Discrétisation de l'équation de Poisson $-\Delta u = f$

On s'intéresse à la discrétisation du problème suivant :

$$\begin{cases} -\Delta u = f & \text{sur } \Omega = [0,1] \times [0,1] \\ u = 0 & \text{sur } \partial\Omega \end{cases}$$

On considère comme fonction f, la fonction suivante :

$$f(x_1, x_2) = 6(1 - 3x_1 + 2x_1^2)(x_2 - 1)^3 x_2 + 6(1 - 3x_2 + 2x_2^2)(x_1 - 1)^3 x_1$$

.

Dans le cadre de notre exercice, nous connaissons l'expression de u, solution exacte, qui est la suivante :

$$u(x_1, x_2) = x_1 x_2 (x_1 - 1)^3 (x_2 - 1)^3$$

Le but de cette première partie est de retrouver u par différences finies. Pour cela nous écrirons une fonction qui retournera la matrice de discrétisation du Laplacien, nous résolverons ensuite le système et comparerons la solution approchée à celle connue. Nous utiliserons comme outil de programmation le langage python.

On commence par importer les packages dont nous avons besoin :

In [1]:
```
import numpy as np                          #Package pour calculs scientifique
s
import scipy.sparse as sparse               #Algèbre linéaire creuse
import matplotlib.pyplot as plt             #Permet la création de graphique
import scipy.sparse.linalg as sci           #Contient plusieurs packages pour
le calcul scientifique
from mpl_toolkits.mplot3d import Axes3D     #Utile pou les graphiques 3D
import time                                 #Affichage du temps de calcul
```

La fonction suivante permet d'écrire la matrice de la discrétisation du Laplacien. On découpe l'intervalle $[0,1] \times [0,1]$ en N intervalles en x et en y. Nous aurons donc $(N+1) \times (N+1)$ points. Cela implique que la matrice retournée est de taille $(N+1) \times (N+1)$.

In [3]:
```
def matrix_lap(N):
    """Retourne une matrice qui discrétise le laplacien de u dans le domaine
Omega = [0,1]x[0,1],
    découpé en N intervalles en x et y. La matrice finale est une matrice sc
ipy.sparse CSR matrix.
    Cette matrice est de taille (N+1)*(N+1)"""

    h = 1./N      #Longueur du pas d'espace
    h2 = h*h

    #On note les inconnues de 0 à Nx suivant x (axe des abscisses)
    #et 0 à Ny suivant y (axe des ordonnées).
    #La taille du problème est donc (Nx+1)*(Ny+1).
    #Ici on prend le même pas d'espace en x et y
    #Cela correspond à x_i = i*h et y_j = j*h
    #et la numérotation (i,j) --> k := (N+1)*j+i.

    taille = (1+N)*(1+N)

    #Rédaction de la matrice, on commence par effectuer
    #le tableau des diagonales

    diags = np.zeros((5,taille))

    #Diagonale principale
    diags[2,:] = 1.
    diags[2, N+2:taille - (N+2)] = -4./h2
    diags[2, np.arange(2*N+1, taille, N+1)] = 1.
    diags[2, np.arange(2*N+2, taille, N+1)] = 1.

    #Diagonale "-1"
    diags[1,N+1:taille-(N+1)] = 1./h2
    diags[1, np.arange(2*N, taille, N+1)] = 0.
    diags[1, np.arange(2*N+1, taille, N+1)] = 0.

    #Diagonale "+1"
    diags[3, N+3:taille-(N+1)] = 1./h2
    diags[3, np.arange(2*N+2, taille, N+1)] = 0.
    diags[3, np.arange(2*N+3, taille, N+1)] = 0.

    #Diagonale "-(N+1)"
    diags[0, 1 : taille - (2*N+3)] = 1./h2
    diags[0, np.arange(N,taille,N+1)] = 0.
    diags[0, np.arange(N+1,taille,N+1)] = 0.

    #Diagonale "+(N+1)"
    diags[4, taille - N*N + 2 : taille - 1] = 1./h2
    diags[4, np.arange(taille - N*N + 1 + N ,taille,N+1)] = 0.
    diags[4, np.arange(taille - N*N + 2 + N ,taille,N+1)] = 0.

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-(N+1),-1,0,1,(N+1)],taille,taille, format = "
csr")

    return A
```

Munis de cette fonction, il ne nous reste plus qu'à résoudre le problème en suivant l'ordre de numérotation des sommets qui est le suivant :

$$k = i + j \times (N+1), \quad i = 0, \dots, N, \quad j = 0, \dots, N$$

Voyons ce que nous retourne la fonction matrix_lap pour N = 2 et N = 3 :

In [4]:
```
print(matrix_lap(2).todense())
```

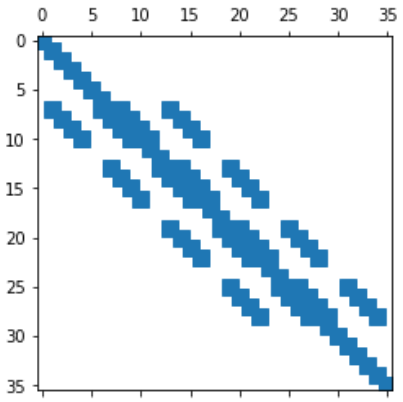
```
[[  1.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   1.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   1.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   1.   0.   0.   0.   0.   0.]
 [  0.   4.   0.   4. -16.   4.   0.   4.   0.]
 [  0.   0.   0.   0.   0.   1.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   1.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   1.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   1.]]
```

In [5]:
```
print(matrix_lap(3).todense())
```

```
[[  1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.   0.]
 [  0.   1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.   0.]
 [  0.   0.   1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.   0.]
 [  0.   0.   0.   1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.   0.]
 [  0.   0.   0.   0.   1.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.   0.]
 [  0.   9.   0.   0.   9. -36.   9.   0.   0.   9.   0.   0.   0.   0.
    0.   0.]
 [  0.   0.   9.   0.   0.   9. -36.   9.   0.   0.   9.   0.   0.   0.
    0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0.   0.   0.
    0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0.   0.
    0.   0.]
 [  0.   0.   0.   0.   0.   9.   0.   0.   9. -36.   9.   0.   0.   9.
    0.   0.]
 [  0.   0.   0.   0.   0.   0.   9.   0.   0.   9. -36.   9.   0.   0.
    9.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.   0.
    0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.
    0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.
    0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    1.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.   1.]]
```

Nous pouvons également avoir un aperçu graphique de la matrice en utilisant la fonction "spy". Regardons pour $N = 5$:

In [6]:
```
plt.spy(matrix_lap(5))
plt.show()
```



Par suite, nous écrivons le code pour définir la fonction f et notre solution exacte u.

In [7]:
```
def f(x1,x2):
    return 6.*(1.-3.*x1+2.*x1**2)*((x2-1.)**3)*x2 + 6.*(1.-3.*x2+2.*x2**2)*(
(x1-1.)**3)*x1

def u(x1,x2):
    return x1*x2*((x1-1.)**3)*((x2-1))**3
```

On va résoudre le problème pour N = 100. Nous commençons par écrire notre maillage à l'aide de la fonction linspace qui permet un découpage homogène de l'intervalle $[0, 1]$ en N intervalles.

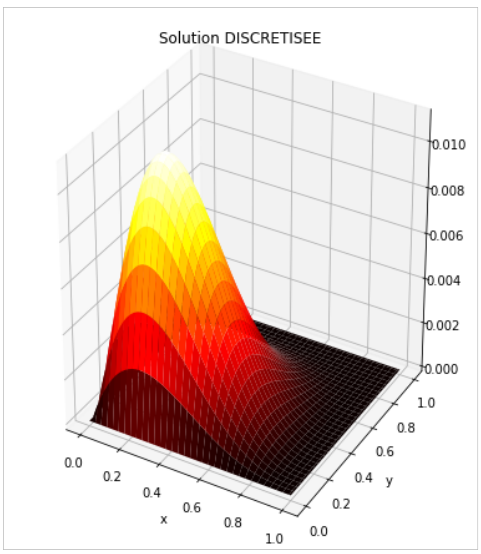
In [8]:
```
N = 100

x = np.linspace(0,1,N+1)
y = np.linspace(0,1,N+1)
```

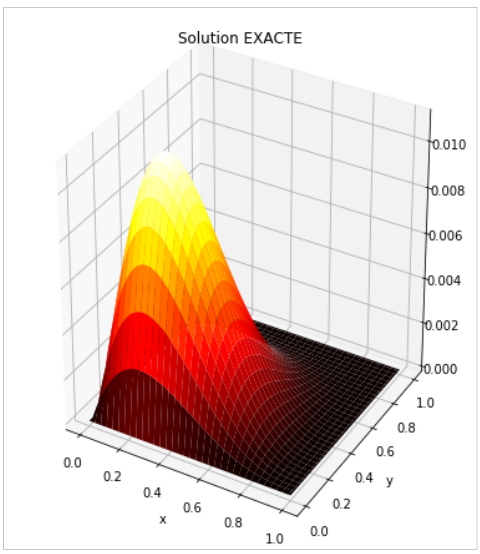
On écrit la fonction qui retourne la solution approchée du problème. Pour résoudre le système linéaire, nous utiliserons dans un premier temps un solveur pour matrice sparse avec méthode directe. Nous verrons par la suite que pour un grand nombre d'intervalles il est préférable d'utiliser un solveur avec une méthode itérative.

In [9]:
```
def sol_disc(N):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    F = np.zeros((N+1)*(N+1))   #Allocation mémoire de f

    for i in np.arange(1,N):    #(1,N) car comme k = i + (N+1)*j
                                #on s'assure que sur tous les bords f = 0.
        for j in np.arange(1,N):
            k = i + j*(N+1)
            F[k] = f(x[i],y[j])

    A = matrix_lap(N)           #Matrice de discrétisation du Laplacien

    U = sci.spsolve(A,F)        #Résolution du système linéaire AU = F
                                #par solveur utilisant une méthode directe

    return U
```

Et nous affichons finalement la solution que nous allons directement comparer à la solution exacte.

In [10]:
```
def sol_exacte(N):                #Fonction pour tracer la solution exacte

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    V = np.zeros((N+1)*(N+1))   #Allocation mémoire sol exacte

    for i in np.arange(N+1):
        for j in np.arange(N+1):
            k = i + j*(N+1)
            V[k] = u(x[i],y[j])

    return V


V = sol_exacte(N)          #On alloue à V la solution exacte
U = sol_disc(N)            #On alloue à U la solution approchée

fig = plt.figure(figsize = [15,8])                  #Création de l'environne
ment pour la figure

ax = fig.add_subplot(1,2,1, projection = '3d')      #Premier subplot pour la
solution approchée
X,Y = np.meshgrid(x,y)
ax.plot_surface(X,Y,U.reshape((N+1,N+1)),cmap = 'hot')
plt.xlabel("x")
plt.ylabel("y")
plt.title("Solution DISCRETISEE")

ax = fig.add_subplot(1,2,2, projection = '3d')       #Second subplot pour la
solution exacte
ax.plot_surface(X,Y, V.reshape((N+1,N+1)), cmap='hot')
plt.xlabel("x")
plt.ylabel("y")
plt.title("Solution EXACTE")

plt.show()
```





On peut remarquer que les graphes sont très ressemblant, ce qui est une bonne nouvelle. Cependant, pour bien se rendre compte de la précision de notre schéma d'approximation, nous allons calculer l'erreur entre la solution approchée et celle exacte. Pour cela on définit deux fonctions d'erreur. La première est l'erreur dite euclidienne, définie par la formule :

$$err_{eucl} = \sqrt{\frac{\sum_{i,j}(ex_{i,j} - app_{i,j})^2}{(N+1)^2}}$$

Puis nous calculerons l'erreur absolue définie par :

$$err_{abs} = \max_{i,j} |ex_{i,j} - app_{i,j}|$$

Nous calculons les erreurs précédentes toujours pour une valeur de $N = 100$.

In [11]:
```
def erreur_eucl(A,E,N):
    return np.sqrt(np.sum((E-A)**2)/((N+1)**2))

err1 = erreur_eucl(U,V,N)
print("{:8s} {:12s}".
    format("Taille", "Erreur demandée"))
print("{:8d} {:12.5e}".
    format((N+1)*(N+1), err1))
```

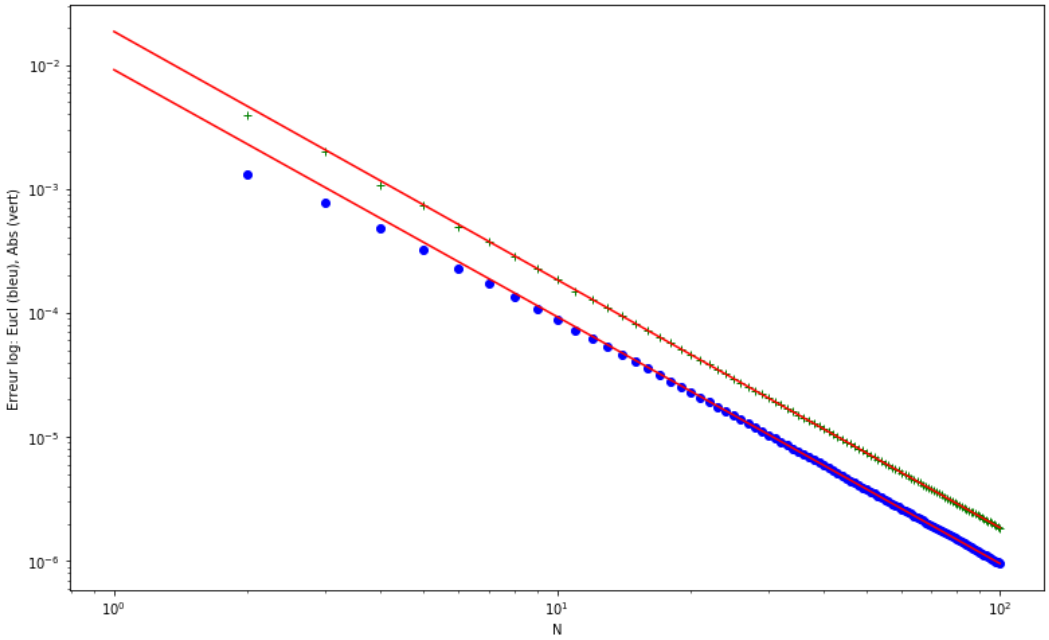
```
Taille   Erreur demandée
   10201  9.55422e-07
```

In [12]:
```
def erreur_abs(A,E,N):
    return np.max(np.abs(E - A))

err2 = erreur_abs(U,V,N)
print("{:8s} {:12s}".
    format("Taille", "Erreur absolue"))
print("{:8d} {:12.5e}".
    format((N+1)*(N+1), err2))
```

```
Taille   Erreur absolue
   10201  1.85769e-06
```

Finalement nous allons écrire le code pour afficher le graphique des 2 erreurs ci-dessus. Nous tracerons également la droite de régression linéaire afin de vérifier la justesse de nos résultats. Utilisant une méthode du second ordre, nous nous attendons à un coefficient directeur de -2. La fonction suivante affichera directement le graphique d'erreur, les 2 droites de régression et retournera les coefficients directeurs.

In [13]:
```
def aff(x,b,a):                         #Création de la fonction pour régressio
n linéaire
    return np.exp(b)*x**(a)

def graphe_erreur(N):                   #Création de la fonction pour représent
er les graphiques d'erreurs
    tab_err1 = np.zeros(N)
    tab_err2 = np.zeros(N)

    ERR1 = np.zeros(N)
    ERR2 = np.zeros(N)

    x = np.linspace(1,N,N)

    for i in range(1,N+1):
        U = sol_disc(i)
        V = sol_exacte(i)

        tab_err1[i-1] = erreur_eucl(U,V,i)
        tab_err2[i-1] = erreur_abs(U,V,i)


    x1 = x[N-10:N]
    Err1 = tab_err1[N-10:N]
    Err2 = tab_err2[N-10:N]
    z1 = np.polyfit(np.log(x1),np.log(Err1),1)
    z2 = np.polyfit(np.log(x1),np.log(Err2),1)

    y1 = aff(x,z1[1],z1[0])
    y2 = aff(x,z2[1],z2[0])

    plt.figure(figsize=[13,8])

    plt.plot(x,tab_err1,color='blue',marker='o', linestyle='none')
    plt.plot(x,tab_err2,color='green',marker='+', linestyle='none')
    plt.plot(x,y1,color='r', linestyle='-')
    plt.plot(x,y2,color='r', linestyle='-')
    plt.xscale("log")
    plt.yscale("log")
    plt.xlabel('N')
    plt.ylabel('Erreur log: Eucl (bleu), Abs (vert)')

    plt.show()

    return z1[0],z2[0]

graphe_erreur(100)
```



Out[13]: (-1.989640803920319, -1.9994143751986317)

Les coefficients directeurs obtenus justifient donc parfaitement la justesse de notre schéma.

Pour conclure, nous allons observer l'intérêt d'utiliser des solveurs avec méthodes itératives pour résoudre nos systèmes linéaires. Pour cela, nous allons utiliser le package time qui renvoie le temps de calcul d'un programme. Nous allons écrire la même approximation mais nous utiliserons la méthode du gradient conjugué (utilisé ici car notre matrice sparse est bien définie positive). Nous allons alors procéder à l'affichage du temps pour $N = 100$ et $N = 1000$.

In [14]:
```
def sol_disc_conjgrad(N):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    F = np.zeros((N+1)*(N+1))   #Allocation mémoire de f

    for i in np.arange(1,N):    #(1,N) car comme k = i + (N+1)*j on s'assure
que
                                #sur tous les bords f = 0.
        for j in np.arange(1,N):
            k = i + j*(N+1)
            F[k] = f(x[i],y[j])

    A = matrix_lap(N)           #Matrice de discrétisation du Laplacien

    U = sci.cg(A,F)             #Résolution du système linéaire AU = F par s
olveur
                                #utilisant une méthode de gradient conjugué


    return U[0]
```

In [15]:
```
def temps_direct(N):
    start_time = time.time()
    U = sol_disc(N)
    print("Temps d execution directe: %s secondes " %(time.time() - start_ti
me))

def temps_iterativ(N):
    start_time = time.time()
    U = sol_disc_conjgrad(N)
    print("Temps d execution itérative : %s secondes " %(time.time() - start
_time))
```

In [20]:
```
temps_direct(100)
temps_iterativ(100)
```

```
Temps d execution directe: 0.1265411376953125 secondes
Temps d execution itérative : 0.11231422424316406 secondes
```

In [17]:
```
temps_iterativ(1000)
```

```
Temps d execution itérative : 83.88120746612549 secondes
```

In [18]:
```
temps_direct(1000)
```

```
Temps d execution directe: 85.36850833892822 secondes
```

On se rend compte que la méthode de résolution itérative, ici par méthode du gradient conjugué est plus rapide que son homologue directe. Comme nous n'utiliserons jamais un grand nombre d'intervalle dans la suite du rapport nous privilégierons l'utilisation de la méthode directe.

