Saad Ben Cherif Ouedrhiri
Constance Goutay

TP 1 : Optimisation continue et optimisation approchée

1.1. Optimisation sans contraintes

1.1.1. Méthode du Gradient

a) *Essayer différentes valeurs de rho et voir que la convergence n'est pas toujours assurée*

Nous notons que pour des pas "grands" (rho = 0.5 ou rho = 0.1), l'algorithme ne converge jamais vers un minimum.

```
*** Méthode du gradient avec rho constant: 0.5 ***
Elapsed time is 0.179173 seconds.
gradient_rho_constant : minimum=NaN
*** Méthode du gradient avec rho constant: 0.1 ***
Elapsed time is 0.175283 seconds.
gradient_rho_constant : minimum=NaN
```

Le pas choisi est donc trop agressif et doit être choisi plus petit pour espérer qu'à chaque itération, x_n se rapproche du minimum.

Le plus grand rho donnant lieu à une convergence du problème avec le point d'initialisation 0 est compris entre 0.22 et 0.23, comme le prouvent les résultats suivants :

```
*** Méthode du gradient avec rho constant: 0.023 ***
GradResults =
  struct with fields:
     initial_x: [5×1 double]
       minimum: [5×1 double]
    f_minimum: NaN
   iterations: 10000
    converged: 0
Elapsed time is 0.232588 seconds.
gradient_rho_constant : minimum=NaN
```

```
*** Méthode du gradient avec rho constant: 0.022 ***
GradResults =
  struct with fields:
     initial_x: [5×1 double]
       minimum: [5×1 double]
    f_minimum: -1.8349
   iterations: 216
    converged: 1
Elapsed time is 0.021910 seconds.
gradient_rho_constant : minimum=-1.8349
```

b) Coder un algorithme du gradient avec choix adaptatif du pas

Algorithme de choix adaptatif du pas :

A chaque itération, on vérifie si $f(x_{n+1}) < f(x_n)$,

- Si c'est le cas, on double la valeur de rho, met à jour x_n <- x_{n+1} et on passe à l'itération suivante
- Sinon, on divise rho par 2, et on "reste au même endroit", i.e. on ne met pas à jour la valeur de x_n lorsqu'on passe à l'itération suivante

L'implémentation MATLAB choisie est la suivante :

```
%% Adaptative step implementation

while ~converged && it < itermax
    it=it+1;
    dfx=han_df(xn);        %Compute the gradient
    xnp1=xn-rho*dfx;       %Compute x(n+1)
    fnp1=han_f(xnp1);      %Compute f(x(n+1))
    call = call + 1;       %Increment the number of calls to the cost function
    if abs(fnp1-f)<tol     %Check convergence
        converged = true;
    end
    if fnp1 < f            %Compare f(x(n)) and f(x(n+1))
        rho = 2*rho;       %If x(n+1) better, move to it and double rho
        xn=xnp1;           %Otherwise, stay at x(n) and divide rho
        f=fnp1;
    else
        rho = rho/2;
    end
end
```

Notons que :

- La méthode converge quel que soit le pas initial choisi, si celui-ci est trop grand, il sera diminué jusqu'à adaptation. Par exemple :

```
*** Méthode du gradient avec rho adaptatif, rho initial = 0.5 ***
Elapsed time is 0.022325 seconds.
gradient_rho_adaptatif : minimum=-1.836
```

- La méthode converge plus rapidement lorsque le pas initial choisi est petit. En effet, il sera doublé pour que la descente soit plus rapide (c.f. question suivante). Par exemple :

```
*** Méthode du gradient avec rho constant: 0.01 ***
Elapsed time is 0.006939 seconds.
gradient_rho_adaptatif : minimum=-1.8345
```

c) Comparer les performances des méthodes (nombre d'itérations, nombre d'appels à la fonction coût, temps d'exécution)

Nous avons ajouté une variable "nb_appels" qui est incrémenté de 1 à chaque fois qu'un appel à la fonction f (mais pas df) est fait.

Pour avoir un temps consistant, nous prenons la moyenne sur 5000 minimisations.

Important : ces métriques dépendent du point d'initialisation, qui sera égal à 0 dans la suite de l'exercice 1.1.

Pour un pas rho = 0.01

	Rho constant	Rho adaptatif
# itérations	542	233
# appels	543	234
Temps moyen	9ms	4ms

Détail de nos résultats :

		0,2	0,1	0,023	0,022	0,01	0,001	0,00001
CONVERGENCE	rho constant	NON	NON	NON	(-1.8349)	(-1.8369)	(-1.8362)	(-1.0714)
	rho adaptatif	(-1.835)	(-1.8356)	(-1.836)	(-1.7169)	(-1.8345)	(-1.8242)	(-1.8355)
ELPASED TIME (s)	rho constant	0.246360	0.234634	0.232588	0.021910	0.038017	0.133744	0.197319
	rho adaptatif	0.025691	0.032208	0.039925	0.030659	0.028579	0.032970	0.032049
ITERATIONS	rho constant	10000	10000	10000	216	542	3753	10000
	rho adaptatif							

1.1.2. Méthode de Quasi-Newton (version BFGS)

On peut récupérer les performances de la méthode de Quasi-Newton en utilisant la variable "output" en sortie de fminunc.

	Rho constant	Rho adaptatif	BFGS sans grad	BFGS avec grad
# itérations	542	233	13	13
# appels	543	234	96	16
Temps moyen	9ms	4ms	8.4ms	4.8ms

On note :

- L'algorithme du gradient avec rho adaptatif fonctionne très bien ici ; on voit la pourquoi les descentes de gradients sont appréciées : c'est un algorithme simple à comprendre et à implémenter qui donne souvent de bons résultats.
- Le nombre d'itérations nécessaires et donc d'appels à la fonction sont significativement inférieurs dans les méthodes BFGS.
- Avoir une expression analytique du gradient permet de faire moins d'appels à la fonction de coût et ainsi de réduire significativement le temps avant convergence.

1.2. Optimisation sous contraintes

1.2.1. Optimisation à l'aide de routines Matlab

On souhaite maintenant minimiser f_1 sue Uad = $[0;1]^5$

Il y a deux manières d'imposer la contrainte Uad = $[0 ;1]^5$ sous Matlab

1. En définissant A et b tels que Ax ≤ b avec $A = [eye(5); -eye(5)]$ et $b = [ones(5,1); zeros(5;1)]$
2. Ou en définissant des bornes supérieure (*ub*) et inférieures (*lb*) de la façon suivante $lb = zeros(5,1)$ et $ub = ones(5,1)$

L'instruction fmincon nous permet ensuite de résoudre ce nouveau problème contraint.

Résultat de l'optimisation du problème sous contraintes :

SQP	**f1**	**f2**
# itérations	9	4
# appels	68	30
Temps moyen	8.7ms	7ms
Minimum	-0.13853	0

Notez que comme U = (0, 0, 0, 0, 0) est la solution optimale de f2, on choisit U0 = (½, ½, ½, ½, ½).

La convergence est rapide.

