# Rapport de stage

Emmanuel Rodriguez

20 août 2017

## 1 Introduction

On utilise des systèmes aléatoires dans de nombreux domaines de l'informatique comme les réseaux de communication, systèmes distribués, systèmes de calculs... Ces systèmes, utilisée pour étudier et améliorer la performance algorithmique distribuée, sont souvent composé de nombreux objets en interaction ce qui rend leur analyse exacte dificile.

Nous nous interesserons dans ce stage a des outils stochiastiques permetant de montrer que pour de nombreux systèmes, la performance d'un système composé de N objets a une limite quand N est grand.
L'objet de ce stage est s'iteresser a ces systèmes et leur convergeance. Ainsi qu'a développer un simulateur permetant de valider cette aproximation.

## 2 Chaînes de markov à temps continue

[3]

### 2.1 Définition

**Lois exponentielles** Une variable aléatoire $T$ à valeurs dans $\mathbb{R}_+$ suis la loi exponentiele de paramètre $\lambda \geq 0$, si sa fonction de répartition est donnée par $F(t) = (1 - e^{-\lambda t})$ sur $\mathbb{R}_+$. (et F(t)=0 sur $\mathbb{R}_-^*$).

**Processus de population markovien a temps continue** On considère le modèle de densité de population de Kurtz suivant :

— N le nombre d'individue répartis dans les diffétents états.
— un ensemble d'état fini $S$. On notera $n$ le nombre d'états.
— pour chaque état $i$ on note $X_i$ la population dans l'état $i$. On notera $X^{(N)} = (X_0, X_1, ..., X_{n-1})$.
— un ensemble de transitions avec pour chaque transition une fonction de transition associé. On représentera une transition par un vecteur $l \in \mathbb{R}^n$. Avec $\beta_l : \mathbb{R}^n \to \mathbb{R}^+$ tels que $X^{(N)}$ passe de l'état $x$ à $x + \frac{l}{N}$ avec un taux $N * \beta_l$.
— $\{X_i\}_{i \in S}$une distribution initiale sur l'ensemble des états.

Pour notre étude nous nous limiterons aux modèles ne possedant qu'un seul point atracteur $x*$.
On défini le drift :

$$f(x) = \sum_{l \in L} l\beta_l(x)$$

**convergence du modèle** Lorsque N tend vers l'infini, le système tend vers la solution de l'ODE (du drift). On note $X_0$ cette solution limite. On peur alors montrer $E(X^{(N)}) = X_0 + \frac{C}{N} + O(\frac{1}{N^2})$ avec $X_0$ le point fixe et $C$ un coefficient de correction dont nous allons chercher la valeur au cours de ce stage.

### 2.2 Simulation d'une CMTC

Pour la simulation, nous suivons le shémat suivant :

```
boucle:
    trouver la transition l qui a lieux en premier
    temps+= temps avant la transition
```

**Methode naive** Soit $l \in L$ un vecteur représentant une transition. Cette transition est associé au taux $\beta_l$ . Le système passe d'un état $X$ à $X + \frac{l}{N}$ avec le taux $N * \beta_l$ . On tire la v.a. $expo(N * \beta_l)$ pour obtenir le temps avant cette transition. Avec expo(i) une loi exponentielle de paramètre i. On tire donc une loi exponentielle pour chaque transition puis seul la transition qui a lieux en premier est prise en compte.

Mais il existe une methode plus rapide et equivalente :

**Amélioration** Pour savoir quelle transition aura lieux en premier on peut plus simplement choisir une transition $l$ avec proba : $\frac{\beta_l}{\sum_{i \in L} \beta_i}$, en tirant juste un reel entre 0 et 1 par exemple. Puis une fois la transition choisi on tire une loi exponentielle de paramètres $\sum_{i \in L} \beta_i$ pour obtenir le temps avant la transition.

