TP Optimisation

Option Mathématiques Aplliquées

Mathieu Chalvidal, Adrien Boilot

Octobre 2018

Préambule : Démarche et choix des outils

Afin de réaliser les différents exercices de ce TP, nous avons choisi de scripter les algorithmes en language **Python** sous format **Jupyter Notebook**.

En effet, Python apparaît comme un outil efficace pour le traitement des problèmes d'optimisation, étant donné la forte communauté d'utilisateurs et la profusion des librairies optimisées disponibles pour traiter ces problèmes. Parallèlement la syntaxe fluide de ce language permet une écriture simple des algorithmes utilisés et proche de leur formulation logique.

Plus précisement, nous utiliserons intensivement les librairies **Numpy** pour la manipulation des vecteurs multidimentionnels (opérations élémentaires, redimensionnement, etc.) La librairie **Scipy** propose des méthodes et outils efficaces et fortement documenté d'optimisation. A cela nous ajoutons, pour le traitement de la partie 2, la libraire **Cvxpy** qui propose des outils de résolution des problèmes linéaires en nombre entier et mixtes ainsi que la librairie **Networkx** qui permet la formulation et la résolution de problème d'optimisation sous forme de graphe.

Enfin, nous utiliserons **Matplotlib** et **Seaborn** pour la visualisation graphique de nos résultats ainsi que le module **Time** pour mesurer les performances temporelles de ces solutions.

1 Optimisation contine et optimisation approchée

1.1 Optimisation sans contraintes

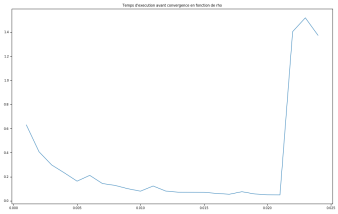
1.1.1 Méthode du gradient

a) Méthode à pas constant

Nous réalisons l'optimisation de f1 par la méthode du gradient à pas constant pour des pas de l'ensemble 0.001*k pour k dans l'intervalle $[\![1, 25]\!]$ et pour des paramètres d'initialisation identiques par ailleurs. Ce choix d'ensemble de pas nous permet de mettre en évidence deux phénomènes pour l'optimisation de f1. Le **ralentissement** du temps de convergence de l'algorithme pour des pas trop faible et la **non-convergence** de l'algorithme pour des pas trop forts (ici > 0.022). Nous donnons ci-dessous le code pour le gradient à pas constant (figure 1) et le graphe du temps de résolution en fonction du pas (figure 2).

```
rho_ = [i*0.001 for i in range(1,25)]
results=[]
for rho in rho_:
    debut = time.time()
    GradResults=gradient_rho_constant(f1,df1,x0,rho=rho,tol=1e-6,args=(B,S))
    tps_ecoule = time.time()-debut
    results.append(tps_ecoule)
    print ('rho', rho, '   converged',GradResults['converged'] ,'  tps ecoule (gradient_rho_constant):',tps_ecoule)

fig, ax = plt.subplots(figsize=(16, 10))
ax.plot(rho_,results)
ax.set_title("Temps d'execution avant convergence en fonction de rho")
plt.savefig('testplot.png')
```

Figure 1: Code de l'algorithme de descente de gradient à pas constant



(a) Temps de convergence en fonction de ρ

```
rho 0.001   converged True   tps ecoule (gradient_rho_constant): 0.6634757518768311
rho 0.002   converged True   tps ecoule (gradient_rho_constant): 0.4257762432009888?
rho 0.003   converged True   tps ecoule (gradient_rho_constant): 0.33278584480285645
rho 0.004   converged True   tps ecoule (gradient_rho_constant): 0.2644131183624267?
rho 0.005   converged True   tps ecoule (gradient_rho_constant): 0.15034031867980957
rho 0.006   converged True   tps ecoule (gradient_rho_constant): 0.1845238208770752
rho 0.007   converged True   tps ecoule (gradient_rho_constant): 0.1800630009262084?6
rho 0.008   converged True   tps ecoule (gradient_rho_constant): 0.1422033309936523?
rho 0.009000000000000001   converged True   tps ecoule (gradient_rho_constant): 0.14203596115112305
rho 0.01   converged True   tps ecoule (gradient_rho_constant): 0.11784696578979492
rho 0.011   converged True   tps ecoule (gradient_rho_constant): 0.10687899589538574
rho 0.012   converged True   tps ecoule (gradient_rho_constant): 0.10131096839904785
rho 0.013000000000000001   converged True   tps ecoule (gradient_rho_constant): 0.1066938835144043
rho 0.014   converged True   tps ecoule (gradient_rho_constant): 0.08557415008544922
rho 0.015   converged True   tps ecoule (gradient_rho_constant): 0.0809230856444335?4
rho 0.016   converged True   tps ecoule (gradient_rho_constant): 0.08115601539611816
rho 0.017   converged True   tps ecoule (gradient_rho_constant): 0.08867192268371582
rho 0.018000000000000002   converged True   tps ecoule (gradient_rho_constant): 0.07127785682678223
rho 0.019   converged True   tps ecoule (gradient_rho_constant): 0.09186124801635742
rho 0.02   converged True   tps ecoule (gradient_rho_constant): 0.06561207771301?27
rho 0.021   converged True   tps ecoule (gradient_rho_constant): 0.06010032257080?8
rho 0.022   converged False   tps ecoule (gradient_rho_constant): 1.4267759323120117
rho 0.023   converged False   tps ecoule (gradient_rho_constant): 1.3652861118316?65
rho 0.024   converged False   tps ecoule (gradient_rho_constant): 1.2923629283905?3
```

(b) Résultats de convergence

Figure 2: Résultats de la méthode de gradient à pas constant en fonction de ρ

Interprétation: Comme indiqué plus haut, nous observons qu'un pas trop petit ralentit l'algorithme puisque la solution courante xn varie trop faiblement entre deux itérations successives. En revanche, un pas trop fort va "empecher" l'algorithme de converger vers la solution, puisque l'itération fera dévier trop fortement x_n.

b) Méthode à pas adaptatif

Une solution simple pour éviter ces problèmes est d'adapter le pas à chaque itération de l'algorithme en fonction de la différence entre la valeur de f en la solution courante et de la solution nouvelle. Le code ci-dessous s'inspire de la partie a) et ajoute la mise à jour itérative du pas d'un facteur 1/2 si la valeur par f de la solution courante est plus élevée que la précedante, et d'un facteur 2 sinon. Intuitivement, si l'on se déplace dans la "bonne direction", on peut en effet, accélérer la descente.

```
def gradient_rho_adaptatif(fun, fun_der, U0, rho, tol,args):

    itermax=10000  #nb d'itérations
    xn=U0          #initialisation
    f=fun(xn,*args)
    it=0   #compteur
    loss_compt=0
    converged = False;

    while (~converged & (it < itermax)):
        it=it+1
        dfx=fun_der(xn,*args)
        xnp1=xn-rho*dfx
        fnp1=fun(xnp1,*args)
        if fnp1<f:
            rho=2*rho
            if abs(fnp1-f)<tol:
                converged = True
            xn=xnp1
            f=fnp1
            loss_compt+=1
        else:
            rho=0.5*rho


    GradResults = {
            'initial_x':U0,
            'minimum':xnp1,
            'f_minimum':fnp1,
            'iterations':it,
            'converged':converged,
            'loss_compt':loss_compt
            }
    return GradResults
```

Figure 3: Code de l'algorithme de descente de gradient à pas adaptatif

c) Comparaison des performances des deux méthodes

Nous présentons ici les deux méthodes proposées plus haut et comparons leurs résultats en temps et nombre d'itérations.

