RAPPORT -EVALUATION DE PERFORMANCES-
Simulation d'une file attente M/M/1

I-Approche du projet:

Le projet consiste a trouver de manière statistiques certaines propriétés théoriques qui caractérise la file d'attente M/M/1 en utilisant une simulation discrète. Le programme de la gestion des événements (Echéancier) est fait dans le langage JAVA ainsi l'affichage des résultats théoriques et pratiques.

Ce rapport entame deux axes principales du projet :

- La comparaison des résultats obtenus par simulation avec celles théoriques en se basant sur des figures, ainsi vérifier l'exactitude des résultats de simulation en faisant varier plusieurs paramètres statistiques par rapport a celles théoriques pour conclure finalement a quelle point le programme permet la simulation du théorie.
- Les performances du simulateur en expliquant comment le programme permet de réduire le temps de simulation (Complexité du programme)

II-Comparaison des résultats pratiques et théoriques fournies par le simulateur:

❖ ***Pratique VS Théorique:***

Le programme JAVA intègre une fonction "ecrirefichier()" qui permet d'écrire les résultats de la simulation dans des fichiers data mis dans le dossier "Data_Simulation". On utilise l'outil Gnuplot pour tracer ces résultats et en déduire des conclusion a propos de la comparaison des résultats pratiques avec celle théoriques.

- **Proportion clients avec attente (Intensité du trafic):**

-L'intensité du trafic est défini théoriquement par la formule: $\rho = \lambda/\mu$
-La chaîne de Markov est ergodique si $\rho < 1$.
-On trace les résultats comparatifs concernant l'intensité dans le fichier 'Intensite.dat' qui contient le rapport ρ (intensité théorique) et l'intensité pratique et la différence entre les deux (Erreur).

Obtention du fichier data:

On fait varier le rapport lambda/mu et à chaque fois on compare avec le résultat pratique obtenu. J'ai opté pour le code suivant pour faire ceci.

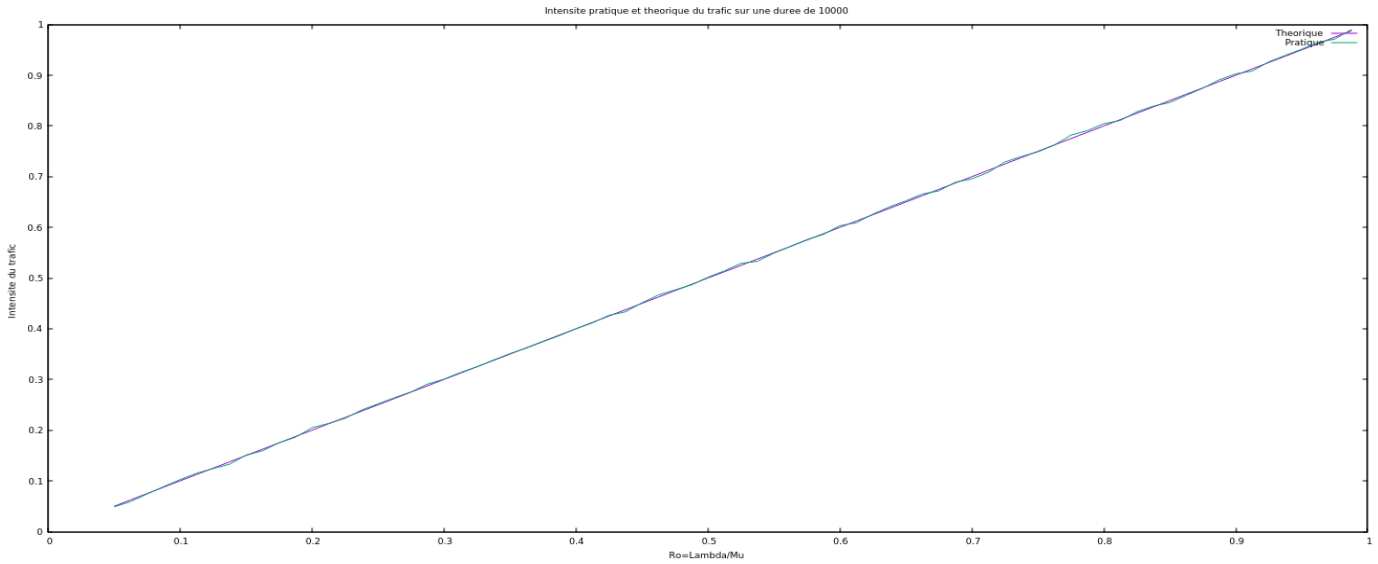
```
double lambda=1;
double mu=20;
//On reste dans le cadre d'une file d'attente stable
while(lambda<mu) {
	//nouvelle statistique
	Utile Simulation = new Stats(lambda,mu);
	//On simule sur un durée de 10000 sans debogage
	Simulation.simuler(10000,0);
	//On écrit les résultats dans le fichier data
	Simulation.ecrirefichier();
	//On varie le lambda(implique varie ro)
	lambda=lambda+0.25;
}
```

Tracé du fichier data:

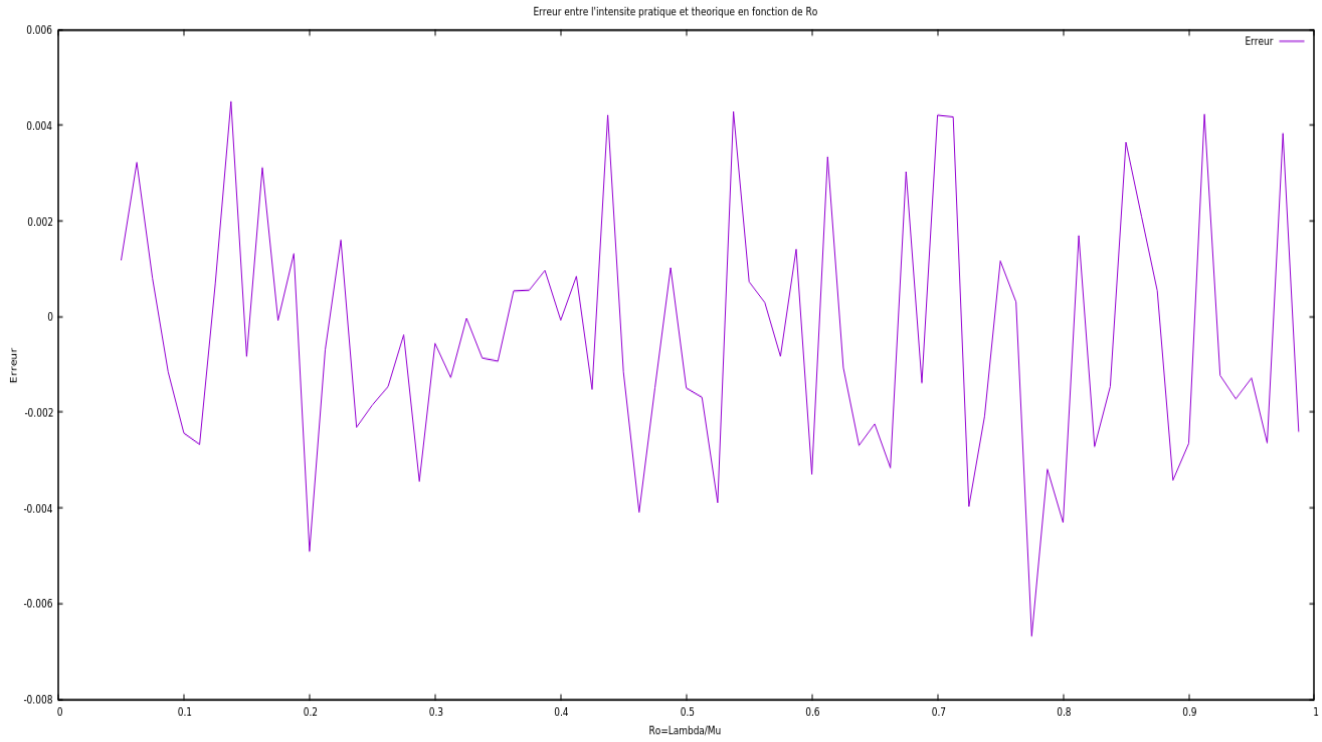
On trace tout d'abord l'intensité théorique et pratique en fonction de Ro=Lambda/mu.
On utilise pour ceci les commandes Gnuplot suivantes:

```
gnuplot> set title "Intensite pratique et theorique du trafic sur une duree de 10000 "
gnuplot> set xlabel "Ro=Lambda/Mu"
gnuplot> set ylabel "Intensite du trafic"
gnuplot> plot "Intensite.dat" u 1:1 w l
gnuplot> plot "Intensite.dat" u 1:1 w l title "Theorique" , "Intensite.dat" u 1:2 w l title "Pratique"
```