**Réalisation d'un simulateur** J'ai codé en python le simulateur suivant prenant en entrée une chaine de markov a temps continue quelconque. La fonction comprend deux options qui sont les deux derniers arguments : on peut fixer la graine qui sert a génerer l'aléatoire du module random de python3. Ce qui sert a pouvoir refaire la même simulation avec exactement les mêmes conditions.(il faut alors ajouter l'argument : $fix = 1234$ par exemple)

La seconde option permet d'écrire dans un fichier (dont on précise le nom avec l'argument : $file ='' mon\_fichier.out''$) le tableau de donnée representant la siulation efectuée.

La simulation renvoie un tableau data contenant : $data[0]$ contient le tableau des temps. $data[1]$ contient les valeursdes v.a. $(X_0, X_1, ..., X_{n-1})$. Par exemple $data[1][i] = [X_0, X_1, ..., X_{n-1}]$ au temps $data[0][i]$

```
def simu_cmtc(taux,L,X,N,time,file="",fix=-1):
    n=len(X)
    nb_trans=len(L)
    t=0

    if fix!=-1:
        seed(fix)

    if file!="":
        data=open(file,"w")
    else:
        data=[[0],[X]]

    while t<time:
        a=random()

        L_poids=array([taux(i,X) for i in range(nb_trans)])
        S=sum(L_poids)

        cumul=0
        for i in range(nb_trans):
            tmp=cumul + L_poids[i]/S
            if a < tmp:
                l=i
                break
            cumul=tmp

        X = X+(1./N)*L[l]
        if max(L_poids)==0:
            break

        expo=expovariate(N*S)
        t+=expo

        if file=="":
            data[0].append(t)
            data[1].append(X[:]*expo)
        else:
            data.write(str(t)+" ")

            for i in range(n):
                data.write(str(X[i])+" ")
            data.write('\n')

    if file=="":
        data[1] = np.array(data[1])
        return(data)
    else:
        data.close()
        return(0)
```

### 2.3 Système d'infection : SIR

On considère le système suivant :

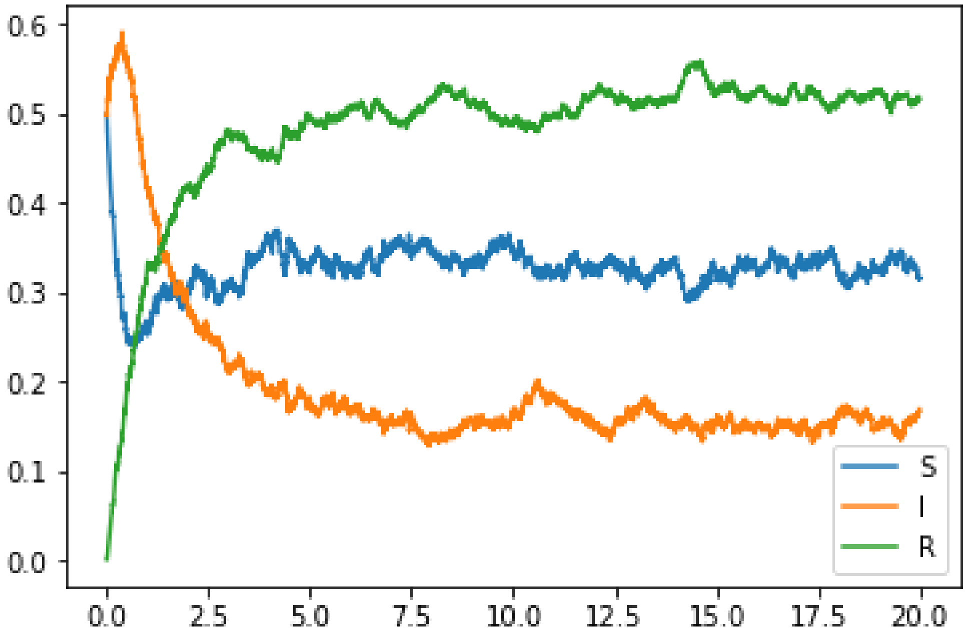
$$
\begin{aligned}
S &\rightarrow I \ \ taux = 3x_0x_1 + \frac{x_0}{2} && (1)\\
I &\rightarrow R \ \ taux = x_1 && (2)\\
I+R &\rightarrow S \ \ taux = 2x_1x_2 && (3)
\end{aligned}
$$