1.2.2. Optimisation sous contraintes et pénalisation

1) Définir une fonction de pénalisation β

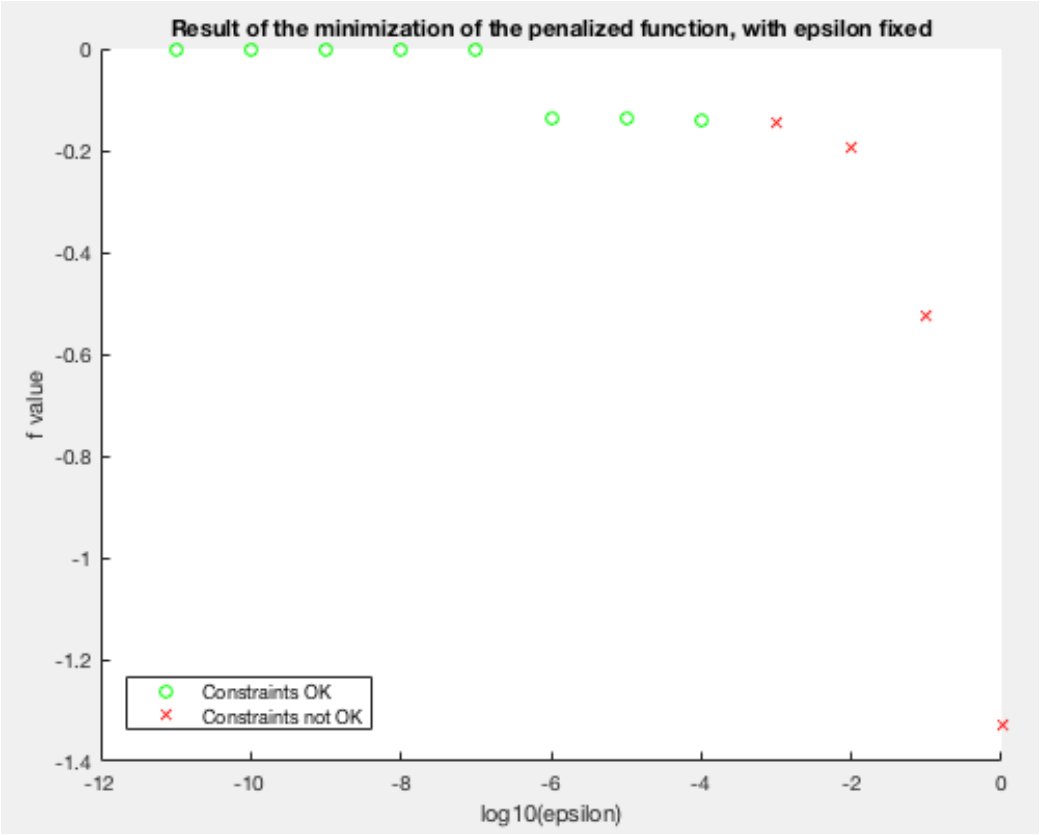
Les 5 axes sont contraints dans le pavé U_{ad}.
On définit donc β(U) =$\sum_{i=1}^{5}((Ui-1)^{+})^2 + ((0-Ui)^{+})^2$, avec $(.)^{+} = \max(0,.)$. On notera que l'élévation au carré permet de rendre la pénalisation différentiable.

Sur Matlab :

```
function beta=penalisation(U)
beta = sum(max(0,U-1).^2 + max(0, -U).^2);
```

2) Mettre en œuvre la méthode de pénalisation

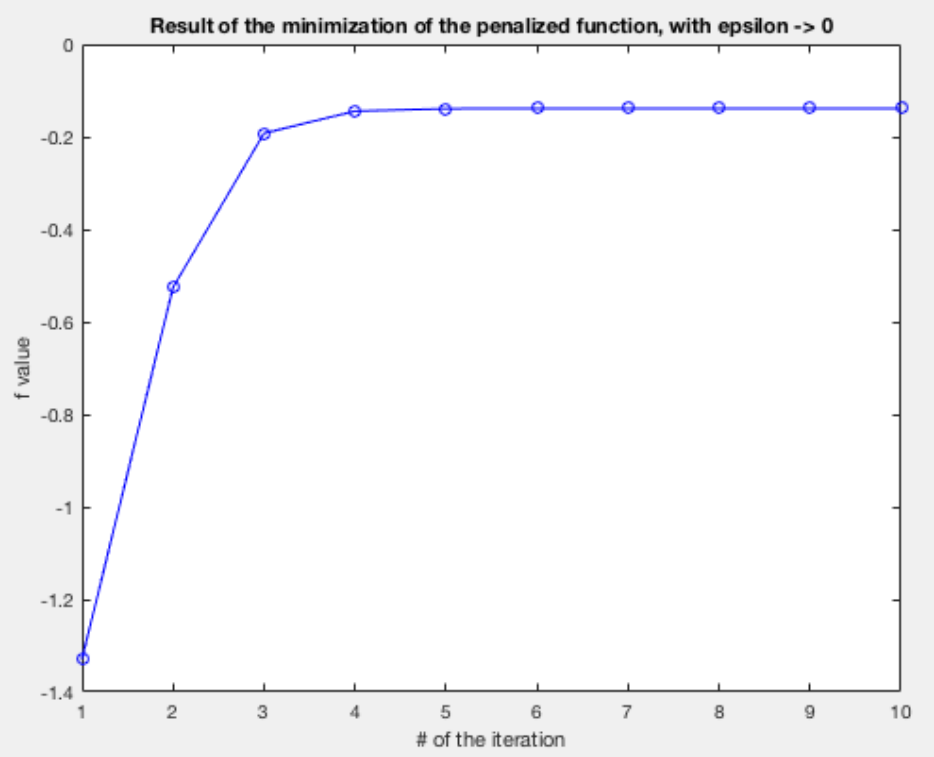
Essayons d'abord de résoudre le problème de minimisation non contraint de la fonction pénalisé, avec ε fixé, i.e. sans faire tendre ε vers 0.



Nous notons trois différents régimes :

- Pour ε fixé très petit, on ne converge pas vers le minimum attendu. Dans ce cas la partie pénalisée « domine » la fonction.
- Pour ε « moyen », nous notons un plateau pour lequel on obtient bien le minimum attendu, et pour lequel les contraintes sont bien respectées.
- Pour ε « grand », nous avons des valeurs de f inférieures à celles attendues, et les contraintes ne sont plus respectées. La partie « pénalisée » n'est plus assez grande pour imposer de rester dans le cadre voulu.

Pour ne pas avoir à choisir ε, on le fait tendre vers 0. A l'étape n+1, on effectue un problème de minimisation non contrait avec $\varepsilon_{n+1} = \varepsilon_n/10$ et le point de départ égal au résultat de la minimisation à l'étape n.



On peut noter qu'après 4 itérations seulement, on a déjà convergé vers le (bon) minimum, -0.139.

1.2.3. Méthodes duales pour l'optimisation sous contraintes

1) Ecrire le Lagrangien associé à f1 et aux contraintes définissant Uad.

Le Lagrangien est :

$$L1 : R^5 \times R_+^{10} \to R$$

$$L1(U, \lambda) = f1(U) + \lambda^T * (AU - b)$$

Sur Matlab, on l'implémente de la sorte :

```
function fU=L1(U, B, S,lambda)
A = [eye(5);-eye(5)];
b = [ones(5,1);zeros(5,1)];
fU = f1(U,B,S) + transpose(lambda)*((A*U)-b);
```

2) Mettre en œuvre l'algorithme d'Uzawa.

Uzawa	f1
# itérations	3010
# appels à L1	88488 (!)
Temps moyen	16.8s (!)

L'implémentation d'Uzawa ici donne des résultats significativement moins bons que les deux précédentes techniques. Pour cause, à chaque itération, on résout un problème de minimisation non contraint.

De plus, bien que la solution obtenue (fval = -0.1389), soit proche de la solution optimale (fval = -0.1385), la solution ne respecte pas au sens stricte du terme les conditions. On obtient certaines composantes Un qui sont (très faiblement) inférieures à 0.

```
 0.0001
 0.1270
-0.0001
 0.0195
-0.0001
```

L'implémentation Matlab est la suivante :

```
tic
call = 0; %Counting the number of function calls
while ~converged && iter <= maxiter
    iter = iter + 1;
    han_L = @(u) L1(u,B,S,lambdan);  %In these 2 lines, we minimize respect to u, L(., lambdan)
    [xn, Ln, EXITFLAG, OUTPUT] = fminunc(han_L, xnm1, options);
    call = call + OUTPUT.funcCount;
    if abs(Ln-Lnm1) < tol
        converged = true;
        break;
    else
        gradLn = (A*xn-b); %Gradient of L1 respect to lambda
        lambdan = max(0, lambdan + rho*gradLn); %the new lambda is the projection of lambdan + rho*gradLn
        Lnm1 = Ln; %Updating the values of of xnm1 and L(xnm1, lambdanm1) for next loop
        xnm1 = xn;
    end
end
toc
```

1.3. Optimisation non convexe – Recuit simulé

a) Définir la fonction f4.

