plans-dexperiences

January 5, 2024

1 TP Plans d'expériences

LAMNAOIR Imane
SROUR Mathieu
ALLOUCH Omar

1.1 Exercice 1

1.1.1 Question 1 :

Nous disposons de 4 variables:

A= L'amplitude du pic de débit $q_{\max}$

B = Le temps de montée du débit t_m

C = La durée totale de la crue d

D = La proportion de la digue rompue p.

Normalement, un plan factoriel complet nécessiterait $2^4 = 16$ expériences pour explorer toutes les combinaisons possibles des niveaux des quatre facteurs. Cependant, en raison de contraintes budgétaires, nous voulons réduire le nombre d'expériences à 8 expériences soit la moitié pour des raisons de coût.

Pour cette raison, on construit un plan factoriel fractionnaire en commençant d'abord par la réalisation d'un plan complet avec $(p-q) = 3$ facteurs A, B et C puis on définit le facteur qui reste par un produit entre les 3 premiers facteurs. Puisque l'interaction ABC est certainement négligeable et du coup on confond la facteur D avec l'interaction $D = A * B * C$.

La matrice du plan est égale donc à avec la première colonne de 1:

$$X = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & 1 & -1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

1.1.2 Question 2 :

[22]:
```
import numpy as np

p = 4
X = np.ones((8, p + 1))

A = np.array([1, 1, 1, 1, -1, -1, -1, -1])
B = np.array([1, 1, -1, -1, 1, 1, -1, -1])
C = np.array([1, -1, 1, -1, 1, -1, 1, -1])
D = A * B * C

X[:, 1:] = np.column_stack((A, B, C, D))

XTX = X.T @ X

print("X:")
print(X)

print("(X'X):")
print(XTX)
```

```
X:
[[ 1.  1.  1.  1.  1.]
 [ 1.  1.  1. -1. -1.]
 [ 1.  1. -1.  1. -1.]
 [ 1.  1. -1. -1.  1.]
 [ 1. -1.  1.  1. -1.]
 [ 1. -1.  1. -1.  1.]
 [ 1. -1. -1.  1.  1.]
 [ 1. -1. -1. -1. -1.]]
(X'X):
[[8. 0. 0. 0. 0.]
 [0. 8. 0. 0. 0.]
 [0. 0. 8. 0. 0.]
 [0. 0. 0. 8. 0.]
 [0. 0. 0. 0. 8.]]
```

Le plan est bien orthogonal par ce que $(X^T X)^{-1}$ est proportionnel à l'identité par un facteur de 8 La clé d'un plan est l'ensemble des relations que l'on exprime sous la forme $1 = ...$ Dans notre cas la clé est égale à $A * B * C * D = 1$. La résolution correspond au nombre inférieur de symboles des éléments de l'alias 1, dans notre cas, on a une résolution de IV. Autrement dit, un effet principal ne peut être confondu avec une interaction "double", mais certaines interactions "doubles" sont confondues entre elles.

1.1.3 Question 3 :

Si on prend en considération tous les interactions, on aura $2^4 = 16$ paramètres à estimer.

Cepandant, si on néglige les interactions de facteurs supérieures ou égales à 3 facteurs on aura $2^4 - \begin{pmatrix} 4 \\ 3 \end{pmatrix} - \begin{pmatrix} 4 \\ 4 \end{pmatrix} = 11$ paramètres à estimer.