On note $X = [x_1; x_2; x_3]$ les proportions de S I R :

$$
\begin{aligned}
x_0 &= \frac{population(S)}{population\ totale}\\
x_1 &= \frac{population(I)}{population\ totale}\\
x_2 &= \frac{population(R)}{population\ totale}
\end{aligned}
$$

On simule cet exemple de la façon suivante avec le simulateur codé précedement avec comme condition initiale $X_0 = [\frac{1}{2}; \frac{1}{2}; 0]$

```
liste_transitions = [ array(l) for l in [ [-1.,+1.,0.], [0.,-1.,1.],
[2.,-1.,-1.] ] ]

def taux ( i, x):
    if i==0:
       return 3*x[0]*x[1]+0.5*x[0]
   elif i==1:
       return x[1]
   else:
       return 2*x[1]*x[2]

labels=["S","I","R"]
X=array([0.5,0.5,0.])

simu_cmtc(taux,liste_transitions,X,1000,20,"data.out",1234)

data=file_to_array("data.out",3)

for i in range(3):
   plot(data[0],data[1][:,i], label=labels[i])

legend()
show()
```



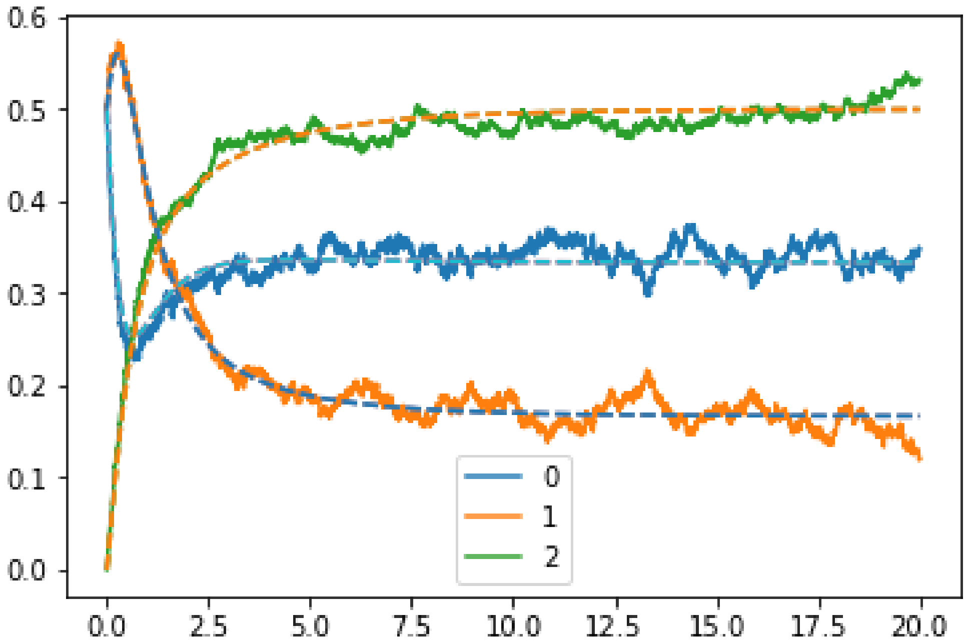

On obtiens bien un resultat qui parait coherent au premier coup d'oeil. Pour vérifier cette simulation on résout l'equation différentielle qui régit ce système :

$$f'(x) \quad = \quad \sum_{x \in L} x\beta_x$$

```
  def print_simu(taux, liste_transition ,X,N,time, fix=-1):
     data=simu_cmtc(taux, liste_transition ,X,N,time,"", fix)
     K=len(X)
     for i in range(K):
         plot(data[0], data[1][:, i], label=str(i))
     def drift(x):
     return (sum([liste_transition[i]*taux(i,x)
for i in range(len(liste_transition))],0))

     t = linspace(0,time,1000)

     for i in range(K):
         x = odeint( lambda x,t : drift(x), X, t)
         plot(t,x,'--')
     legend()
     show()
```

cette fonction nous permet donc de tracer sur la même courbe la simulation et la solution théorique du système. Avec les quelques lignes de code suivantes on obtient :

```
print_simu(taux, liste_transitions ,X,1000,20)
```




On a donc la confirmation que pour ce système la simulation est coherente avec la résultet théorique. (l'ordre de grandeur et le signe sont equivalnts).