On introduit la non-convexité avec le sinus. Nous n'avons plus l'unicité de la solution, pire, nous avons des minimums locaux.

```
function fU=f4(U,B,S)
fU = f1(U,B,S) + 10*sin(2*f1(U,B,S));
```

b) Tester une méthode d'optimisation classique (BFGS). Remarques ?

En utilisant une méthode classique, on remarque que pour différents points d'initialisation (générés aléatoirement dans le code), l'algorithme convergera vers différents minimums, en général vers des minimums locaux.

Dans la simulation faite sur Matlab, pour 100 points d'initialisation différents dans $[0,1]^5$, on obtient 22 minimums différents :

```
Using Quasi-Newton
Elapsed time is 0.758513 seconds.
Minimums found:
  Columns 1 through 11

   20.6180   14.3348    1.7685    4.9101   -7.6563   -4.5147   -1.3731   11.1932  -10.7979   23.7596   17.4764

  Columns 12 through 22

   26.9012   33.1844   52.0340   39.4676   55.1755   36.3260   30.0428   48.8924   64.6003    8.0517   58.3171
```

Figure 1 - Résultat de BFGS avec U0 aléatoires

On notera que « le minimum des minimum » trouvé ici est **-10.7979**. La plupart du temps cependant, on converge vers un minimum local.

c) Tester la méthode du recuit simulé.

On utilise une méthode de type grid-search pour définir les paramètres optimaux « InitialTemperature » et « ReannealInterval ». Pour chaque paire de paramètres, on résout 10 fois le problème avec une initialisation aléatoire U0 dans $[[0,10]]^5$, et on compte le nombre de fois ou le minimum -10,7979 est trouvé. Plus la variable de comptage est proche de 10, et plus les paramètres sont adaptés.

```
Ini Temp = 26, Rean Int = 1, Min found = 6
Ini Temp = 26, Rean Int = 26, Min found = 6
Ini Temp = 26, Rean Int = 51, Min found = 7
Ini Temp = 26, Rean Int = 76, Min found = 5
Ini Temp = 26, Rean Int = 101, Min found = 3
Ini Temp = 51, Rean Int = 1, Min found = 3
Ini Temp = 51, Rean Int = 26, Min found = 8
Ini Temp = 51, Rean Int = 51, Min found = 7
Ini Temp = 51, Rean Int = 76, Min found = 6
Ini Temp = 51, Rean Int = 101, Min found = 6
Ini Temp = 76, Rean Int = 1, Min found = 3
Ini Temp = 76, Rean Int = 26, Min found = 8
```

Figure 2 - Résultat de la grid-search pour quelques paires de paramètres

Pour de faibles températures initiales, l'exploration ne se fait pas sur assez de zones, et on converge rarement vers le minimum. Pour des températures initiales trop grande, on tombe dans le travers inverse. Finalement un bon compromis semble être une température autour de 50/75 et environ 25 transformations par palier.

```
ini_temps = 1:25:101;
rean_ints = 1:25:101;

for ini_temp= ini_temps
    for rean_int = rean_ints
        min_found = 0;
        for i=1:nb_min
        U0 = rand(d,1)*10;
        options = optimoptions('simulannealbnd', 'ReannealInterval', rean_int,'InitialTemperature',ini_temp,'MaxIterations', 10000,
        [x,fval,exitflag,output] = simulannealbnd(han_f4, U0, lb, ub, options);
        if round(fval,4) == -10.7979
            min_found = min_found+1;
        end
        end
        disp(['Ini Temp = ', num2str(ini_temp), ', Rean Int = ' , num2str(rean_int), ', Min found = ', num2str(min_found)]);
    end
end
```

Figure 3 - Implémentation de la grid-search

1.4. Application : Synthèse d'un filtre à réponse impulsionnelle finie

1) Minimiser le critère avec un algorithme d'optimisation sans contraintes (simplexe de Nelder and Mead)

On définit le critère J(h) pour une discrétisation donnée

```
function Jh=J(h, discretisation)

Jh = 0;
for nu=discretisation
    ecart = abs(H0(nu) - H(h,nu));
    if ecart > Jh
        Jh = ecart;
    end
end
```

Figure 4 - Définition de J

avec H0 et H définis comme suit

```
function H0nu=H0(nu)

if and(0 <= nu,nu <= 0.1)
    H0nu = 1;
elseif and(0.15 <= nu, nu <= 0.5)
    H0nu = 0;
else
    H0nu = 50;
end
```

Figure 5 - Définition de H0

```
function Hnu=H(h, nu)


cosi = arrayfun(@(x)double(cos(2*pi*nu*x)), double((1:length(h))'));
Hnu = h'*cosi;

end
```

Figure 6 - Définition de H

Les routines de type simplexe Nelder and Mead ou Newton aboutissent à une erreur que nous ne sommes pas arrivés à contourner…

```
Error using arrayfun
Non-scalar in Uniform output, at index 1, output 1.
Set 'UniformOutput' to false.
```

2) Reformulation du problème comme un problème linéaire

On introduit une constante gamma telle que :

$$\min_{h,\gamma} \gamma$$

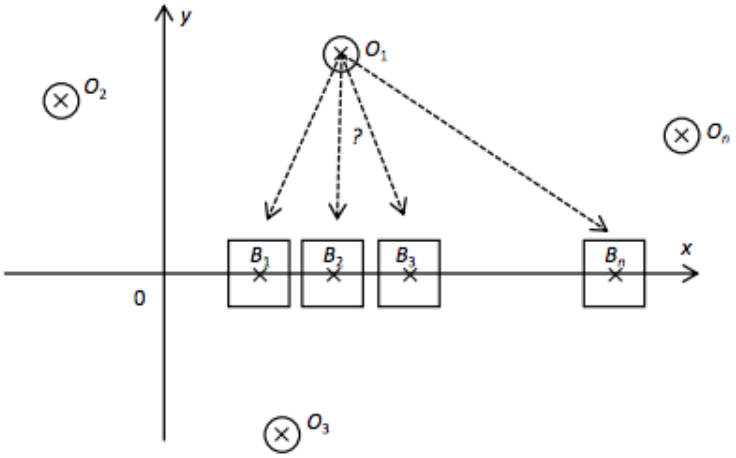
$$s.c.\{\forall j, H(v_j) \leq H_0(v_j) + \gamma$$

TP 2&3 : Optimisation discrète et optimisation multi-objectif

2.

2.1. Rangements d'objets (optimisation combinatoire - PLNE)

Il s'agira dans ce problème d'assigner à chaque objet une unique boîte en minimisant la distance totale de déplacement. Nous avons 15 objets et 15 boites.



1) Traduire mathématiquement que la boîte i contient un objet et un seul et que l'objet j se trouve dans une boîte et une seule.

- La boîte i contient 1 seul objet ⇔ $\sum_j x_{ij}$ = 1
  - Cette contrainte s'applique pour toutes les boites (pour tout i), on obtient donc 15 contraintes
- L'objet j se trouve dans une seule boite ⇔ $\sum_i x_{ij}$ = 1
  - Cette contrainte s'applique pour tous les objets (pour tout j), on obtient donc 15 nouvelles contraintes

Nous pouvons remarquer que $\sum_{ij} x_{ij} = n$ découle des équations précédentes, signifiant que tous les objets sont rangés/toutes les boites sont occupées.

