



ÉCOLE NATIONALE D'INGÉNIEURS DE TUNIS

Département Génie industrielle

**Projet Probabilités**

---

# Bin Packing

---

*Elaboré par :*

Mohamed Fyras TELMINI

Rihab AZZABI

*Encadré par :*

M. Azmi MAKHLOUF

1<sup>ère</sup> Année MIndS

Année universitaire : 2019/2020

# Table des matières

<b>1</b>	<b>Inroduction</b>	<b>1</b>
<b>2</b>	<b>Les algorithmes</b>	<b>2</b>
<b>3</b>	<b>Analyse de la performance</b>	<b>5</b>
<b>4</b>	<b>Etude de la performance des algorithmes pour un problème de gestion d'emplois de temps</b>	<b>19</b>
4.1	Introduction . . . . .	19
4.2	Variables aléatoires . . . . .	19
4.2.1	Loi de la variable aléatoire de la durée d'une tâche . . . . .	19
4.2.2	Variable aléatoire "nombre de journées pas chargés" . . . . .	20
4.3	Simulation . . . . .	20
4.4	Interprétation des Histogrammes . . . . .	24
4.4.1	Loi de $X_n$ . . . . .	24
4.4.2	Loi de $W_n$ . . . . .	26
4.4.3	Loi de $Y_n$ . . . . .	29
4.5	Conclusion et discussion . . . . .	31
<b>5</b>	<b>Simulation de l'occupation des lits de réanimation en tunisie</b>	<b>32</b>
5.1	introduction . . . . .	32
5.2	Liste des nouveaux cas journaliers du cov-19 en tunisie . . . . .	32
5.2.1	Modèle SEIR . . . . .	33
5.3	Liste des durées d'occupation . . . . .	35
5.3.1	Nombre de nouveaux cas graves par journée . . . . .	35
5.3.2	Vecteur durée d'occupation . . . . .	35
5.4	Boucle principale . . . . .	36
5.5	Résultats et Conclusion . . . . .	37
<b>6</b>	<b>Conclusion</b>	<b>39</b>



# 1. Introduction

Dans le cadre de ce projet de probabilités, nous souhaitons reconnaître les lois de probabilités de certaines variables aléatoires par usage de la méthode de Monte-Carlo. Notre étude traite des variables aléatoires issues d'algorithmes de Bin-packing (remplissage d'espace unidimensionnel), nous allons traiter 4 de ces algorithmes. chaque algorithme reçoit une liste d'objets dont les longueurs sont des variables aléatoires qui suivent des lois prédéfinies, nous souhaitons étudier d'autres variables aléatoires tels que le nombre de Boites remplies et l'espace perdu pour chaque algorithme, et cela pour plusieurs lois pour les longueurs d'objets.

Dans la partie un on explicitera le fonctionnement de chaque algorithme. Dans la partie 2 on étudiera par la méthode de Monte-Carlo les lois des variables aléatoires qui représentent le nombre de boites et l'espace perdu pour des listes d'objets dont les longueurs suivent les lois uniformes et géométriques. Dans la partie 3 on essaiera d'implémenter cette étude dans un cadre concret de gestion d'emplois de temps. La partie 4 est un peu hors-sujet mais elle concerne l'implémentation d'un algorithme de bin packing pour un problème de gestion d'espace à la fois déterministe et probabiliste.

## 2. Les algorithmes

Le problème de bin packing consiste à déterminer le nombre minimal de bins nécessaires pour ranger un ensemble d'objets or ce problème est *NP-difficile*, d'où on essaie d'approcher la solution optimale par des algorithmes simples.

Dans tous les algorithmes qui suivent, nous allons procéder à une méthode de remplissage spécifiée, où chaque bloc du Bins (contenant  $n$  blocs) nous donne *bloc.full* : la partie remplie, et *bloc.free* : la partie vide.

La figure 2.1 illustre le principe du remplissage.

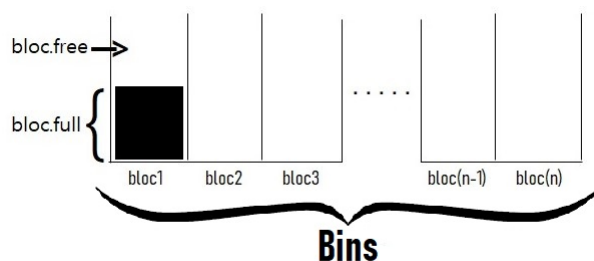


FIGURE 2.1 – Principe du remplissage

- **Algorithme Next-Fit** : le principe de cet algorithme consiste à remplir l'objet courant dans le bloc courant si il peut le contenir, sinon un nouveau bloc sera créé.

En exécutant le code du Next-Fit pour les tailles = (0.2 ; 0.5 ; 0.4 ; 0.7 ; 0.1 ; 0.3 ; 0.8) nous obtenons le résultat présenté par la figure 2.2

Fields	full	free
1	[0.2000 0.5000]	0.3000
2	0.4000	0.6000
3	[0.7000 0.1000]	0.2000
4	0.3000	0.7000
5	0.8000	0.2000

FIGURE 2.2 – Bins remplis par la méthode Next-Fit

- **Algorithme First-Fit** : le principe du First-Fit se présente par parcourir tous les bins et l'objet courant se place dans le premier bloc qui peut le contenir, sinon dans un nouveau bloc.

L'exécution de cet algorithme pour les tailles = (0.2; 0.5; 0.4; 0.7; 0.1; 0.3; 0.8) nous donne le résultat présenté ci-dessous (figure 2.3).

Fields	full	free
1	[0.2000 0.5000 0.1000]	0.2000
2	[0.4000 0.3000]	0.3000
3	0.7000	0.3000
4	0.8000	0.2000

FIGURE 2.3 – Bins remplis par la méthode First-Fit

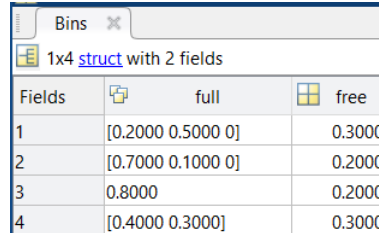
- **Algorithme Best-Fit** : l'algorithme suivant consiste à chercher le bloc le mieux rempli c'est à dire la valeur minimale des textitbloc.free et on range l'objet courant dans ce bloc, sinon un nouveau bin sera créé.

Pour les tailles = (0.2; 0.5; 0.4; 0.7; 0.1; 0.3; 0.8), l'exécution du code Best-Fit donne le résultat suivant (figure 2.4)

Fields	full	free
1	[0.2000 0.5000 0.1000]	0.2000
2	0.4000	0.6000
3	[0.7000 0.3000]	0
4	0.8000	0.2000

FIGURE 2.4 – Bins remplis par la méthode Best-Fit

- **Algorithme Skim-Up** : En se basant sur l'algorithme du Next-Fit, on obtient des bins avec débordement d'un objet. Ainsi et toujours par Next-Fit, on enlève ces objets pour les mettre dans des nouveaux blocs et on itère. Le résultat obtenu lors de l'exécution du code skim-up pour les tailles = (0.2 ; 0.5 ; 0.4 ; 0.7 ; 0.1 ; 0.3 ; 0.8) est présenté ci-dessous (figure 2.5).



Fields	full	free
1	[0.2000 0.5000 0]	0.3000
2	[0.7000 0.1000 0]	0.2000
3	0.8000	0.2000
4	[0.4000 0.3000]	0.3000

FIGURE 2.5 – Bins remplis par la méthode Skim-Up

Après avoir réalisé ces algorithmes approximatifs, on remarque que leurs solutions peuvent ne pas être utiles, ainsi on procède à une analyse de la performance des algorithmes.

### 3. Analyse de la performance

Cette analyse se base sur l'étude (probabiliste) de cas moyen, dont le principe fait appel à la *méthode Monte-carlo* qui vise à approcher toute quantité moyenne par la moyenne empirique sur un grand nombre de simulations et pour garantir la qualité de l'approximation, on effectue une mesure des intervalles de confiance.

L'objectif de cette partie est d'étudier la performance des algorithmes suivant : Next-fit (NF), First-fit (FF), Best-Fit (BF), Skim-Up (SU).

Ainsi pour tout algorithme  $A \in (NF, FF, BF, SU)$ , on note :

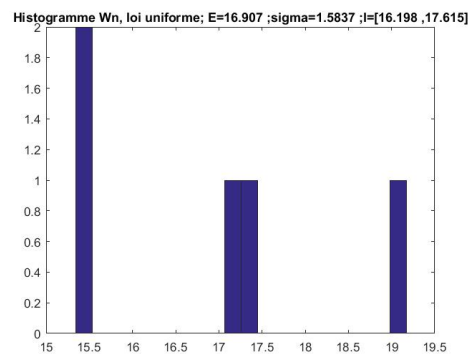
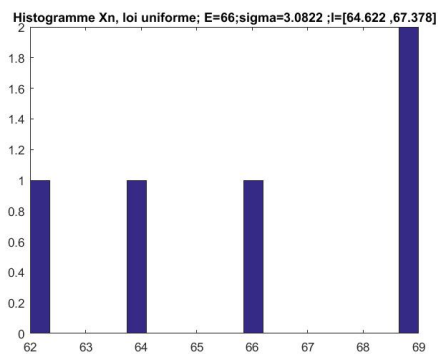
- $X_n^A$  : le nombre de blocs obtenu par A. (où  $n$  correspond au nombre d'objets)
- $W_n^A$  : l'espace non utilisé des bins.

Par la méthode de Monte-Carlo et en tenant compte d'un grand nombre  $M$  de simulations de loi uniforme sur  $[0,1]$  et ensuite de loi exponentielle, on va étudier à chaque algorithme A, en fonction de  $n$  : la moyenne, la variance et les histogrammes de  $X_n^A$  et  $W_n^A$ , avec des intervalles de confiance associés.

