cada nutriente al menor costo, por lo tanto, la elección natural de las variables de decisión a optimizar será x_1, x_2 y x_3 como las cantidades a consumir por día de cada alimento.

El paso siguiente es determinar la función objetivo a maximizar o minimizar. En este ejemplo, queremos minimizar el costo diario total, que se obtiene mediante el cálculo $0{,}82\,x_1 + 0{,}50\,x_2 + 0{,}40\,x_3$.

Por último necesitamos describir las restricciones que deben satisfacer x_1, x_2 y x_3. En primer lugar, estas variables deben ser positivas (no podemos comer cantidades negativas de alimentos). Estas restricciones se denominan "restricciones de no negatividad", y se encuentran muy frecuentemente en los problemas de programación lineal. Además, no todos los valores de x_1, x_2 y x_3 dan lugar a una dieta con las cantidades diarias necesarias de nutrientes. La cantidad de proteínas en x_1 unidades de A_1, x_2 de A_2 y x_3 de A_3 es $4\,x_1 + 2\,x_2 + 2\,x_3$, y esa cantidad debe ser al menos de 15 por día. Esto significa que x_1, x_2 y x_3 deben satisfacer $4\,x_1 + 2\,x_2 + 2\,x_3 \geq 15$.

Entonces, este problema de optimización dietaria se puede formular de la siguiente forma:

$$\begin{aligned} \text{minimizar:} \quad & f(x_1, x_2, x_3) = 0{,}82\,x_1 + 0{,}50\,x_2 + 0{,}40\,x_3 \\ \text{sujeto a:} \quad & x_1 + 7\,x_2 + 2\,x_3 \geq 10 \\ & 4\,x_1 + 2\,x_2 + 2\,x_3 \geq 15 \\ & 3\,x_1 + 2\,x_2 \geq 6 \\ & x_1 \geq 0;\ x_2 \geq 0;\ x_3 \geq 0 \end{aligned} \tag{1.3}$$

Se dice que una solución $\boldsymbol{x} = (x_1, x_2, x_3)$ es factible con respecto al programa lineal anterior si satisface todas las restricciones anteriores. El conjunto de soluciones factibles se denomina espacio factible o región factible. Una solución factible es óptima si el valor de su función objetivo es igual al valor más pequeño que f puede tomar en la región factible.

Para obtener la solución a este problema de programación lineal utilizaremos la función `linprog` de `scipy.optimize`, que puede resolver problemas con la forma:

$$\begin{aligned} \min_{\boldsymbol{x}} \ & \boldsymbol{c}^\mathsf{T} \boldsymbol{x} \\ \text{tal que:} \quad & \boldsymbol{A}_{ub}\, \boldsymbol{x} \leq \boldsymbol{b}_{ub} \\ & \boldsymbol{A}_{eq}\, \boldsymbol{x} \leq \boldsymbol{b}_{eq} \\ & \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u} \end{aligned} \tag{1.4}$$

donde $\boldsymbol{x}$ representa el vector con las variables de decisión, $\boldsymbol{c}$ es el vector con los coeficientes de la función lineal f, $\boldsymbol{A}_{ub}$ es la matriz con los coeficientes de las restricciones de desigualdad, en la que cada fila especifica los coeficientes de una desigualdad lineal sobre $\boldsymbol{x}$, $\boldsymbol{b}_{ub}$ es el vector de las desigualdades lineales en el que cada elemento representa el límite superario del correspondiente valor de $\boldsymbol{A}_{ub}\, \boldsymbol{x}$. Del mismo modo, $\boldsymbol{A}_{eq}$ y $\boldsymbol{b}_{eq}$ representan restricciones mediante igualdades (que en este problema no tenemos). Finalmente, $\boldsymbol{l}$ y $\boldsymbol{u}$ son los límites inferiores y superiores del dominio de búsqueda de los valores óptimos de $\boldsymbol{x}$ (intepretando la desigualdad elemento por elemento).

Podemos ver que en nuestro problema representado en la forma (1.3) las restricciones de desigualdad tienen el sentido inverso al requerido por `linprog`. Esto no representa ninguna dificultad, ya que multiplicando cada ecuación miembro a miembro por (-1) las adaptamos para usarlas en la función.