Important : les solutions (distance optimale et vecteurs) pour toutes les questions sont réunies à la fin de l'exercice.

2) Formuler le problème d'optimisation à résoudre comme un PLNE

Le problème de programmation linéaire en nombre entiers s'écrit :

$$(PLNE)\ \min_x \sum_{ij} ||Oj - Bi|| * x_{ij}\ \ s.c.\ \begin{cases} \sum_i x_{ij} = 1 \text{ pour tout i} \\ \sum_j x_{ij} = 1 \text{ pour tout j} \end{cases}$$

3) Ajouter la contrainte : O1 doit se situer dans la boîte juste à gauche de la boite contenant O2

La formulation mathématique de la contrainte dans le cas général est : $x_{i-1,1} - x_{i,2} = 0$ pour tout i dans [[2,15]].
A ces 14 contraintes, on doit ajouter celle qui empêche O1 d'être dans la boîte 15. En effet, dans ce cas, O2 pourrait être dans la boite 1 et alors on aurait les 14 contraintes respectées, mais O1 ne serait pas juste à gauche de O2. On ajoute donc $x_{15,1} = 0$ (et $x_{1,2} = 0$).

4) Montrer que $x_{i,3} + x_{i+k,4} \leq 1\ \forall i, \forall k$ ⇔ O3 est à droite de O4

<=
Supposons que O3 soit à droite de O4 et qu'il existe $i_0, k_0 > 0$ tels que $x_{i0,3} + x_{i0+k0,4} > 1$. On aura nécessairement $\begin{cases} x_{i0,3} = 1 \\ x_{i0+k0,4} = 1 \end{cases}$, ce qui signifie que O3 est à gauche de O4, ce qui est absurde.

=>

Supposons maintenant qu'on ait $x_{i,3} + x_{i+k,4} \leq 1\ \forall i, \forall k$ et que O3 soit à gauche de O4.

Notons i_3 la position de 3, et i_4 la position de 4. On a $i_4 - i_3 = k' > 0$ car O3 est à gauche. On a donc $x_{i3,3} + x_{i3+k',4} = 2$, ce qui est absurde.

Cependant, le nombre de contraintes induit par cette formulation est trop grand (de l'ordre de $factorielle(14)$). On peut trouver une formulation équivalente en écrivant $\forall i \in [[1,14]], x_{i,3} + \sum_{k>i}^{k=15} x_{k,4} \leq 1$.
On a bien si O3 est dans la boite i, alors tous les $x_{k,4}, k > i$ sont forcément nuls, ce qui signifie O3 est à droite de 4. Nous n'avons plus que 14 contraintes avec cette écriture.

Note : la formulation en haut impose aussi que O3 ne soit pas dans le bord droit.

5) Ajouter la contrainte : O7 se site à côté de O9.

En faisant abstraction des contraintes au bord, la contrainte s'écrit $(x_{i,7} - x_{i-1,9})(x_{i,7} - x_{i+1,9}) = 0$

⇔ $x_{i,7}(x_{i,7} - x_{i-1,9} - x_{i+1,9}) = 0$.

Cette contrainte est non linéaire. La stratégie adoptée pour résoudre cette question est donc la suivante :

1. On impose que O7 soit à gauche de O9, avec des contraintes similaires à la Q3, i.e. pour tout i dans [[2,15]], $x_{i-1,7} - x_{i,9} = 0$ (+ contrainte au bord). On résout le problème de minimisation et on récupère fval1.

2. On impose maintenant que O9 soit à gauche de O7, i.e. $x_{i-1,9} - x_{i,7} = 0$ (+ contrainte au bord). On résout le problème de minimisation et on récupère fval2.
3. On choisit pour le problème de Q5 le minimum entre fval1 et fval2.

Cette stratégie aboutit à un minimum atteint lorsque O7 est juste à gauche de O9 (cf. solutions à la fin de l'exercice)

6) On aimerait vérifier que la solution obtenue est la seule solution optimale.

Soit x_{opt} l'assignement obtenu au terme de la question 5 et fval la distance totale associée. La stratégie pour vérifier l'unicité de la solution est d'ajouter, tour à tour et pour chaque objet, une contrainte bloquant l'assignement proposé par x_{opt}. Si on obtient un nouveau fval' égale, alors on aura exhibé une autre solution x_{opt}' optimale, et la solution n'est pas unique.

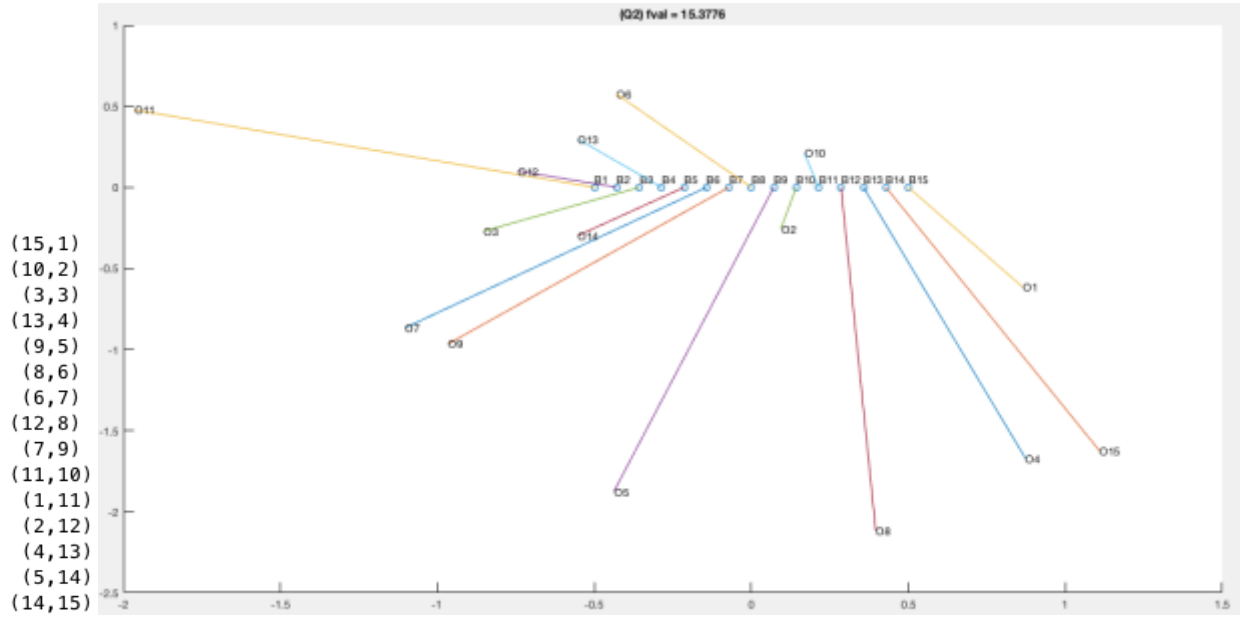
On obtient par exemple que bloquer l'assignement de O14 à B3 (ou de O13 à 4 ; en fait O14 et O13 sont « swapés ») donne une même distance totale (arrondie à 4 chiffres après la virgule). La solution n'est donc pas unique (cf. solutions).

SOLUTIONS

On représentera ici les vecteurs par les x_{ij} non nuls. Par ex, (15, 1) signifie la boite 15 contient l'objet 1.