1.1.4 Question 4 :

L'équation du modèle est la suivante :

$$Y = \alpha_0 + \alpha_A * A + \alpha_B * B + \alpha_C * C + \alpha_D * D + \alpha_{AC} * AC + \alpha_{AB} * AB + \alpha_{BC} * BC + \epsilon$$

En se basant sur la clé du plan, $ABCD$ représente le générateur d'alias. Du coup on multiplie chacun des effets principale et les interaction avec $ABCD$ afin d'obtenir les alias qui vont nous servir pour calculer les 11 pramètres et on trouve les relations suivantes: $A = BCD \quad B = ACD \quad C = ABD \quad AB = CD \quad AC = BD \quad BC = AD$

Sauf que dans notre cas on va suffir juste des interaction de facteurs inférieur strictement à 3 facteurs, et du coup on trouve les relations suivantes:

$$\begin{aligned} \alpha_0 &= \beta_0 \\ \alpha_A &= \beta_A \\ \alpha_B &= \beta_B \\ \alpha_C &= \beta_C \\ \alpha_D &= \beta_D \\ \alpha_{AC} &= \beta_{AC} + \beta_{BD} \\ \alpha_{BC} &= \beta_{BC} + \beta_{AD} \\ \alpha_{AC} &= \beta_{AC} + \beta_{BD} \end{aligned}$$

1.1.5 Question 5 :

Nous débutons en extrayant les valeurs de la réponse (Y) pour 8 simulations. Ensuite, nous calculons les estimations pour les 8 contrastes en utilisant le code suivant et les relations vues en cours:

[23]:
```
import numpy as np
import pandas as pd

def fonction_test(X):
    X = -1 + 6 * (X + 1) / 2
    X[0] = 5 * X[0]
    X[1] = -(X[1] / 5 - 1)

    y1 = X[2] * np.exp(-(10 * X[1]) ** 2 / (60 * X[2] ** 2 + 1))
    y2 = (X[1] + X[3]) * np.exp(X[2] / 500)
    y3 = (X[2] * (X[0] - 2)) * np.exp(-(X[3]) ** 2 / (100 * X[2] ** 2))

    return y1 + y2 + y3 + X[0] * (X[3] / 10)

N = 8
```

```
Y = np.zeros(N)

for i in range(N):
    Y[i] = fonction_test(X[i, 1:5])  # Index 1:5 pour correspondre à X[i, 2:5] en R

X_t = np.column_stack([X, X[:, 1] * X[:, 3], X[:, 2] * X[:, 3], X[:, 1] * X[:, 2]])

alpha = (1/8) * X_t.T @ Y

contrast_names = ["intercept", "A", "B", "C", "D", "AC", "BC", "AB"]
alpha_df = pd.DataFrame(alpha, index=contrast_names, columns=["Estimate"])
print(alpha_df)
```

```
            Estimate
intercept  23.081329
A          33.672746
B          -1.685730
C          26.109105
D           6.339074
AC         43.939126
BC          5.450260
AB         -0.499928
```

1.2 Exercice 2

Import libraries

[24]:
```
import numpy as np
import timeit

from scipy.stats.qmc import Halton, Sobol, scale
```

Define the density function $f_X(x)$, and the function $f(x)$, and a function to test if a point is in the region A

[25]:
```
def f(x):
    if x <= -1:
        return 0
    elif -1 < x <= 0:
        return x + 1
    elif 0 < x <= 1:
        return -x + 1
    else:
        return 0
```

```
def f_x(x):
    return f(x[0]) * f(x[1]) * f(x[2]) * 8  # 8 is the scaling factor to obtain␣
 ↪a valid pdf


def A(x):
    return x[0] > 0 and x[1] > 0.9 and x[2] < -0.4
```

As per the comment above, we multiply the density value of a point x by *8* to account for the volume of the cube in $\mathbb{R}^3$, if we don't do that the integration of the density function over the domain gives *0.125* instead of *1*, which means it's no longer a *pdf*.

