Compte rendu : TP Réseau d'Accès Radio

Quentin Goulas, Lucas Hocquette

This report presents the scripts used and results obtained on the Lab of the Wireless Communications class, focusing on the notions of receivers. Aside from additional comments, all reported scripts are identical to the scripts provided in the Matlab Live Script document provided as support. All plots are direct outputs from each script.

Ce rapport présente les scripts utilisés et les résultats obtenus durant le Travail Pratique de Réseau d'Accès Radio. Mis à part des commentaires additionnels, tous les scripts présentés dans ce document sont identiques aux scripts fournis dans le Jupyter Notebook en guise de support. Tous les graphes sont des outputs directs des scripts fournis.

Librairies requises

```
import numpy as np
import matplotlib.pyplot as plt
```

1 Capacité d'un système CDMA

1 On définit la fonction `sample_users` qui prend en entrée un nombre d'utilisateurs K et un rayon R, et retourne les positions de K utilisateurs uniformément répartis dans un cercle de rayon R.

```
def sample_users(K,R):
    v = np.random.uniform(low=0,high=R**2,size=K)
    theta = np.random.uniform(low=0,high=2*np.pi,size=K)
    r = np.sqrt(v)
    x,y = r*np.cos(theta), r*np.sin(theta)
    return x,y
```

On obtient les échantillonnages présentés en figure 1.

2

```
r = 1
W = 3.84*10**6
theta = 0.4
sigma2 = 10**(-104/10)*1e-3
P = 10**(40/10)*1e-3

def measure_achievement_ratio(K,R,gamma,n_avg=1):
    x,y = sample_users(K,r)
    d = np.sqrt(x**2+y**2)
    L = -128.1 - 37.6*np.log10(d)
    l = 10**(L/10)
    history = np.zeros(n_avg)
```

```
    for i in range(n_avg):
        h = np.random.exponential(0.5,K)
        g = l*h
        p = P/K
        SINR = W/R*p*g/(theta*(K-1)*p*g+sigma2)
        history[i] = np.mean(SINR>=gamma)

    return history

print(f'The percentage of users for which the decoding condition is satisified is : {measure_achievement_ratio(20,32*1e3,10**(7/10))[0]*100}%')
```

Avec la configuration donnée, 100% des utilisateurs satisfont la condition.