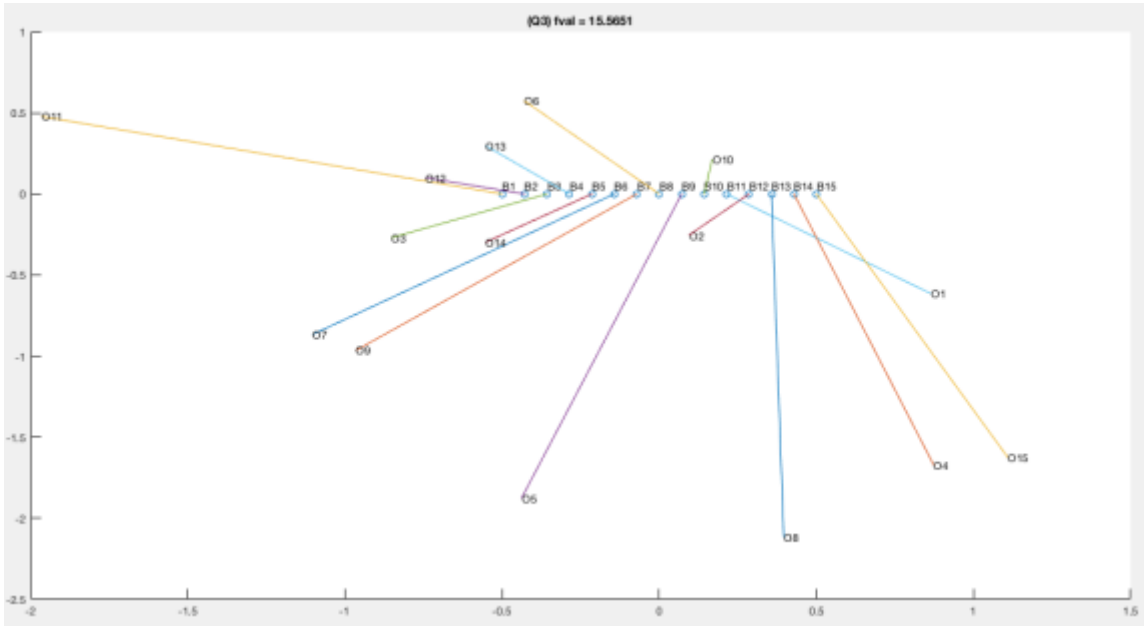
(Q2)

- Distance optimale : **15.3776**
- Vecteur et représentation :



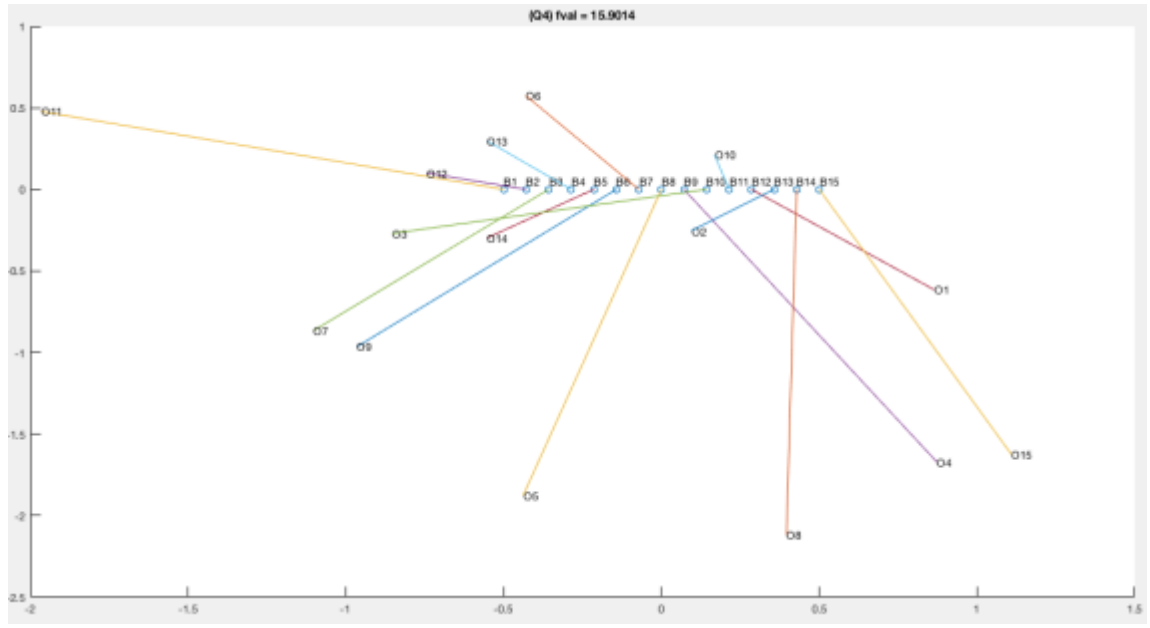
(Q3)

- Distance optimale : **15.5651**
- Vecteur et représentation :



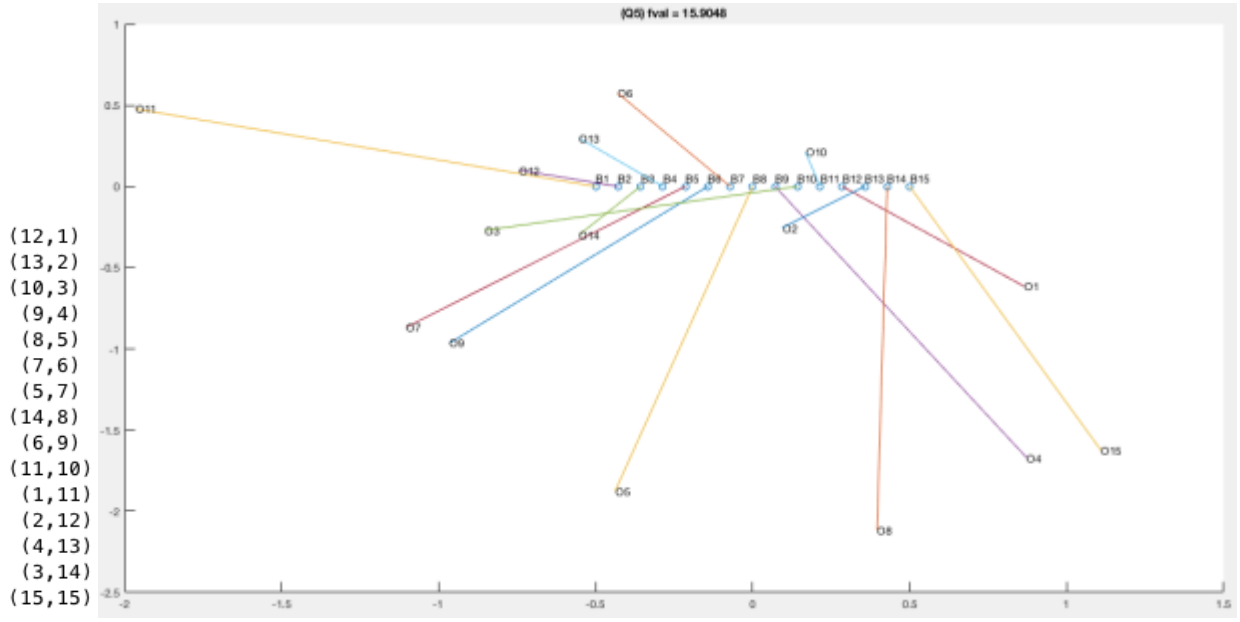
(Q4)