On voit bien que les courbes sont presque identiques.



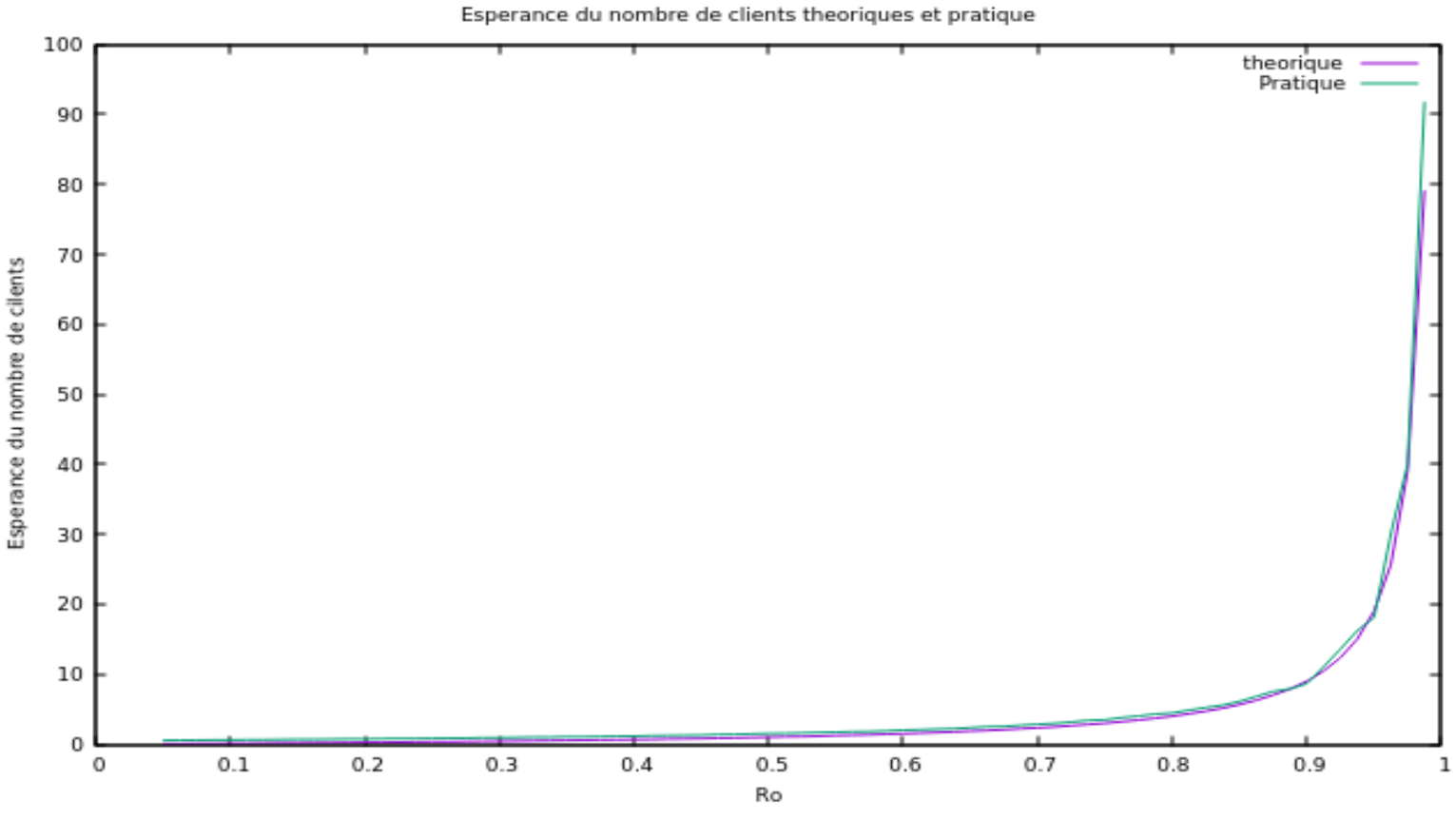
Afin de visualiser mieux l'erreur entre les deux courbes, on trace la différence entre la courbe qui est aussi en fonction du rapport Ro.