# 3 Calcule du coefficient en 1/N

## 3.1 Par simulation

Pour calculer le coefficiet en 1/N on fait la moyenne des points fixe obtenues lors de n simulation puis on soustrait a ça le points fix calculé théoriquement : $C = \frac{\sum_{i=0}^{n-1} E(X_i^{(N)})}{n} - X_0$

Pour calculer le point fixe par simulation on fait une moyenne sur le dernier tier des termes d'une simulation longue (assez longue pour qu'on ateigne largement le point fixe).

Mais le résultat de $C$ ne corespond pas! un autre terme d'erreur en $O(\frac{1}{N}$ est aparue lors du calcul des points fixes par simulation. La moyenne calculé des termes de la simulation ne prend pas en compte le temps entre chaque transition d'où ce terme en $O(\frac{1}{N})$.

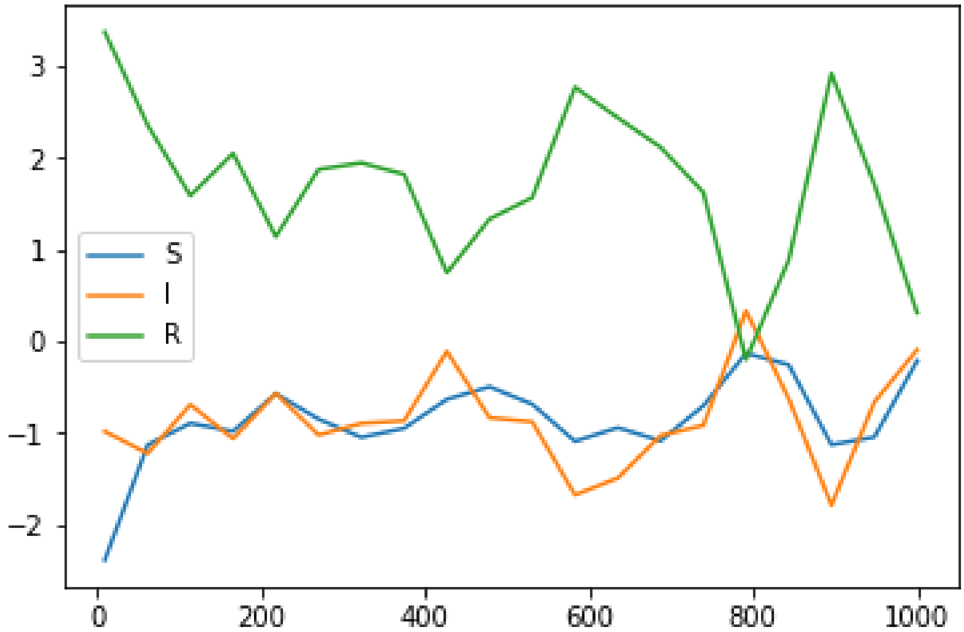
Pour y remédier on calcule une moyenne pondéré par le temps entre chaque transition.

```
def moy_pond(data,deb,fin,i):
    m=0
    for j in range(deb,fin):
        m+= data[1][j][i]*(data[0][j]-data[0][j-1])
    m=m/(data[0][fin-1]-data[0][deb-1])
    return(m)
```

on test donc de calculer ce coefficient sur un grand nombre de simulations et pour des valeurs différentes de N. On est censsé obtenir un résultat indépendant de N.

```
N=linspace(10,1000,20)
diff=array([[0. for i in N] for j in range(3)])
C_i=array([[0. for i in N] for j in range(3)])
X_0=fixed_point(taux,liste_transitions,3)

for n in range(len(N)):
  for j in range(100):
    data=simu_cmtc(taux,liste_transitions,X,N[n],50)
      for i in range(3):
      diff[i,n]+=(moy_pond(data,int(len(data[1])/2),
                  len(data[1]),i)-X_0[i])/100.
  for i in range(3):
    C_i[i,n]=N[n]*diff[i,n]

for i in range(3):
  plot(N,C_i[i,:])
legend(('S','I','R'))
show()
```