We define 2 functions: - `quasi_monte_carlo` generates `n` points following a low discrepancy `sequence` - `repeated_quasi_monte_carlo` repeats the `quasi_monte_carlo` function and returns the mean of the result to reduce the impact of random variability

[26]:
```
def quasi_monte_carlo(n, sequence):
    l_bounds = [-1, -1, -1]
    u_bounds = [1, 1, 1]
    X = sequence(3, scramble=True).random(n)
    X = scale(X, l_bounds, u_bounds)
    A_points = X[np.apply_along_axis(A, 1, X)]
    Y = np.apply_along_axis(f_x, 1, A_points)
    return np.sum(Y) / n


def repeated_quasi_monte_carlo(n, sequence, m):
    return np.mean([quasi_monte_carlo(n, sequence) for _ in range(m)])
```

Run the functions and see the results

[27]:
```
n = 50000

print("###################### Single run ######################")
start_time = timeit.default_timer()
print("Proportion of points in A for Sobol:  ", quasi_monte_carlo(n, Sobol))
print(f"Done in: {timeit.default_timer() - start_time}s")
start_time = timeit.default_timer()
print("Proportion of points in A for Halton: ", quasi_monte_carlo(n, Halton))
print(f"Done in: {timeit.default_timer() - start_time}s")
print("########################################################")

print()

print("###################### Repeated run ######################")
m = 10
start_time = timeit.default_timer()
```

```
print("Proportion of points in A for Sobol:  ", repeated_quasi_monte_carlo(n,␣
 ↪Sobol, m))
print(f"Done in: {timeit.default_timer() - start_time}s")
start_time = timeit.default_timer()
print("Proportion of points in A for Halton: ", repeated_quasi_monte_carlo(n,␣
 ↪Halton, m))
print(f"Done in: {timeit.default_timer() - start_time}s")
print("############################################################")
```

```
###################### Single run ######################
Proportion of points in A for Sobol:   0.000447103139955776
Done in: 0.07205495900052483s
Proportion of points in A for Halton:  0.0004416805242572016
Done in: 0.11663967200001935s
########################################################

###################### Repeated run ######################
```

```
/home/omar/miniconda3/lib/python3.11/site-packages/scipy/stats/_qmc.py:804:
UserWarning: The balance properties of Sobol' points require n to be a power of
2.
  sample = self._random(n, workers=workers)
```

```
Proportion of points in A for Sobol:   0.000451845633658298
Done in: 0.6113522830019065s
Proportion of points in A for Halton:  0.0004525976140567989
Done in: 1.1047922739999194s
############################################################
```

Comparison with the analytical value

$$\mathbb{P}(X \in A) = \int_0^1 \int_{0.9}^1 \int_{-1}^{-0.4} (x_1 + 1) \cdot (x_2 + 1) \cdot (-x_3 + 1)\, dx_1\, dx_2\, dx_3 = 4.5 \times 10^{-4}$$

1.2.1 Conclusion

The solve the problem we wrote the probability as an integral, then estimated the integral using quasi-Monte Carlo (Monte Carlo using low discrepancy sequences)
Since the problem is not too hard to solve analytically, we did the calculations and compared the analytical solution to that of the code, and to no surprise the results were in agreement.

Result-wise there are almost no differences, but performance-wise it seems the `Sobol` sequence is slightly faster, and this difference in execution time accumulated quickly to the point where `Halton` almost took twice the time `Sobol` takes for only 10 runs.

In conclusion, the use of low discrepancy sequences, such as Sobol and Halton sequences, provides a powerful and efficient approach for solving numerical problems, particularly in the context of Monte Carlo simulations and quasi-Monte Carlo integration. These sequences exhibit improved coverage of the sample space compared to traditional random sequences, leading to a more uniform exploration of the domain.

The advantages of low discrepancy sequences include their ability to reduce variance and accelerate

convergence rates, especially in higher dimensions. By systematically distributing points across the space, these sequences help achieve a more accurate and stable estimation of integrals, probabilities, and other numerical quantities.

Moreover, the deterministic nature of low discrepancy sequences ensures reproducibility, a crucial aspect in scientific computing and experimentation. This feature allows for consistent and comparable results across multiple runs, facilitating the analysis of algorithmic performance.

Despite their advantages, it is important to note that the effectiveness of low discrepancy sequences may depend on the specific characteristics of the problem at hand. In certain scenarios, the choice of a particular low discrepancy sequence and the understanding of its properties can significantly impact the success of the numerical solution.