#### — L'étude du Next-Fit :

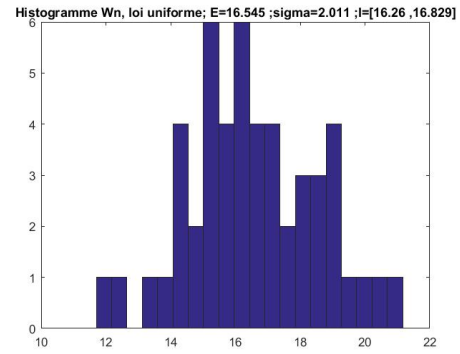
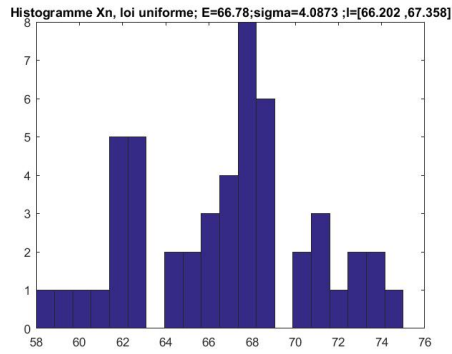
En faisant varier  $n$  et en utilisant la loi uniforme sous Matlab, nous obtenons les résultats suivant :

— Pour  $n=5$  :

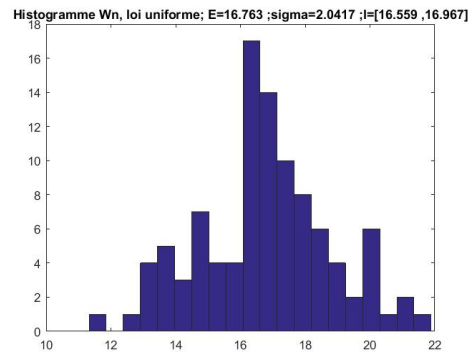
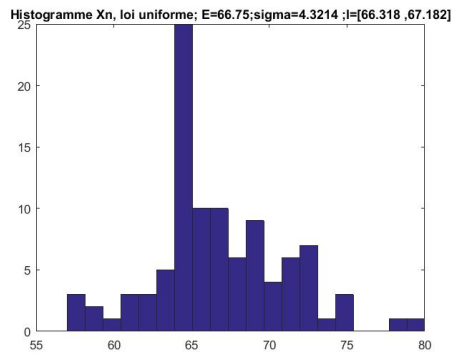


— Pour  $n=50$  :



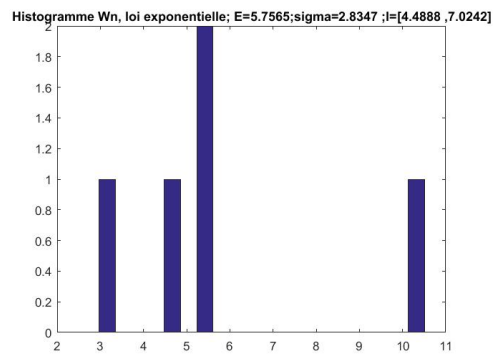
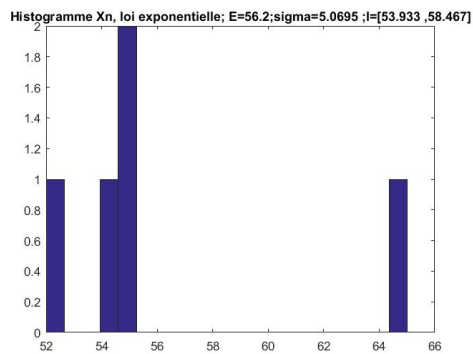


— Pour  $n=100$  :



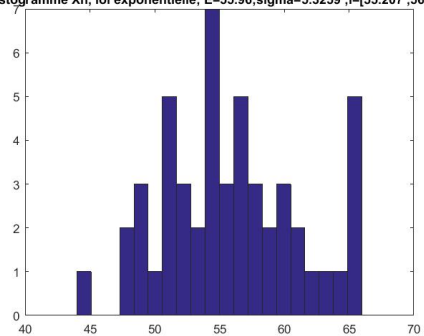
Toujours en variant  $n$ , on utilise maintenant la loi exponentielle.

— Pour  $n=5$  :

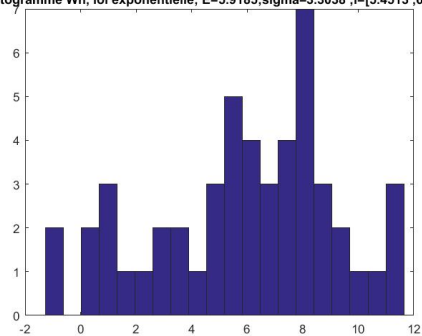


— Pour  $n=50$  :

Histogramme  $X_n$ , loi exponentielle;  $E=55.96$ ;  $\sigma=5.3259$ ;  $I=[55.207, 56.713]$

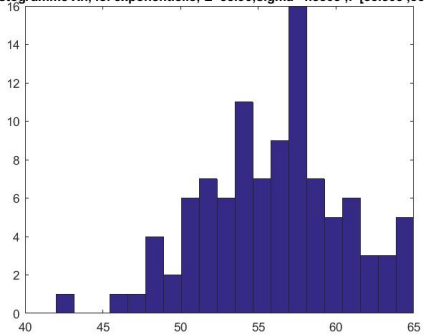


Histogramme  $W_n$ , loi exponentielle;  $E=5.9185$ ;  $\sigma=3.3038$ ;  $I=[5.4513, 6.3857]$

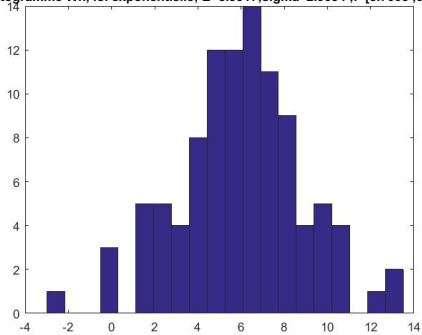


— Pour  $n=100$  :

Histogramme  $X_n$ , loi exponentielle;  $E=55.96$ ;  $\sigma=4.5503$ ;  $I=[55.505, 56.415]$



Histogramme  $W_n$ , loi exponentielle;  $E=5.9917$ ;  $\sigma=2.9094$ ;  $I=[5.7008, 6.2826]$

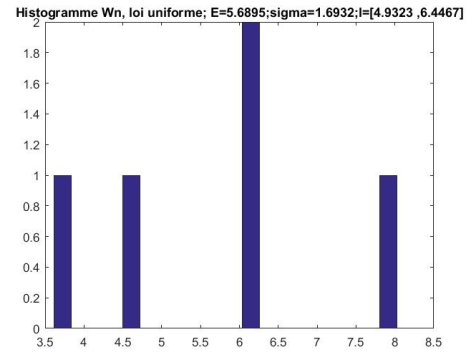
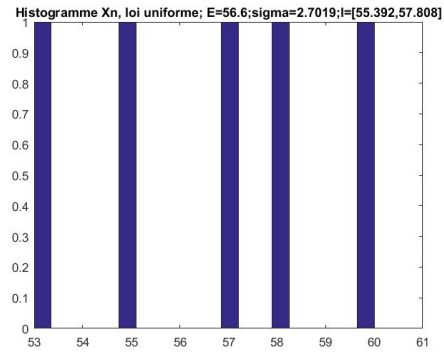


— **L'étude du First-Fit :**

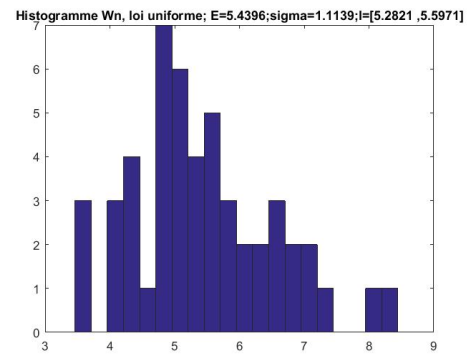
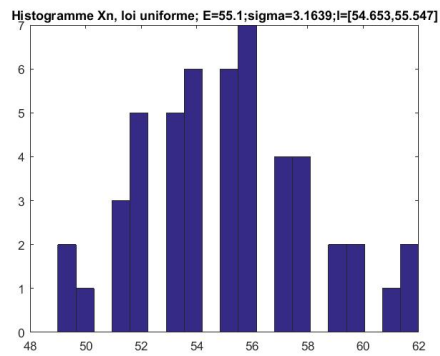
Par la même méthode appliquée sur l'étude du NF, on procède à étudier l'algorithme FF et on a comme résultat :

Pour la loi uniforme :

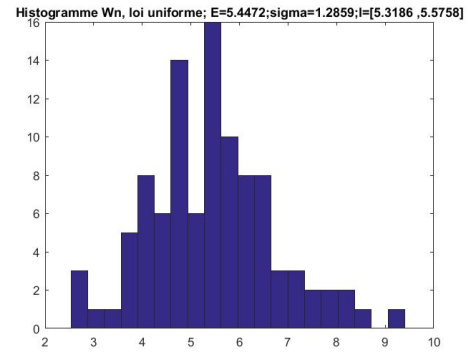
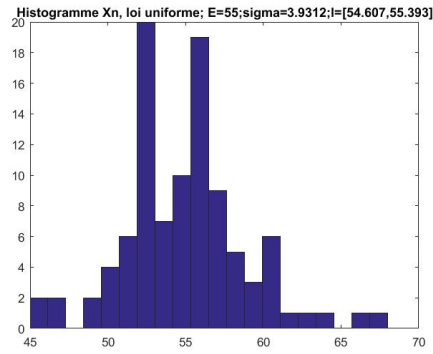
— *Pour  $n=5$  :*



— *Pour  $n=50$  :*

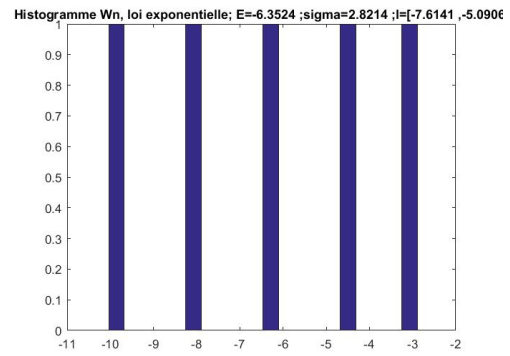
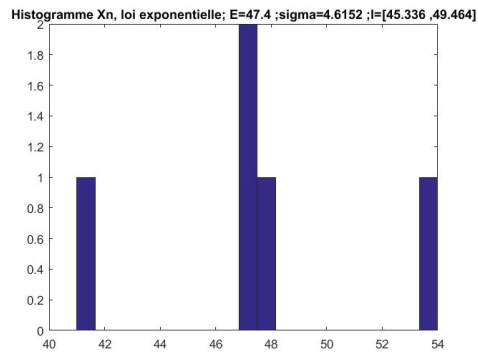


— Pour  $n=100$  :

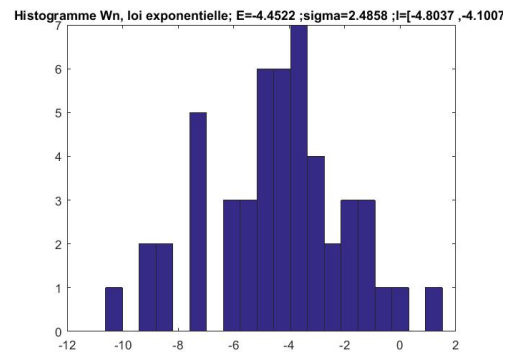
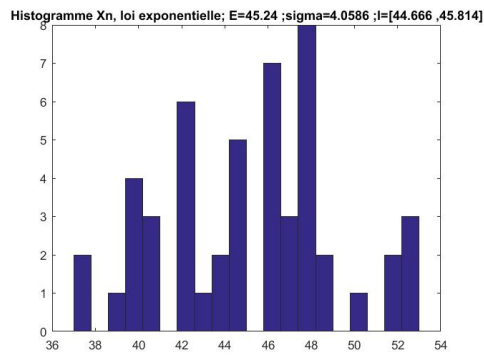


Pour la loi exponentielle :