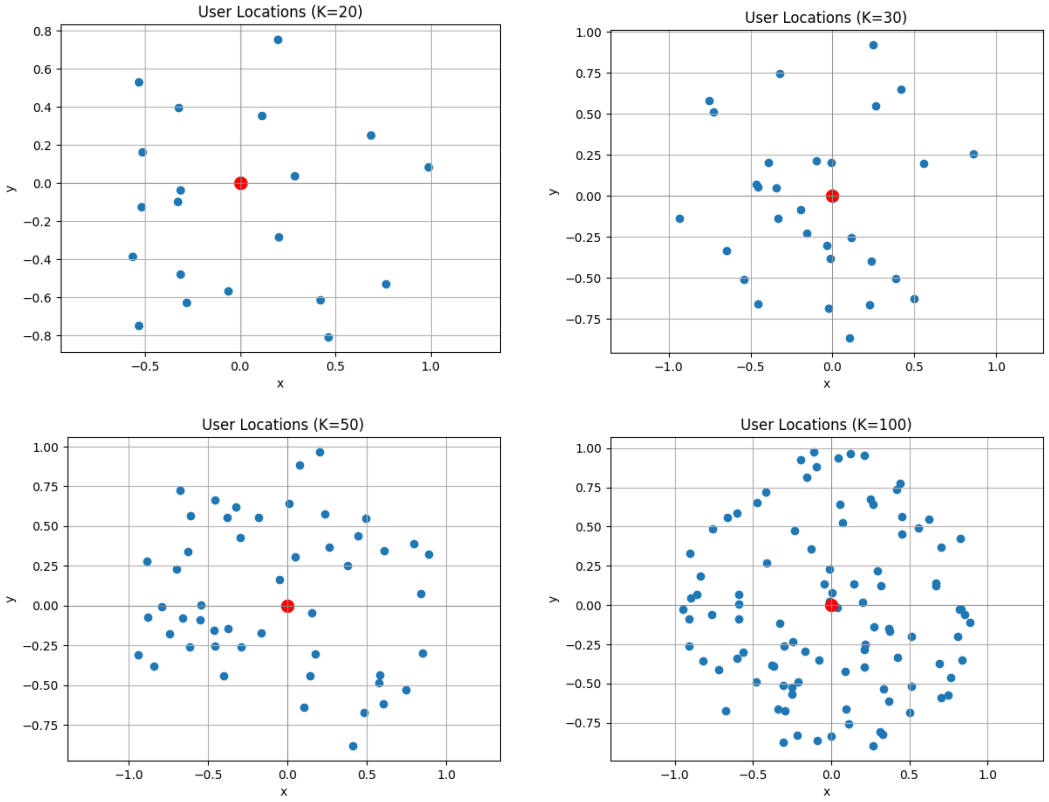

Figure 1: Exemples d'échantillonnages de positions utilisateurs pour 20, 30, 50 et 100 utilisateurs

3 et 4

```
achievement_ratio = measure_achievement_ratio(20,32*1e3,10**(7/10),100)
print(f'delta = {np.mean(achievement_ratio)*100}%')
```

On obtient des résultats de 97.95% et 98.1%.

5

```
K_values = np.array(range(1,100))
deltas = np.zeros(len(K_values))

for i,K in enumerate(K_values) :
    ach_rat = measure_achievement_ratio(K,32*1e3,10**(7/10),100)
    deltas[i] = np.mean(ach_rat)

print(deltas)
print(f'The maximum number of users on the network is {K_values[np.sum(deltas>=0.9)]}')

plt.plot(K_values,deltas)
plt.title('Condition achievement rate by the number of users')
plt.xlabel('Number of users')
plt.ylabel('Proportion of users over SINR threshold')
plt.axhline(0.9,color='0')
plt.axvline(K_values[np.sum(deltas>=0.9)],color='0')
plt.show()
```

Le nombre maximal d'utilisateurs obtenu est 43, comme nous pouvons le voir en figure 2.

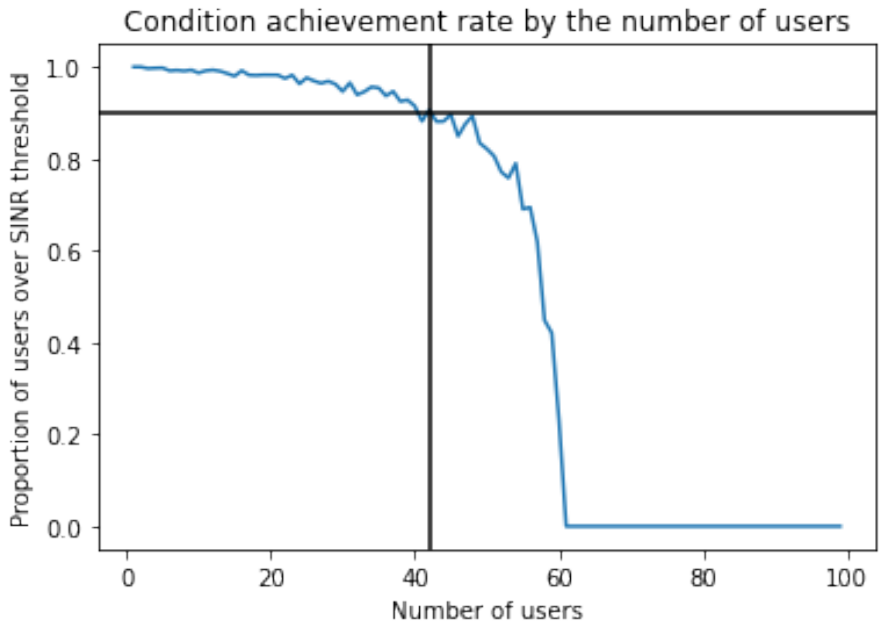

Figure 2: Proportion d'utilisateurs satisfaisant la condition de SINR en fonction du nombre total d'utilisateurs

2 Contrôle de puissance uplink d'un système CDMA

1.a Nous renvoyons à la question 1 de la partie 1 pour cette question.