La función `linprog` utiliza los métodos `highs-ds` y `highs-ipm` para resolver el problema de minimización. que consisten en rutinas desarrolladas en C++ de alto desempeño[6] que implementan los métodos simplex dual revisado y de punto interior, cuyas descripciones exceden el alcance de este libro [3]. En la celda 22 definimos `c` como una lista con los coeficientes de f, `A` representa la matriz $\boldsymbol{A}_{ub}$, la lista `b` contiene los elementos del vector $\boldsymbol{b}_{ub}$, y finalmente establecemos los límites para cada variable de decisión con la tupla `(0, None)` que significa que el límite inferior en cada variable es cero, y el superior es infinito (todas las variables de decisión son no negativas):

Cell 22

```
c = [0.82, 0.5, 0.4]
A = [[-1, -7, -2], [-4, -2, -2], [-3, -2, 0]]
b = [-10, -15, -6]
x0_bounds = (0, None)
x1_bounds = (0, None)
x2_bounds = (0, None)
```

Ahora podemos invocar la función `linprog`, guardando su resultado en un objeto `res` que contiene el valor mínimo de la función objetivo (`res.fun`) evaluada en los valores óptimos obtenidos (`res.x`):

6 Ver https://highs.dev/.

CELL 23

```
from scipy.optimize import linprog

res = linprog(c, A_ub=A, b_ub=b, bounds=[x0_bounds, x1_bounds, x1_bounds])
print(f"Valor de f: {res.fun}, en x = {res.x}")
print(f"c . x = {c @ res.x}")
```

```
Valor de f: 3.052380952380952, en x = [1.9047619  0.14285714 3.54761905]
c . x = 3.0523809523809526
```

que verificamos mostrando el valor del producto `c @ res.x`. El objeto `res` contiene más información acerca de la ejecución de la rutina de optimización:

CELL 24

```
res
```

```
         message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
         success: True
          status: 0
             fun: 3.052380952380952
               x: [ 1.905e+00  1.429e-01  3.548e+00]
             nit: 3
           lower:  residual: [ 1.905e+00  1.429e-01  3.548e+00]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00]
           upper:  residual: [       inf        inf        inf]
                 marginals: [ 0.000e+00  0.000e+00  0.000e+00]
           eqlin:  residual: []
                 marginals: []
         ineqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00]
                 marginals: [-1.238e-02 -1.876e-01 -1.905e-02]
  mip_node_count: 0
  mip_dual_bound: 0.0
         mip_gap: 0.0
```

Para la interpretación de la información contenida en `res`, invitamos a ver la documentación[7].

1.3.3. Programación no lineal

Una vez más, cuando en el problema intervienen relaciones no lineales entre las variables de decisión, la complejidad aumenta y son necesarios métodos más sofisticados para encontrar el valor óptimo, y en muchas oportunidades no es posible obtener una minimización o maximización que respete todas las restricciones.

Una de las técnicas que se pueden utilizar para problemas de optimización con restricciones no lineal es la de los multiplicadores de Lagrange, que convierte un problema con restricciones en uno sin restricciones mediante la incorporación de variables adicionales. La idea es incorporar cada restricción a la función objetivo y resolver el problema como uno sin restricciones.

Por ejemplo, consideremos el problema de optimización mín $f(\boldsymbol{x})$ sujeto a restricciones de igualdad $g(\boldsymbol{x}) = 0$. Si no consideramos esta restricción, el gradiente de $f(\boldsymbol{x})$ se anula en los puntos extremos, $\nabla f(\boldsymbol{x}) = 0$. Se puede mostrar que al incorporar la restricción, el negativo del

7 https://docs.scipy.org/doc/scipy/reference/optimize.linprog-highs.html#optimize-linprog-highs.

gradiente pertenece al espacio soportado por la restricción normal, es decir, $-\nabla f(\boldsymbol{x}) = \boldsymbol{\lambda} J_g^\intercal(\boldsymbol{x})$, donde $J_g(\boldsymbol{x})$ es la matriz jacobiana de la función de restricción $g(\boldsymbol{x})$ y $\boldsymbol{\lambda}$ son las nuevas variables representadas por el vector de los multiplicadores de Lagrange. Esta condición tiene su origen en igualar a cero la función $L(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\lambda}^\intercal g(\boldsymbol{x})$ denominada "función lagrangiana". Si ambas funciones f y g son continuas y diferenciables, un punto estacionario $(\boldsymbol{x}_0, \boldsymbol{\lambda}_0)$ de L corresponde a un $\boldsymbol{x}_0$ que es un óptimo del problema original con restricciones.