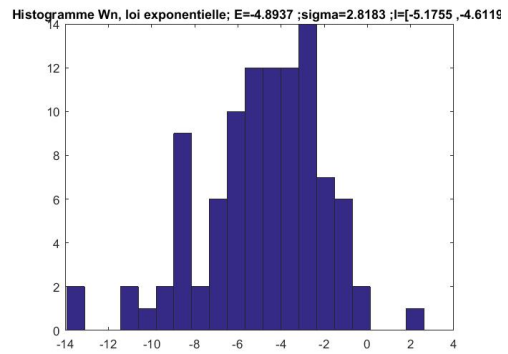
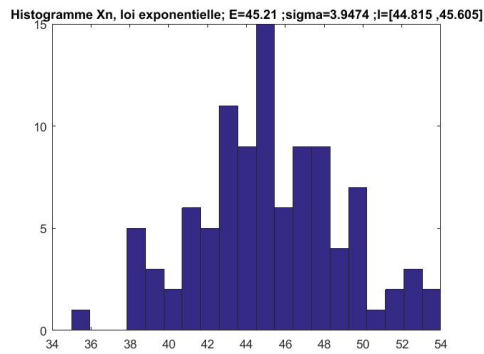
— Pour  $n=5$  :



— Pour  $n=50$  :

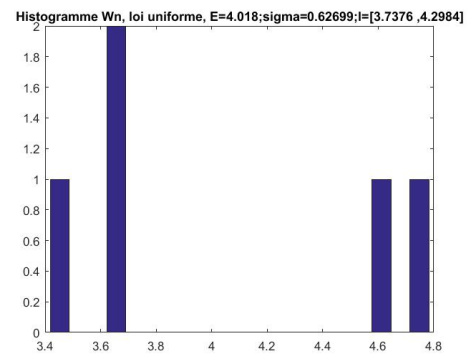
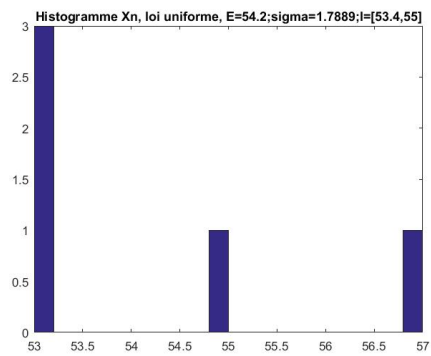


— Pour  $n=100$  :

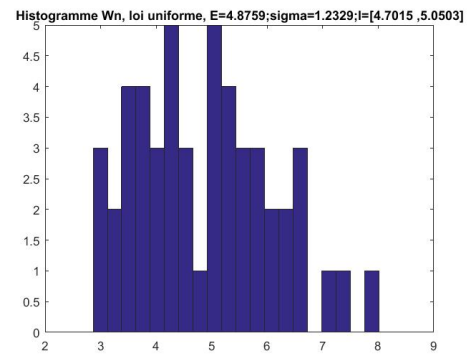
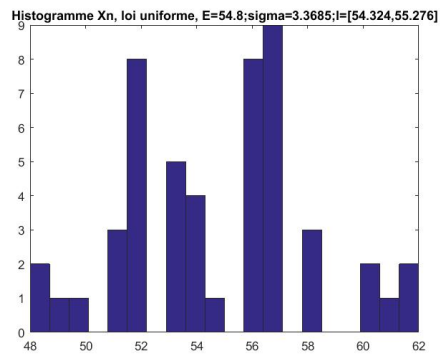


— L'étude du Best-Fit :  
Pour la loi uniforme :

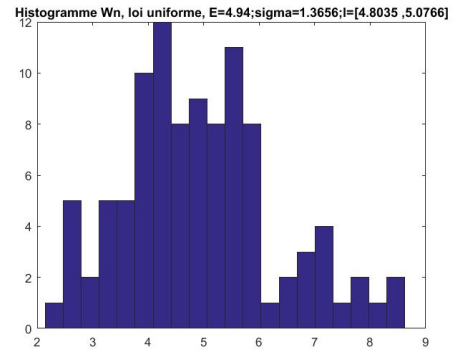
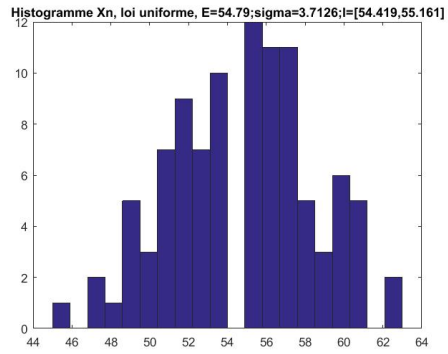
— Pour  $n=5$  :



— Pour  $n=50$  :

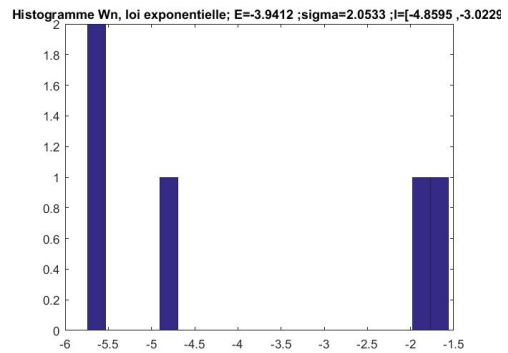
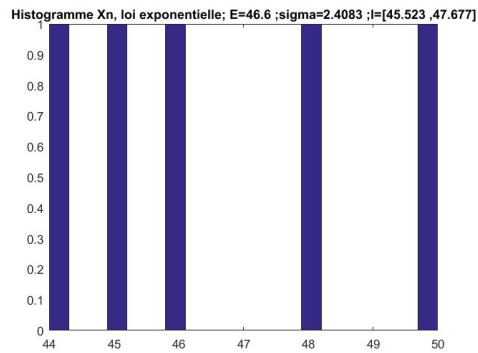


— Pour  $n=100$  :

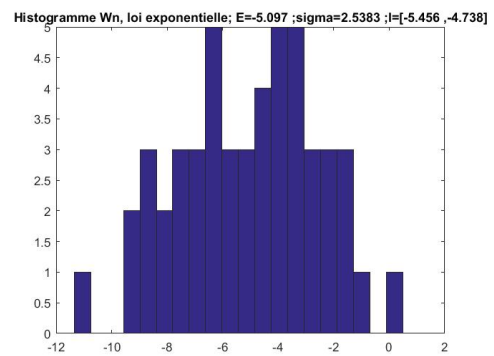
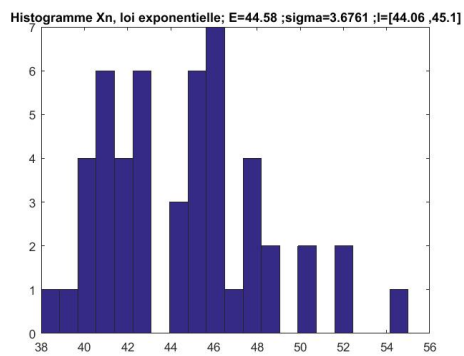


Pour la loi exponentielle :

— Pour  $n=5$  :

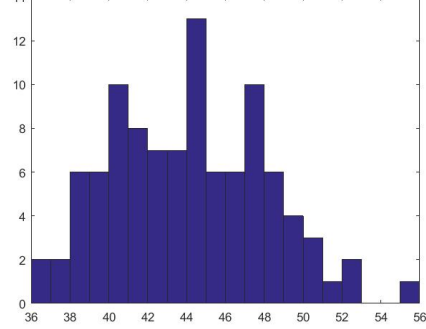


— Pour  $n=50$  :

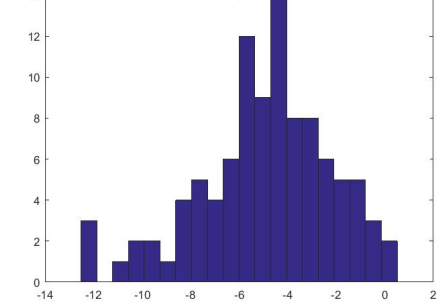


— Pour  $n=100$  :

Histogramme  $X_n$ , loi exponentielle;  $E=44.61$ ;  $\sigma=4.0199$ ;  $l=[44.208, 45.012]$



Histogramme  $W_n$ , loi exponentielle;  $E=-4.9098$ ;  $\sigma=2.7163$ ;  $l=[-5.1815, -4.6382]$

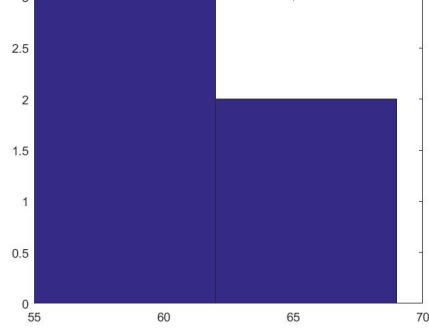


— L'étude du Skim-Up :

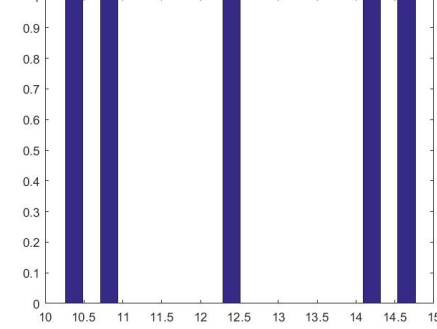
Pour la loi uniforme :

— Pour  $n=5$  :

Histogramme  $X_n$ , loi uniforme;  $E=61.4$ ;  $\sigma=6.3482$ ;  $l=[58.561, 64.239]$

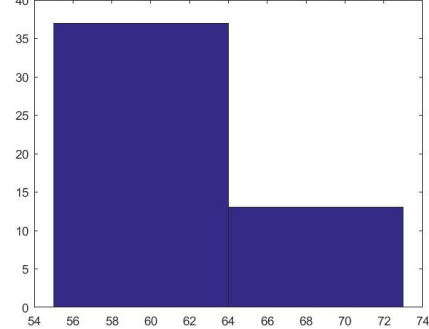


Histogramme  $W_n$ , loi uniforme;  $E=12.477$ ;  $\sigma=2.0307$ ;  $l=[11.569, 13.385]$

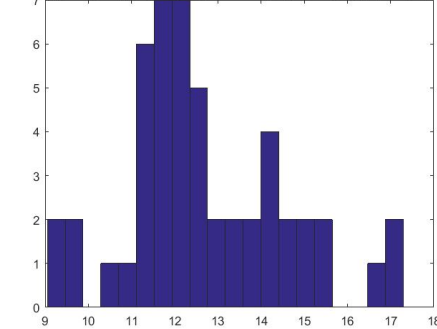


— Pour  $n=50$  :

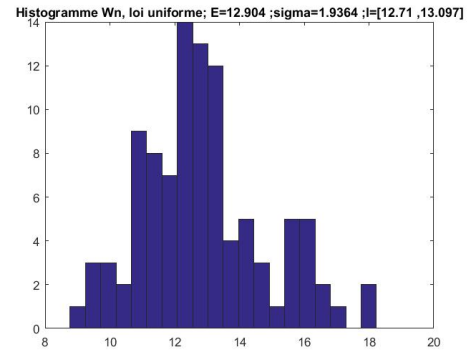
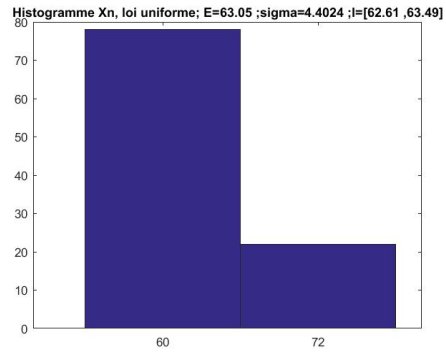
Histogramme  $X_n$ , loi uniforme;  $E=62.3$ ;  $\sigma=3.5066$ ;  $l=[61.804, 62.796]$