On obtient alors :




On observe un bruit important pour les grandes valeurs de N ce qui est surrement du aux erreurs de calculs sur les réels avec python. On observe donc $x_s = -1$ $x_I = -1$ $x_R = 2$ avec un bruit important pour 100 simulations par point. le temps de calcul pour ce résultat est d'environ 40 minutes (avec un code non optimisé).

## 3.2 Comparaison avec le calcul théoriquede ce coefficient

On peut montrer que le coefficient d'erreur en $\frac{1}{N}$ vaut exactement : [2]

$$C_i = \frac{1}{2} * \sum_j ((A^{-1})_{i,j} * \sum_{k_1,k_2} (B_j)_{k_1,k_2} W_{k_1,k_2})$$

avec :

— $A_{i,j} = Df(x^*)_{i,j}$

— $(B_j)_{k,l} =$

— $W$ est la solution de l'équation de Lyapunov suivante : $AX + XA^T + Q = 0$

— Avec $Q = \sum_{l \in L} Q_l$

— et $(Q_l)_{n,m} = -l_n l_m \beta_l(x^*)$

L'implémentation de ce calcule est réalisé a l'aide du module de calcul symbolique de python : sympy.

```python
def theorique(taux, liste_transitions ,n,dim):
  number_transitions = len(liste_transitions)
  X_0 = fixed_point(taux, liste_transitions ,n)

  Var=array([sym.symbols('x_{}'.format(i)) for i in range(n)])

  #print(len(X_0))
  f_x=array([0 for i in range(n)])
  for i in range(number_transitions):
      f_x = f_x + liste_transitions[i]*taux(i,Var)

  if dim==n-1:
      for i in range(n):
          f_x[i]=f_x[i].subs(Var[-1],1-sum(array([Var[i]
                                                for i in range(n-1)]))))

  A=array([[sym.lambdify(Var  ,sym.diff(f_x[i],Var[j]))(*[X_0[k]
            for k in range(n)])
          for j in range(dim)]
         for i in range(dim)])

  B=array([[[sym.lambdify(Var,sym.diff(f_x[j],Var[k],Var[l]))(*[X_0[i]
            for i in range(n)])
           for l in range(dim)]
          for k in range(dim)]
         for j in range(dim)])

  Q=array([[0. for i in range(dim)] for j in range(dim)])

  for l in range(number_transitions):
      Q += array([[liste_transitions[l][p]*liste_transitions[l][m]*taux(l,X_0)
                for m in range(dim)]
               for p in range(dim)])


  #print('A=',A)
  #print('Q=',Q)
  #print('B=',B)

  W = solve_lyapunov(A,Q)
  #print('W=',W)
  #print('sumW=',sum(W,0))

  A_inv=inv(A)

  C=[ 0.5*sum(array([A_inv[i][j]*sum(array([[B[j][k_1][k_2]*W[k_1][k_2]
                                        for k_2 in range(dim)]
                                       for k_1 in range(dim)]))
              for j in range(dim)]))
      for i in range(dim)]
  for i in range(len(C)):
      C[i]=sum(C[i])
  return(C)
```

Cette fonction nous permet donc de calculer le coefficient de manière théorique. Mais les systèmes étudiés ont souvent cette equation de vérifié :

$$\sum_{x \in X} x = 1$$

le système SIR par exemple vérifie $x_S + x_I + x_R = 1$. donc on étudie n système de dimention 2 et non 3. Il faut donc le préciser a l'entrée de la fonction : $n = 3$ *et* $dim = 2$.

```
theorique(taux, liste_transitions ,3 ,2)
```

On obtient : $[-0.75000000000000111, \ -0.96428571428571352]$ Ce qui corespond bien aux valeurs calculés par simulation. Ce calcul n'a pris que quelques secondes a calculer comparé aux 40 minutes du résultat approximatif des simulations.