Veamos un ejemplo minimizando nuevamente la función de Rosembrock

$$f(x_1, x_2) = (a - x_1)^2 + b(x_2 - x_1^2)^2$$

con la condición $g(x_1, x_2) = x_1^2 + x_2^2 = 1$. Al igual que en el ejemplo de la subsección 1.3.2. tomamos $a = 1$ y $b = 1$. Esta vez utilizaremos como argumento de la función `minimize` el método SLSQP (*Sequential Least Squares Programming*), que implementa en forma iterativa la optimización de un modelo cuadrático utilizando aproximaciones de segundo orden del lagrangiano. La condición de restricción debe pasarse como diccionario en el argumento `constraints` de `optimize`, declarando en la definición que se trata de una restricción de igualdad a cero (`type='eq'`):

Cell 25

```
def g(X):
    x1, x2 = X
    return x1**2 + x2**2 - 1

constrain = dict(type='eq', fun=g)
```

Ahora hacemos la minimización del mismo modo que en los ejemplos anteriores:

Cell 26

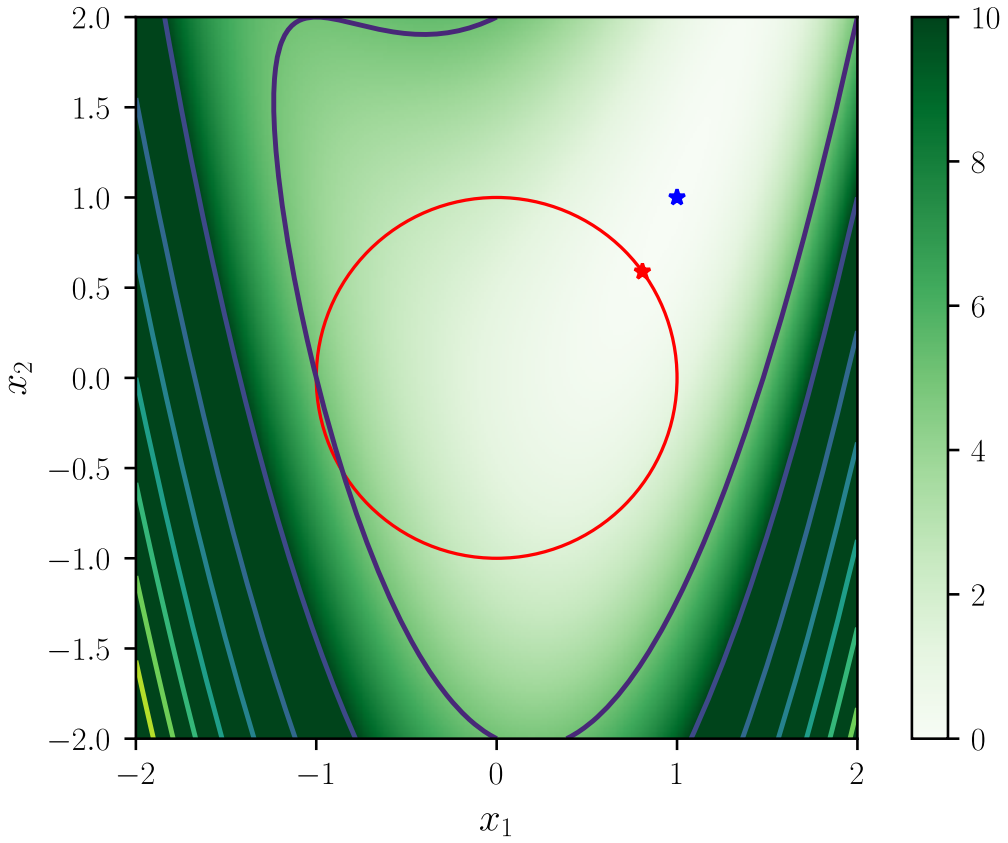
```
res = minimize(f_ros, x0, method='SLSQP', constraints=[constrain])
res
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: 0.04091903706230661
       x: [ 8.082e-01  5.889e-01]
     nit: 8
     jac: [-1.762e-01 -1.284e-01]
    nfev: 33
    njev: 8
```

Y visualizamos el proceso mostrando con una estrella azul el mínimo de la función de Rosembrock sin restricciones, y el nuevo óptimo (en rojo) cuya restricción exige que se encuentre sobre la circunferencia de radio 1:

CELL 27

```
fig, ax = plt.subplots(figsize=(6, 4))
xLim = yLim = 2
x = y = np.linspace(-xLim, xLim, 200)
X, Y = np.meshgrid(x, y)
palette = plt.cm.Greens
im=ax.imshow(f_ros_p(X, Y), extent=[-xLim, xLim, -yLim, yLim], cmap=palette,
             norm=colors.Normalize(vmin=-0.0, vmax=10), origin='lower')
c = ax.contour(X, Y, f_ros_p(X, Y), 10)
circle = plt.Circle( (0, 0), 1, fill=False, color='red')
ax.add_artist(circle)
ax.plot(res_opt.x[0], res_opt.x[1], 'b*', markersize=5)
ax.plot(res.x[0], res.x[1], 'r*', markersize=5)
ax.set_xlabel(r"$x_1$", fontsize=12)
ax.set_ylabel(r"$x_2$", fontsize=12)
plt.colorbar(im, ax=ax)
plt.show()
```



