LinearProblemAnalysis

February 4, 2020

1 Programmation linéaire

Création de la classe LP :

Dans cette partie on va commencer par définir la classe LP qui permet de créer un objet modélisant un problème d'optimisation linéaire.

Les attributs de LP :

- shape : un couple d'entiers (m, n) où m représente le nombre de contraintes (hors contraintes de positivités) et n le nombre de variables du programme linéaire.
- basic_sol : la solution de base du programme linéaire LP donné comme un numpy array.
- basic_feasible : un booléen à True si le programme linéaire a une solution de base admissible et à False sinon.
- basic_vars : un dictionnaire dont les clés sont {1, . . . , m} et les valeurs prises dans {0, . . . , n 1} (les indices des variables du programme).
- table : un objet numpy.array (ou mieux un pandas.DataFrame avec colonnes nommées) de taille (m + 1, n + m + 1).

Les méthodes de LP :

- _feasible_() : retourne True si le problème admet une solution de base admissible.
- basic_vars_() : retourne la base de la solution.
- table_() : retourne la table représentant le programme linéaire.
- basic_sol() : retourne la solution de base du problème linéaire.
- pivot(e,s) : applique le pivotement du problème avec un pivot désigné par (e,s).
- dual() : construit le problème dual.

[2]:
```
import pandas as pd
import numpy as np


class LP(object):
```

```
    def __init__(self, a, b, c, v=0):

        if not all(isinstance(i, np.ndarray) for i in (a, b, c)):
            raise TypeError("Arguments of numpy type array expected")
        # On teste si on a les bonnes dimensions
        if a.shape != (b.shape[0], c.shape[0]):
            raise TypeError("Arguments are not of coherent dimensions")
        # On récupère les m et n à partir de la dimension de la matrice a
        m, n = a.shape
        self.shape = (m,n)
        self.basic_vars={} # Le dictionnaire est vide au départ
        # Remplissage du dictionnaire
        for i in range(self.shape[0]):
            self.basic_vars[i]= self.shape[1]+i

        # On construit la table taille (m + 1, n + m + 1) initialisée à des zéros
        self.table = np.zeros((m+1, n+m+1), dtype=float)
        self.table[0, :] = np.hstack((-c, np.zeros(self.shape[0], dtype=float),␣
↪np.array(v)))
        self.table[1:, :] = np.hstack((a, np.eye(self.shape[0], dtype=float), b))
        self.basic_feasible=all(self.basic_sol() >= 0)



    def basic_feasible_(self):
        return self.basic_feasible
    def basic_vars_(self):
        return basic_vars
    def table_(self):
        return self.table

    def basic_sol(self):
        vect = np.zeros(self.shape[0]+self.shape[1], dtype=float)
        for i in range(self.shape[0]):
            vect[self.basic_vars[i]] = self.table[i+1, -1]
        return vect


    def pivot(self,e,s):
        if (e not in self.basic_vars.values() and s in self.basic_vars.
↪values())==False :

            raise TypeError("Error")

        a = self.table
        k=[c for c,v in self.basic_vars.items() if v==s][0]
```

```
        pivot = a[k+1, e]

        if pivot == 0:
            raise ValueError("Ne peut pas pivoter avec de telles variables.")

        a[k+1, :] = a[k+1, :]/pivot

        for i in range(self.shape[0]):
            if i != k:
                a[i+1, :] = a[i+1, :] - a[i+1,e]*a[k+1, :]

        a[0, :] = a[0, :] - a[0, e]*a[k+1, :]


        self.basic_vars[k] = e



    def dual(self):

        a1=self.table[1:,:self.shape[1]].transpose()
        b1=self.table[0,:self.shape[1]].reshape((self.shape[1],1))
        c1=self.table[1:,-1]
        lp = LP(-a1,-b1, c1)
        return lp
```

Test de la classe LP

Création d'une classe LP:

[3]:
```
borne_sup=100
m=4
n=3
a=np.random.randint(-borne_sup,borne_sup, size=(m, n))
b=np.random.randint(-borne_sup,borne_sup, size=(m, 1))
c=np.random.randint(-borne_sup, borne_sup, n)
lp=LP(a,b,c)
```

La table représentant le problème linéaire :