Pour un exemple simple comme le système SIR on a déjà des problèmes de temps de calcul par simulation. La formule précedente facilite donc grandement le calcul du coefficient en 1/N.

# 4 Exemples étudiés

## 4.1 Bianchi's formula

On s'interesse ici a un modèle [4] pour le protocole 802.11 MAC. On le modelise a l'aide d'une chaine de markov. On obtient la chaine de markov aux transitions suivantes :

$$\begin{aligned}
x &\rightarrow x + e_0 - e_k \ avec\ un\ taux\ q_k x_k \prod_{m=0}^{K} (1 - \frac{q_m}{N})^{N x_m - e_k} \ for\ k \in \{1...K-1\} \\
x &\rightarrow x + e_{k+1} - e_k \ avec\ un\ taux\ q_k x_k (1 - \prod_{m=0}^{K} (1 - \frac{q_m}{N})^{N x_m - e_k}) \ for\ k \in \{1...K-1\} \\
x &\rightarrow x + e_0 - e_K \ avec\ un\ taux\ q_K x_K
\end{aligned}$$

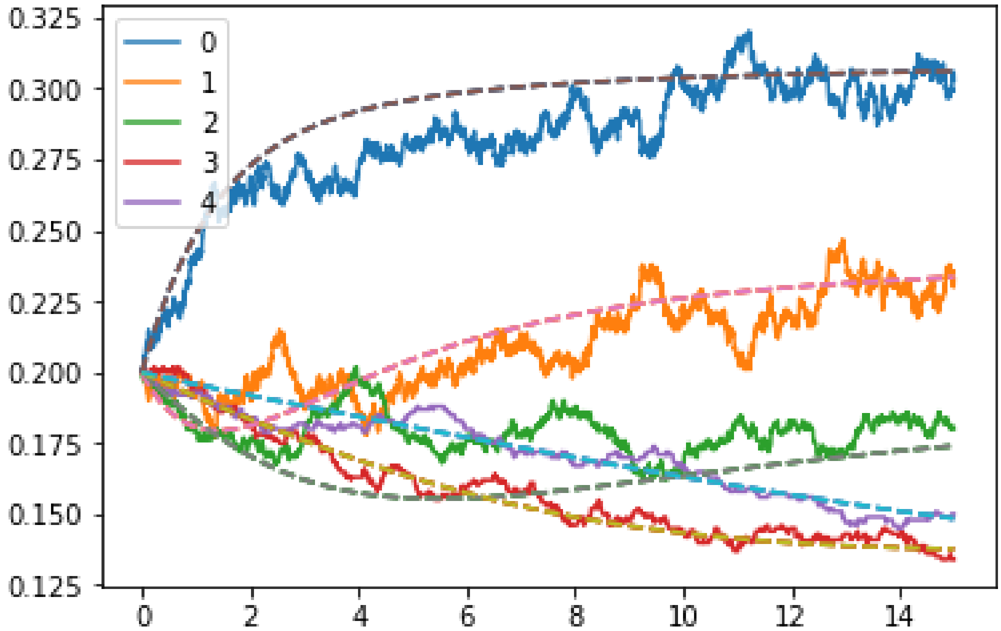
On entre alors le système de la manière suivante dans notre simulateur :

```
N=100
K=5
Var_x=array([sym.symbols('x_{}'.format(i)) for i in range(K)])
#Var_q=array([sym.symbols('q_{}'.format(i)) for i in range(K)])
Var_q=array([2**(-i) for i in range(K)])

def mult_liste(Liste):
    S=1
    for i in Liste:
        S*=i
    return(S)

E=[array([1 if i==j else 0 for j in range(K)]) for i in range(K)]

liste_transitions =[]
taux=[]

liste_transitions.append(E[1]-E[0])
taux.append(Var_q[0]*Var_x[0]*
(1-mult_liste([(1-Var_q[m]/N)**(N*Var_x[m]-(0 if m!=0 else 1))
                                              for m in range(K)])))

for i in range(1,K-1):
    liste_transitions.append(E[0]-E[i])
    taux.append(Var_q[i]*Var_x[i]*
    mult_liste([(1-Var_q[m]/N)**(N*Var_x[m]-(0 if m!=i else 1))
                                              for m in range(K)]))
    liste_transitions.append(E[i+1]-E[i])
    taux.append(Var_q[i]*Var_x[i]*
    (1-mult_liste([(1-Var_q[m]/N)**(N*Var_x[m]-(0 if m!=i else 1))
                                                 for m in range(K)])))

liste_transitions.append(E[0]-E[K-1])
taux.append(Var_q[K-1]*Var_x[K-1])

def fct_taux(i,X):
    return sym.lambdify( Var_x,taux[i])(*X)
```