1.4. Algoritmo genético

En esta sección tomaremos un abordaje cualitativamente diferente al que hemos usado hasta ahora para resolver problemas de optimización. Introduciremos en forma sintética el algoritmo genético (AG), como ejemplo de la clase de métodos heurísticos de optimización. En particular,

los AG están inspirados en el proceso de selección natural en el que una "población" de posibles soluciones (denominados "individuos" o "cromosomas") evolucionan hacia soluciones mejores.

Los individuos de un AG codifican las variables de decisión del problema de optimización en cadenas de longitud finita de algún alfabeto. Estas cadenas que representan candidatos a la solución están compuestas de "genes" dados por el alfabeto utilizado. Para evolucionar hacia mejores soluciones e implementar un mecanismo de selección natural, es necesario una medida que distinga buenas soluciones de las que no son tan adecuadas para resolver el problema. Por lo general, esta medida es una función objetivo, que puede estar dada tanto por una expresión matemática como por el resultado de una simulación, y a la que generalmente se la denomina *fitness*.

Una vez que el problema ha sido codificado a través de un determinado alfabeto en forma de cromosomas, y que disponemos de la correspondiente función de *fitness*, el algoritmo de evolución se implementa de la siguiente manera:

1. Inicialización: se establece una población inicial de inidividuos, generalmente generados de forma aleatoria sobre el dominio de búsqueda. Si es posible incopororar conocimiento específico del problema en cuestión, se puede hacer en este paso.
2. Evaluación: una vez que la población ha sido inicializada, o cuando se obtiene una nueva generación de hijos, es necesario evaluar el *fitness* de cada individuo.
3. Selección: Un conjunto de individuos de la población, priorizando aquellos con mejor *fitness*, es elegido para formar el conjunto de padres de la siguiente generación de individuos.
4. Recombinación (o *crossover*): el cromosoma de dos padres elegidos en el paso anterior es recombinado para obtener nuevos individuos hijos.
5. Mutación: en forma aleatoria se modifica un individuo de la población.
6. Reemplazo: la generación de hijos generada por selección, recombinación y mutación reemplaza (total o parcialmente) la generación anterior de individuos.
7. Se repiten los pasos 2 – 6 hasta que se satisfaga una condición de finalización.

Cada uno de estos pasos requiere de la implementación de operadores (selección, *crossover*, mutación y reemplazo) que puede hacerse de diversas maneras, y de la elección de cada uno de ellos dependerá la eficacia con la que el AG evolucione hacia el óptimo. Mencionaremos algunas opciones a continuación.

Los métodos de selección se pueden clasificar en selección proporcional al *fitness* o en métodos ordinales. Un ejemplo de implementación del primer caso lo constituye el método de la ruleta, en el que a cada individuo se le asigna una "caja" cuyo tamaño es proporcional a su *fitness*, de modo que al lanzar la bola de la ruleta haya más probabilidades de que caiga en la caja de los individuo con mayor fitness. Un ejemplo de método ordinal es la selección por torneo, en el que se selecciona un número arbitrario de candidatos (mínimamente dos), con o sin reemplazo, y el que tiene el mayor *fitness* gana el torneo y es seleccionado como padre para la próxima generación. El torneo debe repetirse hasta seleccionar la cantidad de padres necesaria.

La operación de recombinación o *crossover* consiste en mezclar los genes de los cromosomas de dos padres. Típicamente, esto se logra intercambiando tramos de genes de los padres para

formar a los hijos de la nueva generación, a través del "*crossover* de uno o dos puntos", en los que el o los puntos de corte se eligen aleatoriamente a lo largo del cromosoma; o del "*crossover* uniforme", en el que se genera una máscara binaria y se intercambian los genes a los que les corresponde el valor `1` en la posición de la máscara. Los siguientes esquemas ejemplifican estas variantes del operador de *crossover*.