- Distance optimale : **15.9014**
- Vecteur et représentation



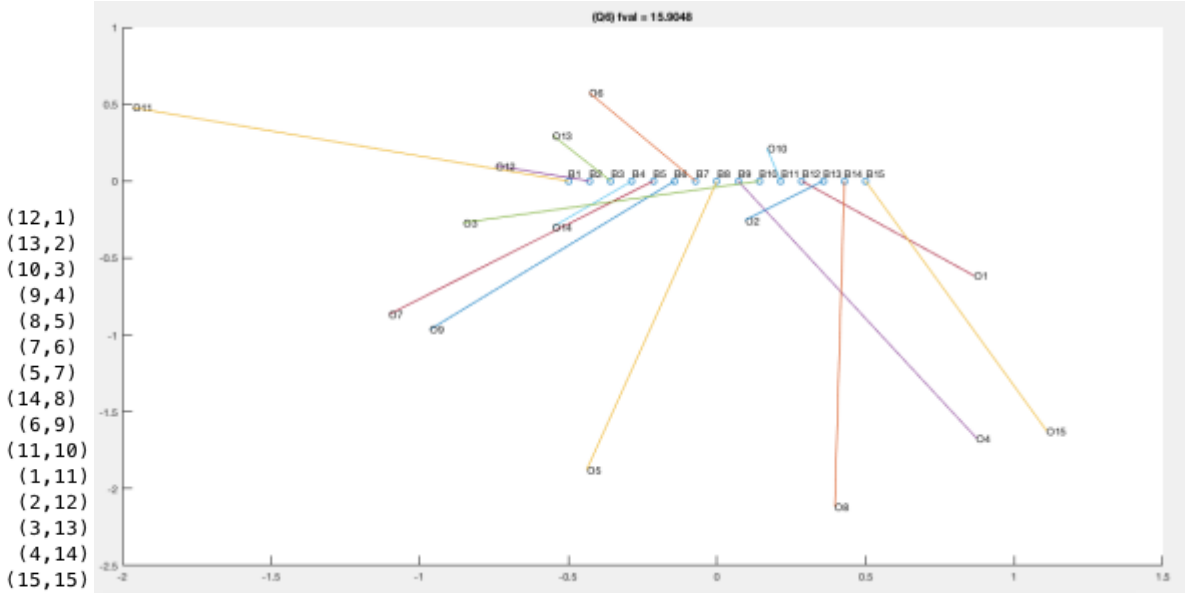
(Q5)

- Distance optimale : **15.9048**
- Vecteur et représentation :



(Q6) Résultat en bloquant l'assignement O14 à B3

- Distance optimale : **15.9048**
- Vecteur et représentation :



2.2. Communication entre espions (optimisation combinatoires)

1) Modélisation du problème

On peut modéliser le réseau d'agents par un graphe non orienté G = (X, E) avec X = {agents} et E = {arêtes entre 2 agents pour lesquels la communication est possible}.

Si on veut que l'agent 1 puisse transmettre son message à tous les autres agents, il faut que pour toute paire de points, il y ait une chaîne reliant ces deux points. C'est la définition d'un graphe connexe.

On peut alors penser au problème d'arbre de recouvrement minimum vu en cours. Il reste à trouver les poids associés à chaque arête e.

Comme on veut minimiser la somme de ces poids, on essaie de se ramener à cette situation. Dans la suite, on notera Aij l'évènement « le message est intercepté pendant une communication entre l'agent i et l'agent j » de probabilité p_{ij}.

Minimiser la probabilité d'interception du message entre tous les agents, c'est maximiser la probabilité de non-interception.

$$P(non\ interception) = P\left(\bigcap_{i\neq j} \overline{Aij}\right) = \prod_{i\neq j} P(\overline{Aij}) = \prod_{i\neq j} (1 - p_{ij})$$

On se ramène à une somme grâce au log. Finalement on veut donc minimiser la fonction :

$$-\sum_{i\neq j} \log(1 - p_{ij})$$

On associe à chaque arête la valeur $-\log(1-p_{ij})$.

D' où la formulation finale suivante du problème du problème de recouvrement minimal :

On cherche le graphe partiel G' = (X, E') tel que $\begin{cases} G' est\ un\ graphe\ connexe \\ \sum_{\{i,j\}\in E'} -\log(1 - p_{ij})\ \ est\ min \end{cases}$

2) Résolution du problème

La résolution du problème donne

- une probabilité d'interception P(interception) = 0.5809
- avec le graphe connexe suivant :

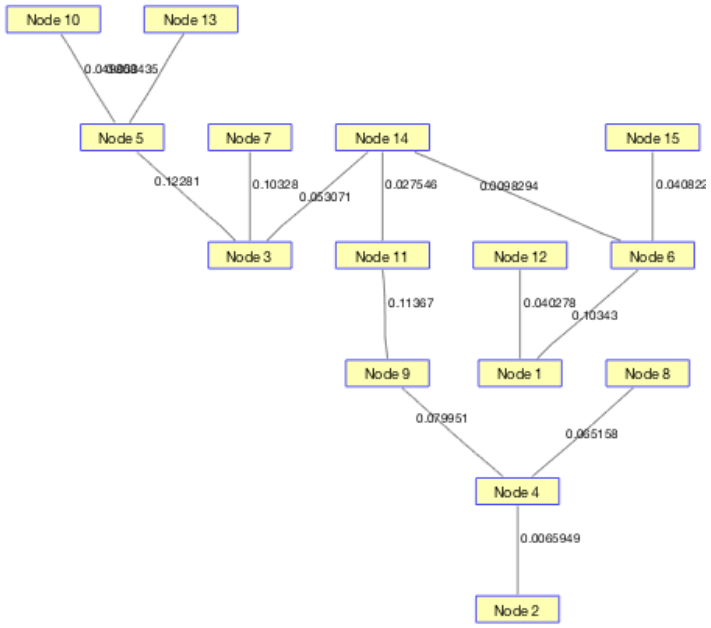


Figure 7 - Graphe connexe minimisant l'interception du message

Note 1 : on obtient exactement le même graphe en minimisant directement le graphe dont les arêtes sont pondérées par la probabilité d'interception p_{ij}. Même si la somme des p_{ij} n'a pas de « sens physique », on comprend cependant qu'elle aboutit à une bonne solution.

Note 2 : Matlab utilise des matrices creuses (« sparse matrix ») pour représenter les graphes. C'est l'entrée de la procédure de minimisation graphminspantree.

Note 3 : soit $m = \min \sum_{\{i,j\}\in E'} -\log(1 - p_{ij})$, on a alors $P(interception) = 1 - e^{-m}$