Interpolation du problème par des fonctions polynômiales

Dans cette seconde partie, nous cherchons à changer la valeur de la solution sur les bords. On s'intéresse à la discrétisation du problème suivante sur un domaine $\Omega = [0,1] \times [0,1]$:

$$-\Delta u = 0$$

Nous allons rajouter les conditions suivantes sur les bords :

$$u(x,\alpha,\beta) = \begin{cases} u(x,\alpha) = x(x-\frac{1}{2})\alpha & \text{si } x \leq 0 \quad \text{sur le bord bas} \\ 0 & \text{sinon} \quad \text{sur le bord bas} \\ u(x,\beta) = (x-1)(x-\frac{1}{2})\beta & \text{si } x \geq 0 \quad \text{sur le bord haut} \\ 0 & \text{sinon} \quad \text{sur le bord haut} \end{cases}$$

L'interpolation dépend donc de deux paramètres réels α et β. On commence par définir les fonctions nécessaires.

In [16]:
```
def fb(x,y,alpha):                      #Fonction pour le bord haut dépendant
de alpha
    return x*(x-(1./2.))*alpha

def fh(x,y,beta):                       #Fonction pour le bord bas dépendant
de beta
    return (x-1.)*(x-(1./2.))*beta
```

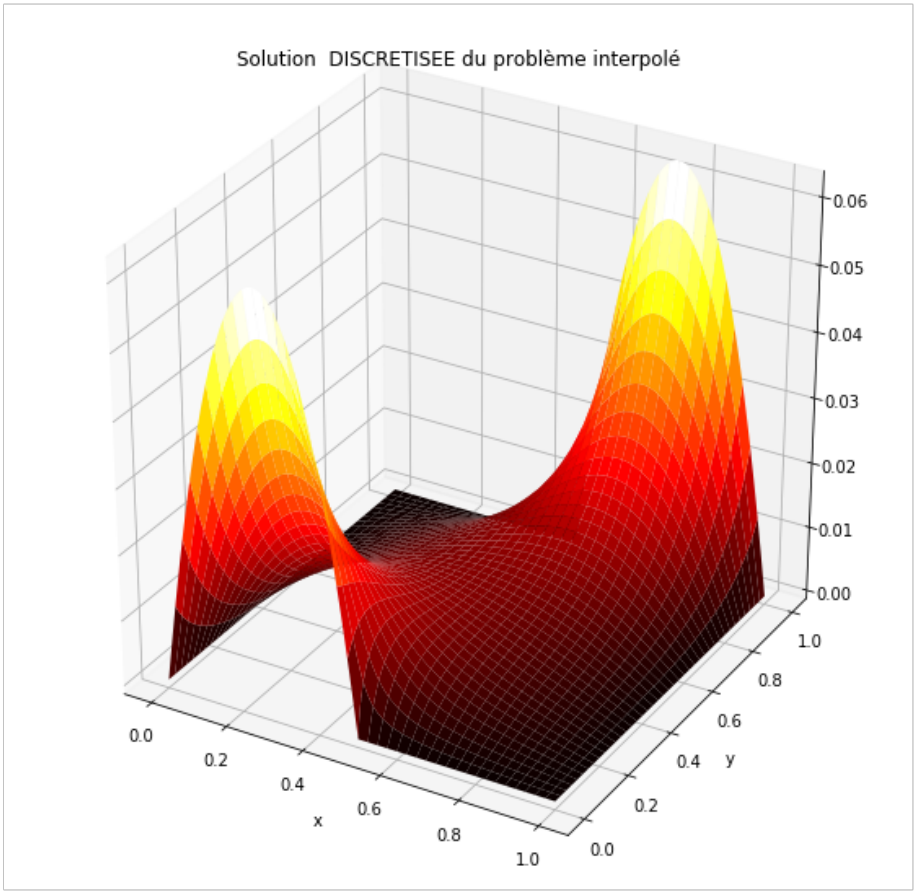
Et on discrétise le problème de la façon suivante :

In [17]:
```
def sol_disc_inter(N, alpha, beta):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    F = np.zeros((N+1)*(N+1))      #Matrice pour stocker les valeurs de f
                                   #avec les fonctions interpolées

    for i in np.arange(0,N+1):
        for j in np.arange(0,N+1):
            k = i + j*(N+1)
            if (i <= (N/2.) and j == 0):
                F[k] = fb(x[i],y[j],alpha)
            elif (i >= (N/2.) and j == N):
                F[k] = fh(x[i],y[j],beta)

    A = matrix_lap(N)              #Matrice de discrétisation du Laplacien

    U = sci.spsolve(A,F)            #On résoud le système linéaire AU = F
                                    #par le solveur avec méthode directe

    return U
```

Finalement, on procède à l'affichage pour une valeur de $N = 200, \alpha = -1, \beta = -1$

In [18]:
```
N = 200

x = np.linspace(0,1,N+1)
y = np.linspace(0,1,N+1)

U = sol_disc_inter(N,-1,-1)

fig = plt.figure(figsize = [10,10])

ax = fig.add_subplot(111 ,projection='3d')
X,Y = np.meshgrid(x,y)
ax.plot_surface(X,Y, U.reshape((N+1,N+1)), cmap='hot')

plt.title("Solution  DISCRETISEE du problème interpolé")
plt.xlabel("x")
plt.ylabel("y")
```

Out[18]:
```
Text(0.5, 0, 'y')
```



Résolution de l'équation de la chaleur par différences finies

Résolution pour une solution u triviale

Dans cette partie, on s'intéresse à la résolution de l'équation de la chaleur, c'est à dire à la discrétisation du problème suivant :

$$\begin{cases} \frac{\partial u}{\partial t} = \Delta u \quad sur \quad \Omega = [0,1] \times [0,1] \\ u = 0 \qquad\quad \text{sur} \quad \partial\Omega \end{cases}$$

Nous remarquons que nous utilisons des conditions de bord de type Dirichlet. Pour approximer la solution, nous allons utiliser une méthode de discrétisation par différences finies de type implicite. Nous obtenons le calcul suivant :

$$\frac{U^{n+1} - U^n}{\delta t} = AU^{n+1}$$

où U représente le vecteur solution et A la matrice de discrétisation du Laplacien. Le réel δt représente le pas de temps. Le vecteur u est dit trivial dans le titre de cette partie car nous l'initialiserons à un vecteur de 1 tout en prenant en compte les conditions de bord. Dans la partie suivante, nous appliquerons cette équation de diffusion à une fonction u donnée.

Nous obtenons l'expression finale :

$$U^{n+1} = (I - \delta t A)^{-1} U^n$$

La stratégie adoptée pour coder ce problème sera la suivante. Dans un premier temps, nous allons écrire la matrice de $(I - \delta t A)$ en s'inspirant de notre travail sur l'équation de Poisson. Dans un second temps, nous ajouterons la boucle temporelle et nous finirons par afficher le résultat final.

Pour commencer, nous allons importer les packages nécessaires.

In [1]:
```
import numpy as np                          #Package pour calculs scientifique
s
import scipy.sparse as sparse               #Algèbre linéaire creuse
import matplotlib.pyplot as plt             #Permet la création de graphique
import scipy.sparse.linalg as sci           #Contient plusieurs packages pour
le calcul scientifique
from mpl_toolkits.mplot3d import Axes3D     #Utile pou le graphiques 3D
import time                                 #Affichage du temps de calcul
from IPython.display import Image           #Affichage d'image dans le Jupyter
```

La fonction suivante retourne la matrice de $(I - \delta t A)$. Ce sera une matrice sparse de taille $(N+1) \times (N+1)$. Elle regroupera donc l'intégralité des points du maillage et nous permettra de résoudre le système linéaire en insérant les conditions de bords ($u = 0$) directement dans le second membre. La fonction suivante prendra en paramètre le nombre N d'intervalles et la pas de temps δt.

In [2]:
```
def matrix_lap2(N,dt):
    """Retourne la matrice de (I -dtA) dans le domaine Omega = [0,1]x[0,1]
    découpé en N intervalles en x et y. La matrice finale est une matrice sc
ipy.sparse CSR matrix.
    Cette matrice est de taille (N+1)*(N+1)"""

    h = 1./N
    h2 = h*h

    #On note les inconnues de 0 à Nx suivant x et 0 à Ny suivant y.
    #La taille du problème est donc (Nx+1)*(Ny+1).
    #Ici le pas d'espace est le même entre x et y.
    #Cela correspond à x_i = i*h et y_j = j*h
    #et la numérotation (i,j) --> k := (N+1)*j+i.

    taille = (1+N)*(1+N)

    diags = np.zeros((5,taille))                          #Création du table
au des diagonales

    #Diagonale principale
    diags[2,:] = 1.
    diags[2, N+2:taille - (N+2)] = 1 + ((4*dt)/h2)
    diags[2, np.arange(2*N+1, taille, N+1)] = 1.
    diags[2, np.arange(2*N+2, taille, N+1)] = 1.

    #Diagonale "-1"
    diags[1,N+1:taille-(N+1)] = -dt/h2
    diags[1, np.arange(2*N, taille, N+1)] = 0.
    diags[1, np.arange(2*N+1, taille, N+1)] = 0.

    #Diagonale "+1"
    diags[3, N+3:taille-(N+1)] = -dt/h2
    diags[3, np.arange(2*N+2, taille, N+1)] = 0.
    diags[3, np.arange(2*N+3, taille, N+1)] = 0.

    #Diagonale "-(N+1)"
    diags[0, 1 : taille - (2*N+3)] = -dt/h2
    diags[0, np.arange(N,taille,N+1)] = 0.
    diags[0, np.arange(N+1,taille,N+1)] = 0.

    #Diagonale "+(N+1)"
    diags[4, taille - N*N + 2 : taille - 1] = -dt/h2
    diags[4, np.arange(taille - N*N + 1 + N ,taille,N+1)] = 0.
    diags[4, np.arange(taille - N*N + 2 + N ,taille,N+1)] = 0.

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-(N+1),-1,0,1,(N+1)],taille,taille, format = "
csc")

    return A
```

Regardons ce qu'affiche la matrice A pour $N = 2$ et un pas de temps $\delta t = 0.1$

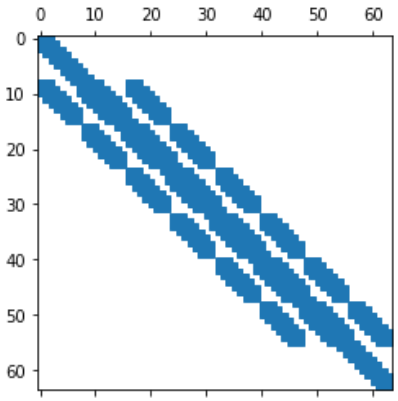
In [3]:
```
A = matrix_lap2(2,0.1)
print(A.todense())
```

```
[[ 1.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   1.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   1.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   1.   0.   0.   0.   0.   0. ]
 [ 0.  -0.4  0.  -0.4  2.6 -0.4  0.  -0.4  0. ]
 [ 0.   0.   0.   0.   0.   1.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   1.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   1.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   1. ]]
```

Nous pouvons aussi, comme en début de rapport de voir un aperçu graphique de la matrice pour une valeur de $N = 7$

In [4]:
```
plt.spy(matrix_lap2(7,0.1))
```

Out[4]: <matplotlib.lines.Line2D at 0x7f363d900198>

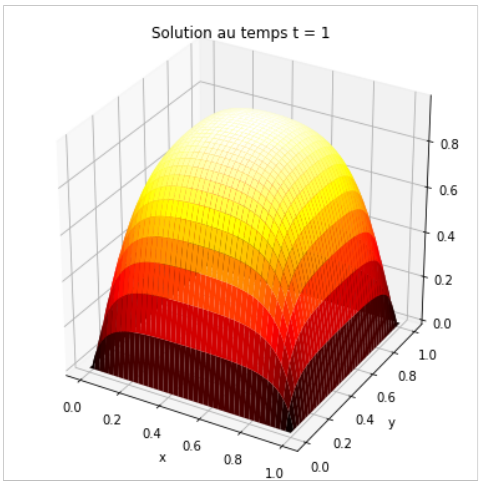


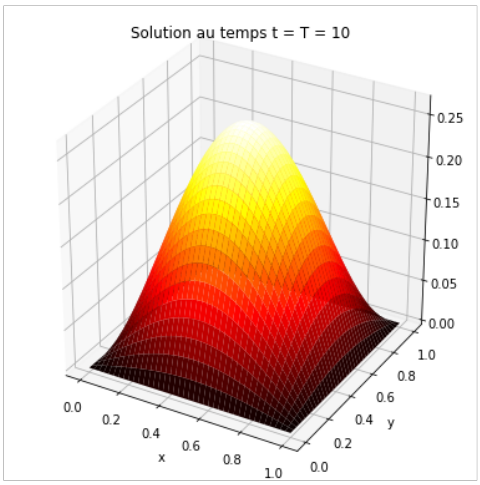
On remarque que la matrice obtenue correspond, visuellement, à celle que l'on doit obtenir. Finalement, on rédige l'algorithme pour résoudre le système linéaire. Nous avons décidé de créer un tableau des temps en 2 dimensions, dans lequel on stockera la solution à chaque temps t. Cela permettra à l'utilisateur de choisir quel temps il voudra afficher dans un intervalle $[0, T]$. Dans le cadre de ce rapport, nous afficherons seulement pour $t = 0$ et $t = T$. Le fait de stocker la solution, à chaque temps, dans un tableau peut-être très coûteuse. Nous verrons par la suite qu'il est également possible de stocker chaque solution dans un fichier, ce qui rend l'opération moins lourde pour l'ordinateur. On obtient la fonction suivante.

In [5]:
```
def chaleur_triv(N,dt,t):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille1 = (N+1)*(N+1)

    #Initialisation du tableau en 2 dimensions, (t+1)
    #en première composante pour le nombre de d'intervalle de temps
    #et (taille1) en seconde composante pour stocker le tableau de solution

    T = np.zeros((t+1,taille1))

    #Initialisation de la solution finale.
    #On met des 0 sur les bords pour valider les conditions initiales
    #et on initialise le reste à 1.

    T[0,N+2:taille1 - N-2] = 1.
    T[0,np.arange(2*N+1, taille1, N+1)] = 0
    T[0,np.arange(2*N+2, taille1, N+1)] = 0

    #On fait la boucle temporelle en stockant la solution à chaque temps.
    #On utilise un solveur avec une méthode directe pour matrice sparse.

    for i in range (t):
         T[i+1,:] = sci.spsolve(matrix_lap2(N,dt),T[i,:])

    return T
```

Finalement on va procéder à l'affichage de la solution discrétisée. On se propose ici d'afficher la solution au temps $t = 1$ et au temps $t = T$. Cependant l'utilisateur peut changer la valeur du paramètre pour l'afficher au temps $t \in [0, T]$. On va utiliser ici une valeur de $N = 200$, $\delta t = 0.01$ et $T = 10$.

In [6]:
```
N = 200
t = 10

#Création du Tableau de temps
T = chaleur_triv(N,0.01,t)

x = np.linspace(0,1,N+1)
y = np.linspace(0,1,N+1)

#Création de l'environnement de la figure
fig = plt.figure(figsize = [15,7])

#Premier subplot pour la solution au temps t = 1
ax = fig.add_subplot(1,2,1, projection = '3d')
X,Y = np.meshgrid(x,y)
ax.plot_surface(X,Y,T[1,:].reshape(N+1,N+1) ,cmap='hot')
plt.title("Solution au temps t = 1")
plt.xlabel("x")
plt.ylabel("y")

#Second subplot pour la solution au temps t = 10
ax = fig.add_subplot(1,2,2, projection = '3d')
ax.plot_surface(X,Y,T[t,:].reshape(N+1,N+1) ,cmap='hot')
plt.title("Solution au temps t = T = 10")
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```





Nous allons maintenant écrire un algorithme permettant de stocker la solution à chaque temps dans un fichier.

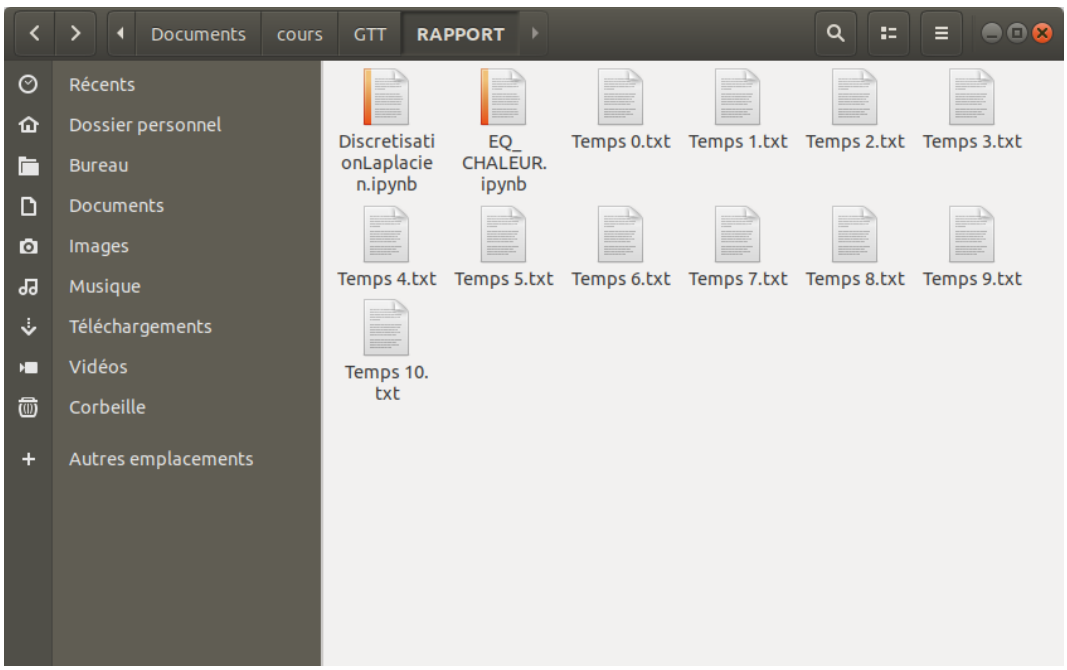
In [7]:
```
def chaleur_triv_files(N,dt,t):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille1 = (N+1)*(N+1)

    #Initialisation du tableau de la solution

    T = np.zeros(taille1)

    #Initialisation de la solution finale.
    #On met des 0 sur les bords pour valider les conditions initiales
    #et on initialise le reste à 1.