Histogramme  $W_n$ , loi uniforme;  $E=12.716$ ;  $\sigma=1.8653$ ;  $l=[12.452, 12.98]$

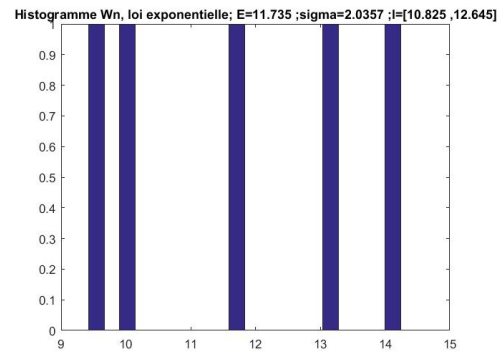
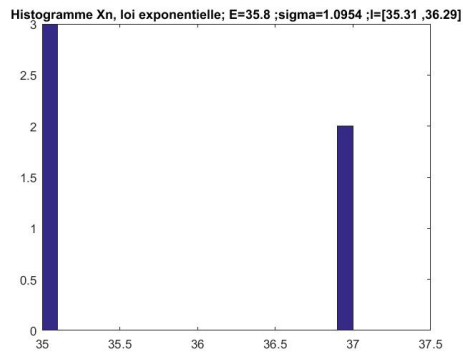


— Pour  $n=100$  :

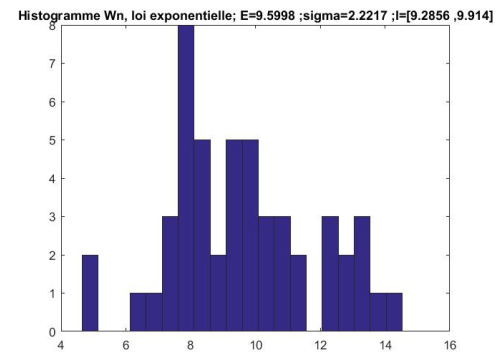
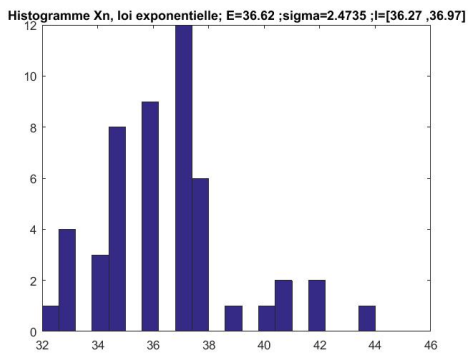


Pour la loi exponentielle :

— Pour  $n=5$  :

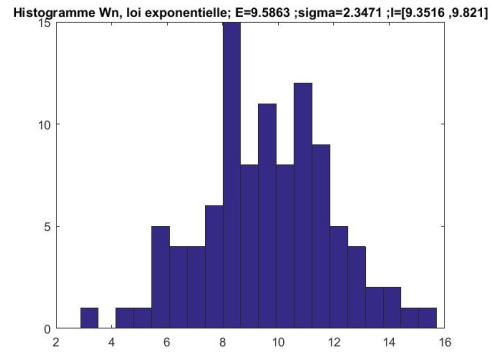
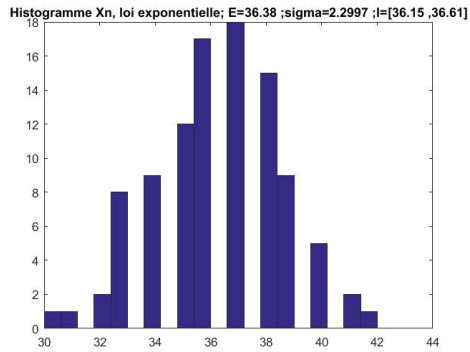


— Pour  $n=50$  :



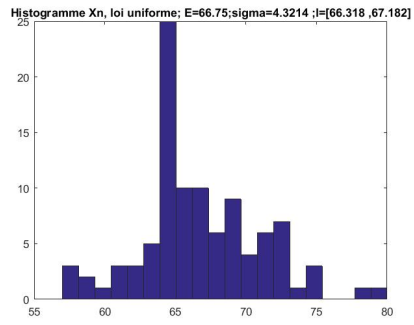


— Pour  $n=100$  :

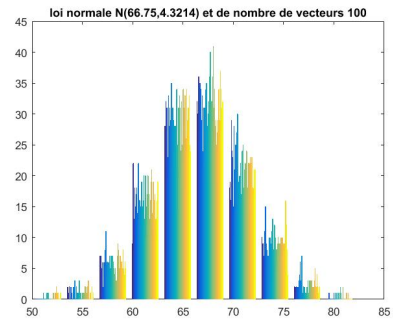


## Interprétation

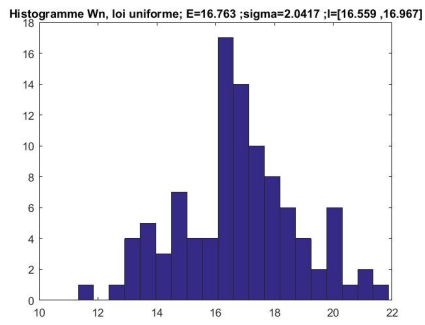
Nous constatons que la simulation de la loi exponentielle est presque semblable à celle de la loi uniforme, c'est à dire le résultat de la méthode Monte-Carlo ne diffère pas d'une loi à une autre. D'après les histogrammes obtenus de  $X_n^A$  et  $W_n^A$  et en augmentant  $n$ , nous remarquons que le résultat est semblable à la loi normale.



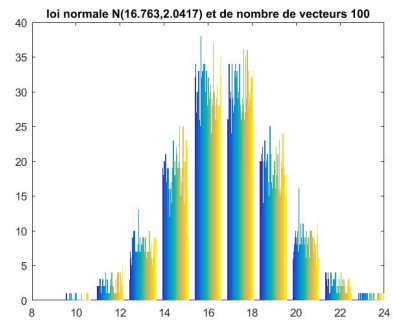
(a)



(b)

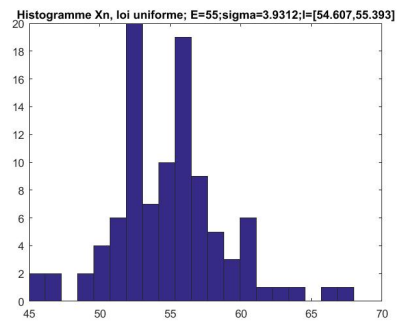


(c)

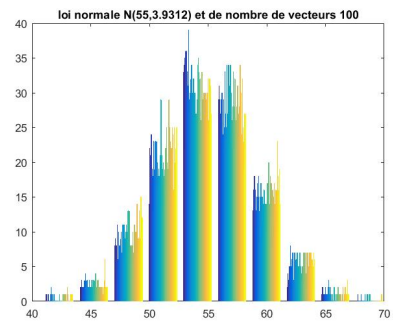


(d)

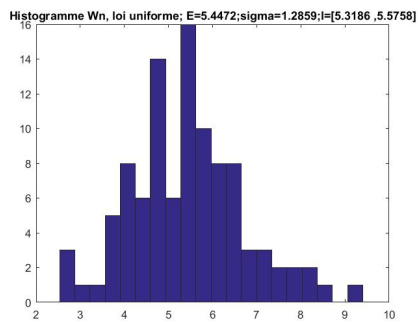
FIGURE 3.1 – comparaison des histogrammes de l'étude du NF et la loi normale pour  $n=100$



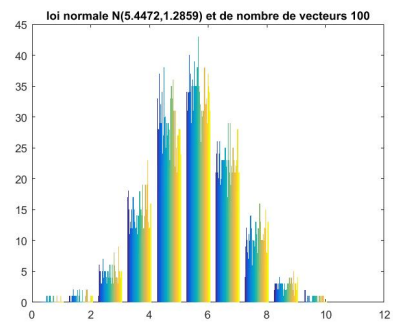
(a)



(b)

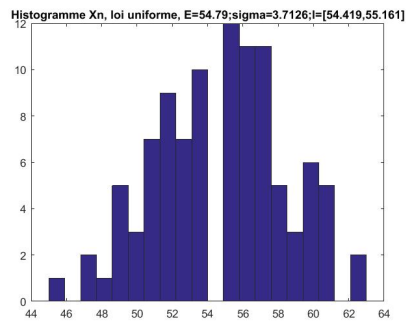


(c)

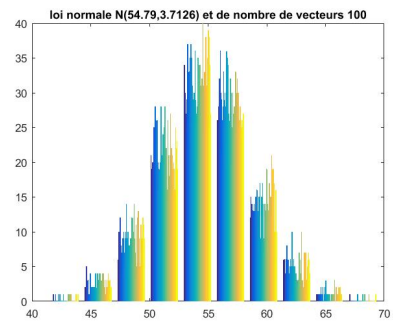


(d)

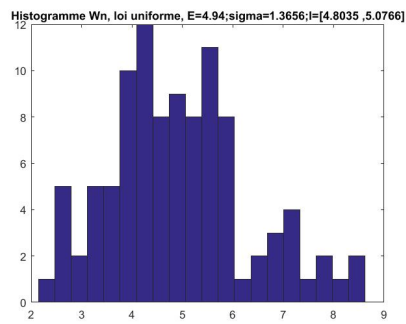
FIGURE 3.2 – comparaison des histogrammes de l'étude du FF et la loi normale pour  $n=100$



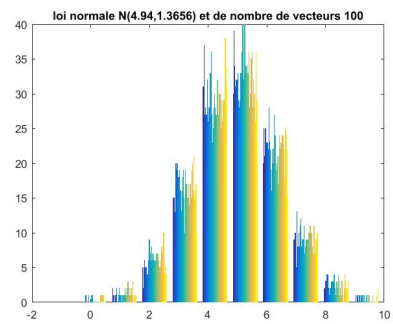
(a)



(b)

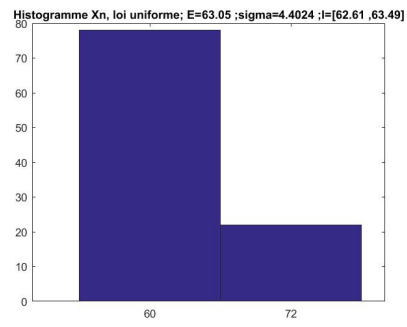


(c)

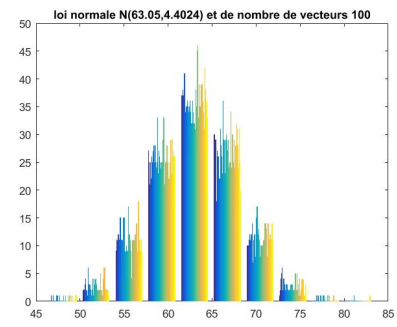


(d)