[4]:
```
lp.table_()
```

[4]:
```
array([[ 14., -18., -84.,   0.,   0.,   0.,   0.,   0.],
       [-32.,  62., -36.,   1.,   0.,   0.,   0.,  52.],
       [ 89.,  47., -82.,   0.,   1.,   0.,   0., -76.],
       [-34., -40.,  52.,   0.,   0.,   1.,   0., -24.],
       [ 43., -38., -58.,   0.,   0.,   0.,   1.,  94.]])
```

[5]:
```
print("La taille du problème est :", lp.shape )
```

```
La taille du problème est : (4, 3)
```

Le problème dual est obtenu en effectuant un certains nombre d'opérations sur la table d'origine.

[6]:
```
lp.dual().table_()
```

[6]:
```
array([[-52.,  76.,  24., -94.,   0.,   0.,   0.,   0.],
       [ 32., -89.,  34., -43.,   1.,   0.,   0., -14.],
       [-62., -47.,  40.,  38.,   0.,   1.,   0.,  18.],
       [ 36.,  82., -52.,  58.,   0.,   0.,   1.,  84.]])
```

[7]:
```
print("La taille du problème dual est :", lp.dual().shape)
```

```
La taille du problème dual est : (3, 4)
```

Dans certains cas, il est plus facil de résoudre le problème dual que le problème primal. Et grâce à un théorème de dualité, on prouve que si le problème dual admet une solution optimale alors il en est de même pour le problème primal et que les deux problèmes ont la même valeur objective optimale.

Implémentation de la classe _Simplex

Cette classe va permettre la résolution d'un problème linéaire ayant une solution de base admissible. **Les attributs de _Simplex :**

- entering : fonction qui prend en entrée une liste correspondante à la première ligne de la table sans le dernier élément et renvoie une variable entrante pour le pivot.
- leaving : fonction qui prend en entrée une liste renvoie une variable.

sortante. **La méthode de _Simplex**

- _Simplex a seulement une méthode call qui s'applique sur un objet de type LP qui est supposé avoir une solution de base admissible. Elle retourne le problème linéaire résolu le temps d'exécution, le nombre d'appel de la fonction pivot, la valeur objective et la solution.

Test de bornitude

Pour le test de la bornitude on utiliste une condition faible vue en cours qui dit que pendant chaque appel de la fonction pivot, s'il existe une colonne entièrement négative alors le problème est non borné. Si cette condition n'est pas satisfaite, rien ne prouve que le problème est borné, mais dans l'algorithme qu'on implémente la variable non_borne devient False pour dire que cette condition est violée.

[8]:
```
import time
class _Simplex(object):


    # attributs :
    # - entering: fonction qui prend en entrée une instance de LP et renvoie une
    #   variable entrante pour le pivot.
```

```
    # - leaving: fonction qui prend en entrée une instance de LP, une variable
    #   entrante et renvoie une variable sortante.

    def __init__(self, leaving_index=None, entering_index=None):

        #Ici l'utilisateur peut faire le choix des fonctions de recherche␣
↪d'indices
        if not leaving_index:
            leaving_index = funcLeavingIndex

        if not entering_index:
            entering_index = funcEnteringIndex

        self.leaving = leaving_index
        self.entering = entering_index

    def __call__(self, lp):

        a = lp.table


        non_borne=True
        n = 0 #itérateur récursif
        cpt=0
        start_time = time.time()#début d'enregistrement du temps
        while any(a[0, :-1] < 0) :

            entering_choices = [i for i in map(lambda x: 0 if x > 0 else x,a[0, :
↪-1])]

            e = self.entering(entering_choices) # on choisit l'indice de la plus␣
↪petite valeur négative

            leaving_choices = [None]*lp.shape[0]
            # On divise la dernière colonne par les coefficients de la colonne␣
↪de l 'indice d'entrée
            for i in range(lp.shape[0]):
                if a[i+1, e] > 0:
                    leaving_choices[i] = (a[i+1, -1]/a[i+1, e])
            if not [i for i in leaving_choices if i]:
                raise OverflowError("Linear program unbounded | check model and␣
↪state.")
            else:
                # l'indice correspont à la valeur de la clé à partir du␣
↪dictionnaire
                s = lp.basic_vars[self.leaving(leaving_choices)]
```

```
            lp.pivot(e, s)
            ####test de bornitude####
            for i in range(lp.shape[0]+lp.shape[1]):
                if any(a[:,i] >= 0):
                    cpt=cpt+1
            if cpt==lp.shape[0]+lp.shape[1]:
                non_borne=False
            cpt=0


            n += 1
        execution_time=time.time()-start_time # fin d'enregistrement du temps
        return lp, lp.basic_sol(), lp.table[0, -1], n, execution_time,non_borne
```