    T[N+2:taille1 - N-2] = 1.
    T[np.arange(2*N+1, taille1, N+1)] = 0
    T[np.arange(2*N+2, taille1, N+1)] = 0

    #On fait la boucle temporelle en stockant la solution
    #à chaque temps dans un fichier txt.
    #On utilise un solveur avec une méthode directe pour matrice sparse.

    for i in range (t+1):
        np.savetxt('Temps {}.txt'.format(i), T.reshape(N+1,N+1))
        T = sci.spsolve(matrix_lap2(N,dt),T)
```

In [11]:
```
chaleur_triv_files(200,0.01,10)
```

Dans notre répertoire, 11 fichiers textes sont apparus, chacun contenant la solution du temps t.

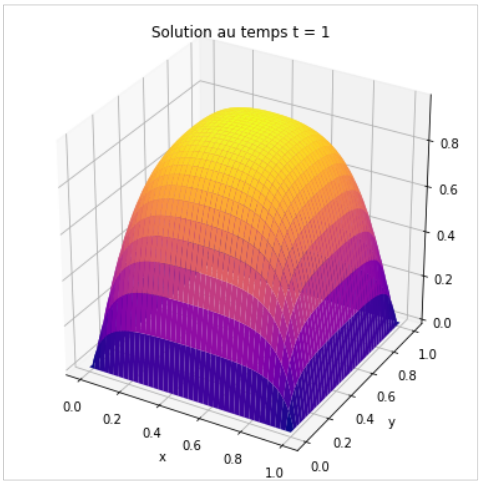
In [9]:
```
Image("/home/matthieu/Documents/cours/GTT/RAPPORT/Solutionstemps.png", width
= 700)
```

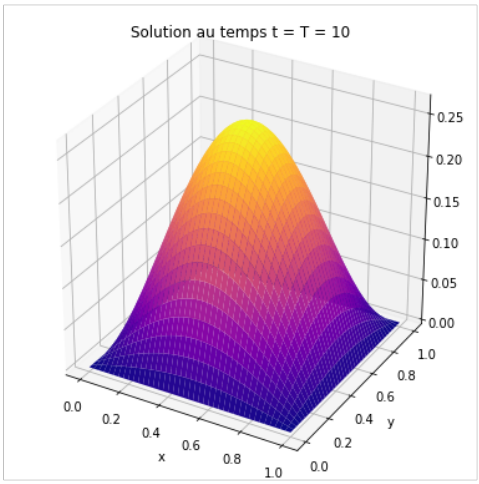
Out[9]:



Nous allons afficher la solution au temps $T = 1$ et $T = 10$ afin de voir si l'on retrouve les mêmes solutions que pour le tableau de temps précédent.

In [12]:
```
solt1 = np.loadtxt("Temps 1.txt")                #On charge le fichier pour
le temps 1
solt10 = np.loadtxt("Temps 10.txt")              #On charge le fichier pour
le temps 10

x = np.linspace(0,1,N+1)
y = np.linspace(0,1,N+1)

fig = plt.figure(figsize = [15,7])

#Premier subplot pour la solution au temps t = 1

ax = fig.add_subplot(1,2,1, projection = '3d')
X,Y = np.meshgrid(x,y)
ax.plot_surface(X,Y,solt1,cmap='plasma')         #On change la couleur de l'
affichage
                                                 #pour le plaisir des yeux
plt.title("Solution au temps t = 1")
plt.xlabel("x")
plt.ylabel("y")

#Second subplot pour la solution au temps t = 10

ax = fig.add_subplot(1,2,2, projection = '3d')
ax.plot_surface(X,Y,solt10 ,cmap='plasma')
plt.title("Solution au temps t = T = 10")
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```





On remarque finalement que les courbes obtenues sont identiques à celles obtenues précédemment. Nous avons donc vu deux méthodes pour stocker les données d'une solution. Il faudra, pour des raisons de coûts de calcul ainsi qu'une éventuelle volonté de les sauvegarder sur le disque dur, privilégier le stockage des tableaux dans des fichiers textes. Cependant, dans la suite du rapport, nous utiliserons la méthode de stockage directement dans un tableaux numpy.

Résolution de l'équation de la chaleur pour une solution u donnée

Dans cette partie, on s'intéresse à la résolution de l'équation de la chaleur avec une fonction u donnée, c'est à dire à la discrétisation du problème suivant:

$$\begin{cases} \frac{\partial u}{\partial t} = \Delta u & sur \quad \Omega = [0,1] \times [0,1] \\ u = 0 & \text{sur} \quad \partial\Omega \end{cases}$$

où

$$u(x_1, x_2) = x_1 x_2 (x_1 - 1)^3 (x_2 - 1)^3$$

On remarque que la fonction donnée répond bien au problème posé, particulièrement au niveau des conditions de bord. En effet, pour $x_1 = 0, x_2 = 0, x_1 = 1, x_2 = 1$ la fonction est nulle. Nous allons utiliser une méthode de discrétisation implicite comme précédemment. Nous allons donc réutiliser les mêmes fonctions. Le seul changement résidera dans le code de la fonction u. On commence par écrire la fonction dont nous avons besoin.

In [13]:
```
def u(x1,x2):
    return x1*x2*((x1-1.)**3)*((x2-1))**3
```

Et on écrit le code qui sera très inspiré de ce qui a été fait avant :

In [14]:
```
def chaleur_ex(N,dt,t):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille1 = (N+1)*(N+1)

    V = np.zeros((t+1,taille1))     #Allocation mémoire sol exacte et tablea
u de temps

    for i in np.arange(N+1):        #Calcul de la solution en tout point du
maillage
        for j in np.arange(N+1):
            k = i + j*(N+1)
            V[0,k] = u(x[i],y[j])

    for i in range (t):             #Boucle temporelle utilisant le même sol
veur qu'avant
        V[i+1,:] = sci.spsolve(matrix_lap2(N,dt),V[i,:])

    return V
```

On affiche finalement le résultat pour des valeurs $N = 200, \delta t = 0.01$.

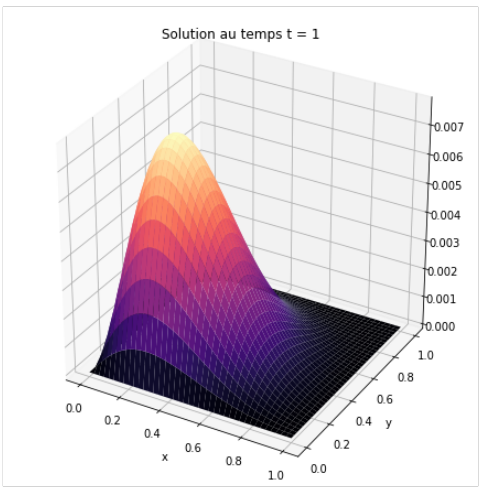
Selon les informations données en cours, nous devrions obtenir les résultats suivants :

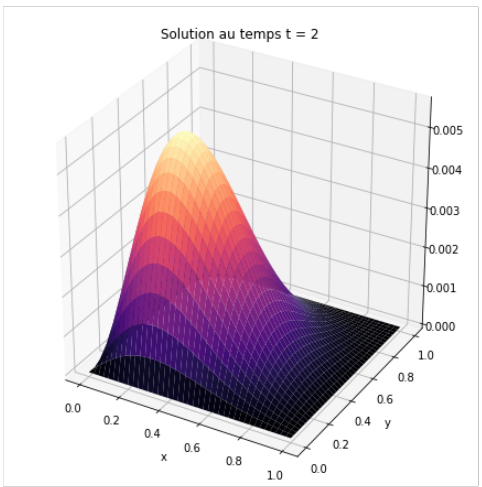
$$t = 1 \implies \max(u) = 0.007773$$
$$t = 2 \implies \max(u) = 0.005619$$

Regardons si nous retrouvons ces résultats.

In [15]:
```
N = 200

x = np.linspace(0,1,N+1)
y = np.linspace(0,1,N+1)

#Création de la solution avec un temps à 2

V = chaleur_ex(N,0.01,2)

#Affichage des maximums

print('Pour t = 1, le maximum atteint est '+str(np.max(V[1,:])))

print('Pour t = 2, le maximum atteint est '+str(np.max(V[2,:])))

#Création de l'environnement pour la figure

fig = plt.figure(figsize = [17,8])

#Premier subplot pour la solution au temps = 1
ax = fig.add_subplot(1,2,1, projection = '3d')
X,Y = np.meshgrid(x,y)
#Les maths, c'est amusant, donc on rechange les couleurs
ax.plot_surface(X,Y,V[1,:].reshape(N+1,N+1) ,cmap='magma')
plt.title("Solution au temps t = 1")
plt.xlabel("x")
plt.ylabel("y")

#Second subplot pour la solution au temps = 2
ax = fig.add_subplot(1,2,2, projection = '3d')
X,Y = np.meshgrid(x,y)
ax.plot_surface(X,Y,V[2,:].reshape(N+1,N+1) ,cmap='magma')
plt.title("Solution au temps t = 2")
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```

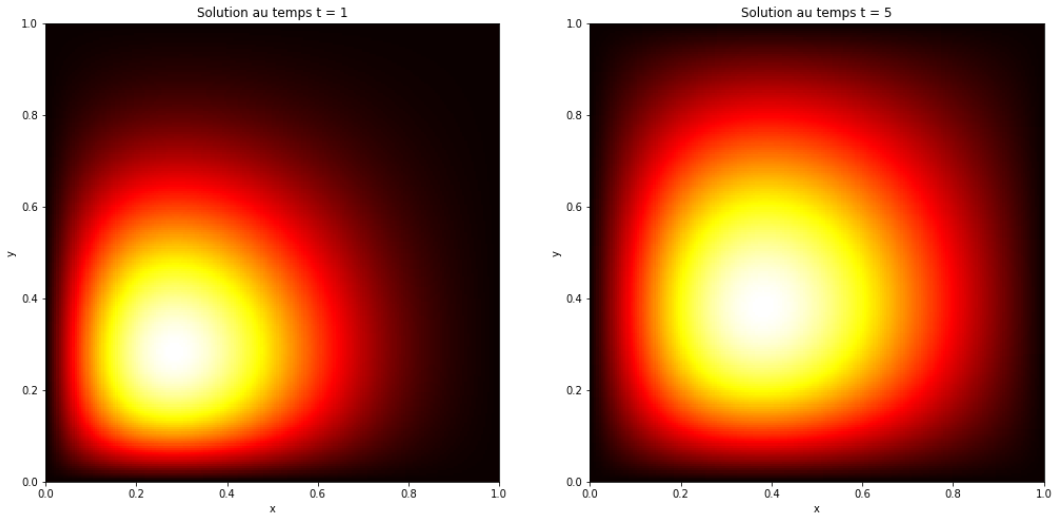
```
Pour t = 1, le maximum atteint est 0.007773079850730673
Pour t = 2, le maximum atteint est 0.0056197497221826655
```





Comme nous obtenons les mêmes résultats que demandés en cours, nous pouvons conclure la justesse de notre approximation. Pour finir cette partie, on se propose d'afficher la solution sur une "grille de chaleur" qui pourrait ressembler, par exemple, à l'évolution de la température sur une plaque de cuisson. Pour cela on réécrit le même code et on change seulement le plot. Nous présentons la solution en utilisant les mêmes paramètres que précédemment sauf pour le temps que nous afficherons pour $t = 1$ et $t = 5$ afin de bien visualiser l'évolution de la solution.

In [16]:
```
N = 200

x = np.linspace(0,1,N+1)
y = np.linspace(0,1,N+1)

#Création de la solution avec un temps à 2

V = chaleur_ex(N,0.01,5)

#Création de l'environnement pour la figure

fig = plt.figure(figsize = [17,8])

#Premier subplot pour la solution au temps = 1
ax = fig.add_subplot(1,2,1)
X,Y = np.meshgrid(x,y)
ax.pcolormesh(X,Y,V[1,:].reshape(N+1,N+1) ,cmap='hot')
plt.title("Solution au temps t = 1")
plt.xlabel("x")
plt.ylabel("y")

#Second subplot pour la solution au temps = 2
ax = fig.add_subplot(1,2,2)
X,Y = np.meshgrid(x,y)
ax.pcolormesh(X,Y,V[5,:].reshape(N+1,N+1) ,cmap='hot')
plt.title("Solution au temps t = 5")
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```



La fonction distance sur un domaine carré

Dans cette section, nous allons afficher la fonction distance sur un domaine carré. C'est à dire que nous allons déterminer une fonction qui va retourner la distance de chaque point du maillage par rapport à un bord. Dans notre cas, le bord sera celui du domaine $\Omega = [0,1] \times [0,1]$.

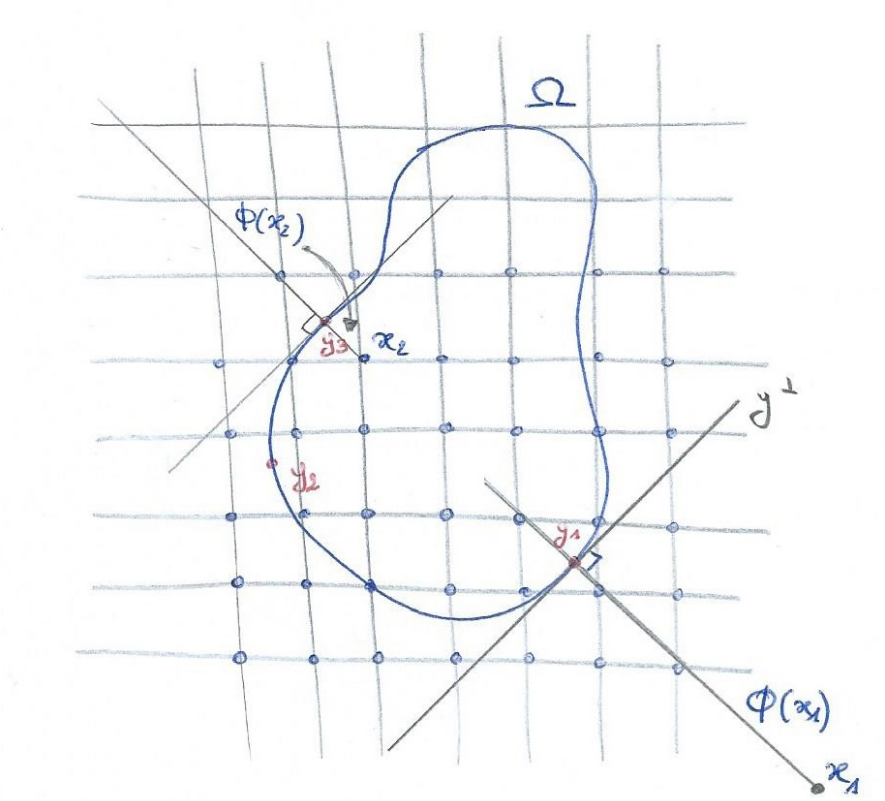
Elle est définie de la manière suivante :

$$\Phi(x) = \min_{y \in \partial\Omega} d(x,y)$$

Graphiquement, on obtient comme représentation, pour un domaine Ω quelconque régulier, le dessin ci-contre :

In [15]:
```
Image("/home/matthieu/Documents/cours/GTT/RAPPORT/Pic1.jpg", width = 600)
```

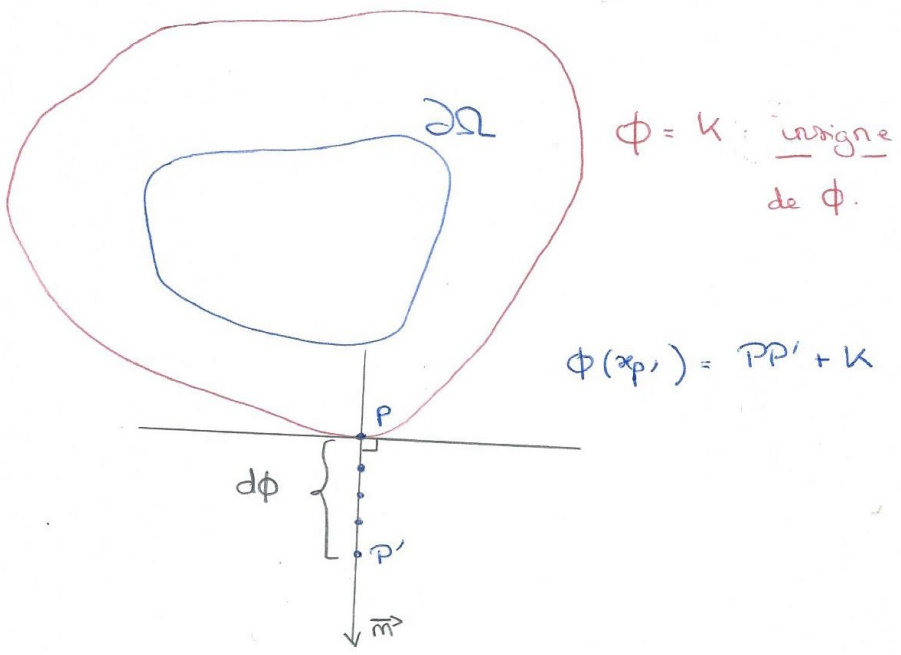
Out[15]:



Il est alors évident que si $x \in \partial\Omega$ alors $\Phi(x) = 0$. A l'inverse, pour retrouver la frontière d'un domaine, il suffira de retourner le zéro de la fonction distance. On propose dans la suite une méthode de calcul de la fonction distance. Pour cela on modélise le problème de la manière suivante :

In [16]:
```
Image("/home/matthieu/Documents/cours/GTT/RAPPORT/Pic2.jpg", width = 600)
```

Out[16]:



$d\Phi$ représente la variation de Φ lorsque l'on passe du point P au point P'. Ainsi, on peut réécrire :

$$d\Phi = \nabla\Phi . \overrightarrow{PP'}$$

Or comme $\overrightarrow{PP'} = n . d\Phi$, où n est le vecteur normal unitaire à Φ au point P on obtient :

$$d\Phi = \nabla\Phi . n . d\Phi$$

On en déduit alors que $\nabla\Phi . n = 1$. Or comme $\Phi = K$, le vecteur gradient lui même est un vecteur normal à Φ. On a donc une expression du vecteur normal unitaire comme suit :

$$n = \frac{\nabla\Phi}{||\Phi||}$$

De plus, on sait que $u . u = ||u||^2$ et donc :

$$\nabla\Phi . n = 1 \Longrightarrow \nabla\Phi . \frac{\nabla\Phi}{||\nabla\Phi||} = \frac{||\nabla\Phi||^2}{||\nabla\Phi||} = ||\nabla\Phi|| = 1$$

On obtient une équation aux dérivées partielles telles que :

$$\begin{cases} ||\nabla\Phi|| &= 1 \quad \text{sur } \Omega \\ \Phi &= 0 \quad \text{sur } \partial\Omega \end{cases}$$

Nous allons donc trouver un moyen de donner une solution approchée de ce problème en utilisant l'équation de la chaleur.