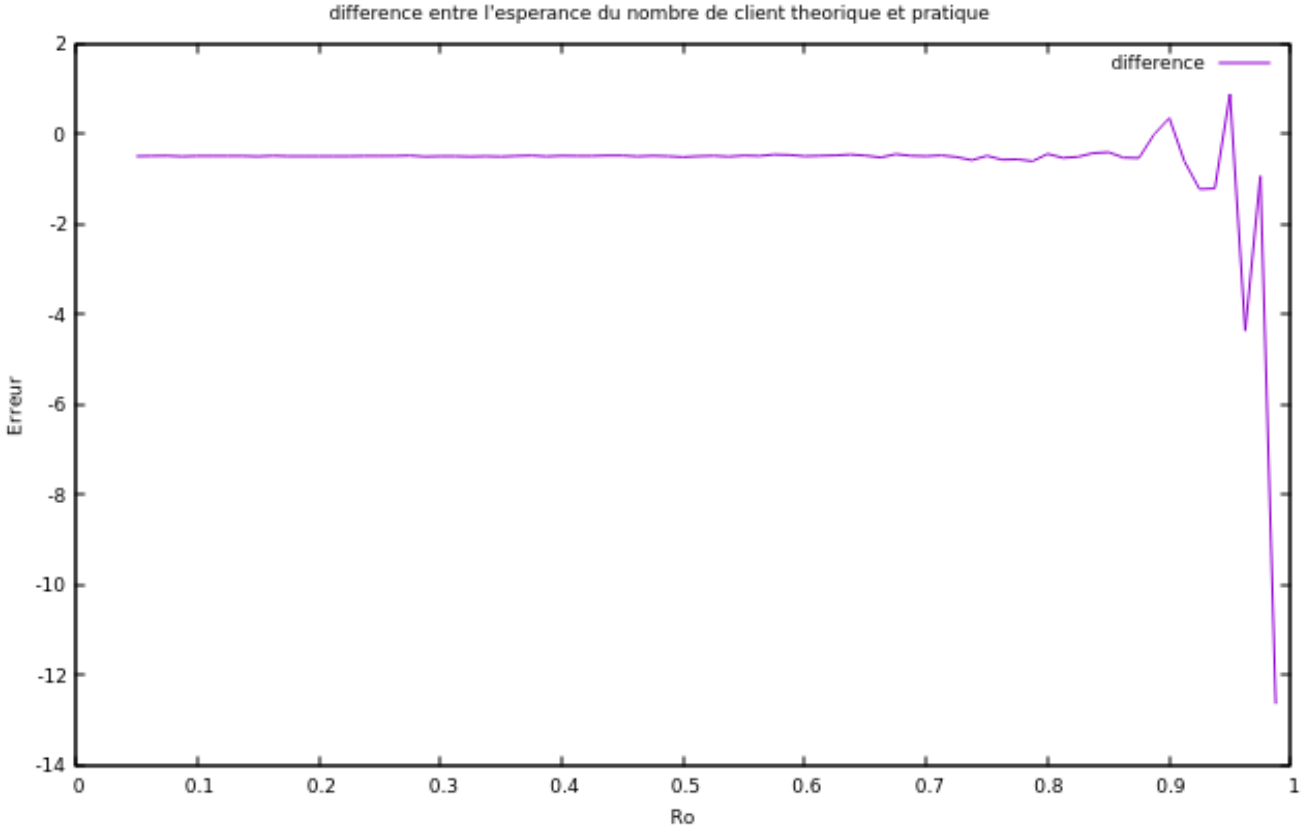
On voit bien que l'erreur est inclus dans [-0.006:0.006] et prend des valeurs aléatoires. (On peut dire qu'il suit une loi uniforme). On constate que le simulateur est top précis au niveau des résultats pratiques de l'intensité du trafic avec un erreur ±0.006.

- **Nombre de clients moyen dans le système:**

-Espérance mathématique du nombre de clients dans la file: $E[N] = \rho / (1 - \rho)$
-On trace les deux courbes correspondants aux nombre de clients moyen dans le système, en suivant les mêmes démarches qu'avant. Les résultats sont stockés dans le fichier "nbr_client_moy.dat".



Tout d'abord la courbe de l'espérance est bien celle d'une fonction f(x)=x/(1-x).
On voit que la courbe correspondant à l'espérance pratique se déstabilise pour des valeurs de Ro proches de 1 . Afin de verifier ceci, on trace la différence entre les deux courbes en fonction de Ro. On obtient la courbe suivante.



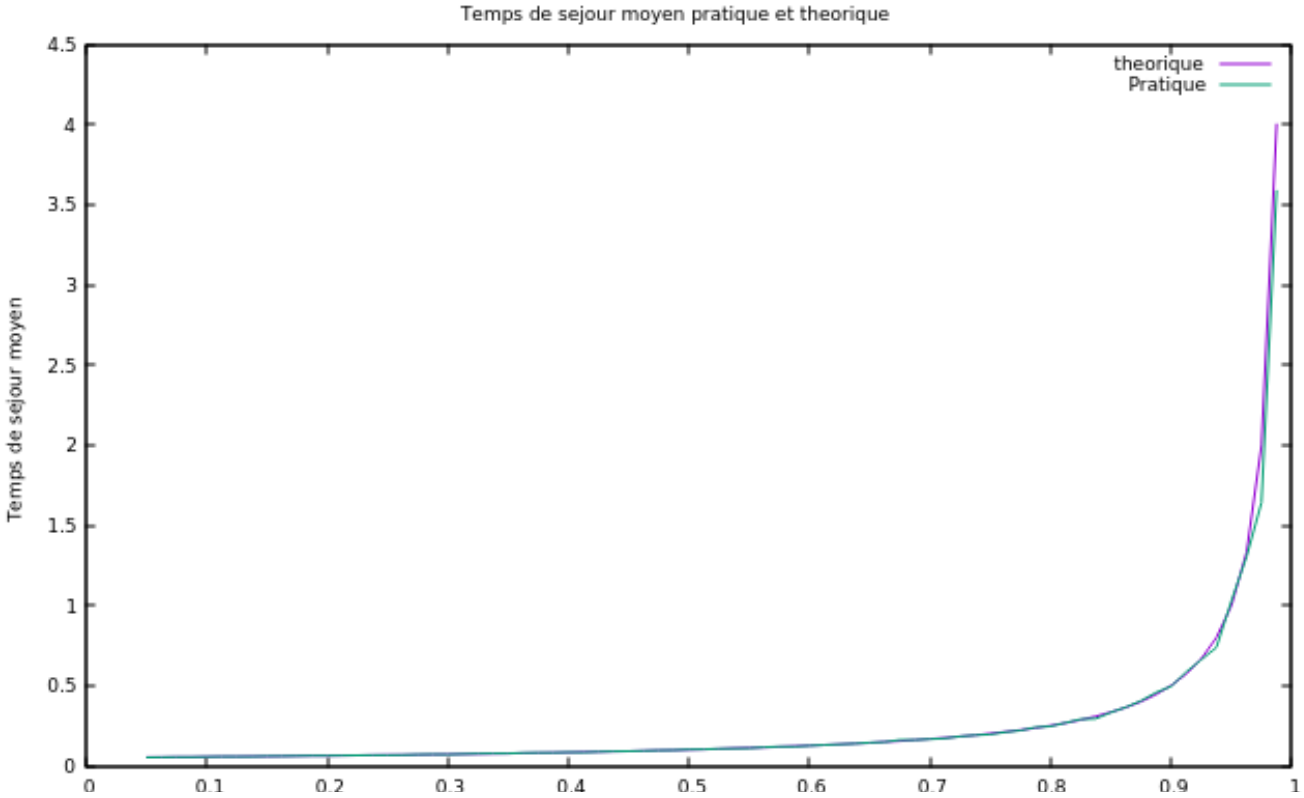
Comme constaté avant, l'espérance est bruité pour des valeurs de Ro proche de 1. On peut conclure que le simulateur ne permet pas d'offrir des résultats pratiques précis pour 0.8<Ro<1. Pareil pour Ro>1 ou la file est apparemment instable. En effet**, pratiquement**, lorsque Ro tend vers des valeurs de 1, implique que Lambda est presque égale a Mu. Le temps des inter-arrivés est le même que le temps de service, ceci provoque un chevauchement des événements lors de la simulation qui est difficile a gérer par le programme . D'autre part, **théoriquement**, on voit apparaître 1-RO dans le dénominateur qui engendre un point de discontinuité pour Ro=1.