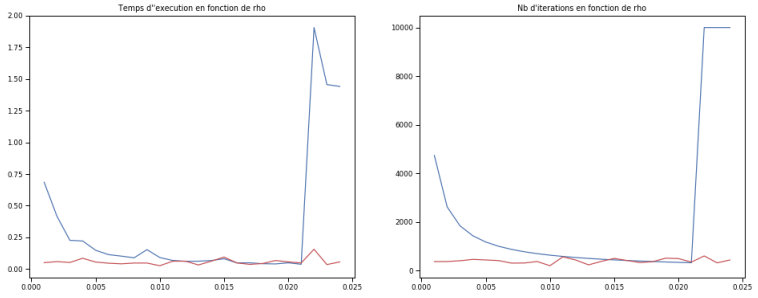


Figure 4: Comparaison des performances temporelles et en nb d'itérations pour différentes initialisations de ρ

Interprétation : On observe que la méthode de pas adaptatif (rouge) est beaucoup plus stable en terme de temps et d'itérations que la méthode de pas constant (bleue) puisqu'elle résoud le problème de non convergence pour un pas trop fort (ici > 0.02) ou de faible convergence pour un pas trop faible (< 0.001)

1.1.2 Utilisation de la Méthode de Quasi-Newton avec l'algorithme BFGS

Le module optimize de la librairie **Scipy** propose différents outils d'optimisation, dont une implémentation de l'algorithme BFGS permettant une mise en oeuvre rapide de la méthode de Quasi-Newton. Nous initialisation l'algorithme avec le même vecteur x_0 afin de comparer les résultats. Le critère d'arrêt de l'algorithme est fixé à $1e^{-6}$.

```
: debut = time.time()
  res=optimize.minimize(f1,x0,method='BFGS',tol=1e-6,args=(B,8))
  tps_ecoule = time.time()-debut
  print(res)
  print('temps écoulé' ,tps_ecoule, 'sec')

      fun: -1.836962311965238
 hess_inv: array([[ 0.5004328 , -0.21223272,  0.05691395, -0.28056012,  0.50919827],
       [-0.21223272,  0.22383377,  0.0370393 ,  0.11542843, -0.34043246],
       [ 0.05691395,  0.0370393 ,  0.1247386 , -0.04905887, -0.09108766],
       [-0.28056012,  0.11542843, -0.04905887,  0.20504998, -0.28961765],
       [ 0.50919827, -0.34043246, -0.09108766, -0.28961765,  0.77691404]])
      jac: array([-2.98023224e-08, -8.94069672e-08,  1.49011612e-08, -1.49011612e-08,
        1.49011612e-08])
  message: 'Optimization terminated successfully.'
     nfev: 91
      nit: 10
     njev: 13
   status: 0
  success: True
        x: array([-0.69603139,  0.15793134, -0.61407083,  0.49414155, -0.05345803])
temps écoulé 0.01946878433227539 sec
```

Figure 5: Mise en oeuvre de l'algorithme BFGS avec Scipy et résultats

Comparaison avec la section 1.1.1 :

La figure 6 nous montre, comme attendu, que l'implémentation Scipy de l'algorithme de Broyden–Fletcher–Goldfarb–Shanno est toujours plus rapide que notre algorithme. Elle est également plus stable au niveau de la solution trouvée. En 25 invocations, l'algorithme converge toujours vers la même valeur, ce qui n'est pas le cas de notre algorithme de gradient à pas adaptatif. (figure de droite) En revanche, il faut noter qu'il n'y aucune garantie de trouver un optimum global par ces méthodes.

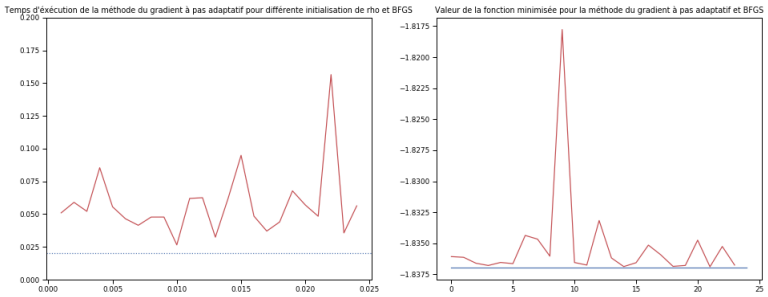


Figure 6: Comparaison des performances temporelles et stabilité de l'optimum pour BFGS et la méthode du gradient à pas adaptatif

1.2 Optimisation sous contraintes

1.2.1 Optimisations à l'aide de routines Python

```
Entrée [51]: bounds = ((0, 1),(0, 1),(0, 1),(0, 1),(0, 1))

             print("optimisation de f1","\n")
             SQP= optimize.minimize(f1,x0,method='SLSQP',bounds=bounds,args=(B,S))
             print(SQP,"\n","\n")

             print("optimisation de f2","\n")
             SQP2= optimize.minimize(f2,x0,method='SLSQP',bounds=bounds,args=(S))
             print(SQP2)

optimisation de f1

     fun: -0.13853161426327318
     jac: array([5.65140143e-01, 1.70478411e-03, 4.86906106e+00, 1.77649595e-03,
       2.70384220e-01])
 message: 'Optimization terminated successfully.'
    nfev: 65
     nit: 9
    njev: 9
  status: 0
 success: True
       x: array([4.70812343e-16, 1.26984957e-01, 8.12072123e-16, 1.94536141e-02,
       1.33997327e-16])


optimisation de f2

     fun: 1.3856916387095l6e-15
     jac: array([1.00000018, 1.00000013, 1.00000035, 1.00000023, 1.00000022])
 message: 'Optimization terminated successfully.'
    nfev: 28
     nit: 4
    njev: 4
  status: 0
 success: True
       x: array([0.00000000e+00, 0.00000000e+00, 1.38569164e-15, 0.00000000e+00,
       0.00000000e+00])
```

Figure 7: Mise en oeuvre et résultats du Sequential Quadratic Programming pour minimiser f_1 et f_2

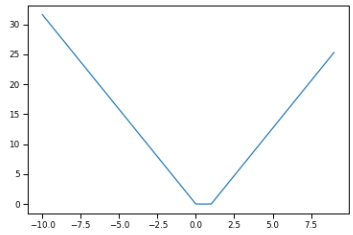
1.2.2 Optimisation sous contraintes et pénalisation

Comme indiqué dans l'énoncé, nous utilisons la fonction numpy.max nous permettant de pondérer notre ensemble U_{ad} par 0, puis d'augmenter la valeur de β proportionellement au module des dimensions de R^n. (figure 8) En faisant tendre ϵ vers 0,

le terme de pénalisation devient de plus en plus fort et force l'algorithme a converger dans la sous-ensemble ou β est nulle.

```python
def Beta(U):
    Uad=True
    vec=[]
    for i in U:
        if i<0 or i>1:
            Uad=False
    if Uad==False:
        for i in U:
            vec.append(np.max([0,i-1])+ np.max([0,-i]))
        return np.linalg.norm(vec)
    else:
        return 0
```