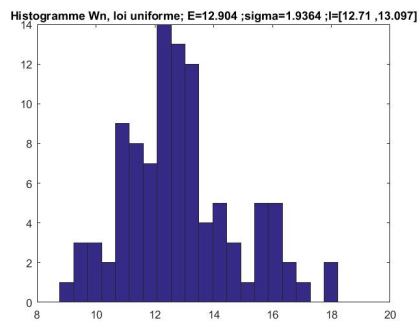
FIGURE 3.3 – comparaison des histogrammes de l'étude du BF et la loi normale pour  $n=100$



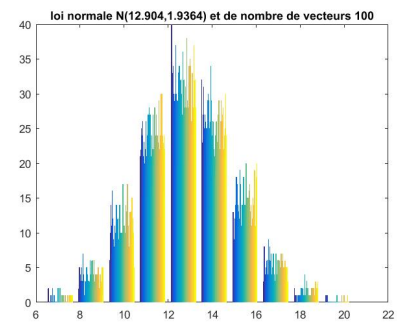
(a)



(b)



(c)



(d)

FIGURE 3.4 – comparaison des histogrammes de l'étude du SU et la loi normale pour  $n=100$

## 4. Etude de la performance des algorithmes pour un problème de gestion d'emplois de temps

### 4.1 Introduction

Cette partie représente notre expansion par rapport à ce sujet. Nous voulons appliquer les outils déjà établis sur un problème de gestion d'un emploi de temps. Notre problème se pose comme suit :

Rami est un employé chez une entreprise. Il reçoit d'une façon périodique mensuelle une liste de tâches à faire. Rami peut estimer le temps que chaque tâche peut lui prendre et cherche à optimiser son emploi du temps selon ses besoins (finir les tâches dans un nombre minimal sans se préoccuper du temps perdu, ou maximiser le temps perdu pour avoir des journées de travail moins lourdes). On sait que Rami peut gérer au plus 7 heures de travail par jour.

On peut comprendre ce problème en étant un problème de bin packing. Les listes des tâches sont des vecteurs aléatoires en supposant que le temps que prend une tâche suit une certaine loi qu'on établira. Les Bins représentent une journée de travail pour Rami et auront une longueur de 7 heures (420 minutes). On cherche à savoir les lois des variables aléatoires  $X_n$  et  $W_n$  établies au chapitre précédent pour chacun des quatre algorithmes de bin packing étudiés. Cette étude est analogue à celle faite dans la partie précédente avec certains changements. On conclura à la fin lesquels des algorithmes correspondent le mieux aux désirs de Rami.

### 4.2 Variables aléatoires

#### 4.2.1 Loi de la variable aléatoire de la durée d'une tâche

On commence l'établissement de la Loi en question en faisant l'hypothèse que la durée moyenne d'une tâche est 2 heures (120 minutes). On peut aussi supposer que

```

1 function resultat = generate_poisson(__m,__n,__s)
2     pkg load statistics;
3     resultat = [];
4     V=[];
5     for i=1:__n
6         U=[];
7         U=poissrnd(__m,1,__s);
8         V=[V;U];
9     endfor
10    resultat = V;
11 endfunction
12

```

FIGURE 4.1 – Fonction qui génère le vecteur des durées des tâches

la durée de chaque tâche est indépendante des tâches précédentes. Pour simplifier encore plus le problème, on choisit de chercher une loi discrète en prenant comme unité de temps 1 minute. La loi discrète qui exprime le mieux le comportement d'une telle variable aléatoire est la loi de Poisson.

Comme dans la partie précédente, on génère un vecteur aléatoire qui suit la loi de Poisson comme suit :

#### 4.2.2 Variable aléatoire "nombre de journées pas chargés"

Pour pouvoir mieux étudier la performance des algorithmes, nous avons choisis d'étudier une 3ème variable aléatoire qui caractérise les journées non chargées. Une journée non chargée est une journée qui a au moins 1 quart de sa durée vide (donc prend la valeur de la somme des journées qui vérifient  $W_n \geq 0.25 \cdot 420$  minutes). Cette variable sera nommée  $Y_n$  est elle est générée par la fonction Generate-Y-nomalgorithme.

### 4.3 Simulation

Le Code est analogue au celui établi dans la partie précédente, La seule différence est que les fonctions qui génèrent  $X_n$  et  $W_n$  prennent un nouveau paramètre qui est la taille de la Bin (420 minutes dans notre cas). La simulation est faite pour des liste de 200 tâches, l'étude par méthode de Monte-Carlo est faite pour  $M=1000$ . Une liste différente est générée par chaque méthode mais cela n'affectera pas nos conclusions vû que  $M$  est assez grand. Le Code de la simulation est le même pour les 4 méthodes, il suffit juste de changer le nom de l'algorithme pour chaque fonction de type Generate-A-nomalgorithme ( $A = X, W$  ou  $Y$ ).

```

function resultat = generate_Y_bestfit (__U,s)
    S=[];
    B=[];
    for j=1:size(__U)(1)
        B=fbestfit(__U(j,:),s);
        count=0;
        for i=1:length(B)
            if (B(i).free>=0.25*s)
                count=count+1;
            endif
        endfor
        S=[S,count];
    endfor
    resultat = S;
endfunction

```

FIGURE 4.2 – Fonction qui génère  $Y_n$

```

1 clear all
2 W=generate_poisson(180,1000,200);
3 S=generate_X_bestfit(W,420);
4 ps=parametres(S)
5 M=generate_W_bestfit(W,420);
6 pm=parametres(M)
7 N=generate_Y_bestfit(W,420);
8 pn=parametres(N)

