

Study case:
**The optimal repartition of customs
officers**

Abstract

This paper is a report of a fictional study case given as homework I have done within the scope of the minor of *Analysis and Prescriptive Analysis* in the Ecole Centrale de Lille in 2024.

It consists in finding the optimal way to organize the customs forces at the terminal of an airport in order to minimize the total number of employees, taking into consideration the planning of the arrivals and the time passengers from a given destination spend at the custom office. The first part is about explaining the methodology and the model used. The second is about finding the optimal repartition if every officer keeps the same the schedule each day of work with two consecutive days off. The third part changes this hypothesis: officer can work at different schedules but cannot do one schedule after another. The fourth consists in testing the hypothesis for employees to work overtime.

All this work has been done using Python for the construction of objects and IBM CPLEX for the resolution of problems, without any IA models.

Part 1

This part aims to explain the approach of the problem chosen and which hypotheses it relies on. The available data are: the schedule of arrivals with the exact hour at the airport and the number of passengers. Also, for each destination the average time for a passenger to pass the custom officer and the probability for an incident to occur and the time it takes to be solved.

The idea of solving this problem with a preliminary approach is as follows:

- For each hour of each day, calculate the number of employees required to meet this waiting time condition.
- Determine a constraint of waiting time
- Establish two work patterns (one for the morning and another for the afternoon).
- Solve a covering problem that considers constraints such as lunch breaks and workdays.

I started by calculating the average waiting time for each flight, considering the probability of issues and the average duration of a problematic passage (*Code 1.1*). Afterward, I decided not to count people but the time they take : If one person spends 4 minutes passing the border, then it is counted as 4 individuals. By doing so, I could mix passengers from different types of flights without any problem. That's why in the following text, I will refer to equivalent-person. I'll spare you all the conversions required to transition from Excel data to a Python list.

```
def avtime():
    t=[4.3,3.6,3.6,2.0]
    p=[80,65,7.5,7.5]
    avt=[]
    for i in range(len(p)):
        avt.append((p[i]*t[i]+(100-p[i]))/100)
    return avt
```

Code 1.1 : The list of average time in minute for each origin

With this data I can, for each day, set up two lists:

- A minute list representing the number of equivalent-person in the queue minute by minute noted *arv* (code 1.2)
- an hour list which represent the number of equivalent-person there are in the queue hour after hour noted as a *paquet* (code 1.3)

```
def arv(D):          #D is the list minute by minute of the number of people arrived
    Arrivée=[]
    cll=convllegadas(D) #traduct the excel table in a list of number of people arrived minute after minute
    S=convllegadas(spe()) #traduct the specificities of the flight
    avt=avtime()
    for i in range(len(time)):
        Arrivée.append(round(cll[i]*avt[S[i]]))
    return Arrivée
```

Code 1.2 : number of equivalent-person in the queue minute by minute

```
def paquet(D):
    P = [0 for i in range(13)]
    H = [i+8 for i in range(13)]
    Arrivée=arv(D)
    for x in time:
        y = (x + 9*60 + 51) // 60
        P[y-9] += Arrivée[x]
    return P
```

Code 1.3 : the number of equivalent-person there are in the queue hour after hour.

During all the process I use a main hypothesis of the *best-worst-waiting-time* which is the maximum amount of time a passenger could wait if the queue is empty at the beginning of the hour of arrival or, to say it in another way, in one hour should pass the same number of passengers than the number of passengers considered as arrived at the beginning of the hour. All the calculations will be done under these main hypotheses.

Since the number of employees could only change at the top of the hour (although it might be interesting to see what happens if it could change every 30 minutes), I grouped arrivals within the same hour and treated them as if they had all arrived at the beginning of the hour. This allowed me to determine how many employees should work to facilitate the passage of all arrivals within a 60-minute timeframe

Let pose:

- A the number of equivalent-person arrived
- N the number of workers
- Q the number of persons in the *queue*