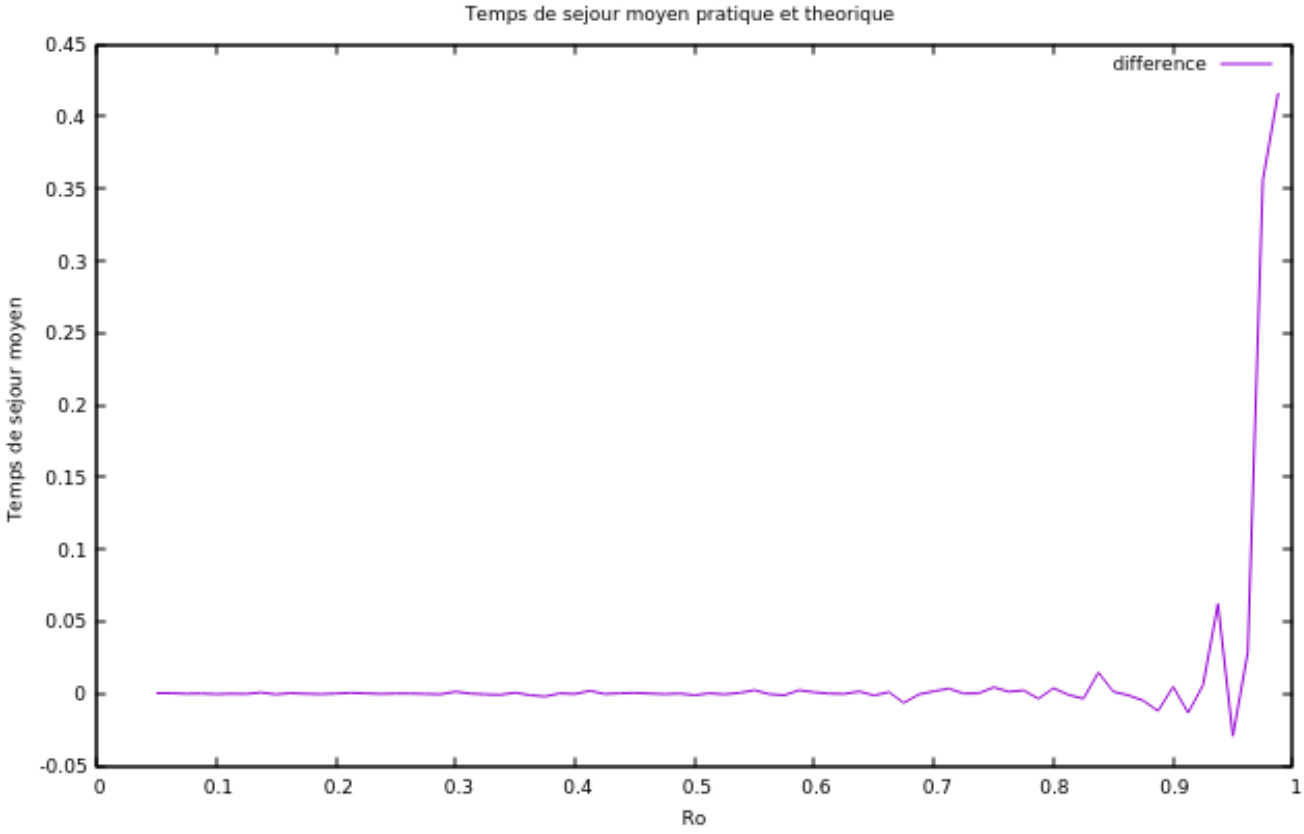
- **Temps moyen de séjour dans le système:**

-Le temps moyen de séjour dans le système est défini théoriquement par:

$$E[T] = \frac{1}{\mu(1 - \rho)}$$

-On trace les deux courbes correspondantes au temps moyen de séjour théorique et pratique, ainsi la différence entre les deux en fonction de Ro comme d'habitude. Les résultats de simulation sont contenus dans le fichier "temps_moy_sej.dat"





On trouve le même résultats que pour le nombre de clients moyen dans le système. Les résultats s'interprète de la même façon. Ceci est attendu puisque la relation entre le temps de séjour moyen et le nombre de clients moyen dans le système est donné par la formule de Little:

$$E[T] = \frac{E[N]}{\lambda}$$

❖ *Comment varie les résultats pratiques en fonction du taille d'échantillon:*

La taille d'échantillon, implique la durée de simulation (car nombre clients=duree_simulation*lambda) pourrait influencer sur les résultats pratiques. Pour vérifier ca, on fait varier l'erreur entre les résultats théoriques et pratiques en fonction de la taille de l'échantillon tout en fixant Lambda et Mu et donc le rapport Ro.

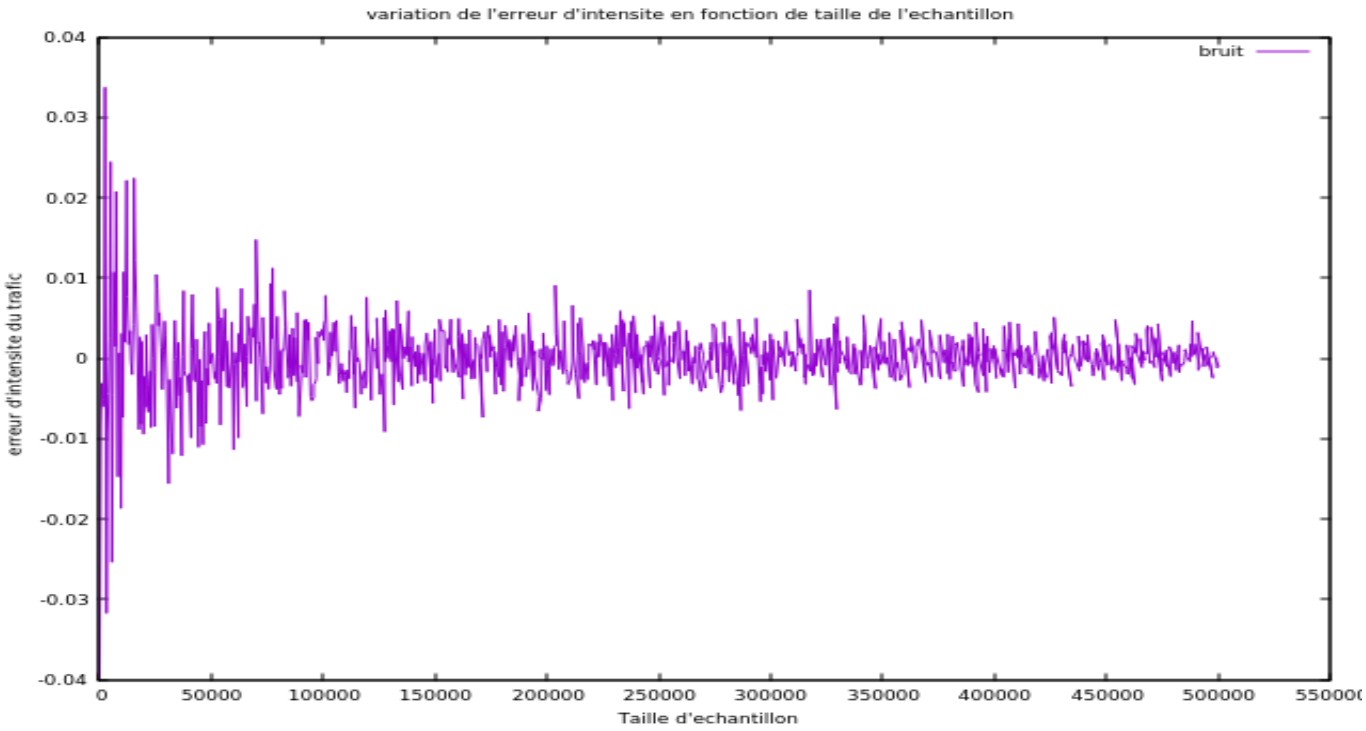
Pour faire ceci j'ai opté pour le code suivant en fixant (Lambda, Mu) en (5,6) :

```
//Variateur de temps de simulation
int variateur=1;
//On limite le variateur a 1000
while(variateur<1000) {
	//on fixe les parametres (lambda,mu) en (5,6)
	Utile Simulation = new Stats(5,6);
	//on simule pour des durée=variateur*100 sans debogage
	Simulation.simuler(variateur*100,0);
	//on ecrit les résultats dans un fichier
	Simulation.ecrirefichier();
	//on incréments le variateur par 1
	variateur++;
}
```