```

FIGURE 4.3 – Code de la simulation



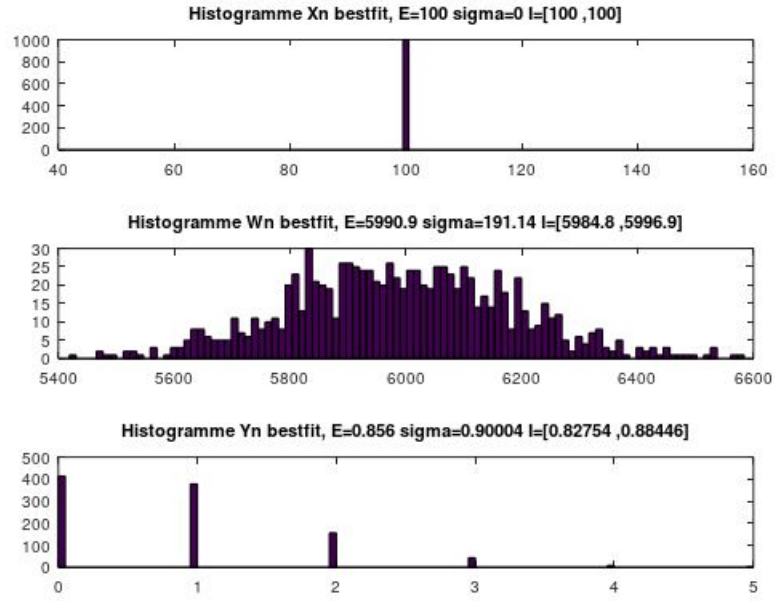


FIGURE 4.4 – Histogrammes des variables aléatoires pour l’algorithme bestfit

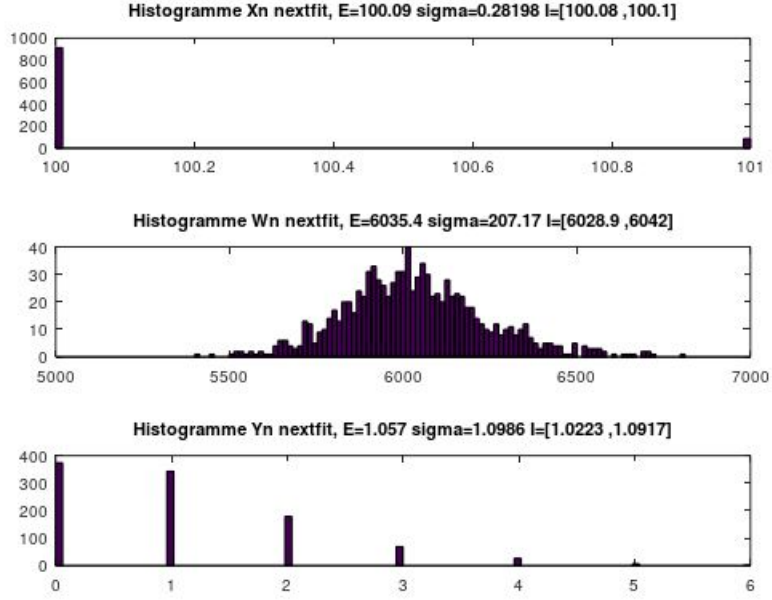


FIGURE 4.5 – Histogrammes des variables aléatoires pour l’algorithme nextfit

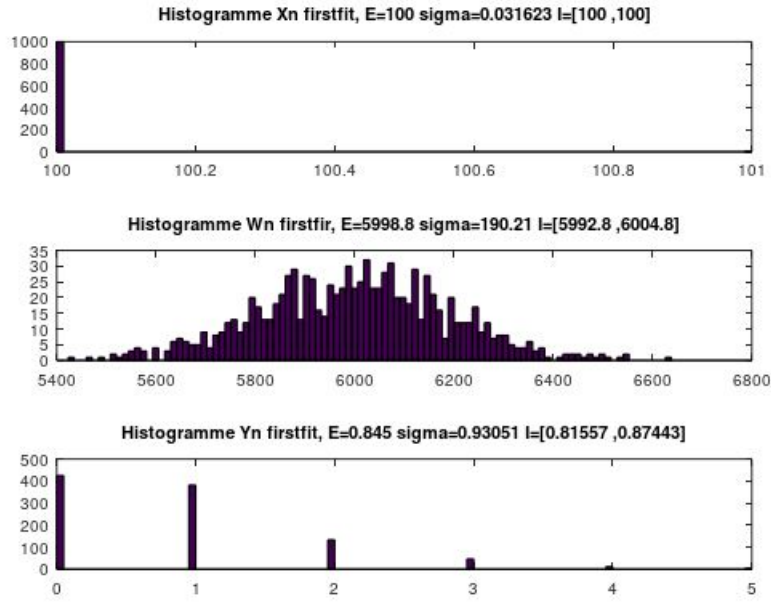


FIGURE 4.6 – Histogrammes des variables aléatoires pour l’algorithme firstfit

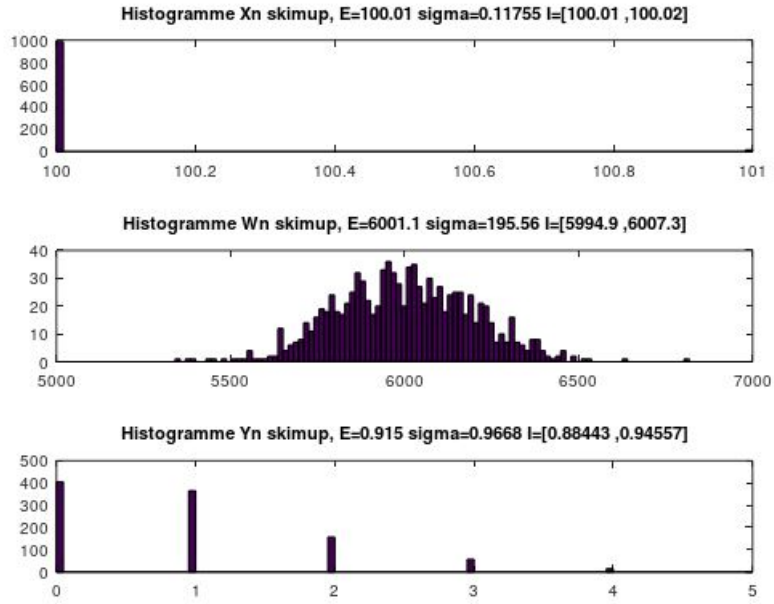


FIGURE 4.7 – Histogrammes des variables aléatoires pour l’algorithme skim-up

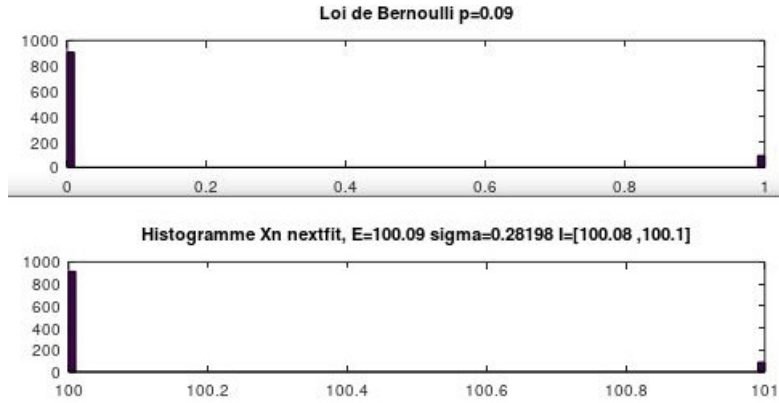


FIGURE 4.8 – Comparaison entre l’histogramme de  $X_{200}$  pour l’algorithme Nextfit et celui d’une loi de Bernoulli de paramètre 0.09

## 4.4 Interprétation des Histogrammes

### 4.4.1 Loi de $X_n$

Les Histogrammes de  $X_n$  pour  $n=200$  ressemblent fortement à ceux d’une loi de Bernoulli qui prend les valeurs 100 ou 101. Le paramètre de cette loi change pour chaque algorithme et il est dans le cas de l’algorithme Bestfit presque égal à 1. On peut l’approximer d’être aux alentours de 0,09 pour l’algorithme Nextfit. Et aux alentours de 0,01 pour l’algorithme Skimup. On pourra l’approximer d’être de l’ordre de  $10^{-3}$  pour l’algorithme FirstFit. Pour l’algorithme Bestfit, comme l’évènement  $X_{200}=100$  est presque sûr, on pourra dire que la loi de Bernoulli associée est de paramètre 0.

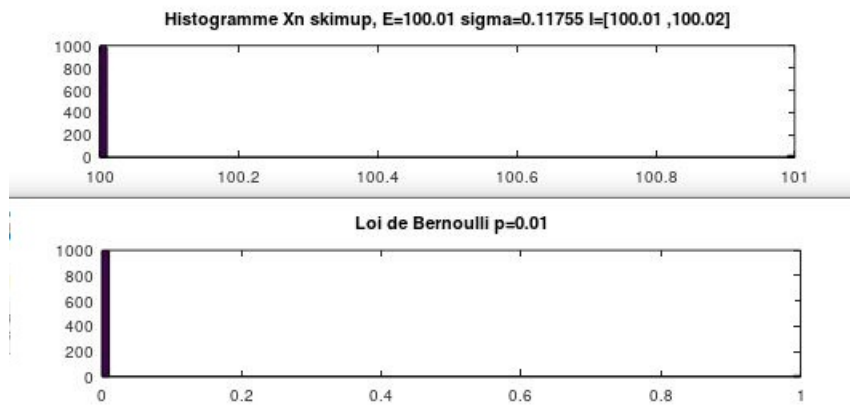


FIGURE 4.9 – Comparaison entre l’histogramme de  $X_{200}$  pour l’algorithme Skim-up et celui d’une loi de Bernoulli de paramètre 0.01

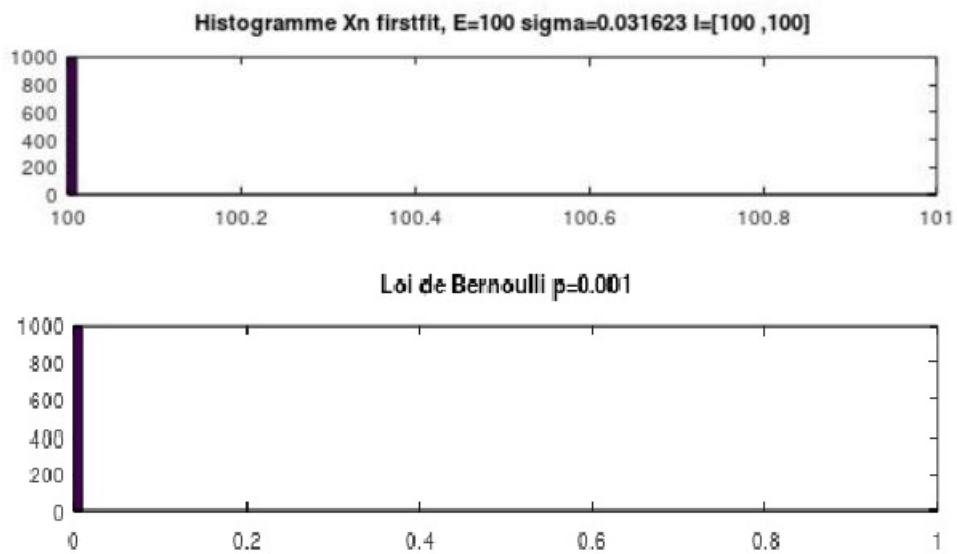


FIGURE 4.10 – Comparaison entre l’histogramme de  $X_{200}$  pour l’algorithme Firstfit et celui d’une loi de Bernoulli de paramètre 0.001

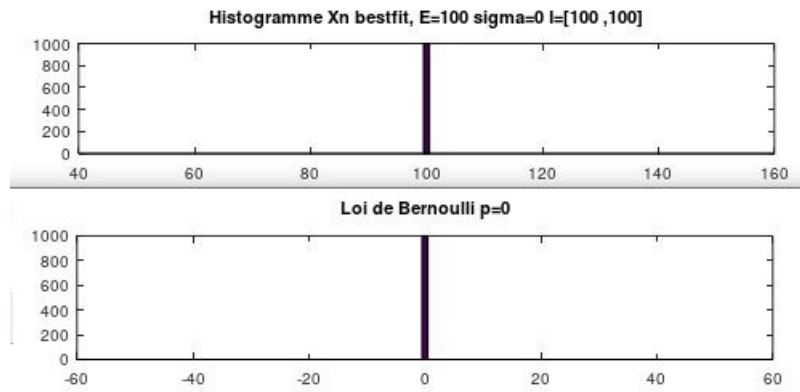


FIGURE 4.11 – Comparaison entre l’histogramme de  $X_{200}$  pour l’algorithme Bestfit et celui d’une loi de Bernoulli de paramètre 0

#### 4.4.2 Loi de $W_n$

Comme fait pour  $X_n$ , une interprétation graphique des histogrammes semble pointer vers le fait que  $W_n$  suit une loi normale. La méthode de Monte-Carlo nous a sorti directement les paramètres  $\mu$  et  $\sigma^2$ .

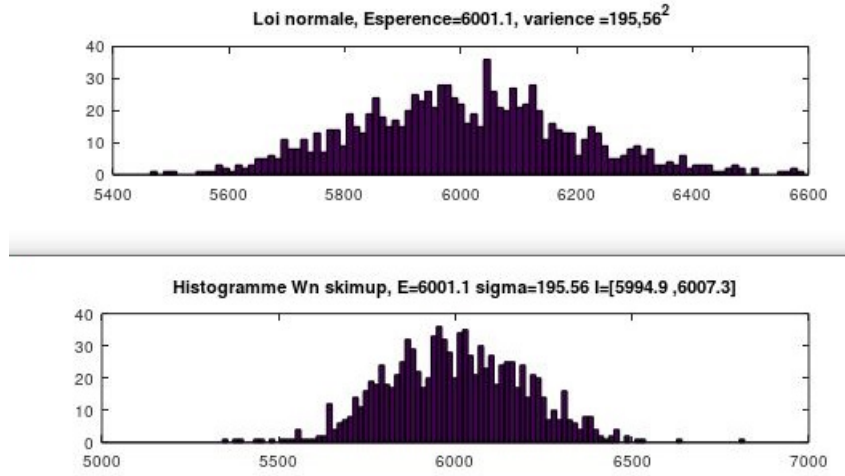


FIGURE 4.12 – Comparaison entre l’histogramme de  $W_{200}$  pour l’algorithme Skimup et de la loi normale associée

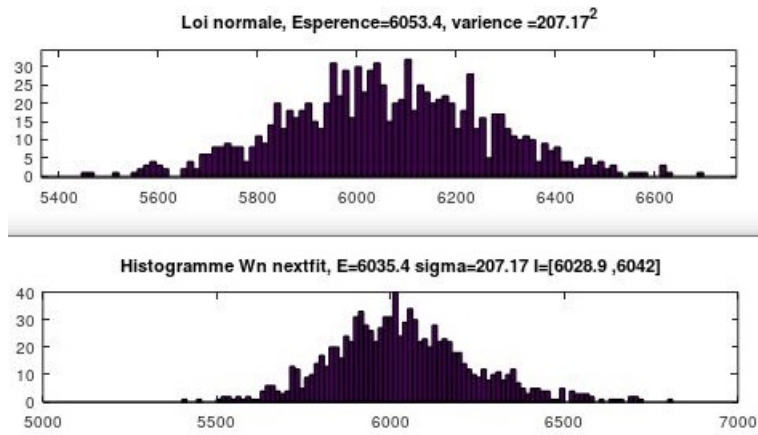


FIGURE 4.13 – Comparaison entre l’histogramme de  $W_{200}$  pour l’algorithme Nextfit et de la loi normale associée

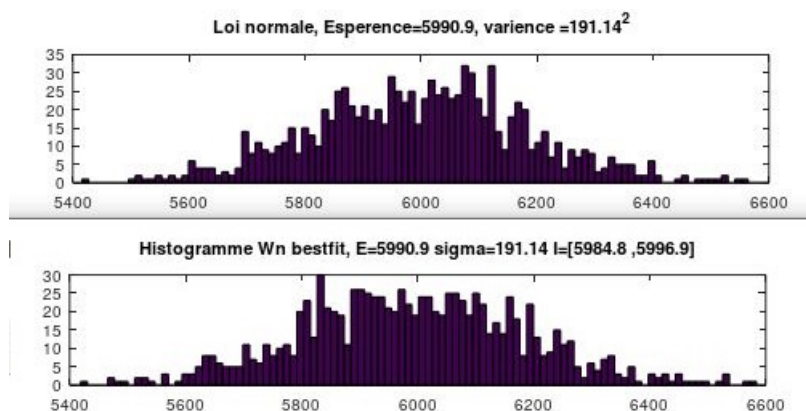


FIGURE 4.14 – Comparaison entre l'histogramme de  $W_{200}$  pour l'algorithme Bestfit et de la loi normale associée