Puis on entre les conditions initiales et on trace le tout :

```
X=[1/K for i in range(K)]
print_simu(fct_taux,liste_transitions,X,1000,15)
```

On obtient la figure suivante :




Le tracé suis bien la solution de l'equation différentielle qui régi le système. Un simple tracé comme celui-ci met déja quelques minutes a se faire. Le calcul du coeficient recherché en 1/N est trop long a calculer avec des simulations.

```
theorique(fct_taux, liste_transitions,5,4)
```

Le calcul a l'aide de la formule se fait en quelques minutes et on obtient :

$$-0.038851395750410606, 0.075038237961333909, 0.037170741481473424, -0.016174636251891139$$

Ce système converge donc rapidement car les coefficient sont de l'ordre de $10^{-2}$

## 4.2 Velo'v

La répartition et l'évolution du nombre de vélo par stations de velo'v peut facilement se traduire en une chaine de markov. On peut utiliser un tel modèle [1] pour optimiser le nombre de velo par station par exemple.

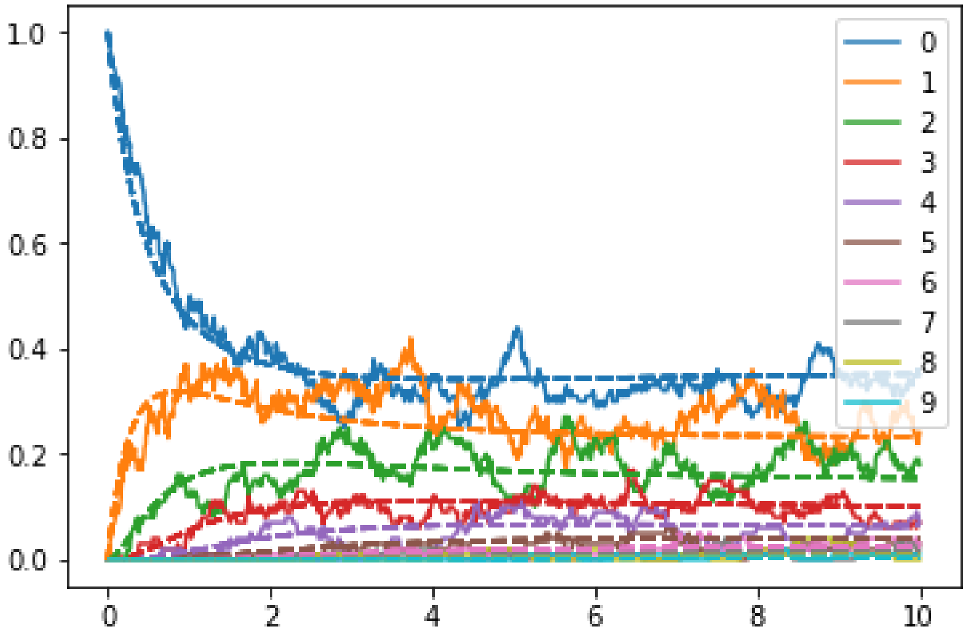
```
K=10
N=20
s=3
Lambda=1
Mu=0.5

y=array([sym.symbols('y_{}'.format(i)) for i in range(K)])
E=[array([1 if i==j else 0 for j in range(K)]) for i in range(K)]
liste_transition=[]
taux=[]

for i in range(K-1):
    liste_transition.append(E[i+1]-E[i])
    taux.append(Mu*y[i]*(s-sum([n*y[n] for n in range(K)])))

for i in range(1,K):
    liste_transition.append(E[i-1]-E[i])
    taux.append(Lambda*y[i])

def fct_taux(i,X):
```

```
    return(sym.lambdify(y,taux[i])(*X))
```