1.b On utilise le script suivant pour obtenir le nombre maximal d'utilisateurs tel que $\rho(F) < 1$ soit toujours vérifée.

```
K = np.array(range(10,100,2), dtype=np.int32)
R = 1
r1,r2 = 15*1e3,32*1e3
```

```
W = 3.84*10**6
theta = 0.4
gamma1, gamma2 = 10**(5/10), 10**(7/10)
Rho = np.array([])

def generate_F(k,R,r1,r2,gamma1,gamma2,W,theta):
    x,y = sample_users(k,R)
    r = np.concatenate((r1*np.ones(k//2),r2*np.ones(k//2)))
    gamma = np.concatenate((gamma1*np.ones(k//2),gamma2*np.ones(k//2)))
    lamda = 0.5
    h = np.random.exponential(1/lamda,k)
    d = np.sqrt(x**2+y**2)
    L = -128.1 - 37.6*np.log10(d)
    l = 10**(L/10)
    g = l*h
    G1,G2 = np.meshgrid(gamma*r*g,g)
    F = (theta/W)*G1 / G2
    F[np.eye(k,dtype=bool)] = 0
    return F, r, gamma, g

for k in K:
    F,_,_,_ = generate_F(k,R,r1,r2,gamma1,gamma2,W,theta)
    rho = np.max(np.abs(np.linalg.eigvals(F)))
    Rho = np.append(Rho,rho)

Kmax = np.max(K[Rho<1])
print(Kmax)
```

On obtient un nombre maximum d'utilisateurs à 92.

1.c Nous calculons l'allocation de puissance optimale par une simple inversion matricielle :

```
k = Kmax

x,y = sample_users(k,R)
r = np.concatenate((r1*np.ones(k//2),r2*np.ones(k//2)))
gamma = np.concatenate((gamma1*np.ones(k//2),gamma2*np.ones(k//2)))
lamda = 0.5
h = np.ones(k)
h = np.random.exponential(1/lamda,k)
d = np.sqrt(x**2+y**2)
L = -128.1 - 37.6*np.log10(d)
l = 10**(L/10)
g = l*h
G1,G2 = np.meshgrid(gamma*r*g,g)
F = (theta/W)*G1 / G2
F[np.eye(k,dtype=bool)] = 0

sigma2 = 10**(-104/10)/1000
b = sigma2*(1/(3.84*10**6))*r*gamma/g
P = np.linalg.inv(np.eye(k) - F)@b

print(10*np.log10(P*1000)) # display the power allocation in dBm

def SINR(W,R,P,G,theta,sigma2):
    alpha = W/R
    pg = P*G
    p,_ = np.meshgrid(P,P)
    p[np.eye(len(p),dtype=bool)] = 0
    sm = np.sum(p,axis=1)*G
    return alpha*pg/(theta*sm+sigma2)
```

```
30
plt.hist(10*np.log10(P*1000))
32 plt.title('Power allocation distribution')
plt.ylabel('Number of users')
34 plt.xlabel('P (dBm)')
plt.show()
```