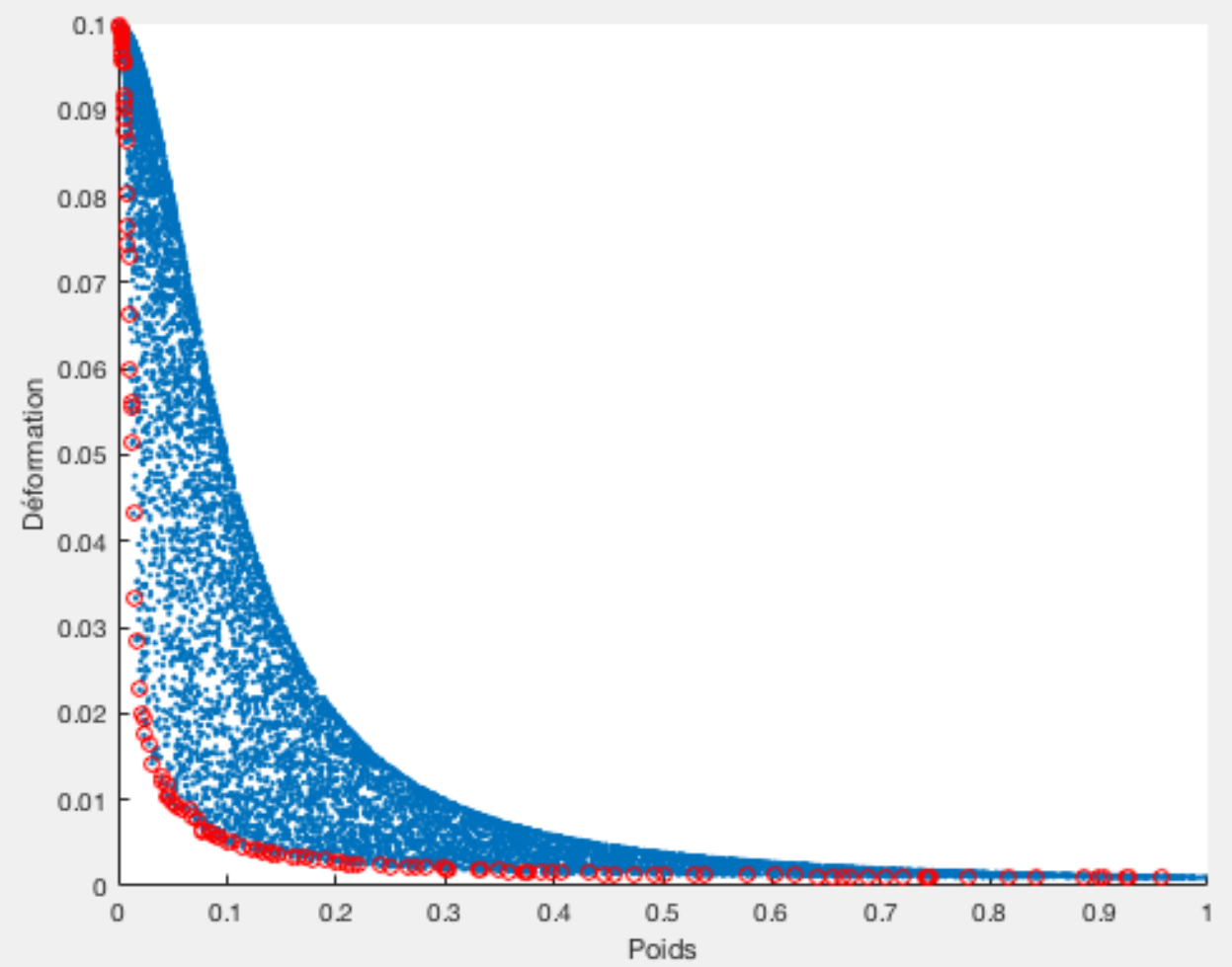
2.3. Dimensionnement d'une poutre (optimisation multi-objectif)

1) Méthode gloutonne :

Dans cette méthode, on génère aléatoirement 10000 points, puis on itère sur tous les points pour définir s'il est dominé par d'autres points ou non. Pour chaque point :

- On récupère l'index des points qui sont situés à sa gauche
- Parmi ces points, on récupère l'index des points qui sont situés en dessous
- Si ce deuxième ensemble est vide, alors le point fait partie du front de Pareto

Résultat :



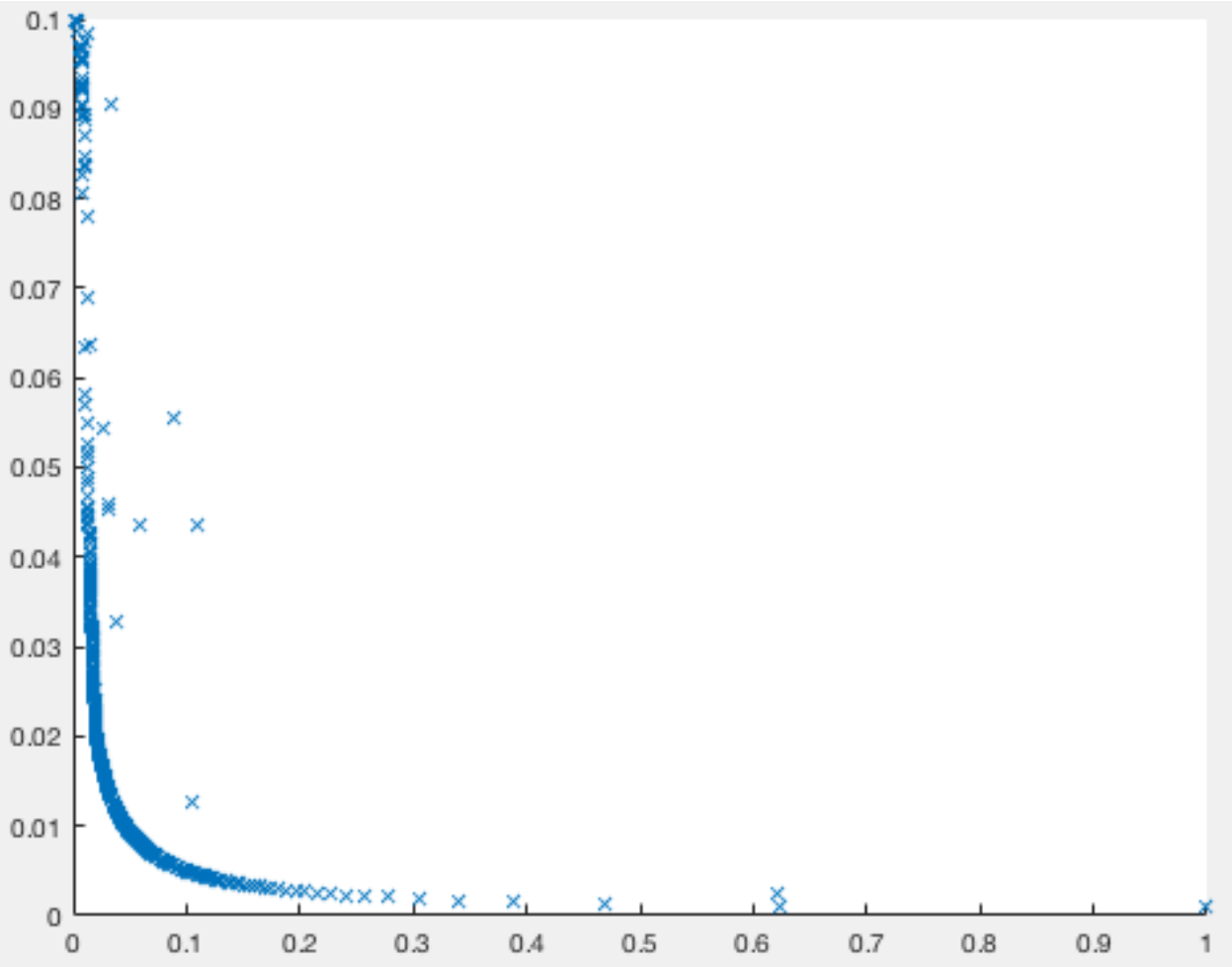
Même si la région ou la pente est forte est moins fournie en points, on se fait une idée claire du front de Pareto.

2) Méthode plus sophistiquée

Ici, on résout, pour plusieurs valeurs de lambda, le problème de minimisation monocritère sans contraintes suivant :

$$\min_{a,b} \lambda * p(a,b) + (1-\lambda) * d(a,b)$$

On obtient pour lambda variant de 0 à 1 avec un pas de 0.001 (1000 minimisations), on obtient le front de Pareto suivant, ressemblant fortement à la figure plus haut :



Implémentation Matlab :

```
disp('SQP');
x_min = [];
Aineq = [1 0;-1 0;0 -1;-1 1];
bineq = [1;-0.002;0;-0.01];
tolerance = 1e-9;

for lambda=0:0.001:1
    han_J = @(x) J(x, lambda);
    options = optimoptions('fmincon', 'TolFun', tolerance, 'Display', 'off');
    [X, FVAL, EXITFLAG, OUTPUT] = fmincon(han_J,rand(2,1),Aineq,bineq,[],[],[],[],[],options);
    x_min = [x_min; X'];
end

p = x_min(:,1).^2 - x_min(:,2).^2;
d = 1e-3./(1e-2 + x_min(:,1).^4 - x_min(:,2).^4);

hold on
plot(p,d, 'LineStyle', 'none', 'Marker', 'x');
axis([0 1 0 0.1]);
hold off
```