Choix de l'heuristique : On choisit deux fonctions entring et leaving

Par défaut la classe _Simplex prend les deux fonctions funcLeavingIndex(), funcEnteringIndex() qui ne sont pas optimales pour le choix des indices et ne permettent pas une convergence rapide vers la solution.

[9]:
```
def funcLeavingIndex(l):
    m = 0
    while not l[m] and m < len(l):
        m += 1
    if m == len(l):
        return 0
    for i in range(len(l)):
        if l[i] and l[m] > l[i]:
            m = i
    return m

def funcEnteringIndex(l):
    return l.index(min(l))
```

Création de la classe Simplex

Cette classe va permettre de traiter n'importe quel problème linéaire ou bien en le résolvant ou alors en retournant un message d'erreur si celui si n'admet pas une solution.

Les méthodes de la classe Simplex

_phase_one(lp) :

- _phase_one : prend en entrée une instance de lp. Si l'instance a une solution de base admissible _phase_one ne modifie pas l'instance en entrée et renvoie un booléen à True. Sinon elle construit l'instance lp marqueur correspondante et fait appel à _Simplex pour un test d'admissibilité. Dans le cas de non-admissibilité _phase_one renvoie un booléen à False.

Dans le cas admissible _phase_one renvoie un booléen à True et modifie en place l'instance lp en entrée pour en faire un programme linéaire équivalent ayant une solution de base admissible.

- _call : En fonction de la nature du problème, cette méthode retourne la solution ou un message d'erreur.

[10]:
```
class Simplex(_Simplex):

    def  _phase_one(self,lp):

        if lp.basic_feasible:
            print("Le problème linéaire admet une solution de base", end="\n\n")
            return True

        gain_fun = np.copy(lp.table[0])

        lp.shape = (lp.shape[0], lp.shape[1] + 1)
        lp.table = np.insert(lp.table, 0, -1, axis=1)
        lp.table[0] = np.hstack((np.ones(1, dtype=float),
                                 np.zeros(lp.shape[1]+lp.shape[0],␣
 ↪dtype=float)))


        for i in lp.basic_vars.keys():
            lp.basic_vars[i]=lp.basic_vars[i]+1



        l =   lp.basic_vars[np.argmin(lp.table[1:, -1])]
        lp.pivot(0, l)

        if _Simplex.__call__(self, lp)[2] == 0:
            print(" ### le problème linéaire est faisable", end="\n\n")

            if 0 in lp.basic_vars.values():
                l = [c for c,v in lp.basic_vars.items() if v==0][0]

                e = 0
                while e < lp.shape[1] and lp.table[l, e] == 0:
                    e += 1
                lp.pivot(e, l)


            for i in lp.basic_vars.keys():
                lp.basic_vars[i]=lp.basic_vars[i]-1

            lp.table = lp.table[:, 1:]
```

```
            lp.shape = (lp.shape[0], lp.shape[1] - 1)

            lp.table[0] = gain_fun
            for i in lp.basic_vars.values():
                l = [c for c,v in lp.basic_vars.items() if v==i][0]
                lp.table[0, :] = lp.table[0, :] - \
                                 lp.table[0, i] * \
                                 lp.table[1 + l, :]
            lp.table[0, -1] = -lp.table[0, -1]

            return True

        else:
            return False


    def __call__(self, lp):
        simplex = _Simplex()

        if lp.basic_feasible:
            print("Le problème a une solution de base admissible")
            return simplex(lp)
        elif not lp.basic_feasible and lp.dual().basic_feasible:
            print("Le problème dual a une solution de base admissible")
            return simplex(lp.dual())
        elif self._phase_one(lp):
            print("Ni le problème ni son dual n'admettent de solutions␣
↪admissibles mais le problème équivalent en admet")
            return simplex(lp)
        else:
            raise Exception("Linear program is not feasible.")
```

Générateur d'instances de LP

Afin de tester les omplémentations, on construit un générateur d'instances de LP.