On considère tout d'abord la fonction suivante, pour tout $\epsilon \geq 0$:

$$T = \exp \frac{-\Phi}{\epsilon}$$

On calcule le Laplacien de T et on trouve :

$$\Delta T = \frac{-T}{\epsilon}\Delta\Phi + \frac{1}{\epsilon^2}||\nabla\Phi||^2 T$$

Si $||\nabla\Phi|| = 1$ alors

$$\Delta T = \frac{-T}{\epsilon}\Delta\Phi + \frac{1}{\epsilon^2}T$$

Dans la limite pour $\epsilon \longrightarrow 0$, le terme $\frac{1}{\epsilon}$ est négligeable devant $\frac{1}{\epsilon^2}$ et on obtient finalement le problème suivant :

$$\begin{cases} ||\Delta T|| &= \frac{1}{\epsilon^2}T \quad \text{sur } \Omega \\ T &= 0 \quad \text{sur } \partial\Omega \end{cases}$$

Si on regarde le problème de conduction de la chaleur non stationnaire suivant :

$$\begin{cases} \frac{\partial T}{\partial t} &= \Delta T \\ T(x,0) &= 0 \end{cases}$$

On effectue une semi-discrétisation implicite et on obtient :

$$\frac{T^{n+1} - T^n}{\delta t} = \Delta T^{n+1}$$

Et on applique finalement au temps t = 0 :

$$\frac{T^1 - 0}{\delta t} = \Delta T^1 \Longrightarrow \frac{T^1}{\delta t} = \Delta T^1$$

Finalement, nous venons de montrer que l'approximation de ce problème nous permet de déterminer une solution approchée de la fonction T. Une fois l'approximation faite, il nous suffira d'écrire :

$$\Phi = -\log(T) \times \sqrt{\delta t}$$

Ainsi nous aurons une approximation de la fonction distance Φ que nous pourrons afficher à l'écran.

In [17]:
```
def chaleurdist(N,dt,t):
    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille1 = (N+1)*(N+1)
    T = np.ones((t+1,taille1))          #Initialisation de la solution final
e,
                                        #1 sur les bords et 0 à l'intérieur
    T[0,N+2:taille1 - N-2] = 0
    T[0,np.arange(2*N+1, taille1, N+1)] = 1.
    T[0,np.arange(2*N+2, taille1, N+1)] = 1.

    #Boucle temporelle

    for i in range (t):
        T[i+1,:] = sci.spsolve(matrix_lap2(N,dt),T[i,:])

    return T
```

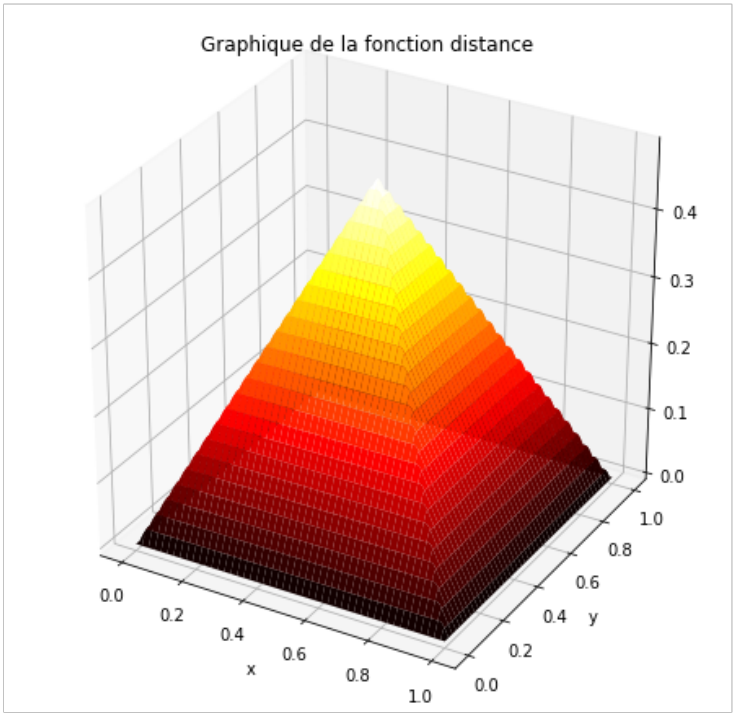
Puis on écrit une fonction qui va retourner la fonction distance :

In [18]:
```
def dist(N,dt):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    #Une itération temporelle de l'équation de la chaleur
    T = chaleurdist(N,dt,1)

    #Calcul de la fonction distance
    dist  = - np.log(T[1,:])*np.sqrt(dt)

    return dist
```

On affiche finalement la solution :

In [19]:
```
N = 1000

DIST = dist(N,0.00001)

x = np.linspace(0,1,N+1)
y = np.linspace(0,1,N+1)

print('La valeur maximale de la fonction distance est '+str(np.max(DIST)))

fig = plt.figure(figsize = [8,8])

ax = fig.add_subplot(111, projection = '3d')
X,Y = np.meshgrid(x,y)
ax.plot_surface(X,Y,DIST.reshape(N+1,N+1) ,cmap='hot')
plt.title("Graphique de la fonction distance")
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```

```
La valeur maximale de la fonction distance est 0.4935559135325694
```



Exemple d'application d'une méthode de pénalisation

Dans cette partie on se propose de résoudre le problème suivant :

$$\begin{cases} \Delta\Phi & = F \quad \text{sur } \Omega_i \\ \Phi_{\partial\Omega_i} & = K \\ \Phi_{\partial\Omega} & = 0 \end{cases}$$

Nous avons vu en cours que ce problème est équivalent au suivant :

$$\begin{cases} \Delta\Phi_\epsilon & = \chi_{\Omega\setminus\Omega_i}\left(\frac{\Phi_\epsilon - K}{\epsilon}\right) + F \quad \text{sur } \Omega \\ \Phi_{\partial\Omega} & = 0 \end{cases}$$

Avec

$$\chi_{\Omega\setminus\Omega_i} = \begin{cases} 0 & \text{si } x \in \Omega_i \\ 1 & \text{sinon} \end{cases}$$

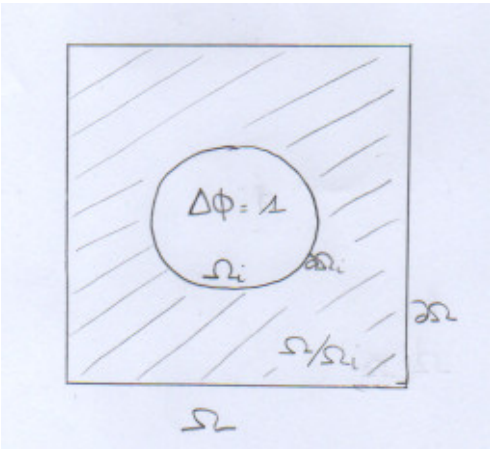
En effet, si l'on multiplie par ϵ de chaque côté de l'équation on obtient :

$$\epsilon\Delta\Phi_\epsilon = \chi_{\Omega\setminus\Omega_i}(\Phi_\epsilon - K) + \epsilon F$$

On observe alors 2 cas de figures :

- $x \in \Omega_i \Longrightarrow \Delta\Phi_\epsilon = F$
- $x \in \Omega\setminus\Omega_i$ alors on fait tendre ϵ vers 0 et on observe $(\Phi_\epsilon - K) = 0 \Longrightarrow \Phi_\epsilon = K$

L'exemple que nous allons résoudre est le suivant :



$$\begin{cases} \Delta\Phi & = 1 \quad \text{sur } \Omega_i \\ \Phi_{\partial\Omega_i} & = K \\ \Phi_{\partial\Omega} & = 0 \end{cases}$$

On connait une solution Φ de ce problème qui est la suivante :

$$\Phi = \frac{1}{4} \times \left(x_1^2 + x_2^2\right)$$

Et $K = \frac{1}{4} \times R^2$ où R est le rayon du cercle Ω_i

Dans un premier temps, nous allons recopier la matrice du Laplacien, puis nous écrirons la fonction χ qui dépendra du rayon du cercle décrivant notre domaine Ω_i. Le but est de pouvoir comparer notre résultat avec la solution exacte Φ et de finalement afficher les graphes d'erreur.

On commence par importer, comme d'habitude, les packages nécessaires.

In [2]:
```
import numpy as np                          #Package pour calculs scientifique
s
import scipy.sparse as sparse               #Algèbre linéaire creuse
import matplotlib.pyplot as plt             #Permet la création de graphique
import scipy.sparse.linalg as sci           #Contient plusieurs packages pour
le calcul scientifique
from mpl_toolkits.mplot3d import Axes3D     #Utile pou le graphiques 3D
import time                                 #Affichage du temps de calcul
from IPython.display import Image           #Affichage d'image dans le Jupyter
```

On écrit la fonction retournant la matrice de discrétisation du Laplacien sur l'intégralité des points du domaine.

In [4]:
```
def matrix_lap(N):
    """Retourne une matrice qui discrétise le laplacien de u dans le domaine
Omega = [0,1]x[0,1],
    découpé en N intervalles en x et y. La matrice finale est une matrice sc
ipy.sparse CSR matrix.
    Cette matrice est de taille (N+1)*(N+1)"""

    h = 1./N      #Longueur du pas d'espace
    h2 = h*h

    #On note les inconnues de 0 à Nx suivant x (axe des abscisses)
    #et 0 à Ny suivant y (axe des ordonnées).
    #La taille du problème est donc (Nx+1)*(Ny+1).
    #Ici on prend le même pas d'espace en x et y
    #Cela correspond à x_i = i*h et y_j = j*h
    #et la numérotation (i,j) --> k := (N+1)*j+i.

    taille = (1+N)*(1+N)

    #Rédaction de la matrice, on commence par effectuer
    #le tableau des diagonales

    diags = np.zeros((5,taille))

    #Diagonale principale
    diags[2,:] = 1.
    diags[2, N+2:taille - (N+2)] = -4./h2
    diags[2, np.arange(2*N+1, taille, N+1)] = 1.
    diags[2, np.arange(2*N+2, taille, N+1)] = 1.

    #Diagonale "-1"
    diags[1,N+1:taille-(N+1)] = 1./h2
    diags[1, np.arange(2*N, taille, N+1)] = 0.
    diags[1, np.arange(2*N+1, taille, N+1)] = 0.

    #Diagonale "+1"
    diags[3, N+3:taille-(N+1)] = 1./h2
    diags[3, np.arange(2*N+2, taille, N+1)] = 0.
    diags[3, np.arange(2*N+3, taille, N+1)] = 0.

    #Diagonale "-(N+1)"
    diags[0, 1 : taille - (2*N+3)] = 1./h2
    diags[0, np.arange(N,taille,N+1)] = 0.
    diags[0, np.arange(N+1,taille,N+1)] = 0.

    #Diagonale "+(N+1)"
    diags[4, taille - N*N + 2 : taille - 1] = 1./h2
    diags[4, np.arange(taille - N*N + 1 + N ,taille,N+1)] = 0.
    diags[4, np.arange(taille - N*N + 2 + N ,taille,N+1)] = 0.

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-(N+1),-1,0,1,(N+1)],taille,taille, format = "
csr")

    return A
```

Puis on code les 2 fonctions dont nous avons besoin.

In [6]:
```
def func_ex(x,y):                                   #Retourne la fonction exa
cte solution
    return 0.25*((x-0.5)**2 + (y-0.5)**2)

def f(x,y):                                         #Retourne l'équation d'un
cercle pour le codage de 'chi'
    return (x-0.5)**2 + (y-0.5)**2
```

Grâce à cela nous rédigeons la fonction χ, fonction qui retourne 0 si x appartient au cercle de rayon R choisi par l'utilisateur et 1 si x est en dehors de ce cercle.

In [9]:
```
def Xhi(f,R):
    """R est le rayon du masque"""
    if f <= R**2:
        return 0
    else:
        return 1
```

Finalement nous pouvons créer notre masque, c'est à dire une fonction qui va retourner une matrice sparse de taille (N+1)x(N+1) où il y aura des 0 partout dans le cercle de rayon R et des 1 à l'extérieur.

In [28]:
```
def masque(f,R,N):
    """Retourne une matrice qui discrétise le masque du domaine grâce à la f
onction Xhi précédente,
    la matrice créée est une matrice sparse de taille (N+1)*(N+1)"""

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille = (N+1)*(N+1)

    diags = np.zeros((1,taille))

    #Diagonale principale
    for i in range(N+1):
        for j in range(N+1):
            k = i + j*(N+1)
            diags[0,k] = Xhi(f(x[i],y[j]),R)

    diags[0,0:N+2] = 1
    diags[0,taille-(N+2):taille] = 1
    diags[0, np.arange(2*N+1, taille, N+1)] = 1
    diags[0, np.arange(2*N+2, taille, N+1)] = 1

    M = sparse.spdiags(diags,0,taille,taille,format = 'csr')

    return M
```

La fonction suivante retourne la solution exacte afin de pouvoir effectuer les comparaisons.

In [30]:
```
def sol_ex(f,R,N):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    E = np.zeros((N+1)*(N+1))

    for i in range(N+1):
        for j in range(N+1):
            k = i + j*(N+1)
            if f(x[i],y[j]) <= R**2 :
                E[k] = func_ex(x[i],y[j])
            else :
                E[k] = 0.25*R**2

    return E
```

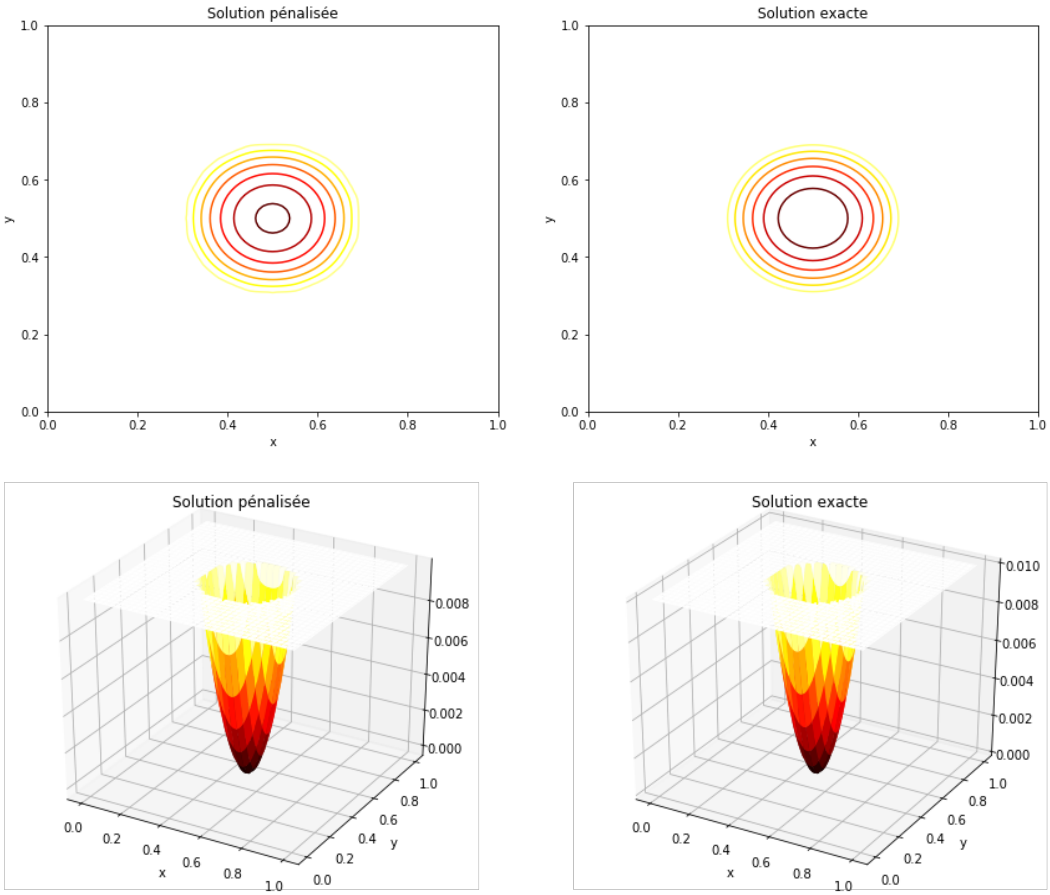
Et finalement on implémente la solution de la pénalisation

In [31]:
```
def sol_penal(f,R,N,eta):
    taille = (N+1)*(N+1)
    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    A = matrix_lap(N)
    B = masque(f,R,N)

    DISC = (A -(1/eta)*B)

    F = np.zeros(taille)

    for i in range (N+1):
        for j in range(N+1):
            k = i + j*(N+1)
            F[k] = - 0.25*R**2*Xhi(f(x[i],y[j]),R)/eta + 1


