Optimisation convexe — TPs

Ivan Lejeune

14 mars 2025

Table des matières

TP1 — Méthodes d'optimisation en 1D

1.1 Méthode de la dichotomie

Exercice 1.1.

1. Quelle équation souhaite-t-on résoudre pour notre problème d'optimisation ? Quelles conditions doit-on vérifier sur f pour appliquer la méthode de la dichotomie ?
2. Ecrire l'algorithme de dichotomie et l'appliquer pour trouver le minimum de la fonction $f = x^2 - 2\sin(x)$ sur $[0, 2]$ avec une précision de 10^{-5}. Comment obtient-on le nombre d'itérations à partir de la précision ?
3. Comparer votre code avec l'implémentation de la fonction `scipy.optimize.bisect`.

Solution.

1. On souhaite résoudre l'équation $f'(x) = 0$ pour trouver le minimum de f. Pour appliquer la méthode de la dichotomie, il faut que f soit continue et unimodale sur $[a, b]$.
   On va alors chercher à résoudre $f'(x) = 0$ pour trouver les points critiques de f.
2. On commence par importer les librairies nécessaires et définir la fonction f :

```python
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import bisect
from scipy.optimize import golden
```

```python
# Define the function to minimise
def f(x):
    return x**2 - 2 * np.sin(x)
```

   Ensuite on définit la fonction `dichotomie` qui prend en argument la fonction f, les bornes de l'intervalle $[a, b]$ sur lequel on cherche le minimum et la précision ε. On l'applique ensuite à notre fonction f :

```python
# Define the dichotomous search algorithm
def dichotomie(f, a, b, epsilon):
    while b - a > epsilon:
        c = (a + b) / 2
        if f(a) * f(c) < 0:
            b = c
        else:
            a = c
    return (a + b) / 2
```

```python
# Define the search conditions
a = 0
b = 2
epsilon = 1e-5
```

```python
# Define the derivative of the function
def df(x):
    return 2 * x - 2 * np.cos(x)
```

```python
# Apply the search algorithm to the function
x_min = dichotomie(df, a, b, epsilon)
print('The minimum of the function is at x =', x_min)
```

   Pour obtenir le nombre d'itérations à partir de la précision, on utilise la formule

$$n = \frac{\log(\frac{b-a}{\varepsilon})}{\log(2)},$$

   où n est le nombre d'itérations, a et b sont les bornes de l'intervalle et ε est la précision.

3. On remarque que la méthode de dichotomie de `scipy.optimize.bisect` donne le même résultat. Un test plus avancé avec des fonctions plus complexes pourrait montrer des différences en termes de performances.

```python
# Comparison with the scipy library
x_min_bisect = bisect(df, a, b, rtol=epsilon)
print('The minimum of the function is at x =', x_min_bisect)
```

1.2 Méthode de Newton

Exercice 1.2.

1. Quelle condition doit vérifier f pour appliquer la méthode de Newton pour le problème d'optimisation ? Comment va être formulé l'itéré de Newton dans ce cas ?
2. Ecrire l'algorithme de Newton dans ce cas et l'appliquer à la fonction $f(x) = x^2 - 2\sin(x)$ avec $x_0 = 1$.

Solution.

1. Pour appliquer la méthode de Newton, il faut que f soit de classe $\mathcal{C}^2$ sur $[a, b]$. L'itéré de Newton est alors donné par
$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}.$$
2. On définit la fonction `newton` qui prend en argument la dérivée première et seconde de f, la valeur initiale x_0 et la précision ε. On l'applique ensuite à notre fonction f :

```python
# Define the Newton search algorithm
def newton(df, df2, x0, epsilon):
    x = x0
    while abs(df(x)) > epsilon:
        x = x - df(x) / df2(x)
    return x
```

```python
# Define the search conditions
x0 = 1
epsilon = 1e-5
```

```python
# Define the function to minimise and its derivatives
def f(x):
    return x**2 - 2 * np.sin(x)


def df(x):
    return 2 * x - 2 * np.cos(x)


def df2(x):
    return 2 + 2 * np.sin(x)
```

```python
# Apply the search algorithm to the function
x_min = newton(df, df2, x0, epsilon)
print('The minimum of the function is at x =', x_min)
```

On remarque que la méthode de Newton converge plus rapidement que la méthode de dichotomie. Cependant, elle nécessite des conditions plus restrictives sur la fonction f. Dans le cas de la dichotomie, f doit être unimodale, tandis que pour Newton, f doit être deux fois dérivable.