Il nous reste plus qu'a choisir une distribution initiale et a tracer l'évolution.

```
X=[1,0,0,0,0,0,0,0,0,0]
print_simu(fct_taux, liste_transition ,X,100,10)
```

On obtient alors la courbe suivante :




```
X=[1,0,0,0,0,0,0,0,0,0]
theorique(fct_taux, liste_transition ,K,K-1)
```

On obtient alors :
-0.10292176084411847,
0.039044474621688807,
0.064763975313775535,
0.048762058614485009,
0.023866044952782651,
0.002612691487885844,
-0.01170356394907981,
-0.019540234122908881,
-0.022526472856673283
On obtien un coefficient de l'ordre du centième.

# 5 projet github

Le travail efectué pendant la durée du stage est contenu dans un projet github[5] contenant :

## 5.1 Librairie python

L'ensemble du code est rassemblé dans une librairie python3 qui contient les fonctions suivantes :
— $simu\_ctmc(fct\_taux, liste\_transition, X_0, N, temps\_final,' mon\_fichier.out', fix\_seed)$ qui simule notre CMTC.
— $file\_to\_array('mon\_fichier.out', len(X))$ qui permet de récuperer les données si on les a stockés dans un fichier précedement.

— $fixed\_point(taux, liste\_transition, n)$ qui renvoit un vecteur des points fixes du système calculés a l'aide de odeint.
— $theorique(taux, liste_transition, n, dim)$ qui calcule le coeficient d'erreur $C$ de manière théorique.

### 5.2 Notebook d'exemples

Ce fichier ipython contient les exemples étudiés pendant le stage et peut donc servir de notice pour l'utilisation du simulateur et des autres fonctions.

## 6 outils utilisés

### 6.1 Org-mode dans emacs

Au cours de ce stage j'ai découvert org, un modude que l'on peut greffer a emacs servant a écrire un journal de bord de manière simple et propre.

```
sudo apt-get install emacs25 org-mode ess r-base auctex
```

### 6.2 jupyter-notebook

Le notebook jupyter est une façon simple et facilement lisible de coder en python par exemple.(en manipulant ici un fichier ipython)

```
sudo apt-get install jupiter-notebook ipython3 ipython3-notebook
```

### 6.3 librairies python3

Dans le code écrit j'utilise les fonctions suivantes provenant de librairies python3 :

```python
from random import random, seed, expovariate
from pylab import array
from numpy import linspace, zeros
from matplotlib.pyplot import *
import sympy as sym
from scipy.integrate import odeint
from scipy.linalg import solve_lyapunov
from numpy.linalg import inv
```

## Références


[1] Nicolas Gast Christine Fricker. Incentives and redistribution in homogeneous bike-sharing systems with stations of finite capacity. 2014.

[2] Nicolas Gast. Can we beat mean-field approximation ? May 15,2017.

[3] Hervé Guiol. Processus aléatoires. 2015/2016.

[4] Y. Jiang J.-w. Cho, J.-Y. Le Boudec. On the asymptotic validity of the decoupling assumption for analyzing 802.11 mac protocol. *IEEE Transactions on Information Theory 58.11*, 2012.

[5] Emmanuel Rodriguez. Random system simulation and calculs. *https ://github.com/manurod/Random-system-simulation-and-calculs*, 2017-09-07.