Crossover de un punto:

Padres	Hijos
$a_1a_2a_3a_4\|a_5a_6a_7$	$a_1a_2a_3a_4b_5b_6b_7$
$\Longrightarrow$	
$b_1b_2b_3b_4\|b_5b_6b_7$	$b_1b_2b_3b_4a_5a_6a_7$

Crossover de dos puntos:

Padres	Hijos
$a_1a_2\|a_3a_4\|a_5a_6a_7$	$a_1a_2b_3b_4a_5a_6a_7$
$\Longrightarrow$	
$b_1b_2\|b_3b_4\|b_5b_6b_7$	$b_1b_2a_3a_4b_5b_6b_7$

Crossover uniforme:

Padres		Hijos
0 1 1 0 1 0 1	$\longleftarrow$	*Máscara*
$a_1a_2a_3a_4a_5a_6a_7$		$a_1b_2b_3a_4b_5a_6b_7$
	$\Longrightarrow$	
$b_1b_2b_3b_4b_5b_6b_7$		$b_1a_2a_3b_4a_5b_6a_7$

También existen diferentes maneras de implementar el operador de mutación. Tal vez la más simple consiste en reemplazar aleatoriamente un gen del cromosoma por un valor arbitrario. Preferentemente este valor debe estar dentro del dominio de búsqueda de la variable asociada al gen por medio del procedimiento de codificación. Si no es el caso, es posible obtener cromosomas que no representen una solución factible. Un método más sofisticado consiste en reemplazar un gen aleatorio por un valor obtenido mediante alguna distribución de probabilidad alrededor del valor del gen que estamos reemplazando. Una representación de este mecanismo se muestra en el siguiente esquema:

Mutación:

Antes	Después
$a_1a_2a_3a_4a_5a_6a_7 \Longrightarrow$	$a_1a_2a_3b_4a_5a_6a_7$

Por último, también existen variantes para el operador de reemplazo. El más simple consiste en reemplazar completamente la población actual con la nueva población de hijos. Esto es sencillo de realizar, pero puede suceder que en la nueva generación los individuos no tengan *fitness* tan buenos como el de algunos padres. Una forma de sortear esta posibilidad es implementar "elitismo", que consiste en reemplazar solamente una porción r de los individuos originales, dejando sobrevivir en la nueva generación la porción $1 - r$ de los que tienen mejor *fitness*. Es usual que a través de la evolución de la población se mantenga constante el número de individuos que la componen.

Implementaremos ahora un AG básico para resolver un problema que, según su tamaño (es decir, el número de variables de decisión), puede ser muy difícil de abordar. El "problema de la mochila" se puede enunciar de la siguiente forma:

> Dado un conjunto de ítems, cada uno con un peso y un valor, determinar qué ítems incluir en una colección de forma tal que el peso total sea menor o igual a la capacidad máxima de la mochila, maximizando el valor de los objetos incluidos en la colección.[8]

Formalmente, si tenemos n ítems cuyos valores son v_i y sus pesos p_i $(0 \leq i \leq n)$, con un valor máximo P de carga, el problema consiste en:

$$\begin{aligned} \text{maximizar:} \quad & f(\boldsymbol{x}) = \sum_{i=1}^{n} v_i \, x_i \\ \text{tal que:} \quad & \sum_{i=1}^{n} p_i \, x_i \leq P \\ \text{y} \quad & 0 \leq x_i \leq q_i, \text{ para } 1 \leq i \leq n \end{aligned} \tag{1.5}$$

Si $q_i = 1$ para cada ítem, se trata del problema de la mochila 0-1. Si uno o más q_i es infinito se trata del problema de la mochila no acotado, mientras que en otro caso se llama problema de la mochila acotado. Abordaremos el caso en el que $x_i \in \{0, 1\}$.

El código a continuación contiene dos clases, que permiten representar objetos de tipo `Individual` y `Population`, siendo este último el objeto que contiene una lista de individuos y métodos que implementan los operadores del AG. Comenzamos describiendo la clase `Individuo`

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

import random


class Individual():
    genes = (0, 1)
    max_weight = 102

    def __init__(self, elements, chromosome=None):
        # structures after elements to simplify later calculus
        self.values, self.weights = zip(*elements)
        self.size = len(elements)

        # if the chromosome is not specified build a random one
        if chromosome is None:
            chromosome = [random.choice(self.genes) for _ in range(self.size)]
        self.chromosome = chromosome

        # these two values will be set on evaluation
        self.fitness = None
        self.weight = None

    def evaluate(self):
        """Calculate own weight and fitness value.

        This is not done on __init__ because the chromosome may change after creation, it
```

8 Recomendamos la lectura de H. Kellerer, U. Pferschy y D. Pisinger. *Knapsack Problems*. Berlin: Springer-Verlag, 2004. DOI: 10.1007/978-3-540-24777-7.