(a) Code de la fonction de pénalisation β



(b) Représentation de beta sur R (U unidimensionel

Figure 8: Définition et forme de β

Nous définissons alors les fonctions f_1penal et f_2penal comme indiqué dans la figure 9. Ensuite nous simulons le comportement des fonctions lorsque ϵ tend vers 0 par une discrétisation de ϵ sur l'intervale $[0.05, 1]$ (respectivement $[0.05, 0.5]$ pour f_2penal) par pas de 0.01. Ceci nous donne les résultats de convergence de la figure 10. La méthode de pénalisation converge en effet vers les résultats trouvés par l'algorithme SQP.

```python
target=SQP.fun
target2=SQP2.fun

def f1_penal(U,B,S,e):
    return f1(U,B,S) + (1/e)*Beta(U)

def f2_penal(U,S,e):
    return f2(U,S) + (1/e)*Beta(U)

#Création de la suite des minimisations de F_1 et F_2

Eps= [0.01*i for i in range(5,100)]
res=[]
iterations=[]
for e in Eps:
    mini=optimize.minimize(f1_penal,x0,method='SLSQP',args=(B,S,e))
    res.append(mini.fun)
    iterations.append(mini.nit)

Eps2= [0.01*i for i in range(5,50)]
res2=[]
iterations2=[]
for e in Eps2:
    mini=optimize.minimize(f2_penal,x0,method='SLSQP',args=(S,e))
    res2.append(mini.fun)
    iterations.append(mini.nit)

#Représentation graphique des résultats
f, (ax0, ax1) = plt.subplots(1, 2, figsize=(16,6))

ax0.plot(Eps,res)
ax0.set_title('Convergence of f1_penal')
ax0.axhline(y=target, color='g', linestyle=':')
ax0.set_xlim(1, 0.05)
ax0.set_ylim(-1.1, 0)

ax1.plot(Eps2,res2)
ax1.set_title('Convergence of f2_penal')
ax1.axhline(y=target2, color='g', linestyle=':')
```

Figure 9: Code de simulation du comportement de f_1penal et f_2penal en fonction de ϵ

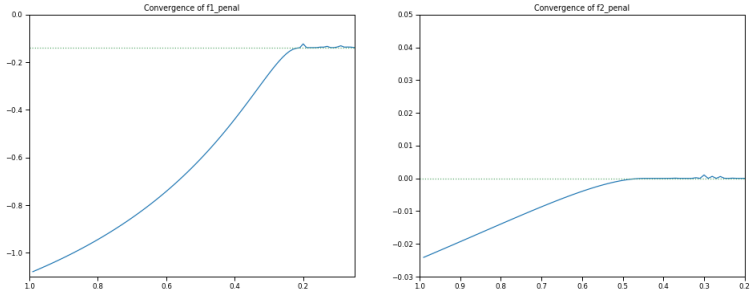


Figure 10: Convergence de f_1penal et f_2penal en fonction de ϵ (l'asymptote horizontale correspond aux solutions trouvées dans la section 1.2.1)

1.2.3 Méthodes duales pour l'optimisation sous contraintes

Nous définissons dans un premier temps le Lagrangien ainsi que son gradient par rapport à Λ (espace dual) puis nous définissons l_0, le vecteur d'initialisation dans cet espace. Les valeurs de ce vecteur ne pouvant être toutes nulles (par résultat du théorème de Fritz Johns), nous décidons de fixer ce vecteur comme $numpy.ones(10)$, vecteur ou toutes les valeurs sont 1.

Enfin nous choisissons d'utiliser le vecteur solution x_n dans l'étape n pour initialiser la recherche de x_{n+1} par minimisation à l'étape n+1 (méthode $Warm\ start$) puisqu'il est raisonnable de penser que cette initialisation est plus proche de la solution et accélère ainsi l'algorithme.

```
def Lagrangien(U,B,S,l):
    l1=l[0:5]
    l2=l[5:10]
    return f1(U,B,S) + np.dot(l1,U-np.ones(5)) + np.dot(l2,-U)
```

```
def Grad_Lagrangien(U,B,S,l):
    vec=np.concatenate((U,-U))
    return 1*vec
```

```
l0=np.ones(10)
```

Figure 11: Définition du Lagrangien et de son gradient et vecteur d'initialisation l_0

Enfin nous savons que l'algorithme d'Uzawa va converger vers un point selle. Car la fonction f1 ainsi que les contraintes sont continûment différentiables et convexes, les contraintes vérifient les hypothèses de qualification, et le problème de minimisation sous contraintes énoncé admet une solution. Le critère d'arrêt porte à la fois sur l'évaluation de deux solutions successives : $(x_n) - f(x_{n+1})$ | ou sur leur distance pour la norme canonique de R^n : $\|x_n - x_{n+1}\|$, avec un seuil de tolérance arbitrairement fixé à $1e^{-6}$, qui rétrospectivement, nous donne une solution proche des méthodes précedemment présentées.

```python
def Uzawa(fun, dfun, U0, l0, args, rho=2, tol=1e-6):

    itermax=1000      #nb d'itérations
    xn=U0
    ln=l0
    it=0               #compteur
    history=[]
    f_history=[]
    converged = False;

    while (~converged & (it < itermax)):
        it=it+1
        xnp1=optimize.minimize(fun,xn,method='BFGS',tol=1e-6,args=(B,S,ln)).x
        lnp1= np.maximum(np.zeros(10), ln + rho*dfun(xnp1,B,S,ln))
        if abs(Lagrangien(xnp1,B,S,ln)-Lagrangien(xn,B,S,ln))<tol and np.linalg.norm(xnp1-xn)<tol:
            converged = True
        xn=xnp1; ln=lnp1; fnp1=Lagrangien(xn,B,S,ln); history.append(xnp1);f_history.append(fnp1)

    GradResults = {
        'initial_x':U0,
        'minimum':xnp1,
        'f_minimum':fnp1,
        'iterations':it,
        'converged':converged,
        'history':history,
        'f_history':f_history
        }

    return GradResults

res=Uzawa(Lagrangien, Grad_Lagrangien, x0, l0, args=(B,S))

print("Minimum de f","\n","\n",res['f_minimum'])

f, ax = plt.subplots(1, 1, figsize=(12,8))
ax.plot(res['f_history'])
ax.plot([f1(k,B,S) for k in res['history']])
ax.axhline(y=target, color='g', linestyle=':')
plt.legend(('f1','Lagrangien','optimum'))
plt.savefig('UzaWa')
```

Figure 12: Implémentation de l'algorithme d'Uzawa

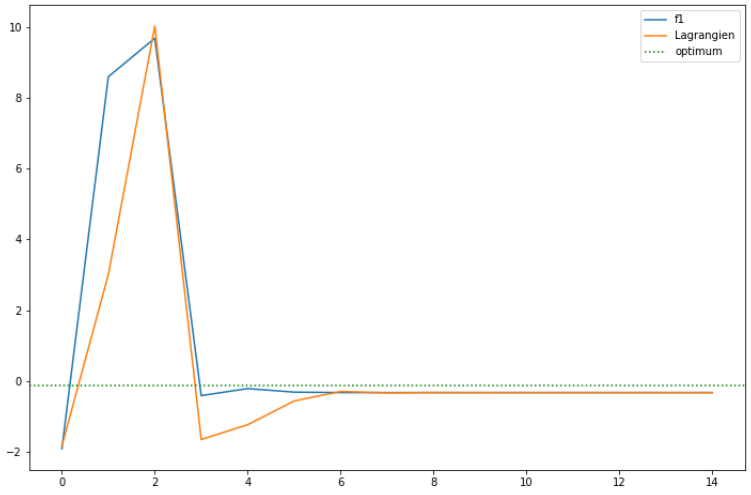


Figure 13: Convergence de l'algorithme d'Uzawa

Interprétation : Nous observons que l'algorithme converge vers une valeur proche de la solution trouvée pour f1 dans la section 1.2.1. Néanmoins celle ci possède

une marge d'erreur significative. Nous attribuons cette erreur au critère d'arrêt choisi.