    U = sci.spsolve(DISC,F)

    return U
```

On affiche les graphes entre la solution de pénalisation et notre solution exacte pour un rayon $R = 0.2$ et un nombre de point $N = 100$. On se propose 2 types d'affichage différents. Le premier sera un affichage '2D'en terme de courbes de niveaux, le second une représentation '3D' de la solution.

In [34]:
```
def graphe(f,R,N,eta):
    taille = (N+1)*(N+1)
    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    A = sol_penal(f,R,N,eta)
    B = sol_ex(f,R,N)

    fig = plt.figure(figsize = [15,6])
    ax = fig.add_subplot(1,2,1)
    X,Y = np.meshgrid(x,y)
    ax.contour(X,Y,A.reshape(N+1,N+1) ,cmap='hot')
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title('Solution pénalisée')

    ax = fig.add_subplot(1,2,2)
    X,Y = np.meshgrid(x,y)
    ax.contour(X,Y,B.reshape(N+1,N+1),cmap='hot')
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title('Solution exacte')

    fig2 = plt.figure(figsize = [15,6])
    ax = fig2.add_subplot(1,2,1, projection = '3d')
    X,Y = np.meshgrid(x,y)
    ax.plot_surface(X,Y,A.reshape(N+1,N+1) ,cmap='hot')
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title('Solution pénalisée')

    ax = fig2.add_subplot(1,2,2,projection = '3d')
    X,Y = np.meshgrid(x,y)
    ax.plot_surface(X,Y,B.reshape(N+1,N+1),cmap='hot')
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title('Solution exacte')

    plt.show()
```

In [36]:
```
graphe(f,0.2,100,0.0000001)
```

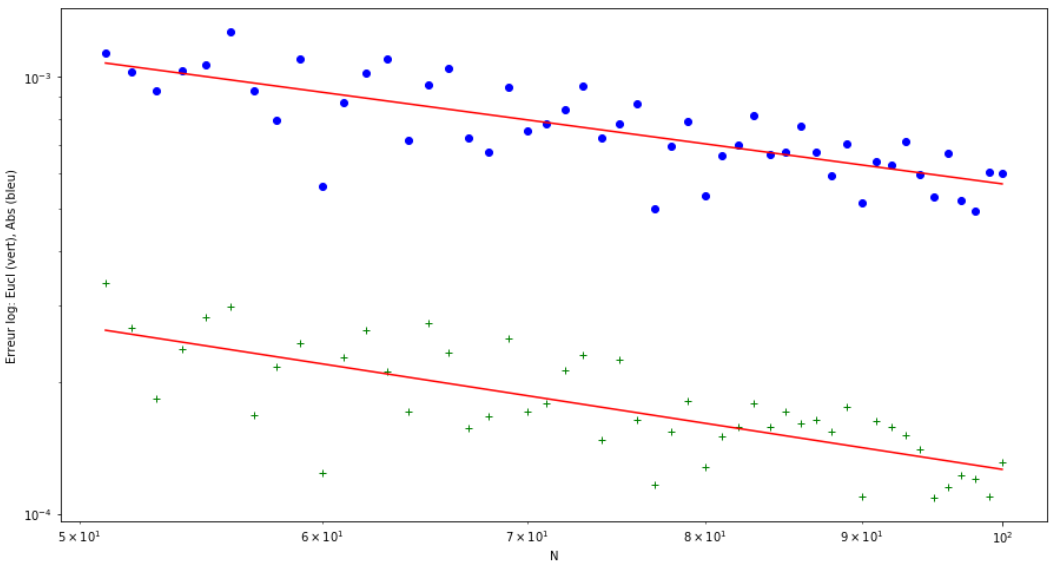


Nous pouvons remarquer que les 2 graphiques sont relativement similaires. On peut donc penser que notre approximation est juste. Cependant les représentations graphiques ne suffisant pas à prouver la justesse de notre travail, nous allons afficher les graphiques d'erreur. Nous étudierons l'erreur absolue ainsi que l'erreur euclidienne. Nous avons vu pendant le cours que la méthode de pénalisation telle que nous l'avons rédigée est une méthode d'approximation d'ordre 1. On s'attend donc à ce que les droites de régression linéaire correspondant aux erreurs aient une pente de -1.

In [37]:
```
def erreur_abs(A,E,N):
    return np.max(np.abs(A - E))

def erreur_eucl(A,E,N):
    return np.sqrt(np.sum(((E-A)**2))/(N+1)**2)

def aff(x,b,a):
    return np.exp(b)*x**(a)
```

In [43]:
```
def graphe_erreur(f,R,N,eta):
    tab_err = np.zeros(N)
    tab_err2 = np.zeros(N)

    ERR1 = np.zeros(N)
    ERR2 = np.zeros(N)

    x = np.linspace(1,N,N)

    for i in range(1,N+1):
        V = sol_penal(f, R, i, eta)
        U = sol_ex(f,R,i)

        tab_err[i-1] = erreur_abs(V,U,i)
        tab_err2[i-1] = erreur_eucl(V,U,i)

    x1 = x[N-50:N]
    Err1 = tab_err[N-50:N]
    Err2 = tab_err2[N-50:N]
    z1 = np.polyfit(np.log(x1),np.log(Err1),1)
    z2 = np.polyfit(np.log(x1),np.log(Err2),1)

    y1 = aff(x,z1[1],z1[0])[N-50:N]
    y2 = aff(x,z2[1],z2[0])[N-50:N]

    plt.figure(figsize = [15,8])

    plt.plot(x1,Err1,color='blue',marker='o', linestyle='none')
    plt.plot(x1,Err2,color='green',marker='+', linestyle='none')
    plt.plot(x1,y1,color='r', linestyle='-')
    plt.plot(x1,y2,color='r', linestyle='-')
    plt.xscale("log")
    plt.yscale("log")
    plt.xlabel('N')
    plt.ylabel('Erreur log: Eucl (vert), Abs (bleu)')

    plt.show()

    return z1[0],z2[0]
```

In [44]:
```
graphe_erreur(f,0.2,100,0.0000001)
```



Out[44]: (-0.9450788534430512, -1.088263481169789)

Nous pouvons finalement remarquer que nos 2 coefficients directeurs ont une pente d'environ -1. Cela prouve la justesse de notre approximation. Nous avons ainsi développé un exemple d'une méthode de pénalisation, méthode permettant d'approcher la solution d'équations aux dérivées partielles dans des domaines précis.

Résolution de l'équation d'élasticité linéaire

Le but de cette section est d'approcher la solution de l'équation aux dérivées partielles relative à l'étude de l'élasticité linéaire. Cela nous permettra de pouvoir décrire le comportement d'un corps subissant une force.

Pour cela, nous ne détaillerons pas l'intégralité des éléments vus lors des premières séances de ce cours. Nous allons commencer par développer certains éléments mathématiques dont nous avons besoin. Nous nous plaçons dans un espace de dimension 2 où $u = (u_1, u_2)$ correspond au déplacement du point après la déformation.

Nous avons défini le tenseur de Cauchy (appelé aussi tenseur de contraintes) $\tau_{i,j} = \{\sigma_{i,j}\}$ tel que :

$$\sigma_{i,j} = \lambda(e_{nn})\delta_{i,j} + 2\mu e_{i,j}$$

où $\delta_{i,j} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{sinon} \end{cases}$ et λ et μ sont appelés les coefficients de Lamé.

L'élément e est appelé le tenseur de déformation et est défini de la manière suivante :

$$e_{i,j} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$$

Le tenseur de déformation est une matrice symétrique , diagonalisable avec des valeurs propres réelles.

De plus, la définition d'équilibre (somme des forces appliquées en un point est nulle) nous montre que, si l'on note $F = \{F_i\}$ la force extérieure par unité de surface, on obtient l'équation suivante :

$$\frac{\partial \sigma_{i,j}}{\partial x_j} + F_i = 0$$

On peut donc essayer de résoudre cette équation en commençant par décrire les $\sigma_{i,j}$:

- $\sigma_{1,1} = \lambda\left(\frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2}\right) + \mu\frac{\partial u_1}{\partial x_1}$
- $\sigma_{2,2} = \lambda\left(\frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2}\right) + \mu\frac{\partial u_2}{\partial x_2}$
- $\sigma_{1,2} = \sigma_{2,1} = \mu e_{1,2} = \frac{1}{2}\mu\left(\frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1}\right)$

On obtient alors deux équations dont la première se développe de la façon suivante :

$$\begin{aligned}
&\implies \frac{\partial \sigma_{1,j}}{\partial x_j} + F_1 &&= 0 \\
&\iff \frac{\partial \sigma_{1,1}}{\partial x_1} + \frac{\partial \sigma_{1,2}}{\partial x_2} + F_1 &&= 0 \\
&\iff \lambda\left(\frac{\partial^2 u_1}{\partial^2 x_1} + \frac{\partial^2 u_2}{\partial x_1 \partial x_2}\right) + 2\mu\frac{\partial^2 u_1}{\partial^2 x_1} + \mu\left(\frac{\partial^2 u_1}{\partial^2 x_2} + \frac{\partial^2 u_2}{\partial x_2 \partial x_1}\right) + F_1 &&= 0 \\
&\iff \mu\left(\frac{\partial^2 u_1}{\partial^2 x_1} + \frac{\partial^2 u_1}{\partial^2 x_2}\right) + (\lambda + \mu)\left(\frac{\partial^2 u_1}{\partial^2 x_2} + \frac{\partial^2 u_2}{\partial x_1 \partial x_2}\right) + F_1 &&= 0 \\
&\iff \mu\Delta u_1 + (\lambda + \mu)\left(\frac{\partial^2 u_1}{\partial^2 x_1} + \frac{\partial^2 u_2}{\partial x_1 \partial x_2}\right) + F_1 &&= 0
\end{aligned}$$

Si l'on pose $\delta = \frac{\lambda + \mu}{\mu}$ et $\widetilde{F}_i = \frac{F_i}{\mu}$ on obtient :

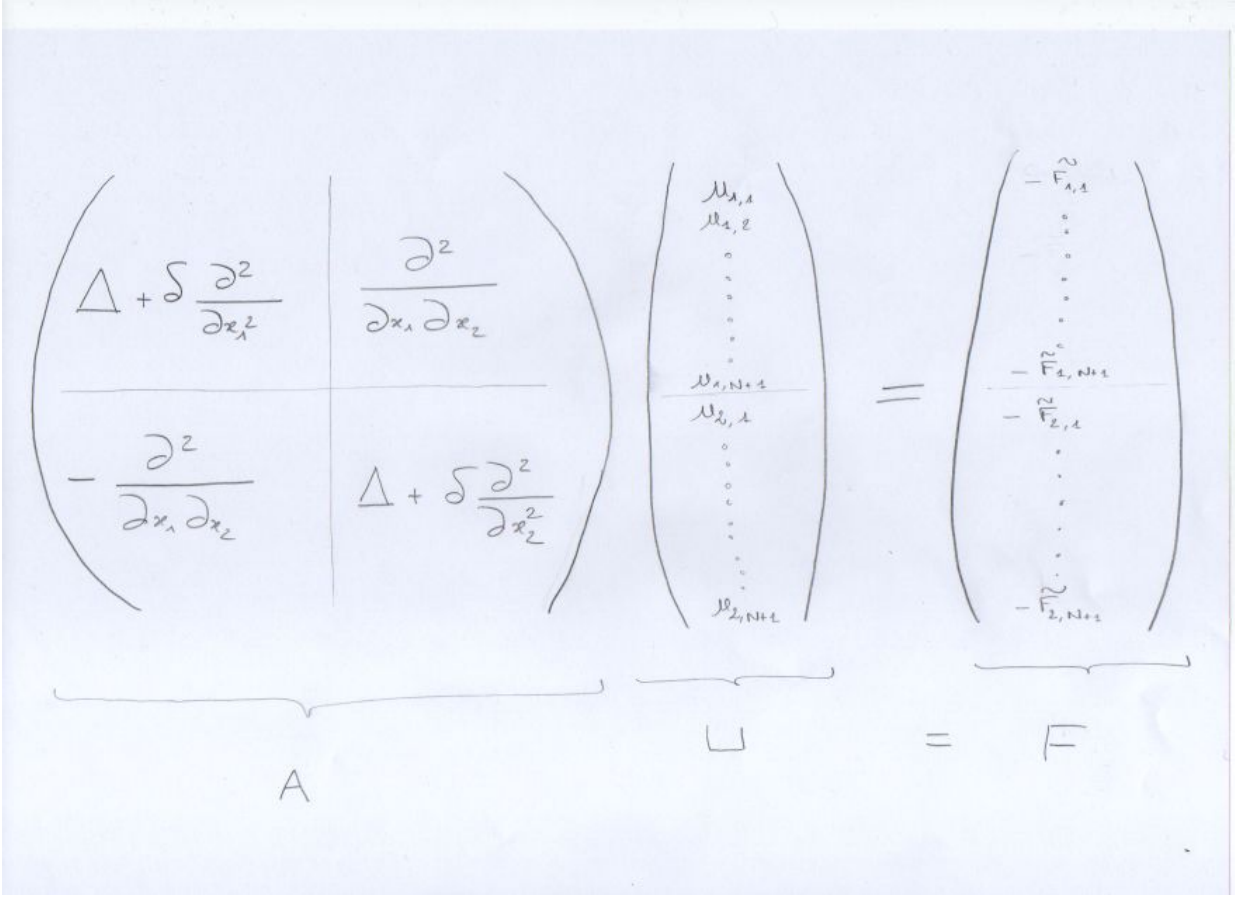
$$\Delta u_1 + \delta\left(\frac{\partial^2 u_1}{\partial^2 x_1} + \frac{\partial^2 u_2}{\partial x_1 \partial x_2}\right) = -\widetilde{F}_1$$

En procédant de même on obtient la seconde équation :

$$\Delta u_2 + \delta\left(\frac{\partial^2 u_2}{\partial^2 x_2} + \frac{\partial^2 u_1}{\partial x_1 \partial x_2}\right) = -\widetilde{F}_2$$

De manière générale l'équation aux dérivées partielles que nous allons discrétiser est la suivante :

$$\mu\Delta u + (\lambda + \mu)\nabla(\nabla u) + F = 0$$



Nous résolverons le système puis afficherons les 2 vecteurs de déformation u_1 et u_2. Finalement, nous connaissons les deux fonctions solution de l'équation (par rapport aux fonctions F données), ce qui va nous permettre, en dernier lieu, d'afficher les graphiques d'erreur et conclure sur la pertinence de notre approximation.

On commence par importer les packages dont nous avons besoin :

In [1]:
```
import numpy as np                          #Package pour calculs scientifique
s
import scipy.sparse as sparse               #Algèbre linéaire creuse
import matplotlib.pyplot as plt             #Permet la création de graphique
import scipy.sparse.linalg as sci           #Contient plusieurs packages pour
le calcul scientifique
from mpl_toolkits.mplot3d import Axes3D     #Utile pou le graphiques 3D
import time                                 #Affichage du temps de calcul
from IPython.display import Image           #Affichage d'image dans le Jupyter
from scipy.sparse import bmat               #Construction d'une matrice par bl
ocs
```

On va maintenant écrire les codes pour les différentes discrétisations des opérateurs différentiels. La description des matrices sera lisible en commentaire dans le code.

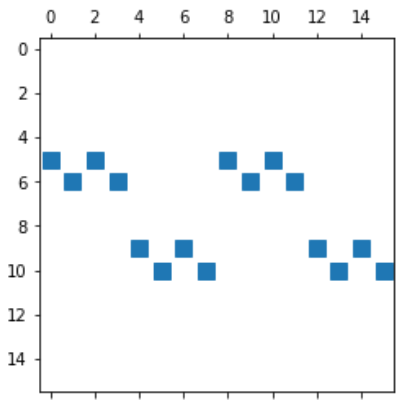
In [2]:
```
def Laplacien(N):
    """Retourne une matrice sparse de taille (N+1)*(N+1) correspondant
    à la discrétisation du Laplacien sur l'intégralité du maillage"""

    h = 1./N
    h2 = h*h
    taille = (N+1)*(N+1)

    diags = np.zeros((5,taille))

    #Diagonale principale
    diags[2,:] = 1.
    diags[2, N+2:taille - (N+2)] = -4./h2
    diags[2, np.arange(2*N+1, taille, N+1)] = 1.
    diags[2, np.arange(2*N+2, taille, N+1)] = 1.

    #Diagonale "-1"
    diags[1,N+1:taille-(N+1)] = 1./h2
    diags[1, np.arange(2*N, taille, N+1)] = 0.
    diags[1, np.arange(2*N+1, taille, N+1)] = 0.

    #Diagonale "+1"
    diags[3, N+3:taille-(N+1)] = 1./h2
    diags[3, np.arange(2*N+2, taille, N+1)] = 0.
    diags[3, np.arange(2*N+3, taille, N+1)] = 0.

    #Diagonale "-(N+1)"
    diags[0, 1 : taille - (2*N+3)] = 1./h2
    diags[0, np.arange(N,taille,N+1)] = 0.
    diags[0, np.arange(N+1,taille,N+1)] = 0.

    #Diagonale "+(N+1)"
    diags[4, taille - N*N + 2 : taille - 1] = 1./h2
    diags[4, np.arange(taille - N*N + 1 + N ,taille,N+1)] = 0.
    diags[4, np.arange(taille - N*N + 2 + N ,taille,N+1)] = 0.

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-(N+1),-1,0,1,(N+1)],taille,taille, format = "
csr")

    return A
```

La visualisation de la matrice a déjà été faite précédemment, nous nous contenterons seulement de regarder la visualisation des autres matrices.

In [3]:
```
def matrix_croi(N):
    """Retourne une matrice sparse de taille (N+1)*(N+1) correspondant
    à la discrétisation des dérivées croisées sur l'intégralité du maillage"
""

    h = 1./N
    h2 = h*h
    taille = (N+1)*(N+1)

    diags = np.zeros((4,taille))

    #Diagonale "-N-2"
    diags[0, 0 : taille - 2*(N+1)] = 1./(4*h2)
    diags[0, np.arange(N-1,taille,N+1)] = 0
    diags[0, np.arange(N,taille,N+1)] = 0

    #Diagonale "-N"
    diags[1, 2 : taille - (2*N+2)] = -1./(4*h2)
    diags[1, np.arange(N+1,taille,N+1)] = 0
    diags[1, np.arange(N+2,taille,N+1)] = 0

    #Diagonale "N"
    diags[2, 2*(N+1) : taille - 2] = -1./(4*h2)
    diags[2, np.arange(2*(N+1)+(N-1),taille,N+1)] = 0
    diags[2, np.arange(2*(N+1)+N,taille,N+1)] = 0

    #Diagonale "N+2"
    diags[3, 2*(N+2) : taille] = 1./(4*h2)
    diags[3, np.arange(2*(N+2)+N-1,taille,N+1)] = 0
    diags[3, np.arange(2*(N+2)+N,taille,N+1)] = 0

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-(N+2),-N,N,(N+2)],taille,taille, format = "cs
r")

    return A
```

In [4]:
```
plt.spy(matrix_croi(3))
```

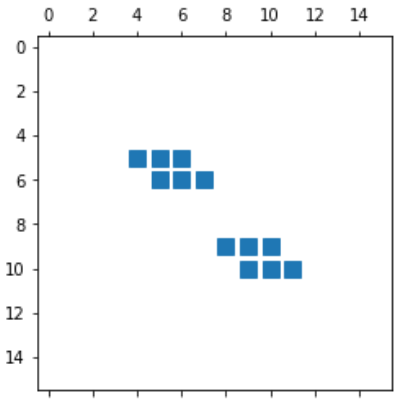
Out[4]:
```
<matplotlib.lines.Line2D at 0x3fefdd8>
```



In [5]:
```
def der_sec1(N):
    """Retourne une matrice sparse de taille (N+1)*(N+1)
    correspondant à la discrétisation de la dérivée seconde
    par rapport à la premiere variable  sur l'intégralité du maillage"""

    h = 1./N
    h2 = h*h
    taille = (N+1)*(N+1)

    diags = np.zeros((3,taille))

    #Diagonale principale
    diags[1,:] = 0
    diags[1, N+2:taille - (N+2)] = -2./h2
    diags[1, np.arange(2*N+1, taille, N+1)] = 0
    diags[1, np.arange(2*N+2, taille, N+1)] = 0

    #Diagonale "-1"
    diags[0,N+1:taille-(N+1)] = 1./h2
    diags[0, np.arange(2*N, taille, N+1)] = 0.
    diags[0, np.arange(2*N+1, taille, N+1)] = 0.

    #Diagonale "+1"
    diags[2, N+3:taille-(N+1)] = 1./h2
    diags[2, np.arange(2*N+2, taille, N+1)] = 0.
    diags[2, np.arange(2*N+3, taille, N+1)] = 0.