Ce qui nous donne la distribution de puissance en figure 3

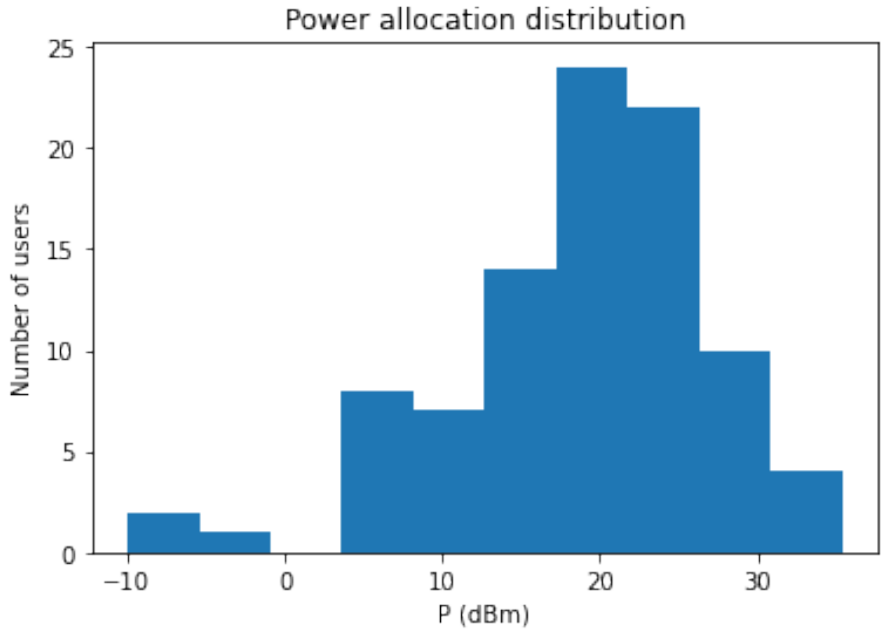

Figure 3: Allocation de puissance optimale

1.d Résoudre le problème comme 1.c peut être très coûteux en calcul puisqu'il demande une inversion matricielle. Pour raccourcir le temps de calcul, nous implémentons une prémière méthode itérative pour arriver à l'allocation optimale.

```
K_list = np.array(range(10,Kmax,2), dtype=np.uint32)
2 epsilon = 0.1
iteration_list = np.zeros(len(K_list))
4
for idx in range(len(K_list)):
6     k = K_list[idx]
      p = np.ones(k)
8     new_p = np.ones(k)*0.1
      num_iteration = 0
10    _, r, gamma, g = generate_F(k,R,r1,r2,gamma1,gamma2,W,theta)
      while not np.all(np.abs(p - new_p) < epsilon):
12        p = new_p
          pg = p*g
14        pg1,_ = np.meshgrid(pg,pg)
          pg1[np.eye(len(pg),dtype=bool)] = 0
16        sm = np.sum(pg1,axis=1)
          new_p = (r*gamma*(theta*sm + sigma2))/(W*g)
18        num_iteration += 1
      iteration_list[idx] = num_iteration
20    print(f"k={k}, num_iteration={num_iteration}")
```

```
plt.plot(K_list, iteration_list, marker='o')
plt.xlabel('Number of Users (k)')
plt.ylabel(f'Number of Iterations with epsilon={epsilon}')
plt.title('Number of Iterations vs Number of Users')
plt.grid(True)
plt.show()
```

Ceci nous permet d'obtenir le nombre d'itérations avant convergence de l'algorithme en fonction du nombre d'utilisateurs, visible en figure 4.