1.3 Optimisation non convexe - Recuit simulé

Nous définissons $f_3 = f_1$ + 10*sin(2*f_1(U)) et nous recherchons l'optimum de cette fonction par les algorithmes BFGS et SQP pour différents points d'initialisation. Ces points d'initialisation sont générés par la fonction *numpy.random.randint* dans les dimensions demandées en 5 exemplaires. La figure 14 indique clairement que les algorithmes convergent vers des solutions différentes à chaque itération et entre chaque itérations. Ceci est du à la grande quantité de minimum locaux de la fonction f_3 vers lesquels convergent BFGS et SQP.

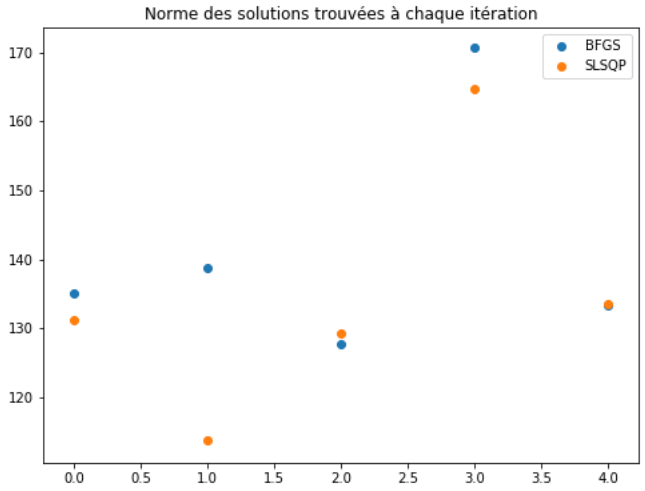


Figure 14: Optimums trouvés par les algorithmes BFGS et SQP pour 5 vecteurs d'initialisation aléatoire dans $[\![1, 100]\!]^5$

Nous allons donc implémenter l'algorithme de recuit simulé. Plusieurs paramètres nécessitent une sélection adéquate. Les paramètres de la descente de température: la température initiale T_0, le paramètre α de descente de la témpérature ainsi que la longueur L de chaque palier. Le nombre d'itération ainsi que le voisinnage selectionné sont également des paramères important.

Puisque notre fonction possède un très grand nombre de minimum locaux, nous voulons que notre solution courante parcourt largement l'espace des solutions sans rester coincée dans ces minimum tout en restant précis dans le parcours autour du minimum global lors des dernières itérations. Par conséquent, nous privilégions une descente de température "douce" avec un $\alpha = 0.95$ et un palier de longueur 100. La température initiale est fixée à 1000 et le nombre d'itérations à 50000 avec

un voisinnage compris dans la boule $B(x_n, 10)$ (NB: A la fin de l'execution, T=$T_0 * 0.95^{n_{iter}/L}$, de plus le voisinnage est fixe mais nous pourrions le restreindre à mesure que les itérations augmentent.)

```python
def Recuit_simule(f, U0, t0, palier, args):

    itermax=10000  #nb d'itérations
    xn=U0
    it=0
    T=t0
    f_history=[]
    current_history=[]

    while (it < itermax):
        it=it+1
        fnpl=f(xn,*args)
        if np.mod(it,palier)==0:
            T=0.95*T
        y = xn + np.random.randint(-10,11,len(xn))*0.5
        f_y=f(y,*args)

        if fnpl>=f_y:
            xn=y
            current_history.append(f_y)
        elif np.random.binomial(1,(np.exp((fnpl-f_y)/T)))==1:
            xn=y
            current_history.append(f_y)
        else:
            current_history.append(f_y)
        f_history.append(f(xn,*args))
    return xn, f_history ,current_history


anneal, f_history, current_history = Recuit_simule(f3, x0, 1000, 100, args=(B,S))
print("L'algorithme converge vers","\n", anneal)

plt.figure(figsize=(16,10))
plt.plot(current_history)
plt.plot(f_history)
plt.title("Evaluation de la valeur de la solution courante et du point voisin en fonction du nb d'itérations")
plt.legend('f_solution courante','f_point du voisinage')
plt.savefig('Recuit_norm')
```

Figure 15: Implémentation de l'algorithme de recuit simulé

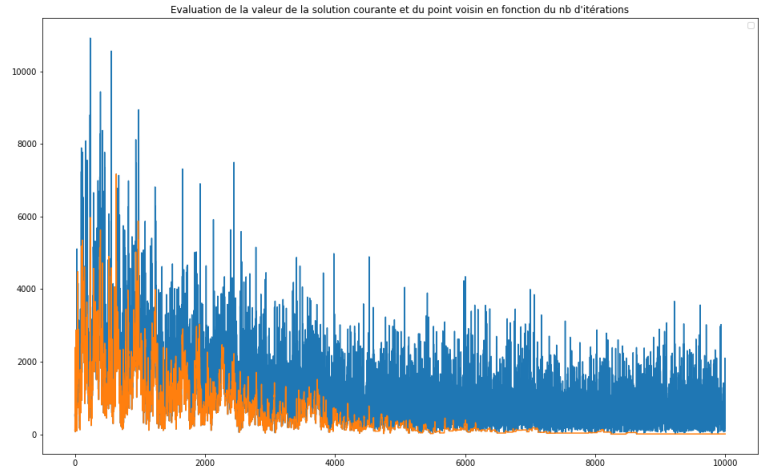


Figure 16: Evaluation de la valeur de la solution courante et du point voisin pour les 10000 premières itérations

La figure 17 montre les solutions trouvées pour 20 itérations de l'algorithme avec une température initiale égale à 50000 et des vecteurs d'initialisation différents. Nous voyons qu'il subsiste au point 10 une variation de la solution. Ceci est dit du à la grossiereté du voisinnage lorsque l'algorithme s'approche de la solution globale. Aussi notre implémentation est améliorable, nous pourrions par exemple imaginer un voisinage qui décroit proportionnellement à la température de descente. En revanche, ces résultats semble valider que l'algorithme converge toujours vers le même point quelque soit l'initialisation, pour un réglage approprié des paramètres de descente et d'itération.

```
[array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ]),
array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ]),
array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ]),
array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-0.5,  0. , -1. ,  0.5,  0.5]), array([-1.5,  0.5, -0.5,  1. , -1. ]),
array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ]),
array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ]),
array([-1.5,  0.5, -0.5,  1. , -1. ]), array([-1.5,  0.5, -0.5,  1. , -1. ])]
```

Figure 17: Solution des 20 itérations de l'algorithme de Recuit simulé avec différentes initialisations