Ce code JAVA permet de faire 1000 simulation avec des durées de simulation variables allant de 100 jusqu'à 100000 , implique pour des échantillons a taille variable allant de 500 jusqu'à 500000. On exécute ce code sur les propriétés de notre simulateur pour voir comment varie l'erreur en fonction de la taille.

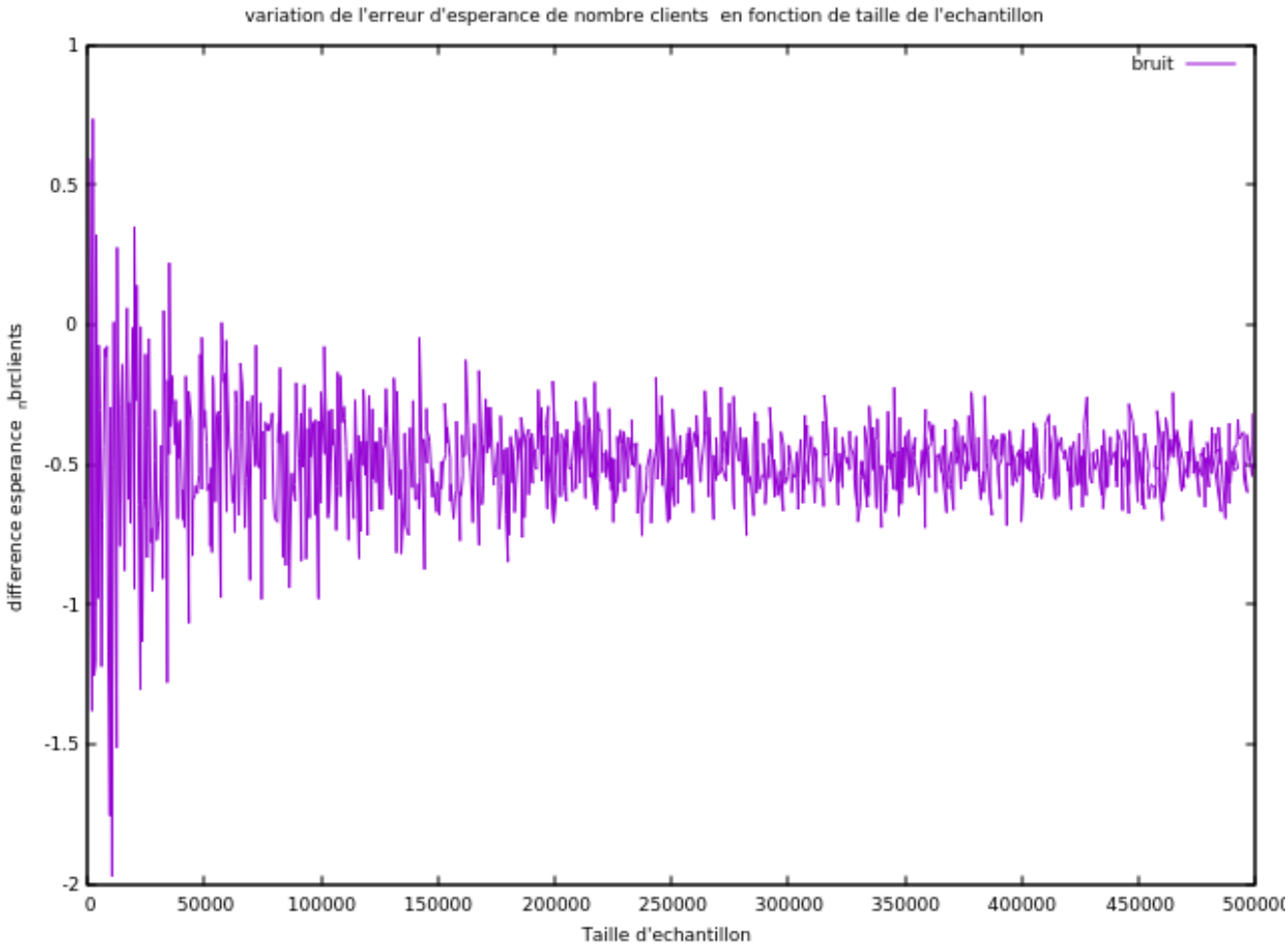
- **Erreur de l'intensité de trafic en fonction du taille d'échantillon:**

-On stocke les résultats concernant cette approche dans le fichier "intensité_echant.dat".