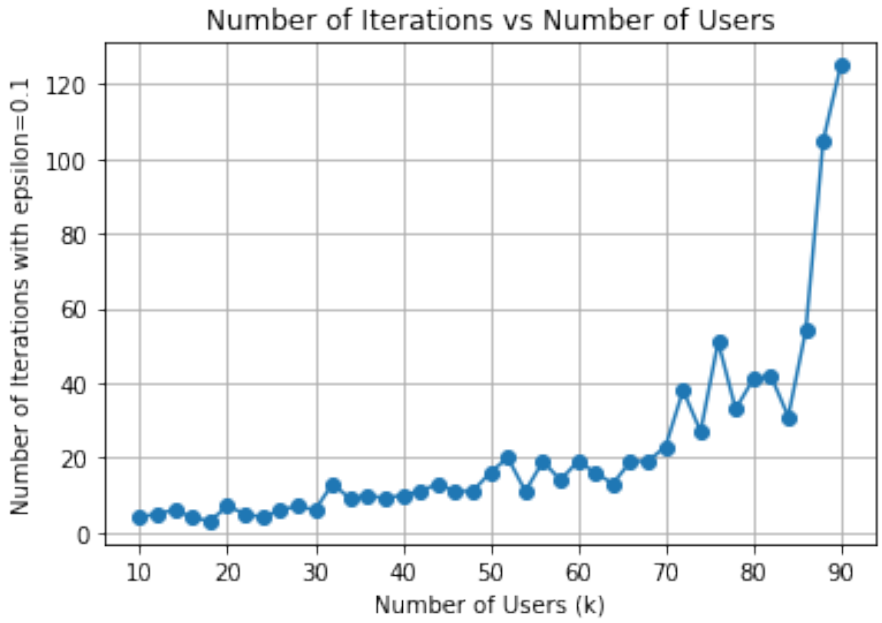

Figure 4: Nombre d'itérations avant convergence de l'algorithme en fonction du nombre d'utilisateurs

1.e On implémente une deuxième méthode itérative pour résoudre le problème d'allocation.

```
def iterativeE(gamma,beta,tol,seed):
    p=seed.copy()
    p_old = np.inf*np.ones_like(p)
    iter = 0
    while np.sum(abs(p-p_old))>=tol:
        p_old = p.copy()
        sinr = SINR(W,r,p,g,theta,sigma2)
        p = (1-beta)*p + beta*gamma/sinr*p
        iter +=1
    return p, iter

Beta = np.arange(0.1,1,0.05)
Niter = np.zeros_like(Beta)
_, r, gamma, g = generate_F(Kmax,R,r1,r2,gamma1,gamma2,W,theta)

seed = np.ones(Kmax)
for b in range(len(Beta)):
    p,iter = iterativeE(gamma,Beta[b],1e-3,seed)
    Niter[b] = iter

print(Niter)
```

```
22
plt.plot(Beta,Niter)
24 plt.title('Number of iterations before convergence by beta')
plt.xlabel('Beta')
26 plt.ylabel('Number of iterations')
```

On considère cette fois le nombre d'itérations avant convergence de l'algorithme en fonction du paramètre β, avec $K_{max} = 92$ utilisateurs. On obtient la courbe en figure 5. On voit très clairement qu'avec un β proche de 1, le nombre d'itérations nécessaire est considérablement plus faible qu'avec β proche de 0.