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-1,0,1],taille,taille, format = "csr")

    return A
```

In [6]:
```
plt.spy(der_sec1(3))
```

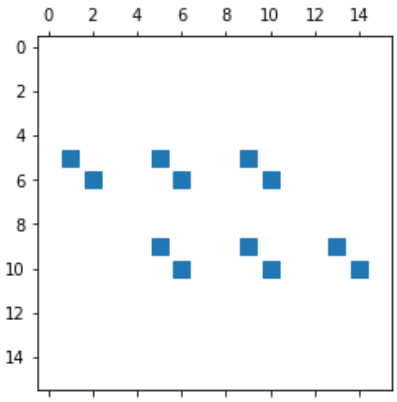
Out[6]:
```
<matplotlib.lines.Line2D at 0x832a9b0>
```



In [7]:
```
def der_sec2(N):
    """Retourne une matrice sparse de taille (N+1)*(N+1)
    correspondant à la discrétisation de la dérivée seconde
    par rapport à la seconde variable  sur l'intégralité du maillage"""

    h = 1./N
    h2 = h*h
    taille = (N+1)*(N+1)

    diags = np.zeros((3,taille))

    #Diagonale principale
    diags[1,:] = 0.
    diags[1, N+2:taille - (N+2)] = -2./h2
    diags[1, np.arange(2*N+1, taille, N+1)] = 0.
    diags[1, np.arange(2*N+2, taille, N+1)] = 0.

    #Diagonale "-(N+1)"
    diags[0, 1 : taille - (2*N+3)] = 1./h2
    diags[0, np.arange(N,taille,N+1)] = 0.
    diags[0, np.arange(N+1,taille,N+1)] = 0.

    #Diagonale "+(N+1)"
    diags[2, taille - N*N + 2 : taille - 1] = 1./h2
    diags[2, np.arange(taille - N*N + 1 + N ,taille,N+1)] = 0.
    diags[2, np.arange(taille - N*N + 2 + N ,taille,N+1)] = 0.

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-(N+1),0,(N+1)],taille,taille, format = "csr")

    return A
```

In [8]:
```
plt.spy(der_sec2(3))
```

Out[8]:
```
<matplotlib.lines.Line2D at 0x8397ef0>
```



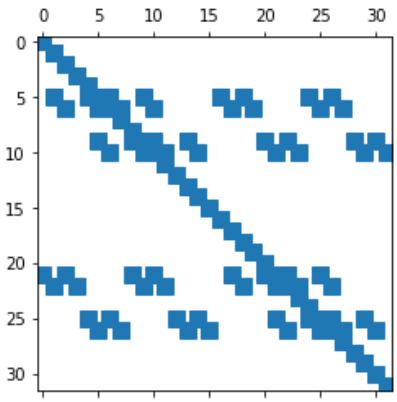
On peut remarquer que si l'on somme les matrices der_sec1 et der_sec2 on retrouve la matrice de discrétsation du Laplacien (ce qui est, par définition, logique).

Ensuite, on assemble la matrice de discrétisation grâce à la fonction suivante :

In [9]:
```
def matrix_elas(N,mu,lamb):
    """Retourne la matrice sparse globale pour la discrétisation du problème
d'élasticité linéaire.
    Cette matrice sera de taille (2*(N+1))^2. Cette fonction prend en paramè
tre N
    le nbr d'intervalle de discrétisation, mu et lamb des scalaires pour l'é
lasticité linéaire."""

    delta = (lamb + mu)/mu

    LAP = Laplacien(N)
    CR = matrix_croi(N)
    DER1 = der_sec1(N)
    DER2 = der_sec2(N)

    #Utilisation de la fonction bmat qui permet la construction d'une matric
e "par blocs".

    MATRIX = bmat([ [LAP + delta*DER1 , CR],[-CR , LAP + delta*DER2]], forma
t = 'csr')

    return MATRIX
```

In [10]:
```
plt.spy(matrix_elas(3,1,0))
```

Out[10]:
```
<matplotlib.lines.Line2D at 0x8403dd8>
```



On écrit les fonctions de second membre et solutions exactes données en cours.

In [11]:
```
def snd_mbr(x,y,delta):
    """Les second membres F1 et F2 sont exprimés directement avec le signe -
devant
    , ce sont les fonctions tildes, i.e pour trouver F1 et F2;
    F_itilde = F_i/mu. On prendra un delta egal à 1"""

    F1 = 6*(1-3*x+2*x**2)*(-1+y)*y**3 + 6*(-1+x)**3*x*y*(-1+2*y) + delta*(-1
+y)*(-1+5*y-4*y**2+6*y**3+12*x**2*y**3-2*x*(-1+5*y-4*y**2+9*y**3))
    F2 = 2*(-1+y)**3*y + 6*(-1+x)*x*(1-3*y+2*y**2) + delta*(-1+x)*(y**2*(-3+
4*y)+4*x**2*y**2*(-3+4*y)+x*(6-18*y+27*y**2-20*y**3))

    return [F1, F2]
```

In [12]:
```
def func_exacte(x1,x2):
    u1 = x1*x2**3*(x1-1)**3*(x2-1)
    u2 = x1*x2*(x1-1)*(x2-1)**3

    return [u1,u2]
```

Finalement on écrit le code pour résoudre le système linéaire.

In [13]:
```
def resolution(N,mu,lamb,force):

    #Pour une valeur de delta égale = 1 il suffit de prendre lambda = 0

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille = (N+1)*(N+1)

    delta = (lamb + mu)/mu

    F = np.zeros(2*taille)

    for i in range(1,N):
        for j in range(1,N):
            k = i + j*(N+1)
            F[k] = force(x[i],y[j],delta)[0]/mu
            F[k+taille] = force(x[i],y[j],delta)[1]/mu

    MAT = matrix_elas(N,mu,lamb)

    U = sci.spsolve(MAT,F)

    return U
```

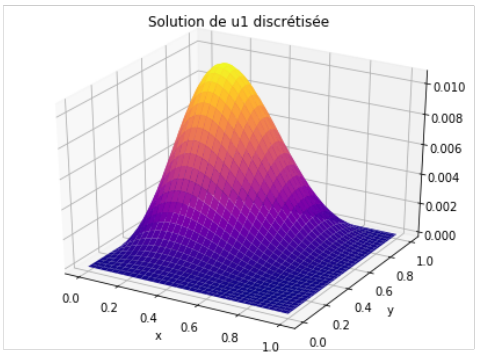
On fait le calcul de la solution exacte afin de pouvoir faire des comparaisons :

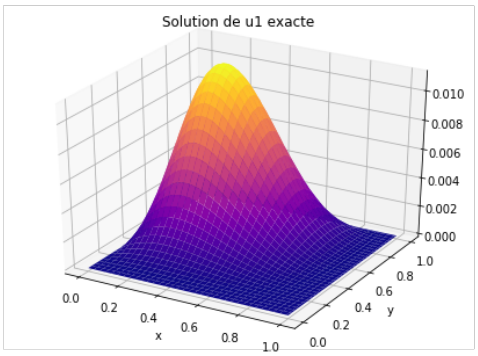
In [14]:
```
def solution_exacte(N,mu,lamb,f_ex):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille = (N+1)*(N+1)

    delta = (lamb + mu)/mu
    E = np.zeros(2*taille)

    for i in range (N+1):
        for j in range (N+1):
            k = i + j*(N+1)
            E[k] = f_ex(x[i],y[j])[0]
            E[k + taille] = f_ex(x[i],y[j])[1]

    return E
```

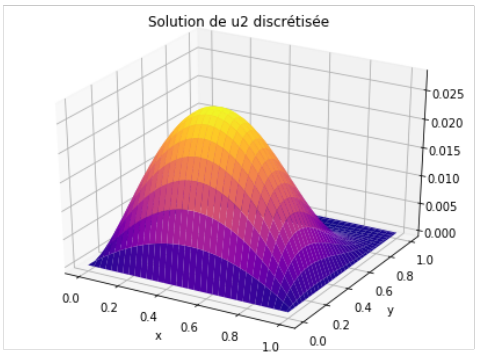
Et on affiche enfin les graphiques :

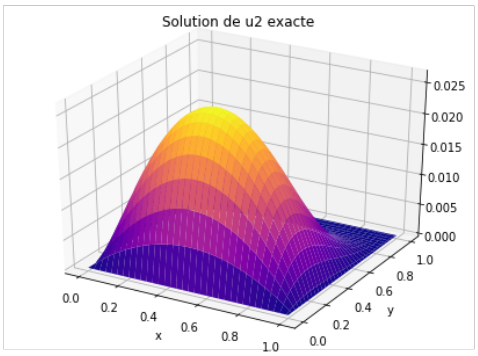
In [15]:
```
def graphe_reso(N,mu,lamb,force,f_ex):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille = (N+1)*(N+1)

    U = resolution(N,mu,lamb,force)
    E = solution_exacte(N,mu,lamb,f_ex)

    u1 = E[0:taille]
    u2 = E[taille : 2*taille]

    U1 = U[0:taille]
    U2 = U[taille : 2*taille]

    fig = plt.figure(figsize = [16,12])

    ax = fig.add_subplot(2,2,1,projection='3d')
    X,Y = np.meshgrid(x,y)
    ax.plot_surface(X,Y, U1.reshape((N+1,N+1)), cmap='plasma')
    plt.title("Solution de u1 discrétisée")
    plt.xlabel("x")
    plt.ylabel("y")

    ax = fig.add_subplot(2,2,2,projection='3d')
    X,Y = np.meshgrid(x,y)
    ax.plot_surface(X,Y, u1.reshape((N+1,N+1)), cmap='plasma')
    plt.title("Solution de u1 exacte")
    plt.xlabel("x")
    plt.ylabel("y")


    ax = fig.add_subplot(2,2,3,projection='3d')
    X,Y = np.meshgrid(x,y)
    ax.plot_surface(X,Y, U2.reshape((N+1,N+1)), cmap='plasma')
    plt.title("Solution de u2 discrétisée")
    plt.xlabel("x")
    plt.ylabel("y")

    ax = fig.add_subplot(2,2,4,projection='3d')
    X,Y = np.meshgrid(x,y)
    ax.plot_surface(X,Y, u2.reshape((N+1,N+1)), cmap='plasma')
    plt.title("Solution de u2 exacte")
    plt.xlabel("x")
    plt.ylabel("y")

    plt.show()
```

In [16]:
```
graphe_reso(100,1,0,snd_mbr,func_exacte)
```







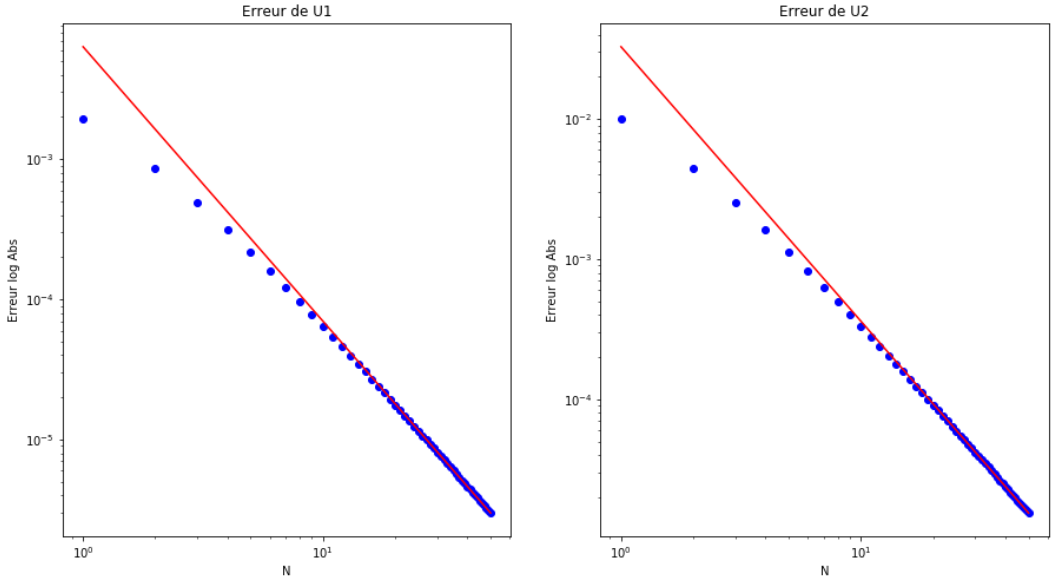


Et enfin on affiche les graphes d'erreurs. Nous regardons, comme d'habitude, les erreurs absolues et euclidiennes. Le schéma utilisé étant un schéma d'ordre 2, nous attendons une pente de régression linéaire de -2.

In [20]:
```
def erreur_abs(A,E,N):
    return np.max(np.abs(E - A))

def erreur_eucl(A,E,N):
    return np.sqrt(np.sum((E-A)**2))/((N+1)**2)

def aff(x,b,a):
    return np.exp(b)*x**(a)
```

In [21]:
```
def graphe_erreur(N,mu,lamb,force,f_ex):
    tab_err1 = np.zeros(N)
    tab_err2 = np.zeros(N)
    ERR1 = np.zeros(N)
    ERR2 = np.zeros(N)
    x = np.linspace(1,N,N)


    for i in range(1,N+1):
        taille = (N+1)*(N+1)
        U = resolution(N,mu,lamb,force)
        E = solution_exacte(N,mu,lamb,f_ex)

        E1 = E[0:taille]
        E2 = E[taille : 2*taille]

        U1 = U[0:taille]
        U2 = U[taille : 2*taille]

        tab_err1[i-1] = erreur_eucl(E1,U1,i)
        tab_err2[i-1] = erreur_eucl(E2,U2,i)

    x1 = x[N-10:N]
    Err1 = tab_err1[N-10:N]
    Err2 = tab_err2[N-10:N]
    z1 = np.polyfit(np.log(x1),np.log(Err1),1)
    z2 = np.polyfit(np.log(x1),np.log(Err2),1)

    y1 = aff(x,z1[1],z1[0])
    y2 = aff(x,z2[1],z2[0])

    plt.figure(figsize = [15,8])
    plt.subplot(1,2,1)
    plt.plot(x,tab_err1,color='blue',marker='o', linestyle='none')
    plt.plot(x,y1,color='r', linestyle='-')
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel('N')
    plt.ylabel('Erreur log Abs')
    plt.title('Erreur de U1')

    plt.subplot(1,2,2)
    plt.plot(x,tab_err2,color='blue',marker='o', linestyle='none')
    plt.plot(x,y2,color='r', linestyle='-')
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel('N')
    plt.ylabel('Erreur log Abs')
    plt.title('Erreur de U2')

    plt.show()

    return z1[0],z2[0]
```

In [22]:
```
graphe_erreur(50,1,0,snd_mbr,func_exacte)
```



Out[22]:
```
(-1.9567584564535896, -1.956758456453595)
```

In [ ]:
```

```

Exemple d'application de l'élasticité linéaire

Dans cette ultime partie, nous présentons un exemple d'application de l'utilisation de l'équation d'élasticité linéaire. Nous allons créer une image représentant un visage avec une bouche neutre et tenter de déformer cette bouche. L'intérêt d'utiliser l'équation d'élasticité linéaire est le suivant : Si nous nous intéressons à la seule déformation de la bouche, on s'attend, via notre discrétisation à déformer tout le domaine et donc à ce que les autres éléments du visage se déforment également. Cette technique fut utilisée notamment dans l'industrie cinématographique. En effet, lorsqu'un acteur parle au travers d'un masque (donc une déformation de sa bouche) on s'attend à ce que tout le visage réagisse pour garantir un maximum de réalisme.

Dans notre exemple, nous implémenterons seulement un visage avec une bouche et deux yeux. Nous nous intéresserons alors à la déformation des yeux lorsque nous forcerons le déplacement de la bouche. La stratégie employée dans cette partie est la suivante. Dans un premier temps, nous illustrerons les domaines (images) qui nous intéressent, puis, dans un second temps, nous appliquerons l'équation d'élasticité et, finalement, nous afficherons les résultats. Il est important de remarquer que la précision de notre travail repose sur celle de l'approximation de l'élasticité linéaire démontré dans le document "ELAST_LIN.ipynb".

Comme à notre habitude, on commence par importer les packages qui nous seront utiles :

In [4]:
```
import numpy as np                          #Package pour calculs scientifique
s
import scipy.sparse as sparse               #Algèbre linéaire creuse
import matplotlib.pyplot as plt             #Permet la création de graphique
import scipy.sparse.linalg as sci           #Contient plusieurs packages pour
le calcul scientifique
from mpl_toolkits.mplot3d import Axes3D     #Utile pou le graphiques 3D
import time                                 #Affichage du temps de calcul
from IPython.display import Image           #Affichage d'image dans le Jupyter
from scipy.sparse import bmat               #Construction d'une matrice par bl
ocs
```

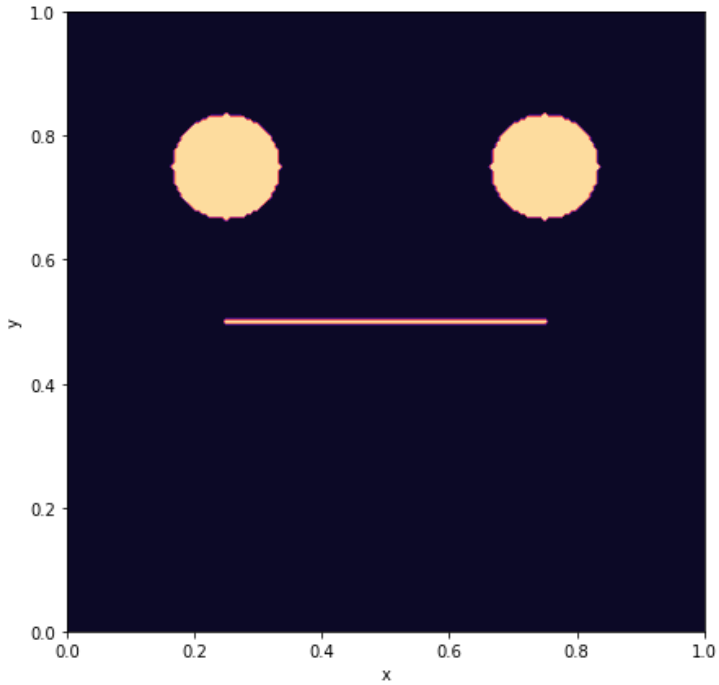
Nous allons commencer par afficher l'image de base que l'on souhaite déformer. Pour aller plus vite par la suite, nous appellerons notre visage Bryan. Ce dernier sera créé de la façon suivante. Nous implémentons une matrice carrée de taille un multiple de 12 (par soucis de symétrie). Nous initialisons la matrice à 0 et mettons des 1 au niveau de la bouche et des yeux. Pour coder ces derniers, nous utilisons une équation de cercle.

In [5]:
```
def f_gauche(x,y,N):
    return (x-int(3*N/4))**2 + (y - int(N/4))**2

def f_droite(x,y,N):
    return (x-int(3*N/4))**2 + (y - int(3*N/4))**2
```

Puis on écrit le code pour Bryan ainsi que sa visualisation.

In [6]:
```
def BRYAN(N):
    """Retourne une matrice de taille NxN avec N = 12k (mutiple de 12) pour
des raisons de symétrie retournant
    le visage de Bryan."""
    ############ PENSER A METTRE UN MULTIPLE DE 12 COMME VALEUR DE N ########
####

    taille = (N+1)*(N+1)

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    MAT = np.zeros((N+1,N+1))

    #Construction Bouche

    for i in np.arange(int(N/4),int(3*N/4) +1):
        MAT[int(N/2)][i] = 1.