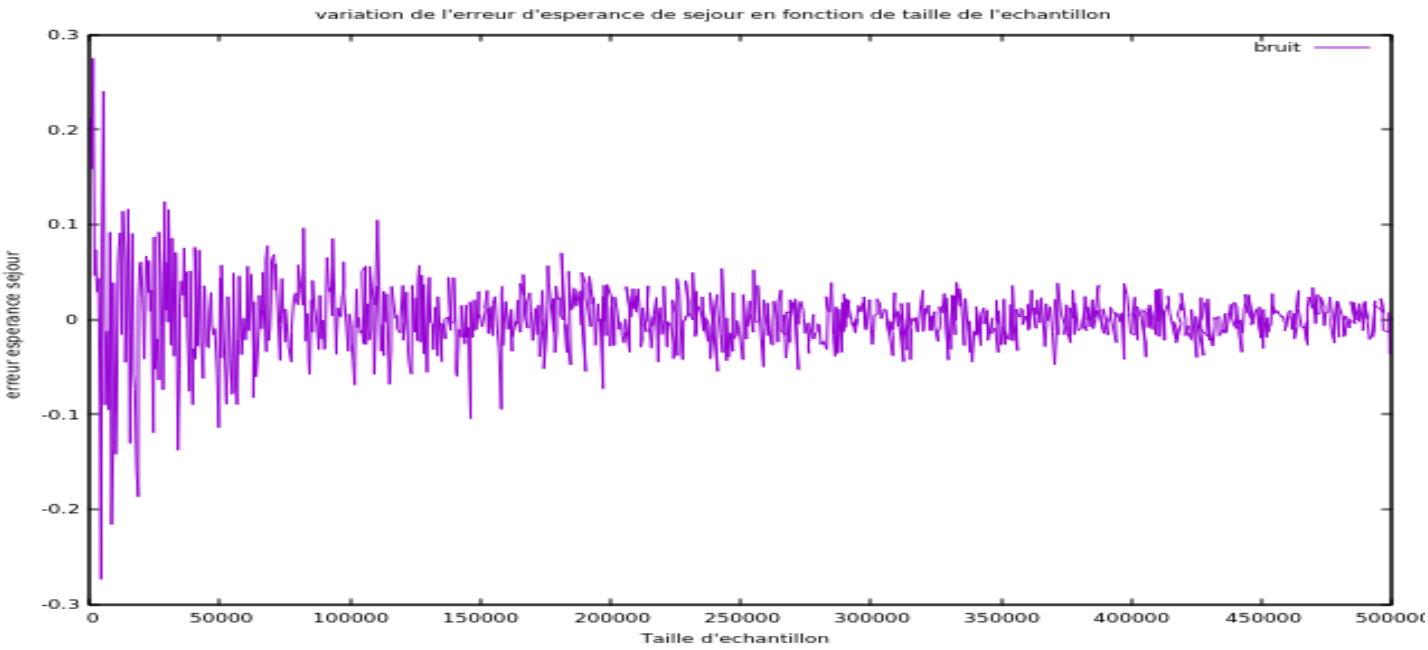
On constate bien que la taille d'échantillon permet de réduire un peu l'erreur entre l'intensité théorique et pratique du trafic.

- **Nombre de clients moyen dans le système:**



On trouve le même résultat comme avant , par contre l'erreur est quasi-symétrique par rapport a -0.5. On peut dire que l'espérance de l'erreur est -0.5. C'est à dire que le résultat pratique moyennement plus que le celui pratique avec un erreur de 0.5. Ainsi le simulateur présente un défaut sur ce niveau.

- **Temps moyen de séjour dans le système:**



On trouve bien le résultat. Notons bien que l'espérance de l'erreur est nulle pour l'espérance de séjour contrairement a celui de l'espérance du nombre de clients dans le système.

❖ ***Calcul d'intervalle de confiance pour l'espérance du nombre de clients :***

Problématique:

On a vu auparavant que l'espérance du nombre de client présente un défaut au niveau de l'erreur(l'espérance de l'erreur est 0.5) ce qui n'est pas normale. J'ai modifié mon programme plusieurs fois pour résoudre ce problème mais en vain. Finalement, j'ai décidé d'étudier à quel point peut on tolérer ce défaut en se basant sur l'intervalle de confiance.

Pour débuter:

Quand on réalise une expérience aléatoire, on observe bien sûr que les résultats obtenus ne sont pas toujours les mêmes, c'est la fluctuation d'échantillonnage. Mais on observe aussi que, plus on répète une expérience un grand nombre de fois, plus la régularité de la fréquence des résultats est grande. On définit les intervalles de fluctuation asymptotique et on en donne un exemple. On peut alors décider si on considère que des résultats obtenus lors d'une expérience sont dus au hasard (c'est-à-dire à la fluctuation d'échantillonnage), ou si on considère qu'ils sont statistiquement significatifs d'une différence avec le modèle choisi. Le but de ce calcul est de voir si il y'a de bonne chance que notre simulateur contient la vraie valeur du paramètre, en d'autres d'être dans l'intervalle de confiance.

Calculer l'intervalle de confiance:

Bien dit il s'agit de calculer l'intervalle de confiance d'une espérance . Pour ceci , on pourrait approcher notre échantillon par une loi normale de paramètres respectivement :

$$\text{Espérance } E[N] = \frac{\rho}{1-\rho} \text{ et variance } \sigma^2 = \frac{\rho}{(1-\rho)^2}$$

Conditions de l'utilisation de la loi normale:

-La condition pour cet approche est d'avoir un échantillon de grande taille.

-La variance du nombre du client moyen est connu. On pourrait calculer l'intervalle facilement. en utilisant la méthode qui approche l'espérance à une loi normale au cas d'une variance connue.

L'intervalle de confiance est définie pour un niveau de confiance de 95% pour (lambda=5,mu=6) par:

$$[E[N] + 1.96*\sigma\sqrt{N}, E[N] - 1.96 *\sigma\sqrt{N}]$$

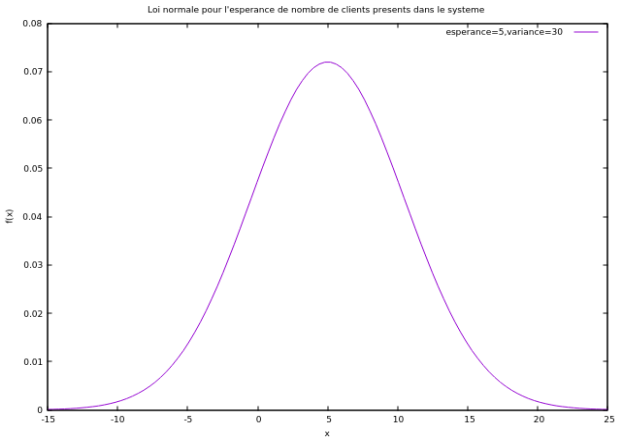
Le simulateur permet de calculer la variance pratique et théorique ainsi l'intervalle de confiance.

On exécute la même simulation plusieurs fois et on observe la fréquence des valeurs obtenus restantes dans l'intervalle de confiance. Cette fois-ci on fixe tous le paramètres(Lambda,Mu,duree_simulation).

```
while(i<1000) {
	//on fixe les paremtres (lambda,mu) en (5,6)
	Utile Simulation = new Stats(5,6);
	//on simule pour durée=100000 sans debogage
	Simulation.simuler(100000,0);
	//on ecrit les résultats dans un fichier
	Simulation.ecrirefichier();
	//on incréments par 1
	i++;
}
```

Intervalle de confiance du nombre moyen théorique de clients dans le système pour (lambda=5,Mu=6) :

[4.995199 : 5.004801] (Calculé par le code Java)



L'espérance du nombre de clients pratique était toujours en dehors de cet intervalle pour tous les simulation faites , ce qui était déjà prévue car l'espérance de l'erreur est -0.5 ce qui est vachement très grande par rapport a l'erreur de confiance de niveau 95% qui est:
[0.004801 : -0.004801] Ainsi on rejette l'hypothèse que notre simulateur est fiable.