```
        will be called automatically before comparison.
        """
        self.fitness = sum(gene * value for gene, value in zip(self.chromosome, self.values))
        self.weight = sum(gene * weight for gene, weight in zip(self.chromosome, self.weights))
        if self.weight > self.max_weight:
            # not fit at all
```

En la clase `Individual` comenzamos estableciendo una tupla que indica el conjunto de genes con el que codificaremos el cromosoma (nuestro alfabeto), que está limitado a `0` y `1`. También indicamos cuál es la capacidad de carga de la mochila, determinado su valor en la variable `max_weight`.

Para la inicialización de los objetos necesitamos pasar como argumentos dos listas: `elements`, y `chromosome`. La primera contiene tuplas de pares `(valor, peso)` que son necesarios para la evaluación del *fitness*, mientras que la segunda lista permite inicializar un individuo con un cromosoma dado. En caso que no suministremos un cromosoma, creamos uno seleccionando al azar elementos de `self.genes`, la cantidad de veces necesaria para completar el tamaño del cromosoma, que queda determinado por la longitud de la lista de los valores.

El método `evaluate` calcula el *fitness* del individuo según (1.5), y también guarda el valor del peso total de los objetos que incluye el cromosoma. Nótese que si el peso del individuo es mayor que la capacidad de carga, asignamos un fitness de `-1`, indicando de este modo que el individuo no es apto como solución del problema ya que no será favorable un cromosoma que represente un valor negativo.

```

    def mutate(self):
        """Change a random gene (maybe)."""
        self.fitness = self.weight = None
        self.chromosome[random.randrange(self.size)] = random.choice(self.genes)

    def __lt__(self, other):
        # ensure fitness is useful
        if self.fitness is None:
            self.evaluate()
        if other.fitness is None:
            other.evaluate()

        return other.fitness < self.fitness

    def __str__(self):
        return f"<Individual {id(self)} fitness={self.fitness}>"
```

El método `mutate` implementa el operador de mutación. Simplemente, determina un gen aleatorio dentro del cromosoma y le asigna una elección al azar de `self.genes`. En este ejemplo, tenemos igual probabilidad de cambiar el valor del gen o dejarlo como estaba. También hacemos `None` los valores de `self.fitness` y `self.weight`, ya que ante un eventual cambio en el cromosoma es necesario evaluar nuevamente estas cantidades. Finalmente, los dos métodos siguientes permiten comparar individuos según su *fitness* (`__lt__`), evaluando sus *fitness* de ser necesario, y realizar una representación prolija de un individuo (`__str__`).

Describiremos ahora la clase `Population`. En su inicialización pasamos como argumentos (con valores por defecto) los parámetros que controlan la ejecución del algoritmo y los correspondientes operadores genéticos, así como una lista con los pares `(valor, peso)` de cada ítem a considerar

durante el proceso de optimización. Para generar la población inicial, construimos una lista de individuos cuyos genes se seleccionan aleatoriamente, hasta obtener el número de elementos establecidos en `size`

```
54
55 class Population():
56
57     def __init__(
58         self,
59         size=20,                # Cantidad de individuos
60         max_gen=50,             # Máximo número de generaciones
61         crossover_p=0.9,        # Probabilidad de crossover
62         mutation_p=0.05,        # Probabilidad de mutación
63         optimo=None,            # Valor óptimo de finalización
64         replacement_ratio=0.5,  # Tasa de reemplazo (elitismo)
65         elements=None,          # Pares (valor, peso) de los ítems
66     ):
67         if size % 2:
68             raise ValueError("Population size must be even.")
69         self.size = size
70
71         # set of individuals, always kept sorted (best at beginning)
72         self.individuals = sorted(Individual(elements) for _ in range(size))
73
74         self.elements = elements
75         self.crossover_p = crossover_p
76         self.mutation_p = mutation_p
77         self.max_gen = max_gen
78         self.optimo = optimo
79         self.replacement_ratio = replacement_ratio
```

Dado que ordenaremos la población desde el mejor individuo en orden decreciente del *fitness*, el método `best` nos permite obtener la mejor solución de la generación actual de la población:

```
81
82     def best(self):
83         """Return the best individual in the population."""
```

Para evolucionar la población utilizamos el método `run` que, mientras no se alcance una situación de finalización, invoca al método `step`. Las condiciones de finalización para este ejemplo son alcanzar el número máximo de iteraciones o el valor óptimo del *fitness*.