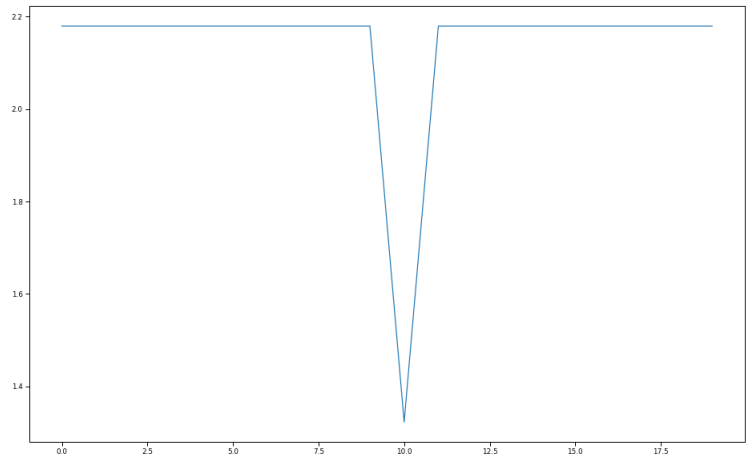


Figure 18: Norme des 20 solutions trouvées par l'algorithme de Recuit simulé avec différentes initialisations

1.4 Application Synthèse d'un filtre à réponse impulsionelle finie

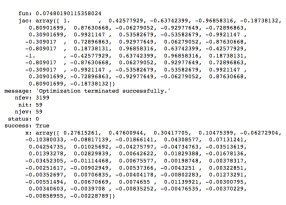
Pour poser le problème, nous définissons dans un premier temps les fonctions H_0 (filtre idéal) et H (filtre synthétisé sur les réponses impulsionelles paires à valeur dans R.) Nous définissons ensuite le critère

$$J(h)=\max_{j} \mid H_0(\nu j) - H(\nu j) \mid$$

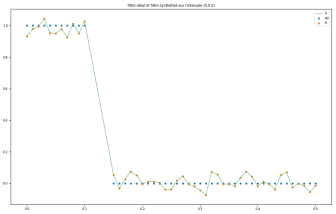
Par ailleurs, le problème spécifiait un filtre à 30 coefficients. Toutefois, afin d'obtenir une discrétisation plus équilibré de l'espace de définition, nous avons décidé de synthétiser un filtre à 47 coefficients (soit une discrétisation de 11 points sur [0,0.1] et de 36 points sur [0.15,0.5]. La formulation du problème reste par ailleurs inchangée.)

```
def H0(nu):
    if 0 <= nu <= 0.1:
        return 1
    else:
        return 0

def H(h,nu):
    n=len(h)
    H_nu = np.dot(h,[np.cos(2*np.pi*nu*i) for i in range(n)])
    return H_nu

def Critere(h,nu_discret):
    H0_discret=np.array([H0(i) for i in nu_discret])
    H_discret=np.array([H(h,i) for i in nu_discret])
    return np.max(np.abs(H0_discret-H_discret))

h0=np.ones(47)
nu_discret = [0.01*i for i in range(11)] + [0.01*i for i in range(15,51)]

Minimisation = optimize.minimize(Critere,h0,method='SLSQP',tol=1e-6,args=(nu_discret))
```

Figure 19: Script de définition du problème de synthèse d'un filtre à réponse impulsionelle finie discrétisé sur 47 points



(a) Résultats de l'algorithme d'optimisation



(b) Superposition du filtre idéal et du filtre synthétisé sur l'intervale [0,0.5]

Figure 20: Résultats et comparaison du filtre synthétisé optimisé sous formulation minimax

2 Seance 2 et 3 : optimisation discrète et optimisation multi-objectif

2.1 Rangement d'objets (optimisation combinatoire)

2.1.1 Question 1

Nous caractérisons le fait que chaque boîte contienne un et un seul objet par le fait que la matrice de $M_n^n(R)$ composé des $x_{i,j}$ du vecteur proposé dans l'énoncé

soit bistochastique. En effet puisque les $x_{i,j}$ sont des éléments de $[\![0,1]\!]$. La contrainte peut s'écrire comme:

$$\sum_{j=1}^{n} x_{i,j} = 1 \forall i \in [\![1,n]\!] \text{ (c.1)}$$
$$\sum_{i=1}^{n} x_{i,j} = 1 \forall j \in [\![1,n]\!] \text{ (c.2)}$$

et la matrice est binaire, les $x_{i,j}$ sont des éléments de $[\![0,1]\!]$. (c.3)

2.1.2 Question 2

Le problème à résoudre peut donc se poser comme un problème de programmation linéaire en nombre entier avec le critère suivant:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{i,j} * \|O_i - B_j\| \text{ sous les contraintes (c.1), (c.2) et (c.3)}$$

Cette formulation nous amène à l'implémentation numérique présentée dans la figure 19. Nous utilisons la librairie cvxpy qui s'avère très pratique pour la déclaration des contraintes et du critère pour un problème de PLNE. Nous obtenons pour cette question les résultats présentés en figure 22 avec une distance minimale de 15,3776 approximativement. La seconde variable correspond au vecteur de sélection objet-boîte.

```
os.chdir('/Users/chalvidalm/Documents/3A OMA/OMA fonda/Optimisation_2017_2018/SÈances_2&3_TP/Les cours/RangerObjets')

Pos_cas = np.loadtxt('PositionCasiers.txt',skiprows=1)
Pos_obj = np.loadtxt('PositionObjets.txt',skiprows=1)

mat=spatial.distance_matrix(Pos_cas,Pos_obj,p=2)
dims=mat.shape

# déclaration de la variable de sélection
selection = cp.Variable((dims[0],dims[0]), boolean=True)

# déclaration des contraintes
constraints = [ cp.sum(selection[i,:]) == 1 for i in range(dims[0]) ]
constraints_col = [ cp.sum(selection[:,j]) == 1 for j in range(dims[1])]

# déclaration du critère
total_distance = cp.sum( mat.flatten() * selection.flatten() )

# déclaration du problème et solution
problem = cp.Problem(cp.Minimize(total_distance), constraints + constraints_col )
problem.solve(solver=cp.ECOS_BB)

if problem.status not in ["infeasible", "unbounded"]:
    # Otherwise, problem.value is inf or -inf, respectively.
    print("Optimal value: %s" % problem.value)
for variable in problem.variables():
    print("Variable %s: value %s" % (variable.name(), variable.value.round(0)**2))
```

Figure 21: Implémentation du problème des casiers

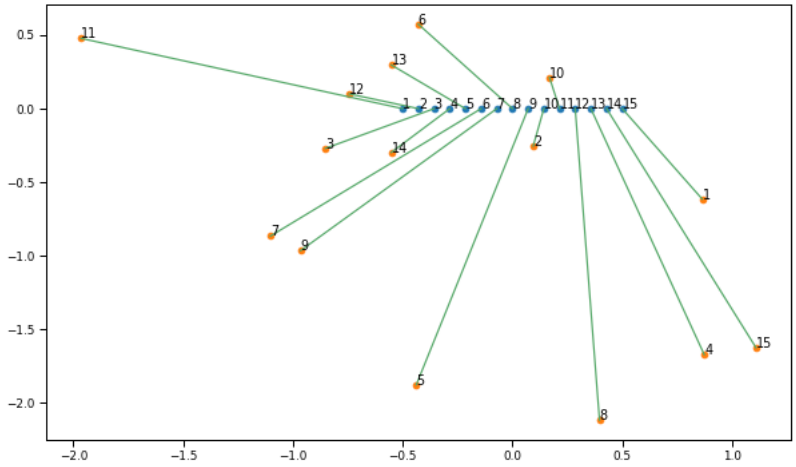


Figure 22: Résultats du problème des casiers

2.1.3 Question 3

Nous conservons la formulation du problème et ajoutons une contrainte supplémentaire, portant sur l'index i:

$$\forall i \in [\![2, n]\!], x_{i-1,1} = x_{i,2}$$

N.B: Le code pour le traitement des questions suivantes ainsi que de la question 3 est proposé dans la figure 23 qui récapitule l'ensemble des contraintes demandées au cours du problème.