The number of equivalent-person in the queue at the beginning of $h + 1$ is :

$$Q[h + 1] = Q[H] + A[h + 1]$$

To be sure that all of these equivalent-person pass in the hour, the number of worker at the hour $h + 1$ is :

$$N[h + 1] = Q[h+1] / 60$$

So when $N[h + 1]$ employees work there is : $Q[H] + A[h + 1]$ at the beginning of the hour $h + 1$ and 0 at the end of the day. The translation in python is done in *code 1.4*.

```
def NU(D):
    #D is the list minute by minute of the number of people arrived
    Q=[0]          #List of equivalent-person empty at the beginning but filled with 0 to avoid any pb.
    N=[0]          #List of worker empty at the beginning but filled with 0 to avoid any pb.
    LL=paquet(D)   # LL is the hour list of the day D
    for i in range(len(LL)):
        Q.append(Q[i]+LL[i]-60*N[i])
        if Q[i+1]<=0:      #the number of equivalent-person must be positive
            Q[i+1]=0
        N.append(int(Q[i+1]//60+1))
    return N[1:] #return the number of worker by hour, without the 0-component
```

Code 1.4 : The calculation of employees needed for each hour of a given day

Thanks to this algorithm, the number of employee needed for each hour of each days are resume in the Figure 1.1:

```
Monday = [1, 1, 6, 2, 2, 6, 2, 1, 7, 1, 1, 1, 1]
Tuesday = [5, 5, 1, 5, 1, 6, 1, 1, 4, 1, 1, 1, 1]
Wednesday = [5, 5, 1, 7, 2, 7, 1, 1, 3, 1, 1, 1, 1]
Thursday = [6, 6, 1, 6, 2, 6, 1, 2, 5, 1, 1, 1, 1]
Friday = [5, 5, 2, 8, 2, 5, 2, 1, 5, 1, 1, 1, 1]
Saturday = [8, 8, 11, 11, 5, 15, 1, 8, 3, 1, 1, 1, 1]
Sunday = [7, 7, 5, 11, 2, 7, 1, 4, 7, 1, 1, 1, 1]
```

Figure 1.1 : the need of officers for each hour of each day

This first approach, and the ambitious hypotheses it relies on, allows us to appreciate the effect on the number of people waiting in the queue. All tests have been done using the real schedules of arrivals and the number of calculated numbers of officers based on precedents hypotheses. Code 1.5 explains how (combined with precedent codes) the number of waiting equivalent-people in the queue for each minute of the day and so, for each day of the week. Figure 1.2 illustrates it for all days, and the second for Saturday.

The consideration of *equivalent-people* in the queue allows the construction of a total demand of officers and the calculation of the minimal number of officers to reach this demand for each hour under the *best-worst-waiting-time hypothesis*. Based on this demand, it is possible to find an upper-bound of the waiting time (not done in this paper but visible in Figure 1.1) and to begin to find the optimal repartition of customs forces to minimize the total number of officers.

```
def QfT(D):          #the quantity of equivalent-person in the queue
    N=NU(D)
    Q=[0]
    LL=arv(D)
    for i in range(len(LL)):
        Q.append(Q[i]+LL[i]-N[i//60])
        if Q[i+1]<0:
            Q[i+1]=0
    return Q[1:]
plt.plot(time,Q[1:])
plt.plot(time, arv(D))
plt.show()

def Etude():
    Qsem=[]
    for x in Semaine:
        Qsem.append(QfT(x))
    for Q in Qsem:

        plt.plot(time,Q)
    plt.show()
```

Code 1.5 : The construction of the number of equivalent people in the queue for each day of the week