    #Construction des yeux

    #Oeil gauche
    for i in range (N+1):
        for j in range (N+1):
            if f_gauche(i,j,N) <= int(N/12)**2 :
                MAT[i][j] = 1.

    #Oeil droit
    for i in range(N+1):
        for j in range(N+1):
            if f_droite(i,j,N) <= int(N/12)**2:
                MAT[i][j] = 1.

    return MAT
```

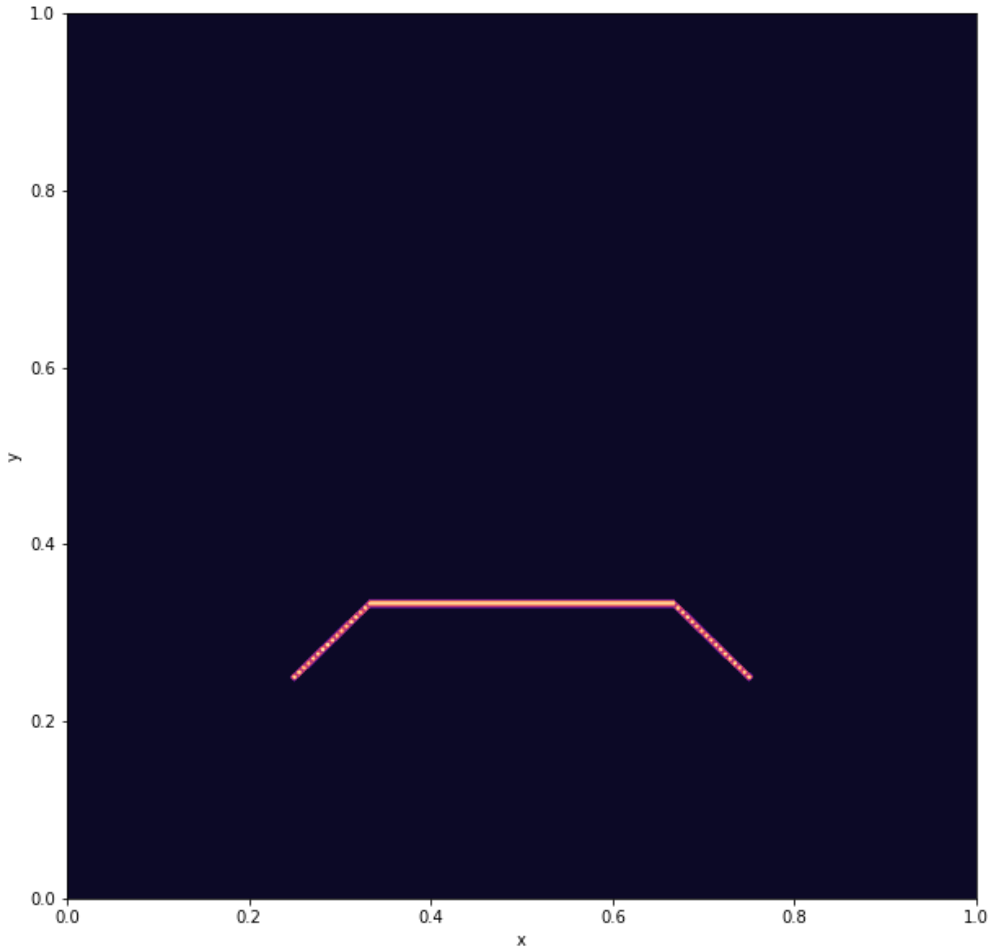
In [8]:
```
A = BRYAN(192)

x = np.linspace(0,1,193)
y = np.linspace(0,1,193)

fig = plt.figure(figsize = [7,7])

ax = fig.add_subplot(111)
X,Y = np.meshgrid(x,y)
ax.contourf(X,Y,A, cmap = 'magma')
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```



Maintenant nous allons créer le domaine représentant la bouche après déformation de la même manière que précédemment en mettant des 1 au niveau de la bouche déformée et des 0 partout ailleurs. On appelle ce domaine *dom_ def(N)*.

In [9]:
```
def dom_def(N):

    ############ PENSER A METTRE UN MULTIPLE DE 12 COMME VALEUR DE N ########
####

    taille = (N+1)*(N+1)

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    y_haut = int(N/3)
    y_bas = int(N/4)
    borne_gauche = int(N/4)
    borne_gaubouche = int(N/3)
    borne_droibouche = int(2*N/3)
    borne_droite = int(3*N/4)

    MAT = np.zeros((N+1,N+1))

    #Construction Bouche pour déformation

    for i in range(y_bas,y_haut) :
        for j in range (borne_droibouche, borne_droite + 1):
            if i == (N - j) :
                MAT[i][j] = 1.

    for i in range (borne_gaubouche, borne_droibouche + 1):
        MAT[y_haut][i] = 1.


    for i in range (y_bas,y_haut):
        MAT[i][i] = 1.


    return MAT
```

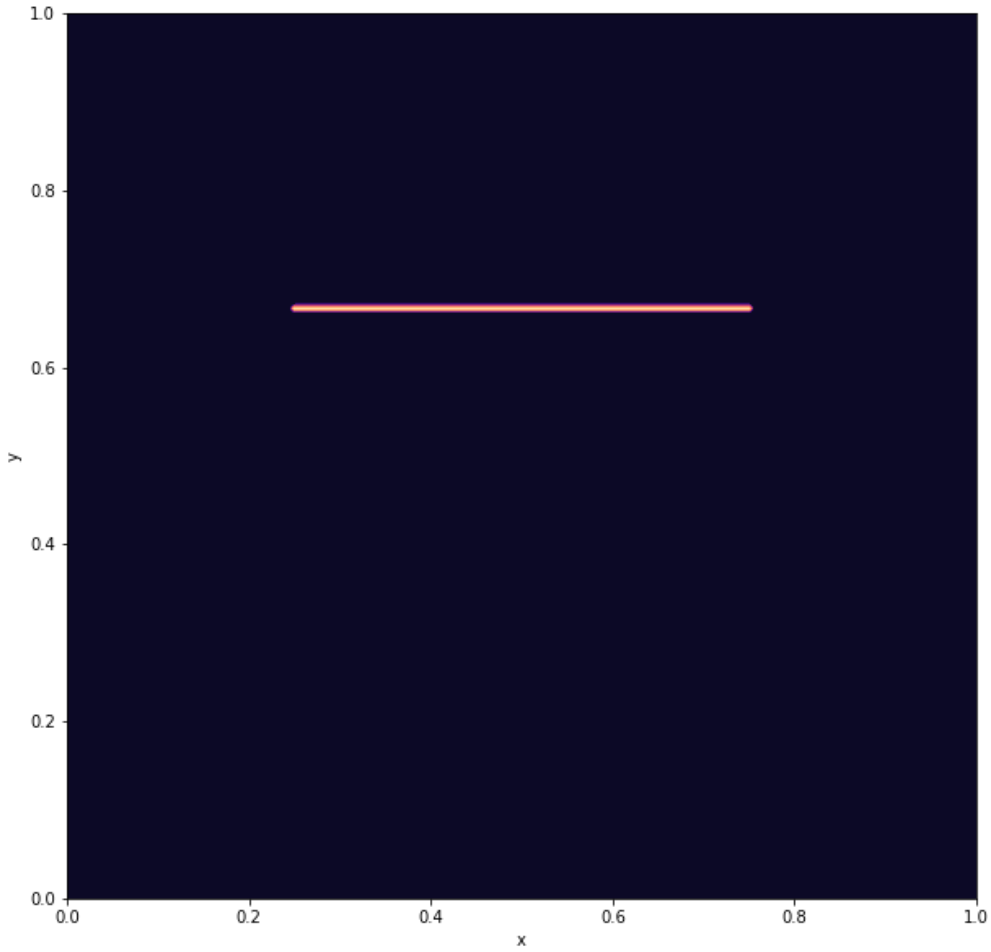
In [10]:
```
B = dom_def(192)

x = np.linspace(0,1,193)
y = np.linspace(0,1,193)

fig = plt.figure(figsize = [10,10])

ax = fig.add_subplot(111)
X,Y = np.meshgrid(x,y)
ax.contourf(X,Y,B, cmap = 'magma')
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```



Finalement, nous allons reproduire seulement la bouche de Bryan que l'on placera plus haute dans notre domaine (nous verrons l'intérêt de cette démarche par la suite). Nous appellerons ce domaine $dom\_\ init(N)$.

In [11]:
```
def dom_init(N):

    ########### PENSER A METTRE UN MULTIPLE DE 12 COMME VALEUR DE N ########
####

    taille = (N+1)*(N+1)

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    MAT = np.zeros((N+1,N+1))

    #Construction Bouche

    for i in np.arange(int(N/4),int(3*N/4) +1):
        MAT[int(2*N/3)][i] = 1.

    return MAT
```

In [12]:
```
C = dom_init(192)

x = np.linspace(0,1,193)
y = np.linspace(0,1,193)

fig = plt.figure(figsize = [10,10])

ax = fig.add_subplot(111)
X,Y = np.meshgrid(x,y)
ax.contourf(X,Y,C, cmap = 'magma')
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```



Maintenant nous allons ajouter les codes pour l'élasticité linéaire. Comme nous l'avons déjà fait dans les fichiers précédents, nous ne les détaillerons pas.

In [14]:
```
def Laplacien(N):
    """Retourne une matrice sparse de taille (N+1)*(N+1) correspondant
    à la discrétisation du Laplacien sur l'intégralité du maillage"""

    h = 1./N
    h2 = h*h
    taille = (N+1)*(N+1)

    diags = np.zeros((5,taille))

    #Diagonale principale
    diags[2,:] = 1.
    diags[2, N+2:taille - (N+2)] = -4./h2
    diags[2, np.arange(2*N+1, taille, N+1)] = 1.
    diags[2, np.arange(2*N+2, taille, N+1)] = 1.

    #Diagonale "-1"
    diags[1,N+1:taille-(N+1)] = 1./h2
    diags[1, np.arange(2*N, taille, N+1)] = 0.
    diags[1, np.arange(2*N+1, taille, N+1)] = 0.

    #Diagonale "+1"
    diags[3, N+3:taille-(N+1)] = 1./h2
    diags[3, np.arange(2*N+2, taille, N+1)] = 0.
    diags[3, np.arange(2*N+3, taille, N+1)] = 0.

    #Diagonale "-(N+1)"
    diags[0, 1 : taille - (2*N+3)] = 1./h2
    diags[0, np.arange(N,taille,N+1)] = 0.
    diags[0, np.arange(N+1,taille,N+1)] = 0.

    #Diagonale "+(N+1)"
    diags[4, taille - N*N + 2 : taille - 1] = 1./h2
    diags[4, np.arange(taille - N*N + 1 + N ,taille,N+1)] = 0.
    diags[4, np.arange(taille - N*N + 2 + N ,taille,N+1)] = 0.

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-(N+1),-1,0,1,(N+1)],taille,taille, format = "
csr")

    return A
```

In [15]:
```
def matrix_croi(N):
    """Retourne une matrice sparse de taille (N+1)*(N+1) correspondant
    à la discrétisation des dérivées croisées sur l'intégralité du maillage"
""

    h = 1./N
    h2 = h*h
    taille = (N+1)*(N+1)

    diags = np.zeros((4,taille))

    #Diagonale "-N-2"
    diags[0, 0 : taille - 2*(N+1)] = 1./(4*h2)
    diags[0, np.arange(N-1,taille,N+1)] = 0
    diags[0, np.arange(N,taille,N+1)] = 0

    #Diagonale "-N"
    diags[1, 2 : taille - (2*N+2)] = -1./(4*h2)
    diags[1, np.arange(N+1,taille,N+1)] = 0
    diags[1, np.arange(N+2,taille,N+1)] = 0

    #Diagonale "N"
    diags[2, 2*(N+1) : taille - 2] = -1./(4*h2)
    diags[2, np.arange(2*(N+1)+(N-1),taille,N+1)] = 0
    diags[2, np.arange(2*(N+1)+N,taille,N+1)] = 0

    #Diagonale "N+2"
    diags[3, 2*(N+2) : taille] = 1./(4*h2)
    diags[3, np.arange(2*(N+2)+N-1,taille,N+1)] = 0
    diags[3, np.arange(2*(N+2)+N,taille,N+1)] = 0

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-(N+2),-N,N,(N+2)],taille,taille, format = "cs
r")

    return A
```

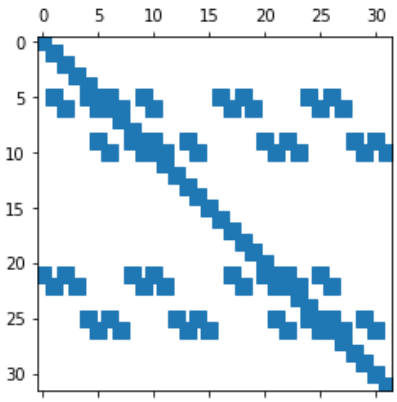
In [16]:
```
def der_sec1(N):
    """Retourne une matrice sparse de taille (N+1)*(N+1)
    correspondant à la discrétisation de la dérivée seconde
    par rapport à la premiere variable  sur l'intégralité du maillage"""

    h = 1./N
    h2 = h*h
    taille = (N+1)*(N+1)

    diags = np.zeros((3,taille))

    #Diagonale principale
    diags[1,:] = 0
    diags[1, N+2:taille - (N+2)] = -2./h2
    diags[1, np.arange(2*N+1, taille, N+1)] = 0
    diags[1, np.arange(2*N+2, taille, N+1)] = 0

    #Diagonale "-1"
    diags[0,N+1:taille-(N+1)] = 1./h2
    diags[0, np.arange(2*N, taille, N+1)] = 0.
    diags[0, np.arange(2*N+1, taille, N+1)] = 0.

    #Diagonale "+1"
    diags[2, N+3:taille-(N+1)] = 1./h2
    diags[2, np.arange(2*N+2, taille, N+1)] = 0.
    diags[2, np.arange(2*N+3, taille, N+1)] = 0.

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-1,0,1],taille,taille, format = "csr")

    return A
```

In [17]:
```
def der_sec2(N):
    """Retourne une matrice sparse de taille (N+1)*(N+1)
    correspondant à la discrétisation de la dérivée seconde
    par rapport à la seconde variable  sur l'intégralité du maillage"""

    h = 1./N
    h2 = h*h
    taille = (N+1)*(N+1)

    diags = np.zeros((3,taille))

    #Diagonale principale
    diags[1,:] = 0.
    diags[1, N+2:taille - (N+2)] = -2./h2
    diags[1, np.arange(2*N+1, taille, N+1)] = 0.
    diags[1, np.arange(2*N+2, taille, N+1)] = 0.

    #Diagonale "-(N+1)"
    diags[0, 1 : taille - (2*N+3)] = 1./h2
    diags[0, np.arange(N,taille,N+1)] = 0.
    diags[0, np.arange(N+1,taille,N+1)] = 0.

    #Diagonale "+(N+1)"
    diags[2, taille - N*N + 2 : taille - 1] = 1./h2
    diags[2, np.arange(taille - N*N + 1 + N ,taille,N+1)] = 0.
    diags[2, np.arange(taille - N*N + 2 + N ,taille,N+1)] = 0.

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-(N+1),0,(N+1)],taille,taille, format = "csr")

    return A
```

In [18]:
```
def matrix_elas(N,mu,lamb):
    """Retourne la matrice sparse globale pour la discrétisation du problème d'élasticité linéaire.
    Cette matrice sera de taille (2*(N+1))^2. Cette fonction prend en paramètre N
    le nbr d'intervalle de discrétisation, mu et lamb des scalaires pour l'élasticité linéaire."""

    delta = (lamb + mu)/mu

    LAP = Laplacien(N)
    CR = matrix_croi(N)
    DER1 = der_sec1(N)
    DER2 = der_sec2(N)

    #Utilisation de la fonction bmat qui permet la construction d'une matrice "par blocs".

    MATRIX = bmat([ [LAP + delta*DER1 , CR],[-CR , LAP + delta*DER2]], format = 'csr')

    return MATRIX
```

In [19]:
```
plt.spy(matrix_elas(3,1,0))
```

Out[19]:
```
<matplotlib.lines.Line2D at 0x7f192d6ec0b8>
```



Maintenant que nous avons tous les éléments à notre disposition, voici la stratégie adoptée pour résoudre notre problème. Nous allons, dans un premier temps, calculer la fonction distance sur la bouche déformée. La matrice obtenue représentera la distance de tous les points du domaine dom_def par rapport à la bouche. Puis, dans un second temps, nous ferons le produit ponctuel entre la bouche de Bryan (dom_init) et la matrice de distance. Les matrices de domaine étant composées de 0 et de 1, le résultat de notre opération sera une matrice composée de 0 sauf au niveau de la bouche de Bryan, où la valeur correspondante sera celle de la distance par rapport à la bouche déformée. Nous serons alors en possession d'une matrice de second membre pour appliquer l'équation d'élasticité linéaire. Cependant, il est important de noter que l'on n'impose pas de forces au sein même de notre domaine, mais plutôt que l'on applique "manuellement" un certain déplacement, c'est pourquoi la matrice de second membre dans notre résolution ne possédra des valeurs que sur les bords. Illustrons maintenant nos propos.