2.1.4 Question 4

Nous rappellons que $\forall i > 0, \forall j > 0$, les $x_{i,j}$ sont à valeurs binaires et puisque chaque objet doit être rangé dans une boite, $\exists i_0$ tel que l'objet 3 soit placé dans cette boite. Par conséquent, si l'on se place en $i = i_0$ alors $x_{i,3} = 1$ et la condition donne $x_{i+k,4} = 0 \forall k > 0$. **Ainsi la boîte contenant l'objet 3 se trouve bien à droite de la boîte contenant l'objet 4**.

2.1.5 Question 5

Cette fois, la contrainte supplémentaire s'écrit:

$$\forall i \in [\![1, n]\!], x_{i,7} = x_{i-1,9} + x_{i+1,9}$$

En effet, une distinction sur les cas ou $x_{i,7}$ vaut 1 où 0 (l'objet 7 est en place i ou non) nous donne l'égalité ci-dessus. Avec la convention que les termes $x_{0,9}$ et $x_{n+1,9}$ sont nuls.

2.1.6 Question 6

Afin de vérifier que la solution optimale trouvée est unique, nous souhaitons retirer ce vecteur de $[\![0,1]\!]^{n^2}$ de l'ensemble des solutions possibles et appliquer à nouveau l'algorithme d'optimisation. Afin que notre algorithme s'applique à ce nouveau problème, nous devons introduire une contrainte qui reste linéaire. Or sachant que le vecteur solution est binaire. Nous pouvons par exemple contraindre le produit scalaire de notre solution ($vec_{nouveau}$) avec la solution optimale (vec_{opt}) à être strictement inférieur à n. (i.e, la nouvelle solution ne coincide pas avec l'ancienne sur au moins un point.)

$$< vec_{opt}, vec_{nouveau} >_{[\![0,1]\!]^{n^2}} < n$$

```
# Déclaration de la variable de selection des combinaisons boite-objet
selection = cp.Variable((dims[0],dims[0]), boolean=True)

#l'ensemble des contraintes proposés dans le problème
constraints = [ cp.sum(selection[i,:]) == 1 for i in range(dims[0])]
constraints_col = [ cp.sum(selection[:,j]) == 1 for j in range(dims[1])]

constraint_3 = [selection[0,i] == selection[1,i+1] for i in range(dims[0]-1)]

constraint_4=[]
for i in range(dims[0]-1):
    for k in range(i+1,dims[0]):
        constraint_4.append(selection[2,i] + selection[3,k] <= 1)

constraint_5 = [selection[6,i] == selection[8,i+1] + selection[8,i-1] for i in range(dims[0]-1)] #+ [selection[0,6]

#Conservation des contraintes dans un dictionnaire
Dic_const={}
Dic_const['2'] = constraints + constraints_col
Dic_const['3'] = Dic_const['2'] + constraint_3
Dic_const['4'] = Dic_const['3'] + constraint_4
Dic_const['5'] = Dic_const['4'] + constraint_5

# fonction critère a minimiser
total_distance = cp.sum( mat.flatten() * selection.flatten() )


#résolution des problèmes
for k in range(2,6):
    print('Question N°{}'.format(k))
    problem = cp.Problem(cp.Minimize(total_distance), Dic_const[str(k)] )
    problem.solve(solver=cp.ECOS_BB)

    if problem.status not in ["infeasible", "unbounded"]:
        # Otherwise, problem.value is inf or -inf, respectively.
        print("Optimal value: %s" % problem.value)
    #for variable in problem.variables():
        #print("Variable %s: value %s" % (variable.name(), variable.value.round(0)**2))
```

Figure 23: Implémentation des contraintes de la question 2 à 6 et de l'algorithme de résolution

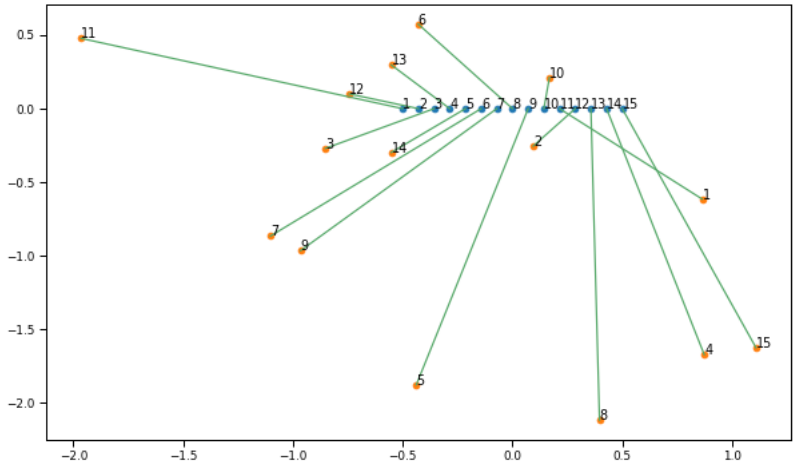


Figure 24: Représentation graphique de la solution du problème de la question 3

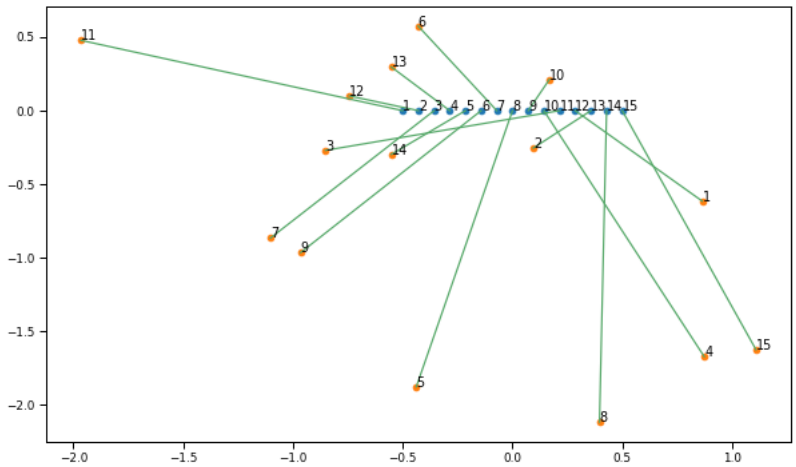


Figure 25: Représentation graphique de la solution du problème de la question 5

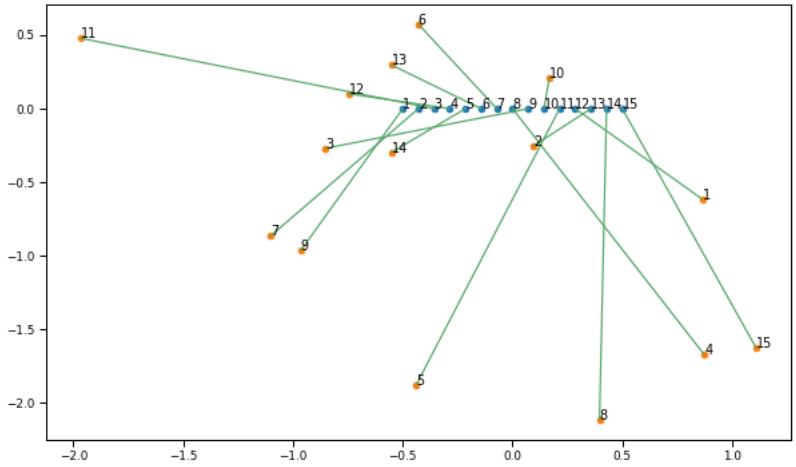


Figure 26: Représentation graphique de la solution du problème de la question 5

```
Question N°2
Optimal value: 15.377627810052134
Question N°3
Optimal value: 15.565123298218957
Question N°4
Optimal value: 15.95691618803305
Question N°5
Optimal value: 16.060295790982952
```