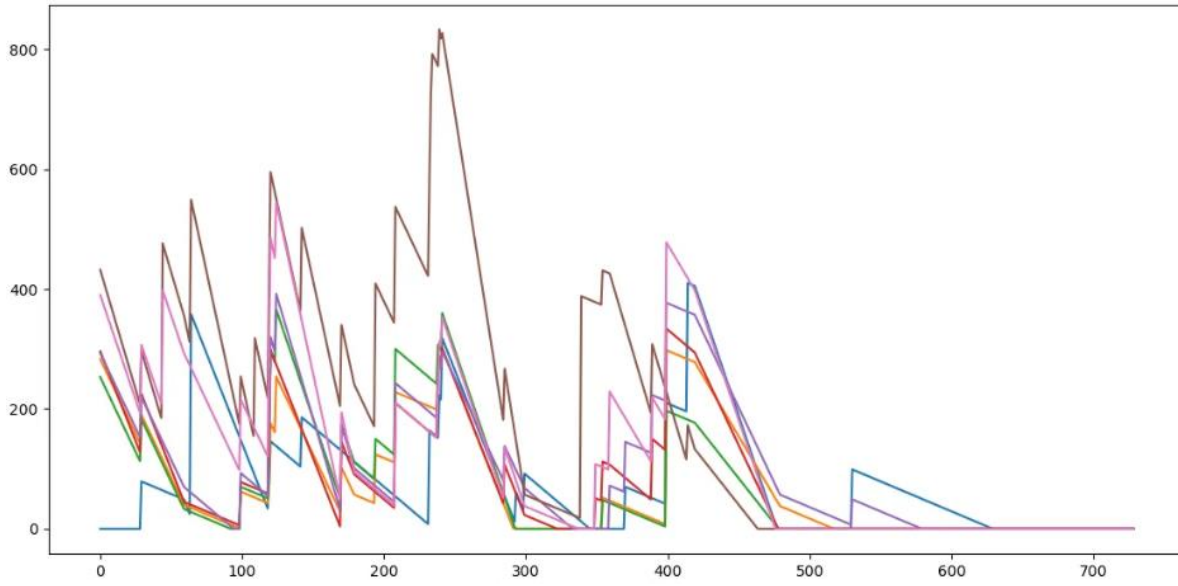


Figure 1.2 : number of equivalent-people in the queue for each day

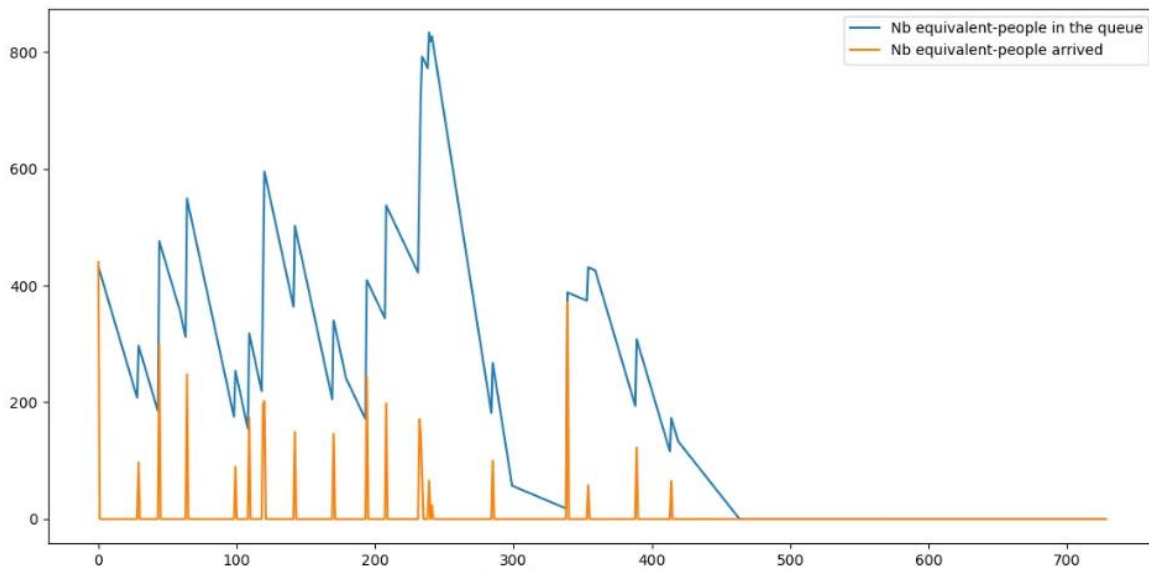


Figure 1.3 : number of equivalent-people in the queue for the Saturday

Part 2

In this part, I shall try to optimize the repartition of the custom officers to answer to these needs. There is a need for officers between 8 a.m. to 7 p.m which. Since all passengers are processed by 7 p.m.,it's not relevant to keep the airport open after that time. Therefore, I maintained the same work patterns for both weekdays and weekends.

There are 3 possible work patterns:

- From 2 p.m. to 10:30 p.m. with a break between 5 p.m. and 6 p.m.
- From 8 a.m. to 4:30 p.m. with a break between 12 p.m. and 1 p.m.
- From 10:30 a.m. to 7 p.m. with a break between 2 p.m. and 3 p.m.