- gen_LP(borne_sup,m,n,i) : Fonction qui prend en argument la taille d'un intervalle symétrique de borne supérieure borne_sup et (m,n) la taille du problème linéaire ainsi qu'un entier i qui désigne le nom du problème et retourne un dictionnaire sérialiser contenant toutes les informations sur le problème ("name", "LP", "heuristique", "shape", "basic_feasible", "feasible", "dual_basic_feasible", "bounded")
- gen_LP_df(borne_sup,m,n,n_instance) : Fonction qui prend en argument la taille d'un intervalle symétrique de borne supérieure borne_sup et (m,n) la taille du problème linéaire ainsi qu'un entier n_instance qui désigne les nombre des problèmes linéaires à générer et retourne un dataframe pandas contenant toutes les informations sur le problèmes générés ("name", "LP"(serialisé), "heuristique", "shape", "basic_feasible", "feasible", "dual_basic_feasible", "bounded").

[11]:
```
import pickle
import pandas as pd

from pickle import load, dump
from io import BytesIO

def gen_LP(borne_sup,m,n,i):

    Ins={}


    a=np.random.randint(-borne_sup,borne_sup, size=(m, n))
    b=np.random.randint(-borne_sup,borne_sup, size=(m, 1))
    c=np.random.randint(-borne_sup, borne_sup, n)
    lp=LP(a,b,c)

    spx =Simplex()
    _spx=_Simplex()
    try:


        Ins["name"]=i
        Ins["LP"]=lp
        Ins["heuristique"]=1
        Ins["shape" ]=lp.shape
        Ins["basic_feasible"]=lp.basic_feasible
        Ins["feasible"]= spx._phase_one(lp)
        Ins["dual_basic_feasible"]=lp.dual().basic_feasible
        Ins["bounded"]=not _spx(lp)[5]
    except :
        print("Error")

    serial=pickle.dumps(Ins)

    return Ins



def gen_LP_df(borne_sup,m,n,n_instance):



    df = pd.DataFrame(columns = ["name", "LP",␣
↪"heuristique","shape","basic_feasible","feasible","dual_basic_feasible","bounded"])
    for i in range(1,n_instance):

        Ins={}
```

```
        a=np.random.randint(-borne_sup,borne_sup, size=(m, n))
        b=np.random.randint(-borne_sup,borne_sup, size=(m, 1))
        c=np.random.randint(-borne_sup, borne_sup, n)
        lp=LP(a,b,c)

        spx =Simplex()
        _spx=_Simplex()
        try:

            df.loc[i,"name"]=i
            df.loc[i,"LP"]=pickle.dumps(lp)
            df.loc[i,"heuristique"]=1
            df.loc[i,"shape" ]=lp.shape
            df.loc[i,"basic_feasible"]=lp.basic_feasible
            df.loc[i,"feasible"]= spx._phase_one(lp)
            df.loc[i,"dual_basic_feasible"]=lp.dual().basic_feasible
            df.loc[i,"bounded"]=not _spx(lp)[5]
        except :
            print("Error")


    return df
```

On génère un dataframe pandas de 100 lignes avec la fonction gen_LP_df()

[12]:
```
df=gen_LP_df(10,3,4,1000)
```

Affichage de quelques premières lignes de la table df

[13]:
```
df.head()
```

[13]:

	name	LP	heuristique	shape	basic_feasible	feasible	dual_basic_feasible	bounded
1	1	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	(3, 4)	True	True	False	True
2	2	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	(3, 4)	False	False	False	False
3	3	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	(3, 4)	False	True	True	NaN
4	4	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	(3, 4)	False	True	False	True
5	5	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	(3, 4)	True	True	False	False

1-Génération d'un problème ayant une solution de base admissible et borné

[14]:
```
df_ad_bo=df[ (df["basic_feasible"]==True) & (df["bounded"]==True)]
```

[15]:
```
df_ad_bo=df_ad_bo.reset_index()
df_ad_bo.head()
```

[15]:

	index	name	LP	heuristique	\
0	1	1	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	
1	80	80	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	
2	90	90	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	
3	94	94	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	
4	107	107	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	

	shape	basic_feasible	feasible	dual_basic_feasible	bounded
0	(3, 4)	True	True	False	True
1	(3, 4)	True	True	False	True
2	(3, 4)	True	True	False	True
3	(3, 4)	True	True	False	True
4	(3, 4)	True	True	True	True

On choisit un des problèmes ayant une solution admissible et borné et on le désérialise.