❖ ***Conclusion:***

- Le simulateur est précis au niveau de calcul de l'intensité de trafic ou file d'attente avec un erreur ±0.006.
- Le simulateur présente un bruit pour l'espérance de nombre de clients et temps de séjour dans le système par rapport au résultats théoriques pour Ro=Lambda/Mu inclus dans [0.8:1], par contre pour Ro<0.8 l'erreur est presque nulle, rendant les résultats pratiques très proches à celle théoriques.
- Le simulateur a un défaut au niveau du calcul pratique de l'espérance du nombre de clients présents dans le système avec une marge d'erreur 0.5.
- Le simulateur donne des résultats un peu plus précises pour des tailles d'échantillon grande.

  Remarque:

  On a traité durant cette chapitre trois propriétés majeurs qui caractérisent la file d'attente M/M/1. On pourrait suivre les mêmes démarches pour les grandeurs: nombre de clients totale, débit et variance de l'espérance du nombre des clients .

  La figure suivante concrétise ces conclusions pour une file d'attente M/M/1 de paramètres (Lambda=6,Mu=10,Duree_simu=1000000,débogage=0):

```
<terminated> Main [Java Application] D:\Javaoracle\bin\javaw.exe (9 oct. 2017 à 01:38:04)
==========================RESULTATS THEORIQUES==================================
lambda<mu: file stable
Temps d'attente moyen ro=lambda/mu : 0.6
nombre de clients attendus (lambda x duree) :6000000.0
Prob de service sans attente (1 - ro) : 0.4
Prob file occupee : 0.6
Debit(lambda) : 6.0
Nombre de clients moyen (Esperance) dans le systeme : 1.4999999999999998
temps moyen de sejour  : 0.25
Variance du nombre de client dans la file d'attente: 3.749999999999999
Intervalle de confiance du nombre moyen de clients dans le systeme : [1.4987035 : 1.5012965]

==========================RESULTATS PRATIQUES==================================
Nombre de clients servis pendant toute la simulation: 6003308.0
Proportion clients avec attente : 0.6000664966715018
Proportion clients sans attente : 0.3999335033284982
Debit pratique : 6.003308
Nombre de clients moyen dans le systeme (Esperance) : 1.9996587714924197
Temps moyen de séjour : 0.249943443628963
Variance du temps d'attente dans le systeme : 4.009484380723309

                Temps de calcul des resultats théoriques et pratiques: 3.162 secondes
```

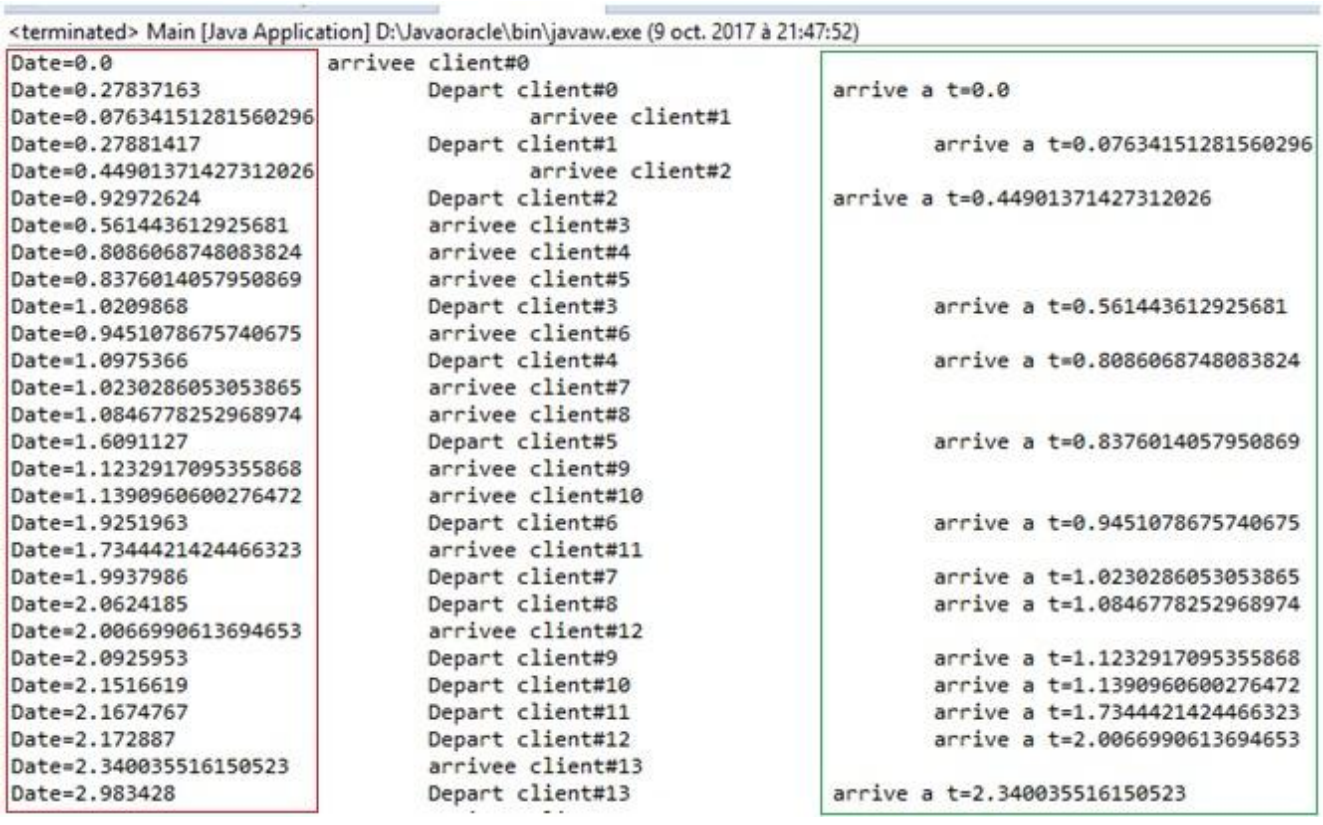
III-Performances du code JAVA:

Le programme utilise cinq classes utilisant des paternes assurant la confidentialité du programme. En effet on utilise des variables protégées (Protected variable) afin d'assurer que les données ne soit modifiables , et donc le programme est confidentiel et sécurisant pour les données de simulation.

- **Echéancier:**

La gestion d'échéancier et l'enchaînement des événements sont faites à l'aide de la classe bibliothèque *"Java.util.Vector"*. Ainsi, on utilise des vecteurs pour insérer les événements. En effet j'ai opté pour structure puisque c'est le meilleur choix en termes de lecture et insertion et suppression puisqu'elle accède en temps constant amortie au valeurs qui sont au fond du vecteur. Par contre cette structure est mauvaise pour insertion et suppression aléatoire. Du coup vue que mon code se base sur les valeurs qui sont au fond du vecteur pour les opération d'insertions et suppression , j'ai choisi cette structure.