To find the best arrangement for a given day D, an IBM CPLEX algorithm (*code 2.1*) is used with two objects (an illustration in *code 2.2*):

- A matrix T which represents the hour of work of the three patterns (1 if it is a work hour, else 0)
- The variable X, a vector whose component i represents the number of employees who work with the pattern i.

The details of needs for each pattern of each day are showed in the *Figure 2.1*.

```

int nbHourDay = 15;
range rangehour = 1..nbHourDay;
range rangetrois = 1..3;

int N[rangehour] = ...; // Nombre d'employés nécessaires
int T[rangetrois][rangehour] = ...; // Temps de travail pour chaque modèle

dvar int+ X[rangetrois]; // Nombre d'employés pour chaque modèle
dexpr float z = sum(k in rangetrois) X[k];

minimize z;

subject to {
  forall(j in rangehour) {
    (sum(k in rangetrois) X[k] * T[k][j]) >= N[j];
  }
}

```

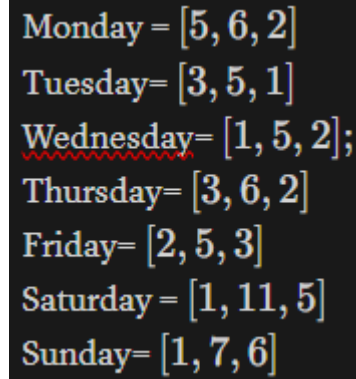
Code 2.1 : optimization of algorithm for a given Day

```

N=[1, 1, 6, 2, 2, 6, 2, 1, 7, 1, 1, 1, 1, 1, 1]; //liste of the needed officer for a given day
T=[
[0,0,0,0,0,0,1,1,1,0,1,1,1,1,1], // when the employee is kept for 30min by hour, it is not counted
[1,1,1,1,0,1,1,1,0,0,0,0,0,0,0],
[0,0,0,1,1,1,0,1,1,1,1,0,0,0,0]
];

```

Code 2.2 : Example of data



```

Monday = [5, 6, 2]
Tuesday = [3, 5, 1]
Wednesday = [1, 5, 2];
Thursday = [3, 6, 2]
Friday = [2, 5, 3]
Saturday = [1, 11, 5]
Sunday = [1, 7, 6]

```

Figure 2.1 : the need for each pattern of each day

The last step consists of finding the schedule repartition that minimizes the number of total employees meeting the demand for the three patterns for each day. Because a schedule is defined as **5** consecutive days of work plus **2** days off, there are **7** possibilities of scheduling which are summarized in the matrix **E** (*Code 2.3*). The demand is given by the transposed matrix **X** built on Figure 2.1. Then, the *Repartition Algorithm* of the *Code 2.4* returns a solution highlighting the need for the schedule **i** the number of officers needed for the pattern **j**. The main issue is to ensure that for a given work pattern the total number of employees is less than or equal to the sum of the number of employees for each schedule. However, our goal is to minimize the total number of employees.

```

6 E=[
7 [1, 1, 1, 1, 1, 0, 0],
8 [0, 1, 1, 1, 1, 1, 0],
9 [0, 0, 1, 1, 1, 1, 1],
10 [1, 0, 0, 1, 1, 1, 1],
11 [1, 1, 0, 0, 1, 1, 1],
12 [1, 1, 1, 0, 0, 1, 1],
13 [1, 1, 1, 1, 0, 0, 1],
14 ];
15
16 X=[
17 [5, 3, 1, 3, 2, 1, 1],
18 [6, 5, 5, 6, 5, 11, 7],
19 [2, 1, 2, 2, 3, 5, 6]
20 ];

```

Code 2.3 : Data of the Repartition Algorithm

```

int nbday = 7;
range rangeday = 1..nbday;
range rangetrois = 1..3;

int X[rangetrois][rangeday] = ...; // nb of needed employees
int E[rangeday][rangeday] = ...; // patterns of work

dvar int+ Y[rangeday][rangetrois]; //number
dexpr float z = sum(l in rangeday, k in rangetrois) Y[l][k];

minimize z;

subject to {
  forall(i in rangetrois){
    forall(j in rangeday){
      sum(k in rangeday) Y[k][i] * E[k][j] >= X[i][j];
    }
  }
}

```

Code 2.4 : The Repartition Algorithm

The solution given by this algorithm is the number of workers for each schedule and each pattern of work:

$$Y = \begin{bmatrix} [4 & 0 & 0] \\ [0 & 4 & 0] \\ [0 & 1 & 4] \\ [1 & 5 & 1] \\ [0 & 1 & 1] \\ [0 & 0 & 0] \\ [0 & 0 & 0] \end{bmatrix};$$

For example, we need **4** employees who work with the 1st pattern with schedule **1** and 5 who work on the 2nd pattern with schedule **4**.