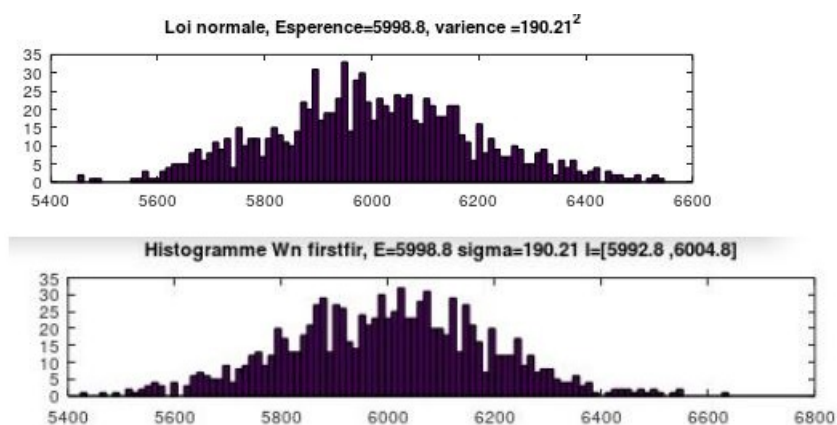


FIGURE 4.15 – Comparaison entre l'histogramme de  $W_{200}$  pour l'algorithme Firstfit et de la loi normale associée

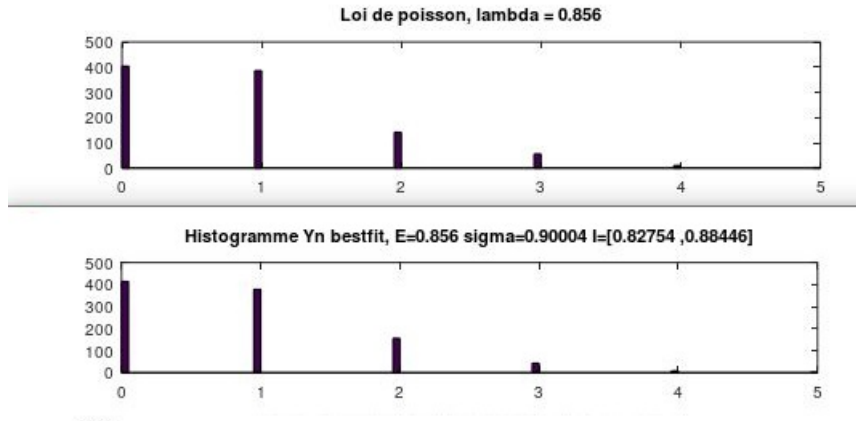


FIGURE 4.16 – Comparaison entre l’histogramme de  $Y_{200}$  pour l’algorithme Bestfit et de la loi de poisson de paramètre  $\lambda = 0.856$

#### 4.4.3 Loi de $Y_n$

D’une manière assez surprenante, la variable aléatoire  $Y_n$  semble suivre aussi une loi de poisson (Comme la liste des durées des taches). Le paramètre de cette loi semble correspondre aux valeurs moyennes estimées par la méthode de Monte-Carlo.



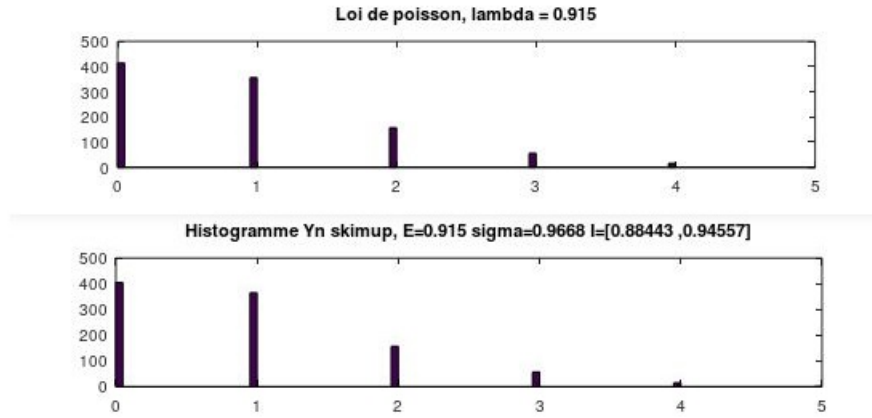


FIGURE 4.17 – Comparaison entre l’histogramme de  $Y_{200}$  pour l’algorithme Skim-up et de la loi de poisson de paramètre  $\lambda = 0.915$

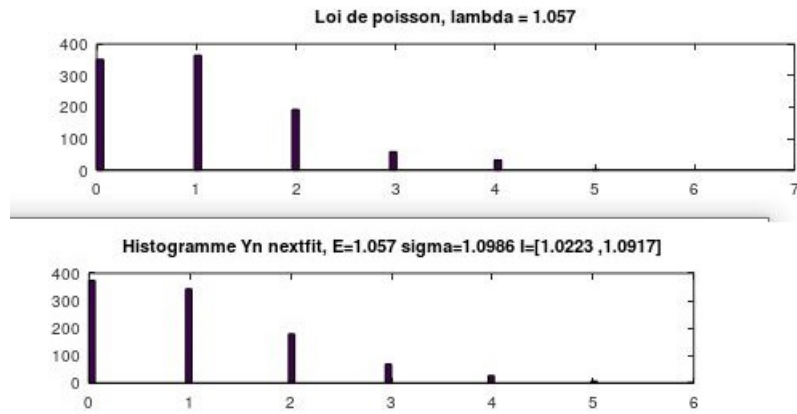


FIGURE 4.18 – Comparaison entre l’histogramme de  $Y_{200}$  pour l’algorithme Nextfit et de la de poisson de paramètre  $\lambda = 1.057$

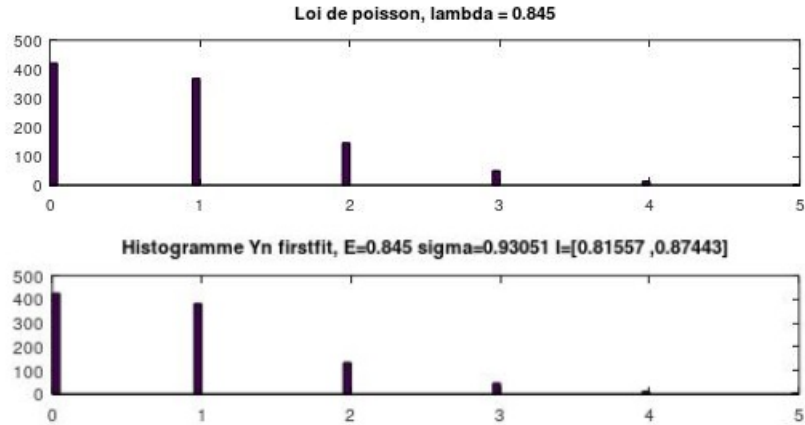


FIGURE 4.19 – Comparaison entre l’histogramme de  $Y_{200}$  pour l’algorithme Firstfit et de la loi de poisson de paramètre  $\lambda = 0.854$

## 4.5 Conclusion et discussion

Nous avons pu modéliser les 3 variables aléatoires  $X, W$  et  $Y$  par des lois usuelles. En revenant à notre problème, on pourra conclure sur laquelle des méthodes est la mieux adaptée à chaque besoin. Si Rami cherche à finir les tâches en un nombre de journées minimal, même si les 4 algorithmes donnent un résultat semblable, l’algorithme Bestfit qui suit une loi de Bernoulli de paramètre 0 est le plus convenable à choisir. Si Rami cherche à avoir un maximum de temps perdu, L’algorithme nextfit est l’algorithme à choisir. Si Rami souhaite maximiser les journées pas chargées, l’algorithme nextfit est aussi celui à choisir !. En prenant en considération le temps d’exécution de chaque algorithme, même si nous ne l’avons pas sauvegardé, nous avons constaté que les algorithmes Skim-up et Nextfit prennent beaucoup moins de temps de calcul que Firstfit, qui lui-même prends beaucoup moins de temps de calcul que Bestfit, cette donnée peut aussi distinguer entre les algorithmes pour arriver vers un choix optimal.

## 5. Simulation de l'occupation des lits de réanimation en tunisie

### 5.1 introduction

Nous avons laissé cette partie du projet en dernier, car elle est plutôt une curiosité qu'une réelle expansion. L'idée qui nous est venue est assez éloignée du cadre original de ce projet, mais exploite certains des outils que nous avons déjà établis.

Nous voulons simuler le taux d'occupation des lits de réanimations ICU (Intensive Care Units) dans le cadre du coronavirus. Nous supposons qu'à  $t = 0$  la tunisie possède 200 lits ICU qui sont dédiés uniquement aux cas graves du Covid-19.

les BINS dans notre cas correspondent aux lits ICU, chaque BIN contient soit 0 (lit vide) soit un entier naturel  $n$  (lit occupé pendant  $n$  jours). La simulation se fait jour par jour, chaque jour la liste des lits est actualisée (durée d'occupation des lits est réduite de 1) puis on y insère les nouvelles durées d'occupation si possible. La simulation se produit comme ainsi :

1. On génère le nombre de cas graves du covid-19 du  $i$ -ème jour
2. On génère la liste des durées d'occupation des cas générés précédemment
3. On actualise puis on insère les nouvelles durées d'occupation
4. On vérifie si il y'a saturation ou pas

Dans ce qui suit, on explicitera la démarche et le code de chaque étape mentionnée.

### 5.2 Liste des nouveaux cas journaliers du cov-19 en tunisie

Cette étape est celle qui nous a posé plusieurs difficultés. L'allure de la courbe de la croissance du nombre d'infections du covid-19 est celle d'une courbe logistique. Intuitivement on croyait pouvoir la modéliser par une loi logistique, mais ce modèle

n'exprimait pas le fait que le nombre d'infectés au temps  $t$  dépend explicitement du nombre d'infectés aux temps inférieurs à  $t$ . Après une recherche plus approfondie sur la modélisation des pandémies, nous avons trouvé que le modèle le mieux adapté pour notre objectif est un modèle probabiliste stochastique. Malheureusement, vu la contrainte du temps, nous n'avons pas pu assimiler la théorie derrière ce type de modèles et nous avons choisi de travailler avec une approche différente. Le modèle qu'on a choisi est un modèle déterministe de type SEIR (susceptible, exposé, infecté, rétabli). Ce modèle décrit la propagation du covid-19 par moyen d'équations différentielles.

### 5.2.1 Modèle SEIR

**aspect théorique** Ce modèle étant nécessaire à notre étude, Nous l'explicitons, tout en le supposant admis. Il est régi par les équations différentielles suivants :

$$\begin{aligned}\frac{dS(t)}{dt} &= -\lambda(t)S(t) \\ \frac{dE(t)}{dt} &= \lambda(t)S(t) - fE(t) \\ \frac{dI(t)}{dt} &= fE(t) - rI(t) \\ \frac{dR(t)}{dt} &= rI(t)\end{aligned}$$

$S(t)$  : population susceptible.  
 $E(t)$  : population infectée mais qui n'infecte pas.  
 $I(t)$  : population infectée qui infecte.  
 $R(t)$  : population rétablie.  
 $\lambda(t), f, r$  : taux relatifs aux variables précédentes.