Figure 27: Distance minimale trouvée pour les questions 2 à 5

2.2 Communication entre espions (optimisation combinatoire)

2.2.1 Question 1

Le problème de communication entre espions s'apparente à la recherche d'un arbre couvrant de poids minimum. En effet, nous pouvons assimiler l'espace des espions ainsi que les probabilités d'interception comme un graphe pondéré non-orienté (car les agents i et j peuvent communiquer avec une probabilité d'interception $p_{i,j} = p_{j,i}$).

Il y a toutefois une discussion sur la pondération du graphe. En effet, notre problème est trouver le chemin tel que la probabilité d'interception entre l'agent 1 et un autre agent soit minimum. Il s'agit de construire les chemins $\{I_2, ..., I_{15}\}$ tels que:

$$\prod_{i,j \in I} (1 - p_{i,j}) \text{ soit maximale}$$

Par conséquent pour se ramener à un problème de plus cours chemin, nous composons par le logarithme néperien (sous réserve qu'aucune communication soit presque surement intercepté, i.e (1-$p_{j,i}$) > 0)et nous prenons l'opposé pour pondérer notre graphe. Ainsi la pondération d'un chemin est donnée par l'opposé du logarithme néperien des probas de non-interceptions:

$$\sum_{i,j \in I} -ln(1 - p_{i,j}) \forall I \in \{I_2, ..., I_{15}\}$$

Notre résolution s'assimilera donc à une recherche de plus court chemin que nous mettrons en oeuvre avec l'algorithme de Bellman-Ford ou Dijkstra (puisque la pondération est positive). Afin de modéliser le graphe en 2 dimensions, nous nous appuyons sur la librairie networkx qui permet de générer des coordonnées pour les points du graphe dont on ne connait que la matrice des distances. (c.f Fruchterman-Reingold force-directed algorithm pour plus de détails) Par la suite nous récupérons le plus court chemin, ainsi que la probabilité d'interception du message lors de sa diffusion depuis l'agent 1 aux autres agents. (figure 25)

2.2.2 Question 2

```
os.chdir('/Users/chalvidalm/Documents/3A OMA/OMA fonda/Optimisation_2017_2018/SÉances_2&3_TP')
Mat_Int = np.loadtxt('ProbaInterception.txt')

#Processing probabilities matrix and create Graph object
Mat_masked = np.ma.masked_invalid(Mat_Int)
Mat_masked = -np.log(np.ones(Mat_masked.shape) - Mat_masked)
Graph=nx.from_numpy_matrix(Mat_masked)
D = nx.DiGraph(Graph)

# Create force-directed layout for the graph
pos = nx.spring_layout(D)
pos_np = np.array(list(pos.values())).T

# Initialize figure
fig, ax = plt.subplots(1, 1,figsize = (16, 10))

ax.scatter(pos_np[0], pos_np[1])
for i in range(Mat_masked.shape[0]):
    ax.annotate(i+1, (pos_np[0,i], pos_np[1,i]))

#compute shortest path and plot edges
from scipy.sparse.csgraph import shortest_path
G, predecessors = shortest_path(Mat_masked, method='BF', return_predecessors=True)
```

Figure 28: Création de la matrice des distances et mise en oeuvre de l'algorithme de Bellman-Ford

Afin de résoudre ce problème, nous avons minimisé la fonction décrite précédemment. Nous trouvons une probabilité d'interception environ égale à **0.58** ainsi que la liste suivantes des communications de probabilité d'interception minimale depuis l'agent 1:

{[[4, 2], [4, 1], [14, 3], [14, 6], [6, 1], [3, 5], [3, 7], [4, 8], [4, 9], [5, 10], [14, 11], [1, 12], [5, 13], [6, 15]]}

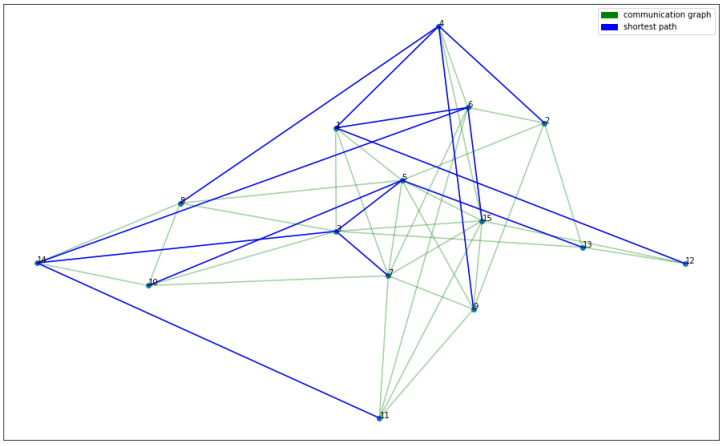


Figure 29: Représentation du graph des communications possible et arbre recouvrant de poids minimal de source 1

2.3 Dimensionnement d'une poutre

2.3.1 Méthode gloutonne

La méthode gloutonne consiste à tirer de façon aléatoire N_{iter} couples (a,b) puis d'estimer le front de Pareto en examinant la dominance de ces points. Pour ce faire, nous utilisons la fonction $numpy.random.random_sample(Niter)$ qui nous donne un échantillon de N tirages de la loi uniforme sur l'intervalle considéré. La loi uniforme semble adéquate pour pouvoir explorer l'ensemble du front de Pareto. En prenant bien compte des contraintes portant sur a et b, la figure 26 donne l'implémentation proposée ainsi que la fonction 'is pareto efficient' d'évaluation de la dominance des points de l'échantillon.

2.3.2 Méthode plus sophistiquée

Nous reformulons le problème multi-objectif comme un problème mono-objectif paramétrisé $J_\alpha(u) = J_1(u) + \alpha J_2(u)$ où alpha est un paramètre nous permettant de faire varier l'importance du critère J_2. Ainsi nous pouvons, pour des valeurs données de α, minimiser le critère J_α associé et reconstruire le front de Pareto.

```
Poids = lambda a: a[0]**2 - a[1]**2
Def = lambda a: 10**(-3)/(10**(-2) + a[0]**4 - a[1]**4)

Niter = 10000
B=[], Pds=[], Defl=[]

A = (1 - 0.02) * np.random.random_sample(Niter) + 0.02
for i in A:
    B.append((i -0.01) * np.random.random_sample())
tirage = list(zip(A,B))

for i in range(len(tirage)):
    Pds.append(Poids(tirage[i]))
    Defl.append(Def(tirage[i]))

Pareto=np.array(list(zip(Pds,Defl)))

def is_pareto_efficient(costs):
    is_efficient = np.ones(costs.shape[0], dtype = bool)
    for i, c in enumerate(costs):
        if is_efficient[i]:
            is_efficient[is_efficient] = np.any(costs[is_efficient]<=c, axis=1)  # Remove dominated points
    return is_efficient

Pareto_efficient=is_pareto_efficient(Pareto)
Pds_non_opti = np.extract(~Pareto_efficient,Pds)
Defl_non_opti = np.extract(~Pareto_efficient,Defl)
Pds_opti = np.extract(Pareto_efficient,Pds)
Defl_opti = np.extract(Pareto_efficient,Defl)
Pds_opti = np.extract(Pareto_efficient,Pds)
Defl_opti = np.extract(Pareto_efficient,Defl)

fig, ax =plt.subplots(1,1, figsize=(16, 10))
ax.scatter(Pds_non_opti,Defl_non_opti)
ax.scatter(Pds_opti,Defl_opti)
```