Summing all the values of **Y**, it comes up that the total number of employees needed is **22**. With this repartition, we ensure that the conditions of demand are met under the hypothesis of 35 hours, 5 consecutive days per week, with a fixed pattern. It must be noted that respect for these hypotheses implies that this number of **22** officers is in fact an upper bound of the minimal number this article tries to reach.

In this condition (number of employees, schedules...) the queue degrowths faster than the limit decided at the beginning of the problem, as the test highlights it in the *Figure 2.2*.

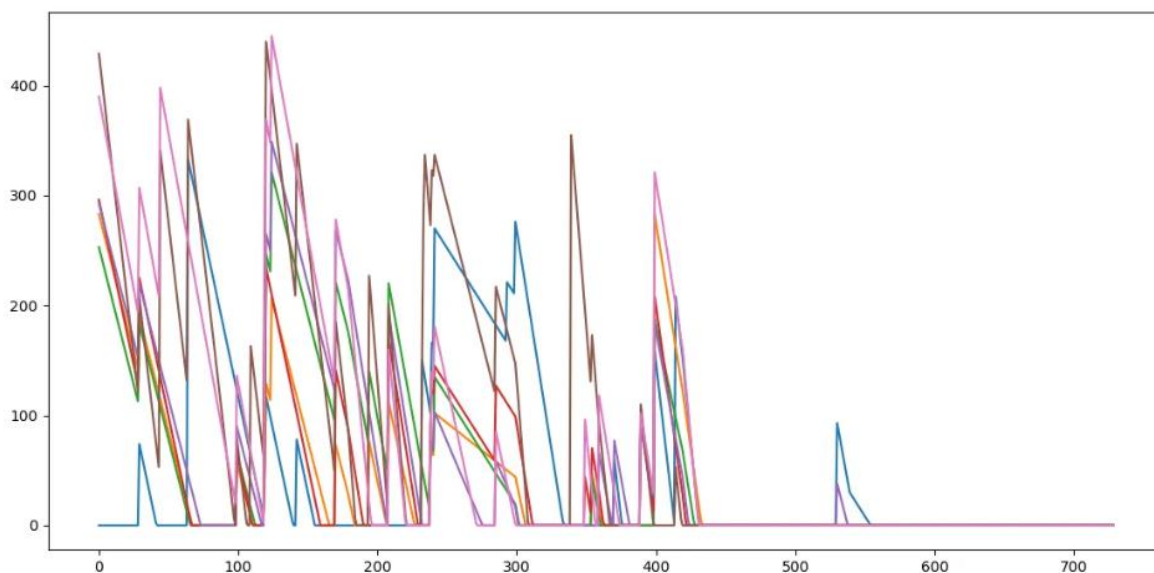


Figure 2.2 : The test of the solution given in Part 1

It's important to note that the algorithm counts in half-hour increments, so in some cases, the curves may decrease slightly faster than what is visible in this graph.

Part 3

In this part, an hypothesis of flexibility is suppose on the contrary with *Part 2* : if employees can change their schedule each day, we can simply consider the total employee needs for each day and determine the number of employees required for each schedule accordingly with *Code 3.1*. This way, there will be enough employees for each day, and they can organize themselves into different patterns depending on the demand. So, the Y variable could shrink to become just a vector.

```

int nbday = 7;
range rangeday = 1..nbday;

int X[rangeday] = ...; // nb of needed employees
int E[rangeday][rangeday] = ...; // patterns of work

dvar int+ Y[rangeday]; //number
dexpr float z = sum(l in rangeday) Y[l];

minimize z;

subject to {
  forall(j in rangeday){
    sum(k in rangeday) Y[k] * E[k][j] >= X[j];
  }
}

```

Code 3.1 : The Repartition Algorithm under the flexible hypothesis

So $Y[i]$ workers are needed with the schedule $E[i]$ must satisfy the demand per day $X[i]$. The whole solution is returns with the matrix :

$$Y = [0, 3, 1, 7, 2, 4, 0]$$

This means that 3 employees should work according to schedule 2. Doing so, we find

that the company just needs **17** employees.