```
85
86     def run(self):
87         while not self.final():
88             self.step()
89
90     def final(self):
```

El método `step` mencionado arriba consiste en aplicar cada uno de los operadores genéticos selección, *crossover*, mutación, y reemplazo de la población en orden secuencial, e incrementar en uno la variable `self.generation`:

```
92
93      def step(self):
94          parents = self.selection()
95          children = self.do_crossover(parents)
96          self.do_mutation(children)
97          self.do_new_population(children)
98          self.generation += 1
99
100     def selection(self):
101         return [self.tournament() for _ in range(self.size)]
102
103     def tournament(self, size=3, goliat=0.9):
104         """Tournament selection."""
105         competidores = random.choices(self.individuals, k=size)
106         competidores.sort()
107         if random.random() < goliat:
108             return competidores[0]
109         else:
```

El operador de selección, implementado en el método `selection`, devuelve una lista de padres seleccionados mediante un torneo, conteniendo la misma cantidad de individuos que la población inicial. Es posible que los individuos con mejor *fitness* sean seleccionados múltiples veces en la lista de padres, pero el torneo permite, en forma aleatoria y con baja probabilidad, que cada tanto un individuo con un *fitness* no muy bueno integre la selección de padres, agregando de esta forma diversidad genética en la población. El torneo implementado en este ejemplo consiste en seleccionar aleatoriamente `size` individuos de la población (por defecto, `size` vale tres), y con una probabilidad definida en la variable `goliat` (con un valor por defecto del 90 %) gana el torneo el individuo con mayor *fitness*, y con una probabilidad de $(1-\texttt{goliat})$ es seleccionado alguno de los restantes participantes del torneo.

El método `do_crossover` implementa el operador de recombinación de dos puntos. Comienza generando una lista vacía de hijos que se irá completando hasta alcanzar la cantidad de individuos originales en la población. El método selecciona al azar dos padres de la lista que constituye el argumento de `do_crossover` (generada por `selection`), y decide si los recombina con una probabilidad `self.crossover_p` (con un valor por defecto de `0.9`), o si incluye en la lista de hijos los cromosomas de los padres sin recombinar. En el caso en que efectivamente se produzca la recombinación, se seleccionan al azar dos puntos en la cadena de cromosomas de los padres y se intercambia el segmento medio de genes entre estos puntos, dando origen a los nuevos cromosomas de los hijos (dos padres generan dos hijos).

```
111
112     def do_crossover(self, parents):
113         """Create a population of children from parents, maybe altering their chromosomes."""
114         children = []
115         crom_size = parents[0].size
116
117         for _ in range(self.size // 2):
118             parent1, parent2 = random.choices(parents, k=2)
119             if random.random() < self.crossover_p:
120                 new_chromo1 = parent1.chromosome.copy()
121                 new_chromo2 = parent2.chromosome.copy()
122
```

```
123                 # cross the "randomly central" part of one chromosome with the other
124                 x_i = random.randrange(0, crom_size - 1)
125                 x_d = random.randrange(x_i + 1, crom_size)
126                 new_chromo1[x_i:x_d] = parent2.chromosome[x_i:x_d]
127                 new_chromo2[x_i:x_d] = parent1.chromosome[x_i:x_d]
128 
129                 child1 = Individual(self.elements, chromosome=new_chromo1)
130                 child2 = Individual(self.elements, chromosome=new_chromo2)
131             else:
132                 child1 = parent1
133                 child2 = parent2
134             children.extend((child1, child2))
```

El operador de mutación se implementa en el método `do_mutation` de `Poblacion`. Simplemente, un bucle recorre la lista de hijos, y con una probabilidad dada por `self.mutation_p` decide mutar o no cada uno de ellos:

```
136 
137     def do_mutation(self, children):
138         """Mutate some of the children."""
139         for child in children:
140             if random.random() < self.mutation_p:
```

Finalmente, el método `do_new_population` genera la nueva población implementando elitismo, esto es, determina qué fracción de la población debe ser reemplazada por los hijos en la generación siguiente. Mantiene entonces a los mejores padres y a los mejores hijos según sus *fitness*.

```
142 
143     def do_new_population(self, children):
144         """Select a portion of the children into current population."""
145         children.sort()
146         cut_position = int(self.replacement_ratio * self.size)
147         self.individuals[self.size - cut_position:] = children[:cut_position]
```

Utilizamos ahora nuestro AG para resolver el ejemplo 2.1 del libro de Martello y Toth [5], que consiste en un conjunto de 8 elementos cuyos valores y pesos están definidos en la lista `elements` del código a continuación, y que tiene como solución óptima $x = (1, 1, 1, 1, 0, 1, 0, 0)$ con valor 280:

```
150 
151 if __name__ == "__main__":
152     # Martello-Toth Example 2.1 pg 31: Optimo en {1, 1, 1, 1, 0, 1, 0, 0) valor=280 cap. 102
153     # conjunto de pares (valor, peso) para cada elemento a poner en la mochila
154     elements = [
155         (15, 2),
156         (100, 20),
157         (90, 20),
158         (60, 30),
159         (40, 40),
```


VERSIÓN PRELIMINAR

```
        (15, 30),
        (10, 60),
        (1, 10),
    ]
    population = Population(elements=elements, size=20, max_gen=50)
    population.run()
    print("Mejor individuo:")
    print(population.best().chromosome)
    print("Valor:", population.best().fitness)
```

Al ejecutar este código obtenemos:

```
$ ./ag.py
Mejor individuo:
[1, 1, 1, 1, 0, 1, 0, 0]
Valor: 280.0
Peso: 102.0
```

Dado el carácter estocástico del AG, este método de optimización no garantiza que se obtenga el valor óptimo global del problema. Es posible que si ejecutamos nuevamente el programa, con otra semilla para el generador de números aleatorios, obtengamos otro resultado:

```
$ ./ag.py
Mejor individuo:
[1, 1, 1, 0, 1, 0, 0, 1]
Valor: 246.0
Peso: 92.0
```

Este efecto se vuelve más crítico cuanto mayor sea la dimensionalidad del problema, por lo que es recomendable realizar múltiples ejecuciones para observar la distribución de los resultados. Por otra parte, para cada problema constituye una cuestión casi artesanal la determinación de los parámetros de ejecución tales como el tamaño de la población, las probabilidades de recombinación y mutación, y la tasa de reemplazo. Cuando el problema es complejo, es conveniente también incorporar otras representaciones de los operadores genéticos. La experiencia en el uso de esta técnica resulta una buena guía para generar el algoritmo genético más eficiente para cada problema.

1.5. Lecturas recomendadas

- Un muy buen libro de optimización convexa es el de Stephen Boyd y Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. DOI: 10.1017/CBO9780511804441, que se puede descargar desde https://web.stanford.edu/~boyd/cvxbook/.
- Wayne L Winston y Jeffrey B Goldberg. *Operations research: Applications and algorithms*. Brooks/Cole, 2004
- Jorge Nocedal y Stephen Wright. *Numerical Optimization*. en. 2.ª ed. Springer Series in Operations Research and Financial Engineering. New York, NY: Springer, jul. de 2006. Una descripción muy completa de una amplia variedad de métodos de optimización.

- El libro clásico de David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.
- Una excelente introducción a los algoritmos genéticos: Melanie Mitchell. *An introduction to genetic algorithms*. Cambridge, Mass: MIT Press, 1996. DOI: https://doi.org/10.7551/mitpress/3927.001.0001.
- El libro de H. Kellerer, U. Pferschy y D. Pisinger. *Knapsack Problems*. Berlin: Springer-Verlag, 2004. DOI: 10.1007/978-3-540-24777-7, presenta con mucho detalle las diversas variantes del problema de la mochila y numerosas técnicas de optimización.

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] M.A. Styblinski y T.-S. Tang. «Experiments in nonconvex optimization: Stochastic approximation with function smoothing and simulated annealing». En: *Neural Networks* 3.4 (1990), págs. 467-483. DOI: https://doi.org/10.1016/0893-6080(90)90029-K. URL: https://www.sciencedirect.com/science/article/pii/089360809090029K.

[3] Q. Huangfu y J. A. J. Hall. «Parallelizing the dual revised simplex method». En: *Mathematical Programming Computation* 10.1 (mar. de 2018), págs. 119-142. DOI: 10.1007/s12532-017-0130-5. URL: https://doi.org/10.1007/s12532-017-0130-5.

[4] H. Kellerer, U. Pferschy y D. Pisinger. *Knapsack Problems*. Berlin: Springer-Verlag, 2004. DOI: 10.1007/978-3-540-24777-7.

[5] S. Martello y P. Toth. *Knapsack Problems*. West Sussex, England: John Wiley & Sons Ltd., 1990.

[6] Stephen Boyd y Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. DOI: 10.1017/CBO9780511804441.

[7] Wayne L Winston y Jeffrey B Goldberg. *Operations research: Applications and algorithms*. Brooks/Cole, 2004.

[8] Jorge Nocedal y Stephen Wright. *Numerical Optimization*. en. 2.ª ed. Springer Series in Operations Research and Financial Engineering. New York, NY: Springer, jul. de 2006.

[9] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[10] Melanie Mitchell. *An introduction to genetic algorithms*. Cambridge, Mass: MIT Press, 1996. DOI: https://doi.org/10.7551/mitpress/3927.001.0001.

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