[16]:
```
Lp=pickle.loads(df_ad_bo.loc[0]["LP"])
```

[17]:
```
Lp.table
```

[17]:
```
array([[-3.,  1., -3.,  5.,  0.,  0.,  0.,  0.],
       [ 4.,  2., -8., -6.,  1.,  0.,  0.,  9.],
       [ 5.,  0.,  8., -1.,  0.,  1.,  0.,  0.],
       [ 1., -8.,  4.,  3.,  0.,  0.,  1.,  7.]])
```

[18]:
```
spx=Simplex()
L=spx(Lp)
print("La solution du problème est :", L[1])
print("La valeur objective est",L[2])
print("Le temps d'exécution est de",L[4],"s")
```

```
Le problème a une solution de base admissible
La solution du problème est : [ 1.     0.    -0.625  0.     0.     0.     8.5  ]
La valeur objective est 1.125
Le temps d'exécution est de 0.00038504600524902344 s
```

2-Génération d'un problème ayant une solution de base admissible et non borné

[19]:
```
df_ad_bo=df[ (df["basic_feasible"]==True) & (df["bounded"]==False)]
df_ad_nb=df_ad_bo.reset_index()
df_ad_nb
```

[19]:
```
    index name                                                LP heuristique  \
0       5    5  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
1     123  123  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
2     223  223  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
3     250  250  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
4     450  450  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
5     531  531  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
6     565  565  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
7     731  731  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
8     848  848  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
9     864  864  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
10    886  886  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
11    980  980  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
12    982  982  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1

     shape basic_feasible feasible dual_basic_feasible bounded
0   (3, 4)           True     True                False   False
1   (3, 4)           True     True                False   False
2   (3, 4)           True     True                False   False
3   (3, 4)           True     True                False   False
4   (3, 4)           True     True                False   False
5   (3, 4)           True     True                False   False
6   (3, 4)           True     True                False   False
7   (3, 4)           True     True                False   False
8   (3, 4)           True     True                False   False
9   (3, 4)           True     True                False   False
10  (3, 4)           True     True                False   False
11  (3, 4)           True     True                False   False
12  (3, 4)           True     True                False   False
```

On choisit un des problèmes ayant une solution admissible et non borné et on le désérialise.

[20]:
```
buffer = BytesIO(df_ad_nb.loc[0]["LP"])
Lpb = load(buffer)
Lpb.table
```

[20]:
```
array([[ 9.,  6.,  8.,  9.,  0.,  0.,  0.,  0.],
       [ 4.,  4.,  8.,  4.,  1.,  0.,  0.,  9.],
       [-9.,  2.,  3.,  4.,  0.,  1.,  0.,  4.],
       [ 7.,  0.,  3.,  2.,  0.,  0.,  1.,  7.]])
```

[21]:
```
spx=Simplex()
M=spx(Lpb)
print("La solution du problème est :", M[1])
print("La valeur objective est",M[2])
print("Le temps d'exécution est de",M[4],"s")
```

```
Le problème a une solution de base admissible
```

```
La solution du problème est : [0. 0. 0. 0. 9. 4. 7.]
La valeur objective est 0.0
Le temps d'exécution est de 5.0067901611328125e-05 s
```

3-Génération d'un problème n'ayant pas de solution de base admissible mais un dual ayant une solution de base admissible

[22]:
```
df_pa_da=df[ (df["basic_feasible"]==False) & (df["dual_basic_feasible"]==True)]
df_pa_da=df_pa_da.reset_index()
df_pa_da.head()
```

[22]:

	index	name	LP	heuristique	\
0	3	3	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...		1
1	10	10	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...		1
2	12	12	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...		1
3	25	25	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...		1
4	34	34	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...		1

	shape	basic_feasible	feasible	dual_basic_feasible	bounded
0	(3, 4)	False	True	True	NaN
1	(3, 4)	False	True	True	NaN
2	(3, 4)	False	True	True	NaN
3	(3, 4)	False	True	True	True
4	(3, 4)	False	True	True	NaN

On choisit un des problèmes ayant une solution non admissible et mais son dual l'est.