1.3 Méthode de la section dorée

Exercice 1.3.

1. Ecrire l'algorithme et l'appliquer à la fonction $f(x) = x^2 - 2\sin(x)$ sur $[0, 2]$
2. Comparer votre code avec l'implémentation de la fonction `scipy.optimize.golden`.
3. Comparer les 3 méthodes pour $f = -\frac{1}{x} + \cos(x)$ sur $[a, b] = [2, 4]$ ou pour $x_0 = 2.5$ au niveau du nombre d'itérations et du temps de calcul.

   Représenter le graphique de la fonction en plaçant les résultats des itérations successives de Newton.

Solution.

1. On définit la fonction `golden` qui prend en argument la fonction f, les bornes de l'intervalle $[a, b]$ sur lequel on cherche le minimum et la précision ε. On l'applique ensuite à notre fonction f :

```python
# Define the golden search algorithm
def golden(f, a, b, epsilon):
    rho = (1 + np.sqrt(5)) / 2
    x1 = 1 / rho * a + (1 - 1 / rho) * b
    x2 = (1 - 1 / rho) * a + 1 / rho * b
    while b - a > epsilon:
        if f(x1) < f(x2):
            b = x2
            x2 = x1
            x1 = 1 / rho * a + (1 - 1 / rho) * b
        else:
            a = x1
            x1 = x2
            x2 = (1 - 1 / rho) * a + 1 / rho * b
    return (a + b) / 2
```

```python
# Define the search conditions
a = 0
b = 2
epsilon = 1e-5
```

```python
# Apply the search algorithm to the function
x_min = golden(f, a, b, epsilon)
print('The minimum of the function is at x =', x_min)
```

2. On remarque que la méthode de la section dorée de `scipy.optimize.golden` donne le même résultat. Un test plus avancé avec des fonctions plus complexes pourrait montrer des différences en termes de performances.

```python
# Comparison with the scipy library
x_min_golden = golden(f, a, b, epsilon)
print('The minimum of the function is at x =', x_min_golden)
```

3. On définira la fonction $f = -\frac{1}{x} + \cos(x)$ et on appliquera les 3 méthodes pour trouver le minimum sur $[2, 4]$ avec $x_0 = 2.5$. On comparera les 3 méthodes en termes de nombre d'itérations et de temps de calcul. On représentera graphiquement les itérations successives des 3 méthodes.

   Commençons par définir la fonction f, ses dérivées première et seconde et les contraintes de recherche :

```python
# Define the function to minimise and its derivatives
def f(x):
    return - 1 / x + np.cos(x)

def df(x):
    return 1 / x**2 + np.sin(x)

def df2(x):
    return -2 / x**3 + np.cos(x)
```

```python
# Define the search conditions
a = 2
b = 4
x0 = 2.5
epsilon = 1e-5
```