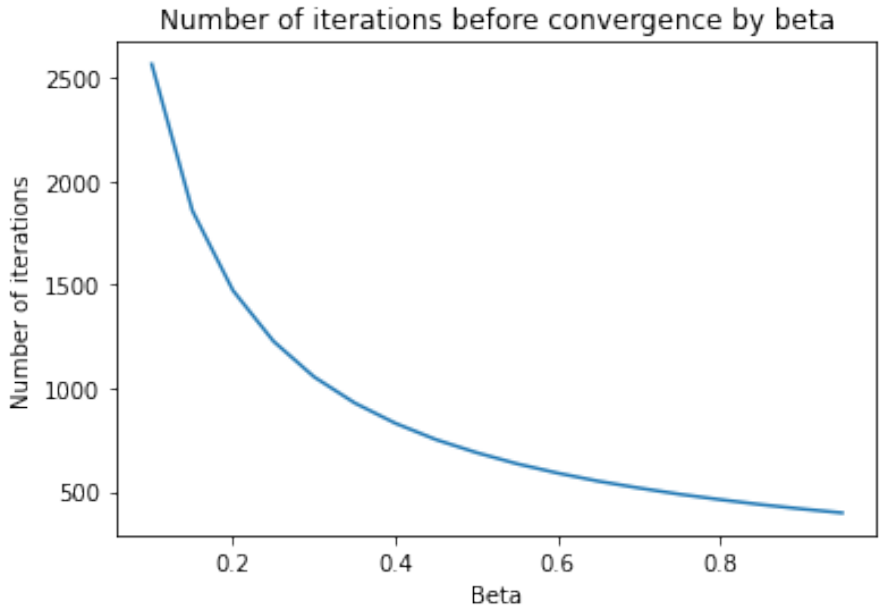

Figure 5: Nombre d'itérations avant convergence en fonction de β

1.f On implémente la méthode proposée dans l'énoncé :

```
def iterativeF(gamma,alpha,tol,seed,maxIter):
2     p = seed.copy()
      p_old = np.inf*np.ones_like(p)
4     iter = 0
      while (np.sum(abs(p-p_old))>=tol) & (iter<maxIter):
6         p_old = p.copy()
          sinr = SINR(W,r,p,g,theta,sigma2)
8         p[sinr<gamma] = alpha*p_old[sinr<gamma]
          p[sinr>gamma] = p_old[sinr>gamma]/alpha
10        iter +=1
      return p, iter
12
Alpha = 10**(np.linspace(0.25,1.5,7)/10)
14 Niter = np.zeros_like(Alpha)
_, r, gamma, g = generate_F(Kmax,R,r1,r2,gamma1,gamma2,W,theta)
16
seed = np.ones(Kmax)
18 for a in range(len(Alpha)):
      p,iter = iterativeF(gamma,Alpha[a],1e-3,seed,5e3)
20    Niter[a] = iter
```

```
print(Niter)
```

Nous constatons que, pour les valeurs d'α proposées, l'algorithme atteint systématiquement le nombre maximum d'itérations, fixé à 5000 itérations.

1.g A l'aide du script suivant : nous implémentons la nouvelle méthode itérative proposée.

```
def iterativeG(gamma,alpha,tol,maxIter):
    p = np.ones(Kmax)
    p_old = np.inf*np.ones_like(p)
    iter = 0
    while (np.sum(abs(p-p_old))/Kmax>=tol) & (iter<maxIter):
        p_old = np.copy(p)
        sinr = SINR(W,r,p,g,theta,sigma2)
        p[sinr > alpha*gamma*p_old] = p_old[sinr > alpha*gamma*p_old]/alpha
        p[sinr < gamma*alpha**(-1)] = p_old[sinr < gamma*alpha**(-1)]*alpha
        iter += 1
    return p, iter

Alpha = 10**(np.linspace(0.25,1.5,7)/10)
#Alpha = [10**(0.25/10), 10**(0.25/10), 10**(0.25/10)]
Niter = np.zeros_like(Alpha)
_, r, gamma, g = generate_F(Kmax,R,r1,r2,gamma1,gamma2,W,theta)
epsilon = 5e-3

for a in range(len(Alpha)):
    p_alpha,iter = iterativeG(gamma,Alpha[a],epsilon,5e3)
    Niter[a] = iter

print(Niter)
```

Nous constatons, encore une fois, que l'algorithme atteint systématiquement le nombre maximum d'itérations (fixé à 5000 itérations) pour toutes les valeurs d'α testées.

2.a Cette étape revient à faire la même chose qu'en question 1.a.