[27]:
```
buffer = BytesIO(df_pa_da.loc[0]["LP"])
Lp_pa_da = load(buffer)
Lp_pa_da.table
```

[27]:
```
array([[-5., -4., -5., -7.,  0.,  0.,  0.,  0.],
       [-8., -8., -1.,  0.,  1.,  0.,  0., -2.],
       [-4.,  0., -9., -6.,  0.,  1.,  0.,  8.],
       [ 9., -1.,  1.,  5.,  0.,  0.,  1., -9.]])
```

[28]:
```
spx=Simplex()
M=spx(Lp_pa_da)
print("La solution du problème est :", M[1])
print("La valeur objective est",M[2])
print("Le temps d'exécution est de",M[4],"s")
```

```
Le problème dual a une solution de base admissible
La solution du problème est : [0.         0.55555556 0.         2.77777778 4.
0.
 3.66666667]
La valeur objective est 4.444444444444445
Le temps d'exécution est de 0.0001900196075439453 s
```

4-Génération d'un problème n'ayant pas de solutions de bases admissibles à la fois pour le primal le dual mais une solution faisable

[29]:
```
df_pa_pa=df[ (df["basic_feasible"]==False) &␣
 ↪(df["feasible"]==True)&(df["dual_basic_feasible"]==False)  ]
df_pa_pa=df_pa_pa.reset_index()
df_pa_pa.head()
```

[29]:
```
   index name                                                LP heuristique  \
0      4    4  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
1      6    6  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
2      7    7  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
3      8    8  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1
4      9    9  b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...            1

    shape basic_feasible feasible dual_basic_feasible bounded
0  (3, 4)          False     True               False    True
1  (3, 4)          False     True               False     NaN
2  (3, 4)          False     True               False     NaN
3  (3, 4)          False     True               False    True
4  (3, 4)          False     True               False     NaN
```

[30]:
```
buffer = BytesIO(df_pa_pa.loc[0]["LP"])
Lp_pa_pa = load(buffer)
Lp_pa_pa.table
```

[30]:
```
array([[10.,  8., -1.,  0.,  0.,  0.,  0.,  0.],
       [ 2., -1.,  3., -3.,  1.,  0.,  0., -1.],
       [-9., -8., -2.,  3.,  0.,  1.,  0., -1.],
       [ 7.,  6.,  9., -3.,  0.,  0.,  1., -3.]])
```

[31]:
```
spx=Simplex()
M=spx(Lp_pa_pa)
print("La solution du problème est :", M[1])
print("La valeur objective est",M[2])
print("Le temps d'exécution est de",M[4],"s")
```

```
 ### le problème linéaire est faisable

Ni le problème ni son dual n'admettent de solutions admissibles mais le problème
équivalent en admet
La solution du problème est : [ 0.  2.  0.  5. 16.  0.  0.]
La valeur objective est 24.0
Le temps d'exécution est de 0.00017189979553222656 s
```

5-Génération d'un problème n'ayant pas de solutions

[32]:
```
df_ps=df[ (df["basic_feasible"]==False) &␣
 ↪(df["feasible"]==False)&(df["dual_basic_feasible"]==False)  ]
df_ps=df_ps.reset_index()
df_ps.head()
```

[32]:

	index	name	LP	heuristique	\
0	2	2	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	
1	17	17	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	
2	29	29	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	
3	31	31	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	
4	32	32	b'\x80\x03c__main__\nLP\nq\x00)\x81q\x01}q\x02...	1	

	shape	basic_feasible	feasible	dual_basic_feasible	bounded
0	(3, 4)	False	False	False	False
1	(3, 4)	False	False	False	False
2	(3, 4)	False	False	False	False
3	(3, 4)	False	False	False	False
4	(3, 4)	False	False	False	False

[33]:
```
buffer = BytesIO(df_ps.loc[0]["LP"])
L_ps = load(buffer)
L_ps.table
```