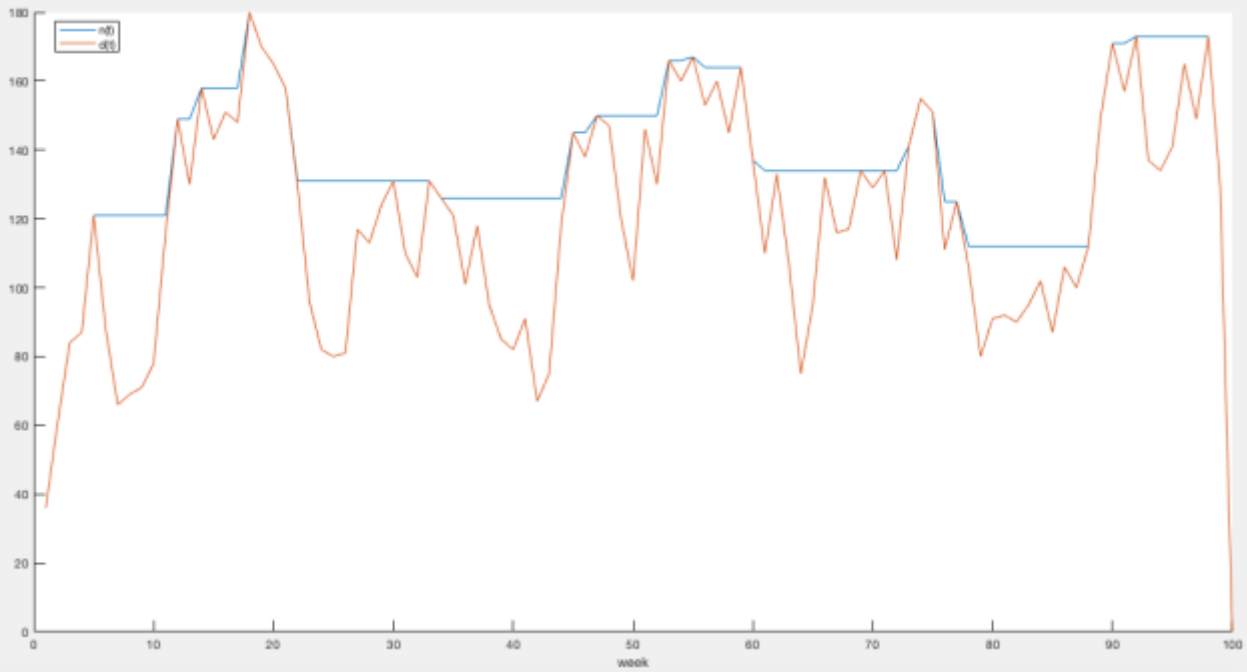
2.4. Approvisionnement d'un chantier

On commencera par donner le résultat (coût minimal). On expliquera ensuite notre formulation du problème. Enfin, on analysera certains résultats pour différents coûts.

RESULTAT :

- Coût minimal : 3 311 200
- Représentation graphique du résultat :

n(t) est le nombre de machines que l'entreprise de construction possède à l'instant t, et d(t) et le nombre de machines dont l'entreprise a besoin à l'instant à t



L'algorithme donne une solution qui suit « à peu près » la stratégie suivante : « on ne rend une machine que si on ne l'utilise pas pendant plusieurs semaines », i.e. si le prix de location pendant ces semaines est supérieur au prix de sortie.

FORMULATION

Ce problème consiste à trouver les 2 fonctions :

- a(t), qui, à une semaine t, associe le nombre de machines à acquérir à t
- r(t), qui, à une semaine t, associe le nombre de machine à rendre à t

On peut déduire de ces deux fonctions n(t), le nombre de machines disponibles à t, définie par :

$$n(t) = \sum_{i=1}^{t} (a(i) - r(i))$$

Soit $x = (a(1), a(2), \dots a(100), r(1), \dots, r(100))$ l'argument et $f^T x$ la fonction de coût linéaire à minimiser. Les contraintes à respecter sont :

- le nombre de machines disponibles est toujours supérieur au besoin : $\forall t, n(t) \geq d(t)$
- On ne doit avoir aucune machine à la dernière semaine : $n(100) = 0$
- On ne peut rien rendre à la première semaine : $r(0) = 0$
- $\forall t, a(t) \geq 0 \; et \; r(t) \geq 0 \; est \; sont \; entiers$

Il s'agit donc d'un problème de programmation linéaire en nombre entiers. Une autre formulation possible aurait été un plus court chemin dans un graphe (non exploré ici), car la situation est assez similaire à celle de la gestion des approvisionnements dans le temps, vue en cours.

REMARQUES SUR RESULTAT :

Plus les coûts d'entrée p^{init} et de sortie p^{fin} sont petits devant le prix de location, moins l'entreprise a intérêt à garder des machines qu'elles n'utiliseraient pas, et plus les deux courbes n(t) se rapproche de d(t). Ainsi, dans le cas extrême ou $p^{init} = p^{loc} = 0$, n(t) et d(t) sont confondus.

Inversement, si ces prix sont très grands devant le prix de location, l'entreprise garde la machine dans tous les cas, à part si elle ne la réutilisera pas au terme des 100 semaines.

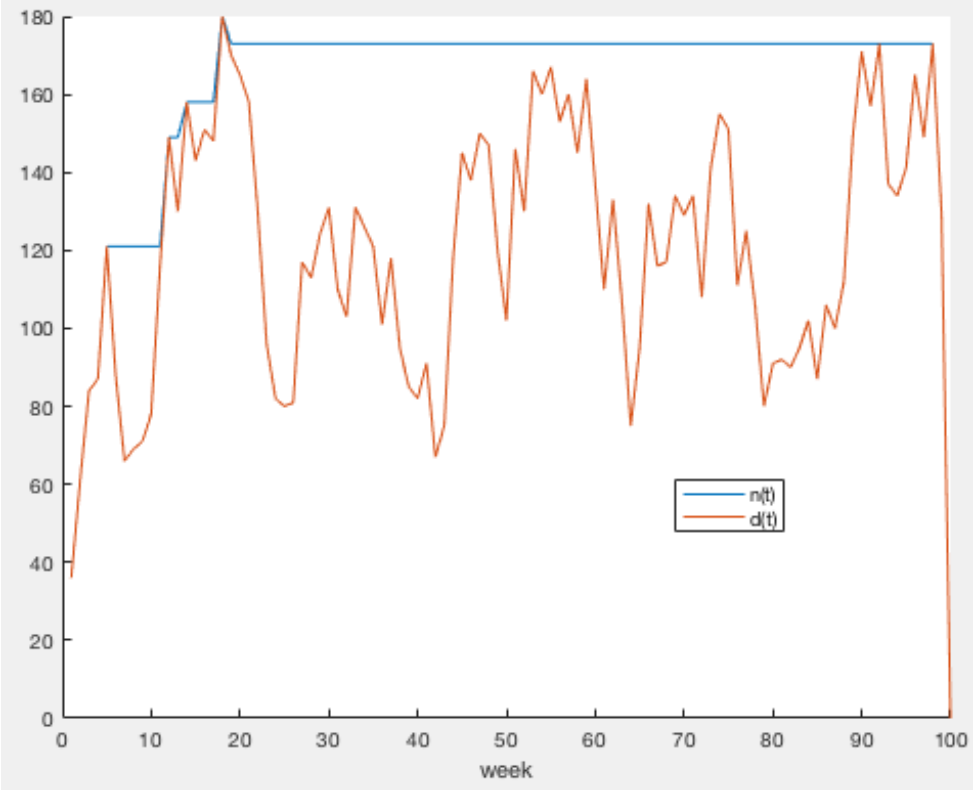


Figure 8 - Stratégie avec pinit=pfin >> ploc