2.b Nous implémentons la méthode itérative vue en 1.e, avec cette fois une mise à jour du paramètre h à chaque étape de l'itération.

```
def generate_F2(k,R,r1,r2,gamma1,gamma2,W,theta):
    x,y = sample_users(k,R)
    r = np.concatenate((r1*np.ones(k//2),r2*np.ones(k//2)))
    gamma = np.concatenate((gamma1*np.ones(k//2),gamma2*np.ones(k//2)))
    d = np.sqrt(x**2+y**2)
    L = -128.1 - 37.6*np.log10(d)
    l = 10**(L/10)
    return r, gamma, l

def iterativeE(gamma,beta,tol,seed,k,l,r,lim_iteration):
    p=seed.copy()
    p_old = np.inf*np.ones_like(p)
    iter = 0
    lamda = 0.5
    while lim_iteration > iter:

        h = np.random.exponential(1/lamda,k)
```

```
        g = l*h

        p_old = p.copy()
        sinr = SINR(W,r,p,g,theta,sigma2)
        p = ((1-beta)*p + p*beta*gamma/sinr)
        iter +=1

    return p, iter, sinr

lim_iteration = 1000
K = 10
Beta = np.arange(0.05,1,0.01)
Niter = np.zeros_like(Beta)
ach_rat = np.zeros_like(Beta)
r, gamma, l = generate_F2(K,R,r1,r2,gamma1,gamma2,W,theta)

seed = np.ones(K)
for b in range(len(Beta)):
    p,iter,sinr = iterativeE(gamma,Beta[b],1e-3,seed,K,l,r,lim_iteration)
    Niter[b] = iter
    ach_rat[b] = np.mean(sinr>=gamma)

plt.figure()
plt.plot(Beta,ach_rat)
plt.title('SINR condition achievement rate by beta')
plt.xlabel('Beta')
plt.ylabel('Proportion of users satisfying SINR condition')
```

Après avoir fait tourner la méthode pendant 1000 itérations pour différents β, on obtient le pourcentage d'utilisateurs satisfaisant la condition de SINR en fonction de beta disponible en figure 6. Nous constatons, malgrès une variance très élevée, une tendance à la baisse de la proportion d'utilisateurs vérifiant la condition de SINR lorsque β se rapproche de 1. Il faut donc faire un compromis dans le choix de β entre performance du système lorsque le fading est stable et performance du système lorsqu'il évolue très rapidement.