Figure 30: Méthode glouton pour l'évaluation du front de Pareto

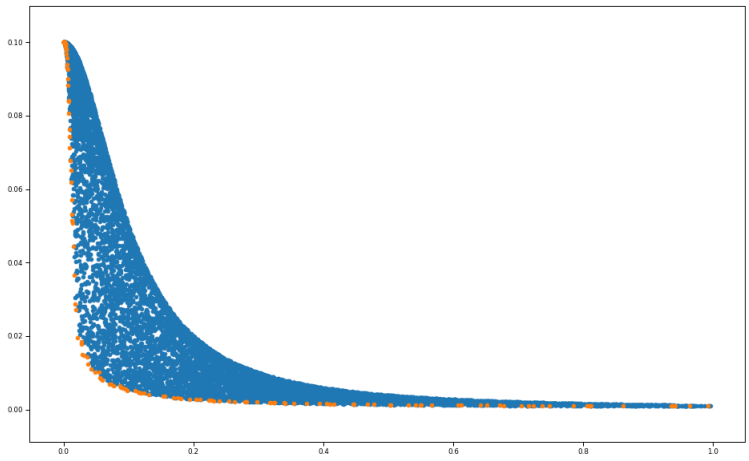


Figure 31: Représentation de l'échantillon tiré et du front de Paretto pour une résolution $N_{iter} = 10\ 000$ points

Question bonus

La résolution de ce problème peut également s'effectuer à partir d'algorithme génétique en prenant compte des potentiels phénomènes de nichage. Nous souhaiterions également obtenir des solutions stables par croisement. Pour cela, nous pourrions déterminer dans un premier temps une fonction de notation f permettant de prendre

```
cons={'type': 'ineq',
      'fun' : lambda x: - x[1] + x[0] - 0.01}

bounds = ((0.02, 1),(0, None))


fig, ax =plt.subplots(1,1, figsize=(16, 10))
#ax.scatter(Pds_non_opti,Defl_non_opti)
#ax.scatter(Pds_opti,Defl_opti)
list_=[]
for k in range(1,100):
    SQP = optimize.minimize((lambda v : Poids(v) + k*100*Def(v)),(0.5,0.48),method='SLSQP',
    constraints=cons,bounds=bounds)
    if SQP.success == True:
        ax.scatter(SQP.x[0],SQP.x[1], c='b')
```

Figure 32: Implémentation de la méthode par pondération pour l'estimation du front de Pareto

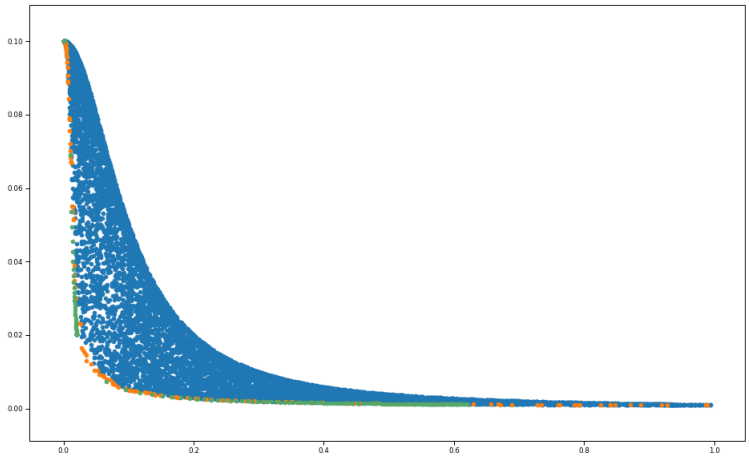


Figure 33: Résultat de la méthode par pondération pour l'évaluation du front de Pareto

en compte le rang de l'individu x de manière à pouvoir classer les individus par leur rang. f(x)=1+rang(x) où le rang serait le nombre de solutions qui dominent la solution x. Dans un second temps, nous pourrions introduire la notion de distance entre individus afin d'obtenir des solutions de manière régulière.

2.4 Approvisionnement d'un chantier (optimisation combinatoire)

Afin de résoudre ce problème nous pouvons directement implémenter un algorithme de type programmation dynamique car le problème vérifie le principe d'optimalité de Bellman. En effet, à chaque itération le choix optimal est fait pour minimiser le cout des équipements.

Une deuxième méthode consiste à représenter le chemin de décision commme un graphe pondéré par le cout de changement de statut de location d'un temps i à un temps $i+1$. Pour ce faire, il s'agit de représenter un graphe G possédant $n_{dates}xdmax$ avec n_{dates} le nombre de dates considérés et $dmax$ la demande maximale sur ces dates. Nous implémentons dans la figure 24 la matrice des distances relatives à ce graphe. Ensuite il suffit de résoudre un problème de plus court chemin.

Une troisième méthode consiste à construire un algorithme combinatoire. Nous pouvons dans un premier temps remarquer que les coûts pour rendre une machine s'élèvent à 1200 euros. Par conséquent, il est optimal de rendre une machine si et seulement si nous n'aurons pas à utiliser cette machine lors des 10 prochaines semaines. 10 semaines étant le point d'équilibre car 10*200=1200+800. De plus le coût de la première semaine est connu et vaut d1*800+0*200=28800 euros. Nous avons ensuite établi l'algorithme suivant:

Algorithme combinatoire

```
#initlisation
d1=demands.iloc[0]
c1=(d1-0)*800+200*d1
cost=[]
cost.append(c1)
print('La demande pour la semaine 1 est',d1,'machines','et le coût de',c1,'euros')

for i in range(0,98):
    if demands.iloc[i+1]>demands.iloc[i]:
        c=(demands.iloc[i+1]-demands.iloc[i])*800+demands.iloc[i+1]*200
    else:
        if max(demands.iloc[i+1:i+11])-demands.iloc[i]>=0:
            c=(demands.iloc[i])*200 # c'est plus intéressant de garder la machine
        else:
            c=(demands.iloc[i]-max(demands.iloc[i+1:i+11]))*1200+max(demands.iloc[i+1:i+11])*20
            #nouveau nombre de machine à louer à reporter dans df.Demand
            df.Demand_dt.iloc[[i+1]] = max(demands.iloc[i+1:i+11])
    cost.append(c)
sum(cost)
print('Le coût minimal est de',sum(cost),'euros')
```

```
La demande pour la semaine 1 est 36.0 machines et le coût de 36000.0 euros
Le coût minimal est de 3447800.0 euros
```

Figure 34: Algorithme combinatoire pour l'approvisionnement d'un chantier

Nous trouvons un coût minimum de 3,447,800 euros.