[33]:
```
array([[ -1.,   2.,  -3.,  10.,   0.,   0.,   0.,   0.],
       [ -4.,   8.,  -2.,  -6.,   1.,   0.,   0.,  -9.],
       [-10.,  -9.,   4.,   0.,   0.,   1.,   0.,  -7.],
       [  4.,   5.,   5.,   3.,   0.,   0.,   1.,  -8.]])
```

Analyse des implémentations de Simplex

Dans ce paragraphe nous allons découvrir l'évolution du nombre d'appels de la fonction pivot en fonction de la taille du problème. Pour faire simple, nous allons choisir des problème carrés et incrémenter leurs tailles à chaque itération.

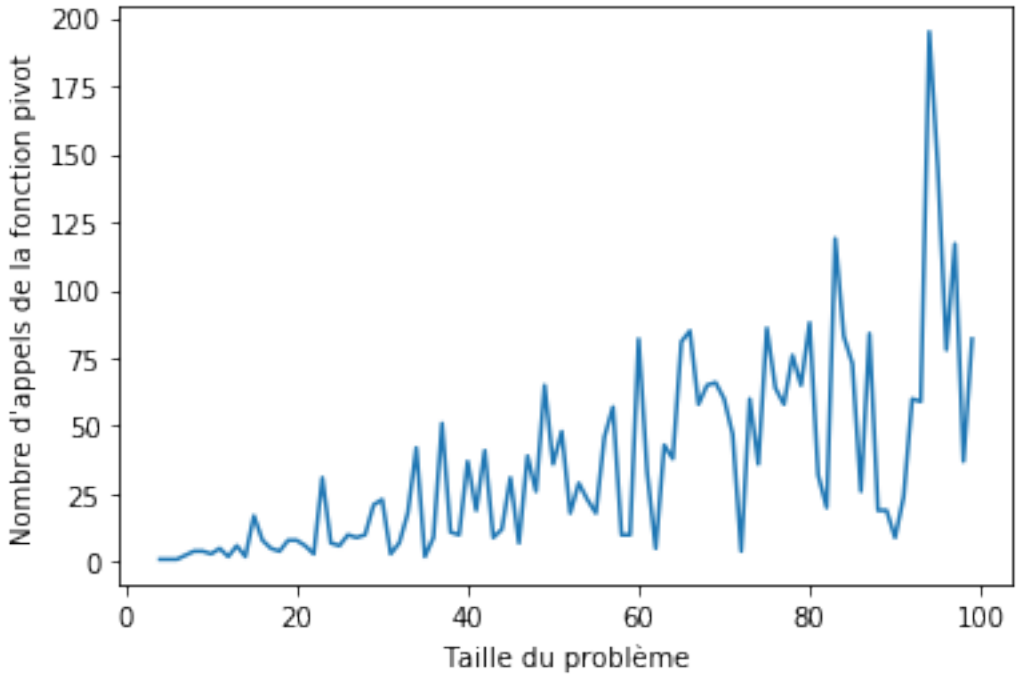
[40]:
```
t_max=100
borne_sup=1000
Dic={}
var=False
for i in range(4,t_max):
    while(var==False) :
        a=np.random.randint(-borne_sup,borne_sup, size=(i, i))
        b=np.random.randint(-borne_sup,borne_sup, size=(i, 1))
        c=np.random.randint(-borne_sup, borne_sup, i)
        lp=LP(a,b,c)
        spx =Simplex()
        try:
            Dic[i]=spx(lp)[3]
            var=spx._phase_one(lp)
```

```
        except:
            var=False
        del spx,lp
    var=False
```

[44]:
```
X=[c for c,v in Dic.items() if v!=0]
```

[45]:
```
Y=[v for c,v in Dic.items() if v!=0]
```

[53]:
```
import matplotlib.pyplot as plt
import numpy as np
plt.plot(X,Y)
plt.xlabel('Taille du problème')
plt.ylabel("Nombre d'appels de la fonction pivot")
plt.show()
```



Dans la théorie, le nombre d'appel de la fonction pivot pourrait évoluer d'une manière exponentielle en fonction de la taille du problème.Alors que dans la pratique, on remarque que cela reste polynomial.