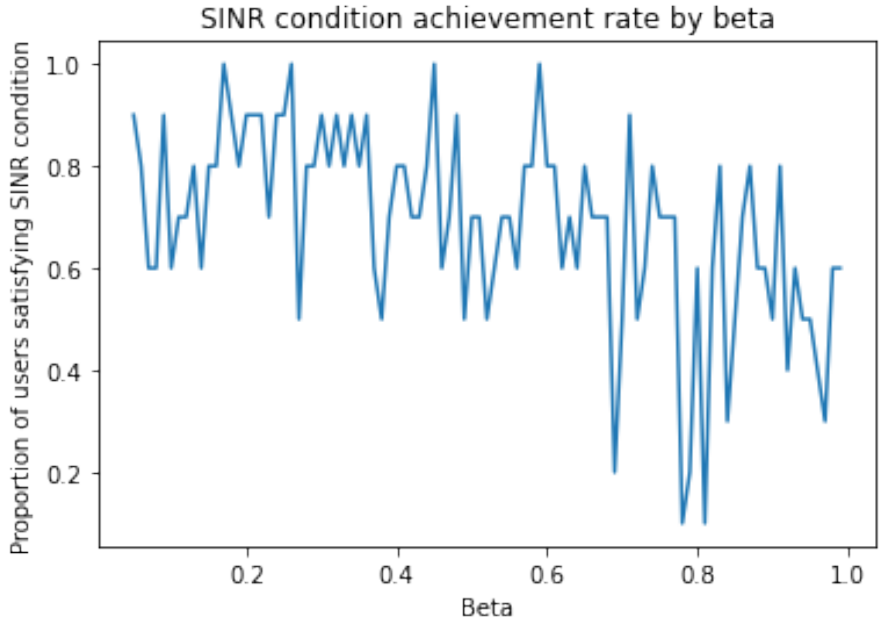

Figure 6: Taux de satisfaction de la condition de SINR en fonction de β

3 Comparaison entre systèmes CDMA et TDMA

1 Nous réutilisons les fonctions introduites en partie 1 et reprocédons à l'analyse de la capacité du système :

```
r = 1
W = 3.84*10**6
theta = 0.4
sigma2 = 10**(-104/10)*1e-3
P = 10**(40/10)*1e-3

def measure_achievement_ratio(K,R,gamma,n_avg=1):
    x,y = sample_users(K,r)
    d = np.sqrt(x**2+y**2)
    L = -128.1 - 37.6*np.log10(d)
    l = 10**(L/10)
    history = np.zeros((n_avg,1))
    for i in range(n_avg):
        h = np.random.exponential(0.5,K)
        g = l*h
        p = P/K
        SINR = W/R*p*g/(theta*(K-1)*p*g+sigma2)
        history[i] = np.mean(SINR>=gamma)

    return history

K_values = np.array(range(1,100))
deltas = np.zeros(len(K_values))

for i,K in enumerate(K_values) :
    ach_rat = measure_achievement_ratio(K,240*1e3,10**(10/10),100)
    deltas[i] = np.mean(ach_rat)

print(deltas)
print(f'The maximum number of users on the network is {K_values[np.sum(deltas>=0.9)]}')
```

Nous obtenons un nombre d'utilisateurs maximum de 4.

2 Nous considérons cette fois un système TDMA, où un seul utilisateur a accès au réseau à la fois. Nous implémentons un algorithme d'ordonnancement de type Proportional Fairness et estimons les statistiques de débit instantané pour les utilisateurs avec le script suivant.

```
T = 1000
I = 50
gamma_i = 10**(10/10)
R_i = 240e3
L = 16                    # = W/R_i
P = 10**(40/10)*1e-3
K_list = [10, 20, 30, 40]
radius = 1
lamda = 0.5
sigma2 = 10**(-104/10)*1e-3

for K in K_list:
    try_averages = np.zeros(I)
    for idx in range(I):
        # Generate K users
        x, y = sample_users(K, radius)
```

```
        # Compute distance and pathloss effect
        d = np.sqrt(x**2+y**2)
        L = -128.1 - 37.6*np.log10(d)
        l = 10**(L/10)
        Average_rates = np.zeros(K)

        for t in range(T):
            h = np.random.exponential(1/lamda,K)
            g = l*h
            SNR_base = P*g/sigma2
            SNR = SNR_base.copy()
            C = np.ones(K)
            current_rates = np.zeros(K)
            proportional_fairness = np.zeros(K)

            for i in range(K):
                while SNR[i] > gamma_i and C[i] < 15:
                    SNR[i] = SNR_base[i]/C[i]
                    C[i] += 1
                # print(f"SNR[i] = {SNR[i]} and gamm_i = {gamma_i}")

                current_rates[i] = C[i] * R_i
                if Average_rates[i] != 0:
                    proportional_fairness[i] = np.argmax(current_rates[i]/
Average_rates[i])
                else:
                    proportional_fairness[i] = np.inf
            selected_user = np.argmax(proportional_fairness)
            Average_rates[selected_user] += -(1/(t+1))*Average_rates[selected_user] +
current_rates[selected_user]/(t+1)
            Average_rates[range(K) != selected_user] -= Average_rates[range(K) !=
selected_user]/(t+1)
        try_averages[idx] = np.mean(Average_rates)
    print(f"K={K}, Average Rate={np.mean(try_averages):.2f}, Standard Deviation={np.
std(try_averages):.2f}")
```

Nous obtenons les résultats en table 1. Nous constatons alors que pour un débit équivalent, le système TDMA permet de servir beaucoup plus d'utilisateurs (environ 10 pour le TDMA contre 4 pour le CDMA à la question précédente). Par ailleurs, nous constantons pour un même nombre d'utilisateurs, le système TDMA peut fournir un débit plus élévé comme en comparant à la question 5 de la partie 1, où le système CDMA n'arrive à fournir que 32 kbits/s de débit pour 44 utilisateurs, où le système TDMA arrive à proposer 72 kbits/s de débit pour 40 utilisateurs (plus du double).

K	Débit instantané moyen (kbit/s)	Ecart-type du débit instantané (kbit/s)
10	307,5	50,0
20	140,7	28,1
30	98,12	16,1
40	71,97	14,0

Table 1: Statistiques des débits instantanés en fonction du nombre d'utilisateurs