Dans l'autre part, même si cette structure de vecteurs présente une bonne complexité au niveau du gestion des événements, on note qu'il y'a des problèmes au niveau chronologique des dates d'arrivée et départ comme le montre la figure ci_dessous par contre les dates d'arrivées et départ de chaque client sont respectées:



En effet le code JAVA traite les temps d'inter arrivées et de service des clients indépendamment tout en respectant les dates d'arrivée pour chaque client grâce au classes Client et FileAttente.

- **FileAttente :**

J'ai ajouté une classe "FileAttente" qui stocke les clients qui sont dans la queue a chaque instant, avec cette méthode notre code traite les événements par bloc (qui est le Queue) au lieu de traiter la file M/M/1 dans sa globalité permettant ainsi de réduire la complexité. Et cette classe utilise aussi la structure Vector pour les mêmes raisons cité avant, en plus les vecteurs sont généralement disposés de manière contiguë dans la mémoire, de sorte que la traversée est efficace parce que le cache de la mémoire du CPU est utilisé efficacement.

Vérifier les performances du simulateur:

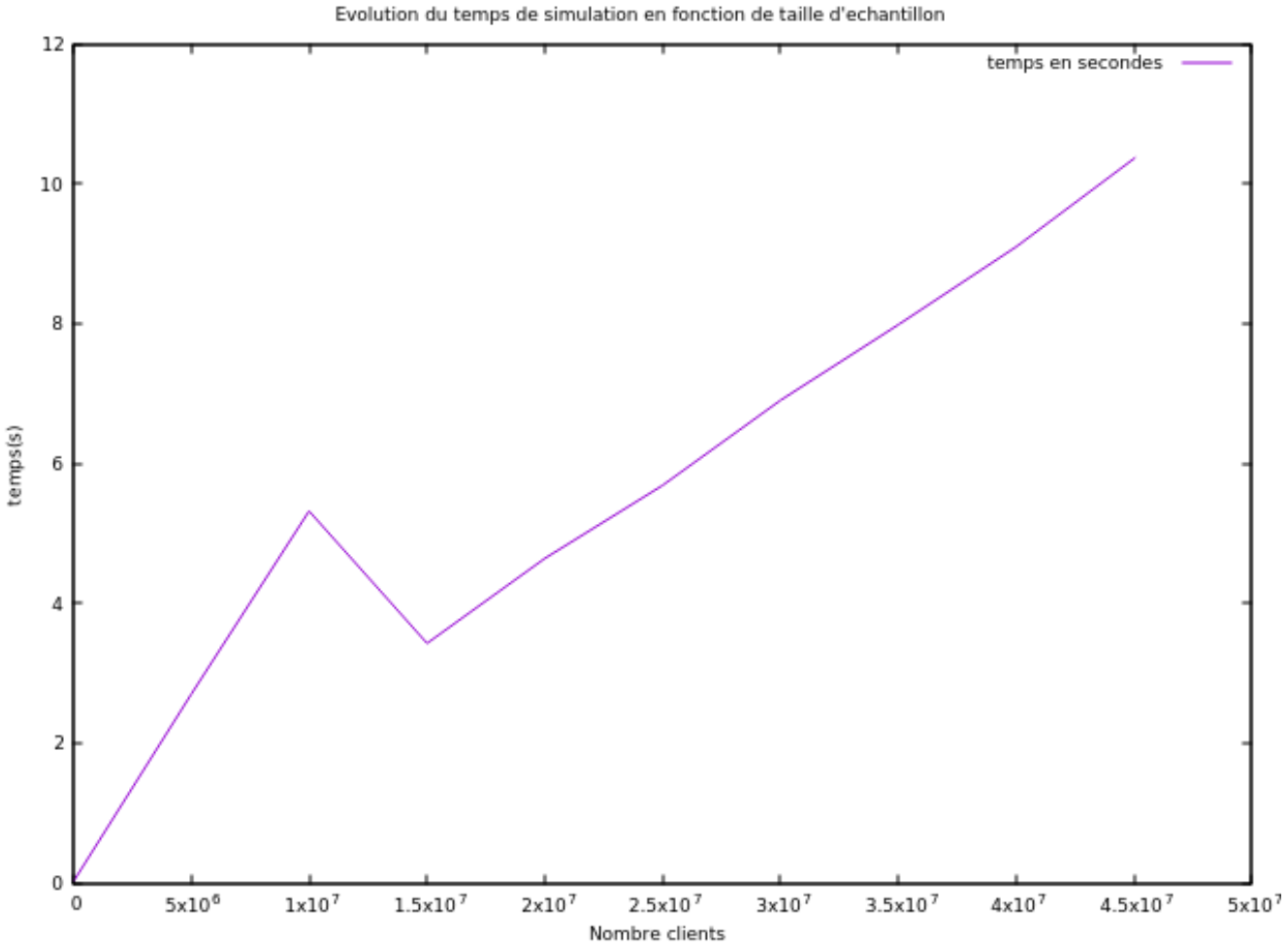
On exécute le simulateur pour (lambda=5,Mu=6) dans une durée de simulation égale à 10000000 sans débogage.

```
<terminated> Main [Java Application] D:\Javaoracle\bin\javaw.exe (9 oct. 2017 à 22:22:17)

                Temps de calcul des resultats théoriques et pratiques: 26.684 secondes


==========================RESULTATS THEORIQUES==================================
lambda<mu: file stable
Temps d'attente moyen ro=lambda/mu : 0.8333333333333334
nombre de clients attendus (lambda x duree) :5.0E7
Prob de service sans attente (1 - ro) : 0.16666666666666663
Prob file occupee : 0.8333333333333334
Debit(lambda) : 5.0
Nombre de clients moyen (Esperance) dans le systeme : 5.000000000000002
temps moyen de sejour  : 1.0000000000000002
Variance du nombre de client dans la file d'attente: 30.000000000000014
Intervalle de confiance du nombre moyen de clients dans le systeme : [4.9984818 : 5.0015182]

==========================RESULTATS PRATIQUES===================================
Nombre de clients servis pendant toute la simulation: 5.0005889E7
Proportion clients avec attente : 0.8334152803482806
Proportion clients sans attente : 0.1665847196517194
Debit pratique : 5.0005889
Nombre de clients moyen dans le systeme (Esperance) : 5.499463958140371
Temps moyen de séjour : 1.0001031425962492
Variance du temps d'attente dans le systeme : 30.20419558107403
```

Le temps de calcul bien vu est de 26.684s ce qui est pas mal pour un échantillon de taille qui vaut cinquante millions (Nombre de population d'un pays). On note bien que le programme calcule en plus l'espérance, ainsi des opération de calcules mathématiques (puissance carrée) sont faites à chaque instant.

Calcul graphique de complexité du code JAVA:

Afin de calculer la complexité, on opte pour la méthode graphique en traçant le temps que met le programme en fonction du nombre de clients dans la file d'attente.



On voit bien que le temps de simulation croit linéairement en fonction du nombre de clients dans le système . Ainsi la complexité est polynomiale.