On redéfinit ensuite les fonctions **`dichotomie`**, **`newton`** et **`golden`** pour renvoyer le nombre d'itérations en plus du minimum trouvé (cela nous permettra de comparer les 3 méthodes) :

```python
# Redefine the search algorithms to return the number of iterations
def dichotomie(f, a, b, epsilon):
    n = 0
    while b - a > epsilon:
        c = (a + b) / 2
        if f(a) * f(c) < 0:
            b = c
        else:
            a = c
        n += 1
    return (a + b) / 2, n
```

```python
# Redefine the search algorithms to return the number of iterations
def newton(df, df2, x0, epsilon):
    x = x0
    n = 0
    while abs(df(x)) > epsilon:
        x = x - df(x) / df2(x)
        n += 1
    return x, n
```

```python
# Redefine the search algorithms to return the number of iterations
def golden(f, a, b, epsilon):
    rho = (1 + np.sqrt(5)) / 2
    x1 = 1 / rho * a + (1 - 1 / rho) * b
    x2 = (1 - 1 / rho) * a + 1 / rho * b
    n = 0
    while b - a > epsilon:
        if f(x1) < f(x2):
            b = x2
            x2 = x1
            x1 = 1 / rho * a + (1 - 1 / rho) * b
        else:
            a = x1
            x1 = x2
            x2 = (1 - 1 / rho) * a + 1 / rho * b
        n += 1
    return (a + b) / 2, n
```

Enfin, on applique les 3 méthodes pour trouver le minimum de f sur $[2, 4]$ avec $x_0 = 2.5$:

```python
# Apply the search algorithms to the function
x_min_dicho, n_dicho = dichotomie(df, a, b, epsilon)
print('The minimum of the function is at x =', x_min_dicho, 'after', n_dicho, 'iterations')

x_min_newt, n_newt = newton(df, df2, x0, epsilon)
print('The minimum of the function is at x =', x_min_newt, 'after', n_newt, 'iterations')

x_min_gold, n_gold = golden(f, a, b, epsilon)
print('The minimum of the function is at x =', x_min_gold, 'after', n_gold, 'iterations')
```

Pour l'affichage graphique des itérations successives des différentes méthodes, il faut encore modifier les fonctions **`dichotomie`**, **`newton`** et **`golden`** pour qu'elles renvoient les itérés successifs :

```python
# Redefine the search algorithms to return the successive approximations
def dichotomie(f, a, b, epsilon):
    x = []
    while b - a > epsilon:
        c = (a + b) / 2
        if f(a) * f(c) < 0:
            b = c
        else:
            a = c
        x.append((a + b) / 2)
    return x
```

```python
# Redefine the search algorithms to return the successive approximations
def newton(df, df2, x0, epsilon):
    x = x0
    x_list = []
    while abs(df(x)) > epsilon:
        x = x - df(x) / df2(x)
        x_list.append(x)
    return x_list
```

```python
# Redefine the search algorithms to return the successive approximations
def golden(f, a, b, epsilon):
    rho = (1 + np.sqrt(5)) / 2
    x1 = 1 / rho * a + (1 - 1 / rho) * b
    x2 = (1 - 1 / rho) * a + 1 / rho * b
    x = []
    while b - a > epsilon:
        if f(x1) < f(x2):
            b = x2
            x2 = x1
            x1 = 1 / rho * a + (1 - 1 / rho) * b
        else:
            a = x1
            x1 = x2
            x2 = (1 - 1 / rho) * a + 1 / rho * b
        x.append((a + b) / 2)
    return x
```

Et on peut vérifier que les fonctions marchent correctement :

```python
# Apply the search algorithms to the function
x_min_dicho = dichotomie(df, a, b, epsilon)
print('The minimum of the function is at x =', x_min_dicho[-1], 'after', len(x_min_dicho), 'iterations')

x_min_newt = newton(df, df2, x0, epsilon)
print('The minimum of the function is at x =', x_min_newt[-1], 'after', len(x_min_newt), 'iterations')

x_min_gold = golden(f, a, b, epsilon)
print('The minimum of the function is at x =', x_min_gold[-1], 'after', len(x_min_gold), 'iterations')
```