This process has caused a significant degrowth of the total number of employees dropping from **24** to **17**. That explains why the *hypothesis of flexibility* is retained.

Part 4

If we consider now that employees can work overtime, we can extend the schedule to minimize the total amount of work. In order to minimize the number of overtimes, I decided that they can't overpass half-hour by day. The total amount of work hours does not exceed **40h** by week.

I take again the algorithm of the Part 2 (*Code 4.1*), adding two patterns with overtimes possibilities (*Code 4.2*).

We find for each day :

Monday = [1, 0, 2, 6, 0]

Tuesday=[1, 0, 1, 5, 0]

Wednesday=[1, 5, 2, 0, 0]

Thursday=[1, 0, 2, 6, 0]

Friday=[1, 4, 3, 1, 0]

Saturday =[1, 11, 5, 0, 0]

Sunday=[1, 5, 4, 2, 0]

With this new demand, the Repartition Algorithm returns the Matrix **Y** :

```
[0 5 1 0 0]
[1 0 2 0 0]
[0 0 2 1 0]
[0 6 0 0 0]
[0 0 0 0 0]
[0 0 0 5 0];
```

So, it is possible to know the total number of employees needed for each schedule, normal or overtime. We just must adapt to these two new possibilities of overtimes patterns of work: **24**. This too high number is due to the pattern of work : the total of consecutive hours is limited by law to 9hours and 40h by week which reduce strongly the potential gain of efficiency they could have offer. Of course, it is a upper-bound of the minimal number and deeper exploration half-hour by half hour may reduce it. However, the flexibility hypothesis is more efficient respecting more decent work hours for employees.

Talking about it, what happens if the problem is placed in Part 3 ?

Using the same Repartition Algorithm of Part 3, it returns:

$$Y = [0, 5, 3, 7, 2, 0, 0].$$

This means that 5 employees should work according to schedule **2**. Doing so, we find that the company just needs: **17** employees.

It appears that if the airport needs less employees by day to function, the conditions of work (schedule & pattern) do not allow a more efficient solution than with the original condition.

```

int nbHourDay = 15;
range rangehour = 1..nbHourDay;
range rangecinq = 1..5;

int N[rangehour] = ...; // Nombre d'employés nécessaires
int T[rangecinq][rangehour] = ...; // Temps de travail pour chaque modèle

dvar int+ X[rangecinq]; // Nombre d'employés pour chaque modèle
dexpr float z = sum(k in rangecinq) X[k];

minimize z;

subject to {
  forall(j in rangehour) {
    (sum(k in rangecinq) X[k] * T[k][j] ) >= N[j];
  }
}

```

Code 4.1 : adaptation of repartition algorithm of the Part 2

```

//data :
N=[7, 7, 5, 11, 2, 7, 1, 4, 7, 1, 1, 1, 1, 1, 1]; //needs of the day
T=[
[0,0,0,0,0,0,1,1,1,0,1,1,1,1,1], // when the employee is kept for 30min by hour, it is not counted
[1,1,1,1,0,1,1,1,0,0,0,0,0,0,0],
[0,0,0,1,1,1,0,1,1,1,1,0,0,0,0],
[1,1,1,1,0,1,1,1,1,0,0,0,0,0,0],
[0,0,0,1,1,1,0,1,1,1,1,1,1,0,0,0]
];

```

Code 4.2 : data used for the schedule with overtimes

Conclusion

Enlightened by the different models and the tests done, the main hypothesis which minimizes the total number of officers is the flexibility hypothesis. This has the double-advantage to respect all precedent hypothesis (number of hours, forms of schedules...) and to be able to absorb the absence of one of them without too many consequences because it is an upper bound of the minimize number of officers to reach the demand of arrival under *best-worst-waiting-time hypothesis*.

For a deeper analysis, it could be interesting to explore the impact of a change in the patterns or schedules, a minimization of the maximal waiting time for passengers and a calculation of the impact of the absence of $p\%$ of officers on this waiting time.