**Code Matlab** Nous avons pris ce modèle déjà écrit sur matlab et nous l'avons adapté à notre étude en changeant certaines variables. Le principe du fonctionnement du code est assez simple. On commence par l'initialisation des variables qui correspondent à notre étude :

```
%%% Initialisations
Pre_infec = 6; %nombre initial d'infectés non infectants
f = 1/Pre_infec; %fréquence
Duration = 5; %temps passé en tant que infecté qui n'infecte pas
r=1/Duration; %fréquence
R_0 = 5; % R0 sans confinement, chaque infecté tend à infecter 5 susceptibles
N = 11000000; % Population of tunisia (2020)
beta = R_0/(N*Duration);
```

FIGURE 5.1 – Initialisation des variables du modèle SEIR

par la suite on crée les vecteurs temps et données initiales :

```
tspan = 0:1:30; % durée de la simulation = 30+1 jours
y0 = [N-1, 1, 0, 0, 0]; % [population susceptible, infectés à t=0, NAN, NAN, NAN]
```

FIGURE 5.2 – Initialisation du vecteur temps et vecteur données initiales

La fonction `ode45()` munie de la fonction `ode_fun_simple()` permet de résoudre le système différentiel énoncé plus haut. Cette fonction nous retourne une matrice `y` qui contient le nombre de chaque population étudiée pour chaque journée. Le vecteur `t` correspond à l'axe des temps (une journée par colonne).

```
[t,y]=ode45(@ (t,y) ode_fun_simple(t,y,beta), tsparn, y0);
```

FIGURE 5.3 – Solution du système différentiel

population infectée					
y	1	2	3	4	5
1	1.1e+07	1	0	0	0
2	1.1e+07	0.90902	0.17411	0.0061774	0.000217...
3	1.1e+07	0.98396	0.33559	0.023745	0.000835...
4	1.1e+07	1.1983	0.5131	0.05286	0.0018605
5	1.1e+07	1.5538	0.73123	0.095484	0.0033607
6	1.1e+07	2.0745	1.0152	0.15528	0.0054654
7	1.1e+07	2.8048	1.3952	0.23779	0.0083693
8	1.1e+07	3.8129	1.9093	0.35089	0.01235

FIGURE 5.4 – Matrice `y`

Le seul résultat qui nous est utile après cette partie est la donnée de la deuxième colonne de la matrice `y`. Nous exploiterons celle ci pour en déduire le nombre de patients qui nécessitent un lit ICU pour chaque journée de la simulation.

## 5.3 Liste des durées d'occupation

### 5.3.1 Nombre de nouveaux cas graves par journée

Avant de parler des durées d'occupation, on doit passer par une étape intermédiaire pour pouvoir récupérer le nombre des nouveaux cas graves journaliers. Le principe est de partir de la 2ème colonne de la matrice "y" générée précédemment, et de multiplier chaque case par le facteur 0.00044 qui représente la fraction des malades qui nécessiteront un lit ICU (valeur trouvée sur internet). Par la suite une simple différence entre chaque deux colonnes nous donne la liste cherchée qu'on nommera h.

```
x=round(0.00044.*(y(:,2)));  
h=zeros(length(x),1);  
for i=1:length(x)  
    if (x(i)>0)&&(x(i)!=x(i-1))  
        h(i) = x(i)-x(i-1);  
    endif  
endfor
```

FIGURE 5.5 – Vecteur h : nombre des nouveaux malades nécessitant un lit de réanimation par jour

### 5.3.2 Vecteur durée d'occupation

**Détermination de la loi de probabilité de la durée d'occupation** Toutes les données qu'on avait obtenus jusqu'à cette étape sont déterministes. Comme on est dans le cadre d'un projet de probabilités, nous avons choisi de considérer que la durée d'occupation d'un lit ICU par un malade est une variable aléatoire. Des données sur internet disent que le séjour moyen d'un malade est de l'ordre de 5 jours, nous savons aussi que 99% des malades restent moins de 11 jours. En faisant l'hypothèse que la durée d'occupation suit une loi gaussienne de valeur moyenne 5, un simple calcul de probabilité nous fait sortir l'écart type.

Données de l'hypothèse :

$$X \rightsquigarrow N(5, \sigma^2), P(X < 11) = 0.99$$

on se ramène à une loi normale centrée réduite :

$$P\left(\frac{X-5}{\sigma} < \frac{6}{\sigma}\right) = 0.99$$

Par la méthode de l'inverse de la loi normale on obtient :  $\frac{6}{\sigma} = 2.326$  d'où  $\sigma = 2.58$

**Code Matlab** comme la i-ème colonne du vecteur h représente le nombre de nouveaux malades lors du i-ème jour, on génère alors pour chaque jour un vecteur

v qui a pour longueur la valeur de  $h(i)$ , et qui contient les valeurs prises par la variable aléatoire qui suit la loi établie précédemment. La fonction generate-gauss prend en paramètres la moyenne et l'écart type, et génère un singleton suivant la loi normale. On prend l'arrondi pour faciliter les futurs calculs, et on vérifie aussi si la valeur générée est significative (temps de séjour minimum est 1).

```

4 function res = generate_gauss(moy,e)
5     x=0;
6     while (x<1)
7         x=normrnd(moy,e);
8     endwhile
9     res=round(x);
10 endfunction

```

FIGURE 5.6 – Fonction generate-gauss

## 5.4 Boucle principale

La boucle principale du programme marche comme suit :

Chaque jour, on vérifie si il y'a nouveaux occupants ou non

Si oui, on actualise (on réduit la durée des occupants déjà présents par 1) puis on génère un vecteur de durées d'occupation, on utilise un algorithme Firstfit pour remplir la liste des lits par les nouveaux occupants. Sinon on actualise seulement.

On vérifie si il y a saturation, si oui on sauvegarde l'indice de la journée de saturation.

**Code Matlab** Ci dessous le code qui exprime ce qu'on vient de dire :

```

0 function res = refresh(c)
1     c2=c;
2     for i=1:length(c2)
3         if (c2(i)!=0)
4             c2(i)=c2(i)-1;
5         endif
6     endfor
7     res=c2;
8 endfunction

```

FIGURE 5.7 – First-Fit

```

for i=1:length(h) %%compteur nombre de jours
    if (h(i)<=0) %pas de nouveaux malades qui nécessitent des lits de réanimation
        sprintf("skip");
        if i==1
            capacity(i,:) = zeros(1,200);
        else
            capacity(i,:)=refresh(capacity(i-1,:));
        endif
    else %si il y'a des nouveaux malades qui nécessitent des lits de réanimation
        v=[];
        for j=1:h(i) %génère un vecteur de durée de séjour, chaque colonne correspond à un malade
            tmp=generate_gauss(5,2.58);
            v=[v,tmp];
        endfor
        if i==1 %pour éviter des erreurs sur i, on isole le cas du premier jour de la simulation
            capacity(i,:) = firstfit(v,zeros(1,200));
        else
            capacity(i,:)=refresh(capacity(i-1,:)); %actualise la capacité des lits
            capacity(i,:) = firstfit(v,capacity(i,:)); %remplit les lits par les durées
        endif
    endif
    capl(i)= counter(capacity(i,:)); %sauvegarde l'occupation des lits au i-ème jour
    %%affichage
    ch="";
    ch=strcat("jour ",num2str(i-1)," : ",num2str(counter(capacity(i,:))),"/200");
    %sprintf(ch) %décommenter pour voir l'occupation pour chaque journée
    %%vérification si il y'a saturation
    if ((verif(capacity(i,:))==1)&&(e==0))
        e=i-1; %si oui, sauvegarde la première occurrence
    endif
endfor

```

FIGURE 5.8 – Boucle principale

## 5.5 Résultats et Conclusion

On lance la simulation deux fois, une pour le cas où il n'y a pas de confinement, et l'autre pour le cas où il y'a confinement. Les deux simulations sont lancées pour une durée de 180 jours. On remarque que pour le cas du déconfinement, on trouve toujours qu'il y'aura saturation au 46-ème jour. Le cas avec confinement ne sature jamais. Ces deux résultats sont assez déterminites, Pourtant notre simulation comprend un aspect probabiliste par rapport à la durée de séjour. Cela s'explique par le fait que la surcharge des lits provient principalement de la croissance de la population malade qu'on a établi par un modèle déterministe. Il faudra aussi noter que notre simulation n'est valide que pour les parties croissantes des courbes, car une erreur implicite existe dans le code qui fait en sortes que dès le moment où la population des malades commence à diminuer, on suppose que la nouvelle demande pour les lits est nulle. Cependant cela n'influe pas les résultats car la saturation arrive au moment où la courbe croit. Dans le cas avec confinement le seul résultat



qui nous est significatif est celui de la non existence d'une saturation. L'effet de la variable aléatoire de la durée de séjour s'observe sur certaines parties des courbes où il y'a fluctuations.

```

# Affichage des commandes
ans = sans confinement:
ans = saturation le46ème jour
ans = avec confinement:
ans = pas de saturation

```

FIGURE 5.9 – Résultats de la simulation

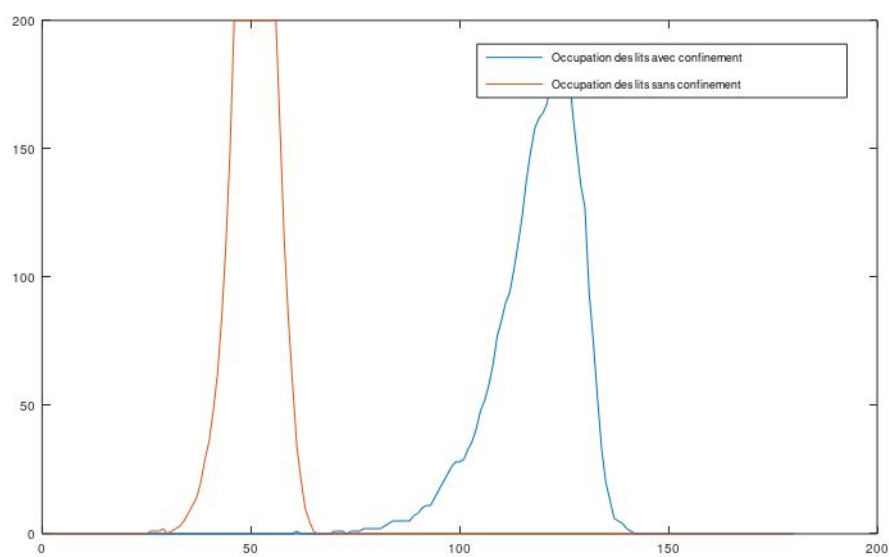


FIGURE 5.10 – Courbes de l'occupation des lits ICU

## 6. Conclusion

Pour conclure, ce projet nous a permis d'assimiler plusieurs nouvelles notions qui concernent la statistique autant que la probabilité, l'implémentation de ces algorithmes sur matlab nous a permis de mieux appréhender cet outil. La méthode de Monte-Carlo nous a aussi permis de savoir comment on peut déduire des lois de probabilités d'une façon assez précise à partir d'histogrammes.