On peut alors afficher les itérations successives des 3 méthodes sur le même graphique avec les lignes suivantes :

```python
# Graph the function and the search algorithms
x = np.linspace(a, b, 100)
y = f(x)

fig, ax = plt.subplots(3, 1, figsize=(10, 15))
```

```python
# Dichotomous search
ax[0].plot(x, y, label='f(x)')
ax[0].scatter(x_min_dicho, [f(x) for x in x_min_dicho], c=range(len(x_min_dicho)), cmap='plasma', label='Dichotomous search')
ax[0].set_title('Dichotomous search')
ax[0].set_xlabel('x')
ax[0].set_ylabel('f(x)')
ax[0].legend()
```

```python
# Newton search
ax[1].plot(x, y, label='f(x)')
ax[1].scatter(x_min_newt, [f(x) for x in x_min_newt], c=range(len(x_min_newt)), cmap='plasma', label='Newton search')
ax[1].set_title('Newton search')
ax[1].set_xlabel('x')
ax[1].set_ylabel('f(x)')
ax[1].legend()
```

```python
# Golden search
ax[2].plot(x, y, label='f(x)')
ax[2].scatter(x_min_gold, [f(x) for x in x_min_gold], c=range(len(x_min_gold)), cmap='plasma', label='Golden search')
ax[2].set_title('Golden search')
ax[2].set_xlabel('x')
ax[2].set_ylabel('f(x)')
ax[2].legend()
```

```python
# Save the plot with high resolution
plt.savefig('search_algorithms.png', dpi=300)

# Display the plot
plt.show()
```

On peut aussi visualiser ces résultats sur la première fonction étudiée, $f = x^2 - 2\sin(x)$: On commence par redéfinir les paramètres de recherche :

```python
# Define the function to minimise and its derivatives
def f(x):
    return x**2 - 2 * np.sin(x)


def df(x):
    return 2 * x - 2 * np.cos(x)


def df2(x):
    return 2 + 2 * np.sin(x)
```

```python
# Define the search conditions
a = 0
b = 2
x0 = 1
epsilon = 1e-5
```

```python
# Apply the search algorithms to the function
x_min_dicho = dichotomie(df, a, b, epsilon)
print('The minimum of the function is at x =', x_min_dicho[-1], 'after', len(x_min_dicho), 'iterations')

x_min_newt = newton(df, df2, x0, epsilon)
print('The minimum of the function is at x =', x_min_newt[-1], 'after', len(x_min_newt), 'iterations')

x_min_gold = golden(f, a, b, epsilon)
print('The minimum of the function is at x =', x_min_gold[-1], 'after', len(x_min_gold), 'iterations')
```

Puis on utilise les fonctions redéfinies précédemment pour afficher les itérations successives :

```python
# Graph the function and the search algorithms
x = np.linspace(a, b, 100)
y = f(x)

fig, ax = plt.subplots(3, 1, figsize=(10, 15))
```

```python
# Dichotomous search
ax[0].plot(x, y, label='f(x)')
ax[0].scatter(x_min_dicho, [f(x) for x in x_min_dicho], c=range(len(x_min_dicho)), cmap='plasma', label='Dichotomous search')
ax[0].set_title('Dichotomous search')
ax[0].set_xlabel('x')
ax[0].set_ylabel('f(x)')
ax[0].legend()
```

```python
# Newton search
ax[1].plot(x, y, label='f(x)')
ax[1].scatter(x_min_newt, [f(x) for x in x_min_newt], c=range(len(x_min_newt)), cmap='plasma', label='Newton search')
ax[1].set_title('Newton search')
ax[1].set_xlabel('x')
ax[1].set_ylabel('f(x)')
ax[1].legend()
```

```python
# Golden search
ax[2].plot(x, y, label='f(x)')
ax[2].scatter(x_min_gold, [f(x) for x in x_min_gold], c=range(len(x_min_gold)), cmap='plasma', label='Golden search')
ax[2].set_title('Golden search')
ax[2].set_xlabel('x')
ax[2].set_ylabel('f(x)')
ax[2].legend()
```

```python
# Save the plot with high resolution
plt.savefig('search_algorithms_2.png', dpi=300)

# Display the plot
plt.show()
```