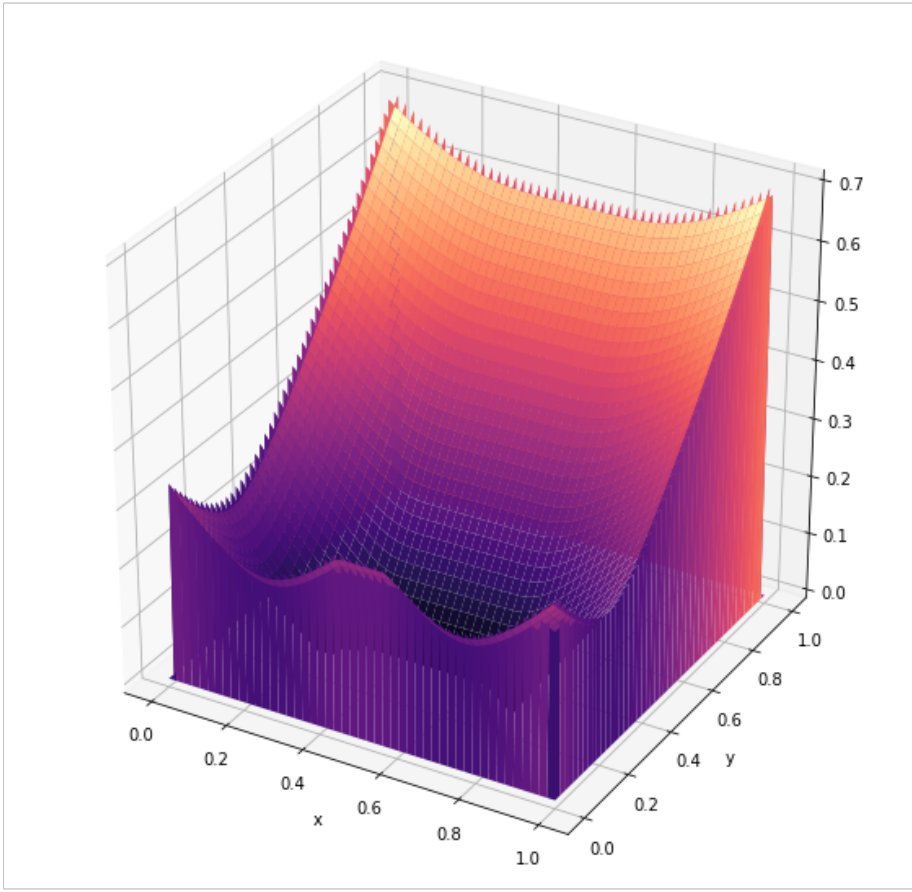
On commence par exprimer les codes pour le calcul de la fonction distance (voir fichier "EQ_CHALEUR.ipynb" pour plus de détails).

In [25]:
```
def matrix_lap(N,dt):
    """Utilisé pour le calcul de la distance par résolution de l'équation de la chaleur"""

    h = 1./N
    h2 = h*h

    #On note les inconnues de 0 à Nx suivant x et 0 à Ny suivant y. La taille du problème est donc (Nx+1)*(Ny+1).

    #Cela correspond à x_i = i*h et y_j = j*h et la numérotation (i,j) --> k := (N+1)*j+i.

    taille = (1+N)*(1+N)

    diags = np.zeros((5,taille))

    #Diagonale principale
    diags[2,:] = 1.
    diags[2, N+2:taille - (N+2)] = 1 + ((4*dt)/h2)
    diags[2, np.arange(2*N+1, taille, N+1)] = 1.
    diags[2, np.arange(2*N+2, taille, N+1)] = 1.

    #Diagonale "-1"
    diags[1,N+1:taille-(N+1)] = -dt/h2
    diags[1, np.arange(2*N, taille, N+1)] = 0.
    diags[1, np.arange(2*N+1, taille, N+1)] = 0.

    #Diagonale "+1"
    diags[3, N+3:taille-(N+1)] = -dt/h2
    diags[3, np.arange(2*N+2, taille, N+1)] = 0.
    diags[3, np.arange(2*N+3, taille, N+1)] = 0.

    #Diagonale "-(N+1)"
    diags[0, 1 : taille - (2*N+3)] = -dt/h2
    diags[0, np.arange(N,taille,N+1)] = 0.
    diags[0, np.arange(N+1,taille,N+1)] = 0.

    #Diagonale "+(N+1)"
    diags[4, taille - N*N + 2 : taille - 1] = -dt/h2
    diags[4, np.arange(taille - N*N + 1 + N ,taille,N+1)] = 0.
    diags[4, np.arange(taille - N*N + 2 + N ,taille,N+1)] = 0.

    #Construction de la matrice creuse
    A = sparse.spdiags(diags,[-(N+1),-1,0,1,(N+1)],taille,taille, format = "csr")

    return A

def chaleurdist(MAT,N,dt,t):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille = (N+1)*(N+1)

    T = np.zeros(taille)          #Initialisation de la solution finale

    for i in range(N+1):
        for j in range(N+1):
            k = i*(N+1) + j
            T[k] = MAT[i][j]

    for i in range(t):
        T = sci.spsolve(matrix_lap(N,dt), T)

    return T
```

Affichons la matrice de la fonction distance.

In [27]:
```
DIST = dist(dom_def(192),192,0.00001)

x = np.linspace(0,1,193)
y = np.linspace(0,1,193)

fig = plt.figure(figsize = [10,10])

ax = fig.add_subplot(111,projection = '3d')
X,Y = np.meshgrid(x,y)
ax.plot_surface(X,Y,DIST.reshape(193,193), cmap = 'magma')

plt.xlabel("x")
plt.ylabel("y")

plt.show()
```

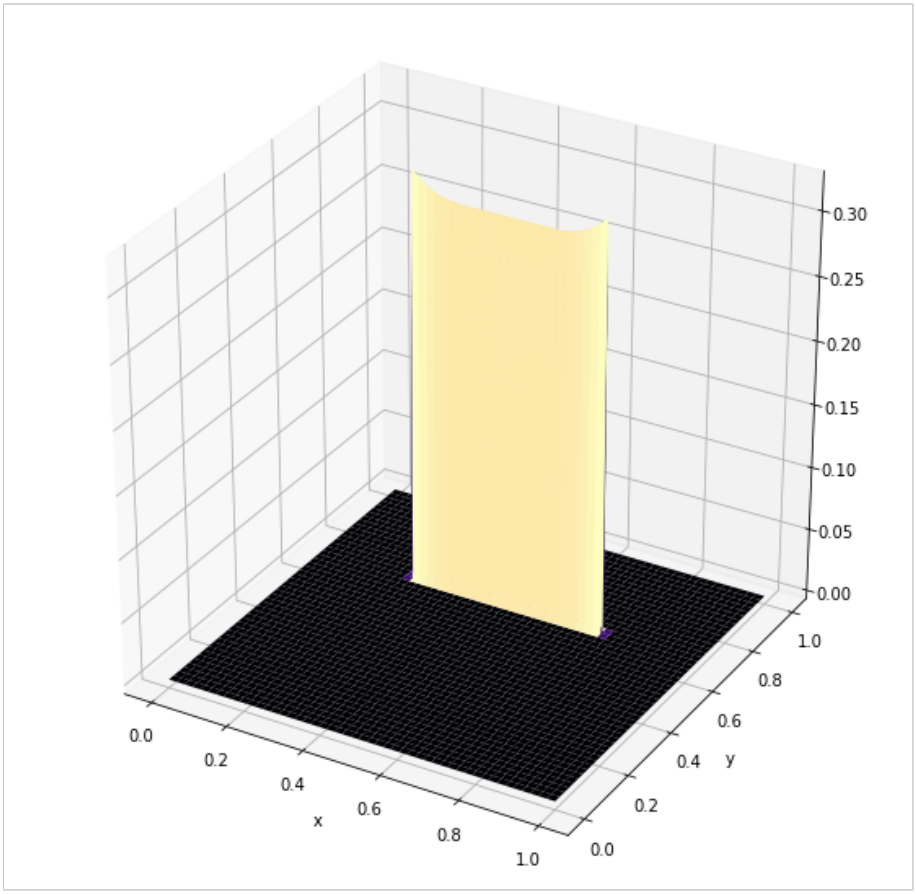
```
/home/matthieu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:72
: RuntimeWarning: divide by zero encountered in log
```



Regardons le résultat lorsque nous multiplions cette matrice par $dom\_ init$.

In [32]:
```
RES = dist(dom_def(192),192,0.00001)*dom_init(192)

x = np.linspace(0,1,193)
y = np.linspace(0,1,193)

fig = plt.figure(figsize = [10,10])

ax = fig.add_subplot(111,projection = '3d')
X,Y = np.meshgrid(x,y)
ax.plot_surface(X,Y,RES.reshape(193,193), cmap = 'magma')

plt.xlabel("x")
plt.ylabel("y")

plt.show()
```

```
/home/matthieu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:72
: RuntimeWarning: divide by zero encountered in log
```



On remarque bien que la matrice obtenue à la valeur 0 partout sauf au niveau de la bouche de Bryan où les valeurs extrêmes sont plus grandes que celles du milieu de la bouche. On a donc bien la valeur d'un déplacement entre la bouche déformée et celle de Bryan. Nous sommes en mesure d'écrire la matrice de second membre de l'équation d'élasticité.

In [34]:
```
def second_membre(N,mu,lamb):

    taille = (N+1)*(N+1)

    DEF = dom_def(N)
    INIT = dom_init(N)

    DIST = dist(dom_def(N),N,0.00001)

    F = DIST*INIT

    M = np.min(F[int(2*N/3),int(N/4):int(3*N/4)])

    F[0,:] = F[int(2*N/3),:]
    F[N,:] = F[int(2*N/3),:]

    for i in range(0,int(N/4)):
        F[0,i] = M
        F[N,i] = M
        F[0,N-i] = M
        F[N,N-i] = M

    F[int(2*N/3),:] = np.zeros(N+1)

    S = np.zeros(2*taille)

    S[taille:2*taille] = np.ravel(F)/mu

    return [F,S]
```

Finalement, nous allons résoudre notre problème d'élasticité linéaire exactement comme dans le code précédent avec cette matrice de distance par rapport à la bouche déformée utilisée en tant que second membre.

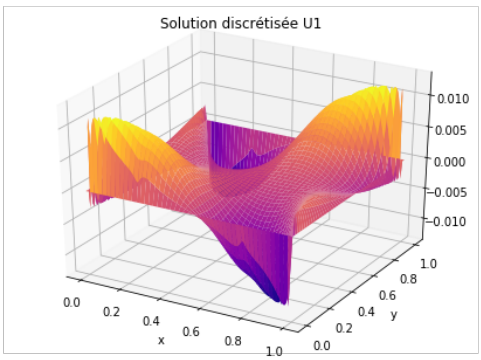
In [36]:
```
def resolution(N,mu,lamb):

    #Pour une valeur de delta égale = 1 il suffit de prendre lambda = 0

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille = (N+1)*(N+1)

    delta = (lamb + mu)/mu

    MAT = matrix_elas(N,mu,lamb)

    [F,S] = second_membre(N,mu,lamb)

    U = sci.spsolve(MAT,S)

    U1 = U[0:taille]
    U2 = U[taille : 2*taille]

    return [U1,U2]
```

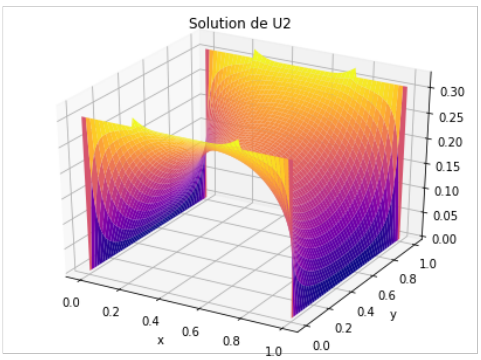
On va afficher les matrices des vecteurs de déplacement obtenues.

In [37]:
```
def graphe_reso(N,mu,lamb):

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)

    taille = (N+1)*(N+1)

    [U1,U2] = resolution(N,mu,lamb)

    fig = plt.figure(figsize = [16,12])

    ax = fig.add_subplot(2,2,1,projection='3d')
    X,Y = np.meshgrid(x,y)
    ax.plot_surface(X,Y, U1.reshape(N+1,N+1), cmap='plasma')
    plt.title("Solution discrétisée U1")
    plt.xlabel("x")
    plt.ylabel("y")

    ax = fig.add_subplot(2,2,2,projection='3d')
    X,Y = np.meshgrid(x,y)
    ax.plot_surface(X,Y, U2.reshape(N+1,N+1), cmap='plasma')
    plt.title("Solution de U2")
    plt.xlabel("x")
    plt.ylabel("y")


    plt.show()
```

In [38]:
```
graphe_reso(192,1,0)
```

```
/home/matthieu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:72
: RuntimeWarning: divide by zero encountered in log
```



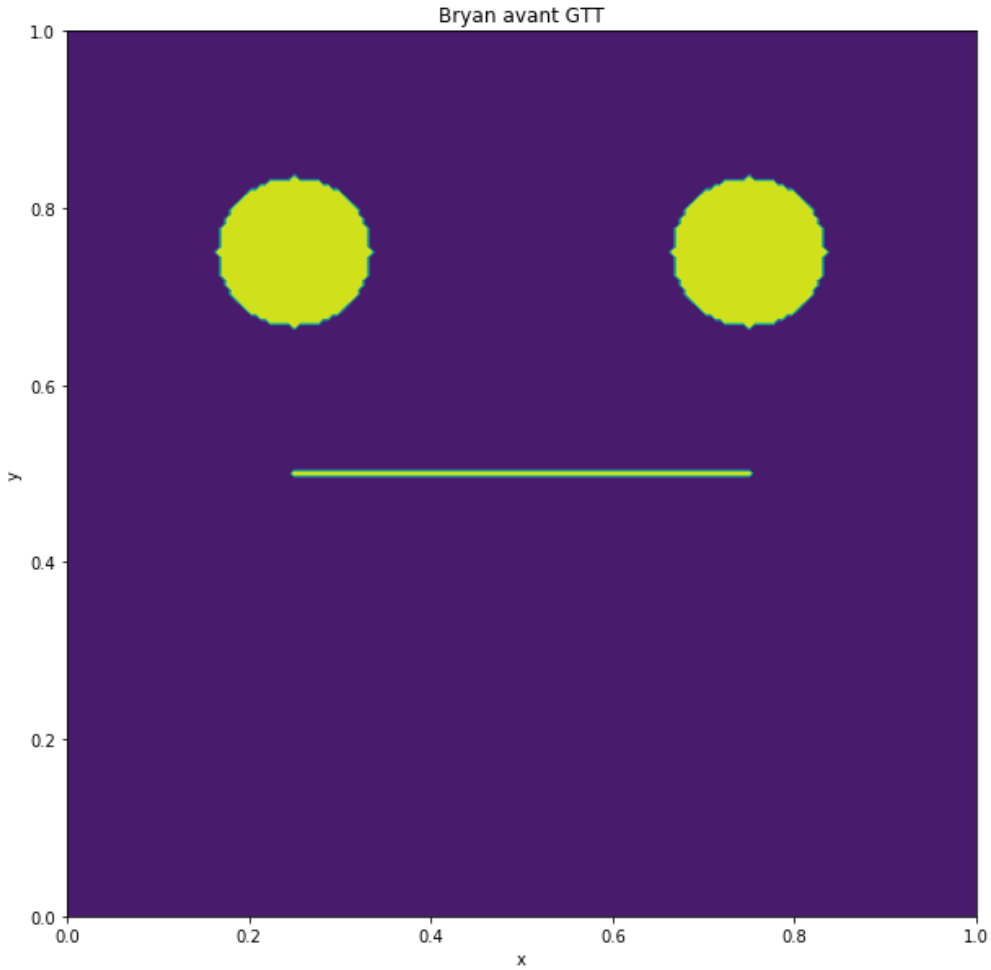


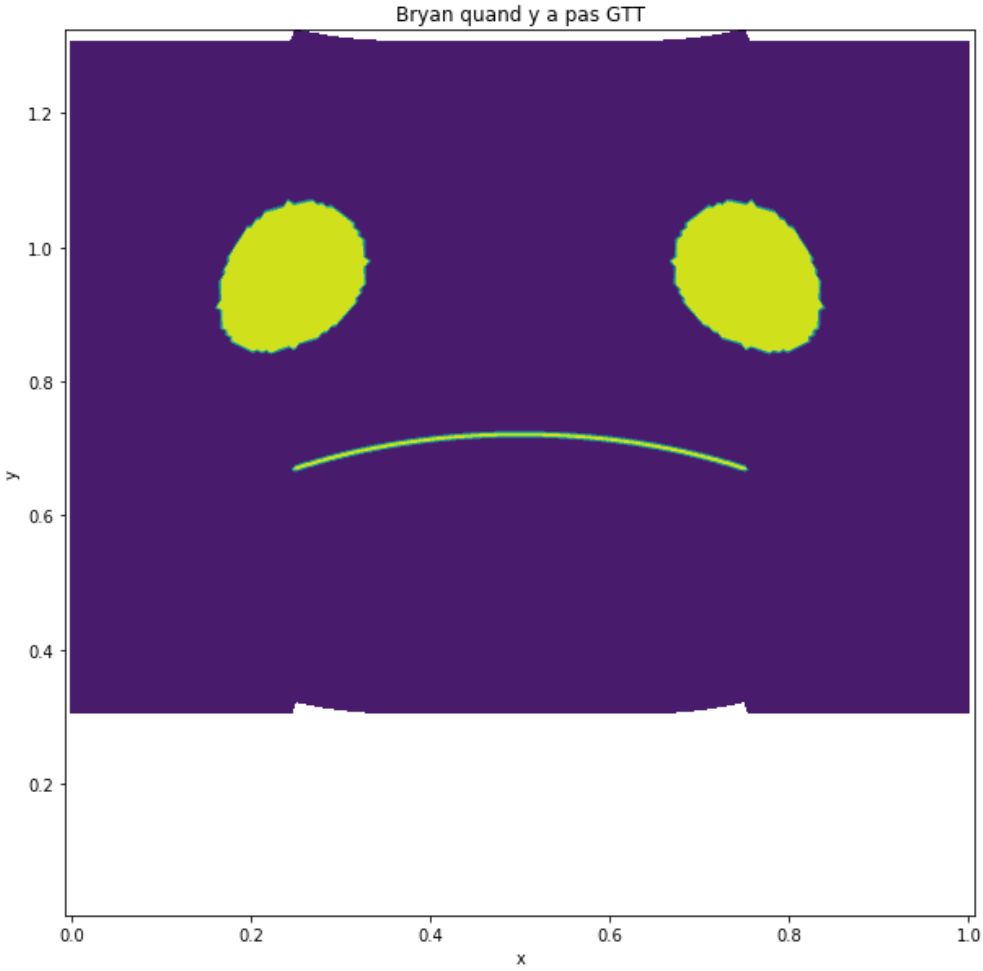
Pour finir, nous allons afficher notre Bryan initial, ensuite nous allons afficher Bryan après deux déformations différentes: la première nous affiche un Bryan triste d'apprendre qu'il n'aura pas de séance de groupe de travail thématique (GTT) cette semaine, puis un Bryan très heureux en sortant d'un cours de GTT. Il est important de noter que l'on applique notre déformation au niveau du maillage en le déformant de plus ou moins U1 et U2.

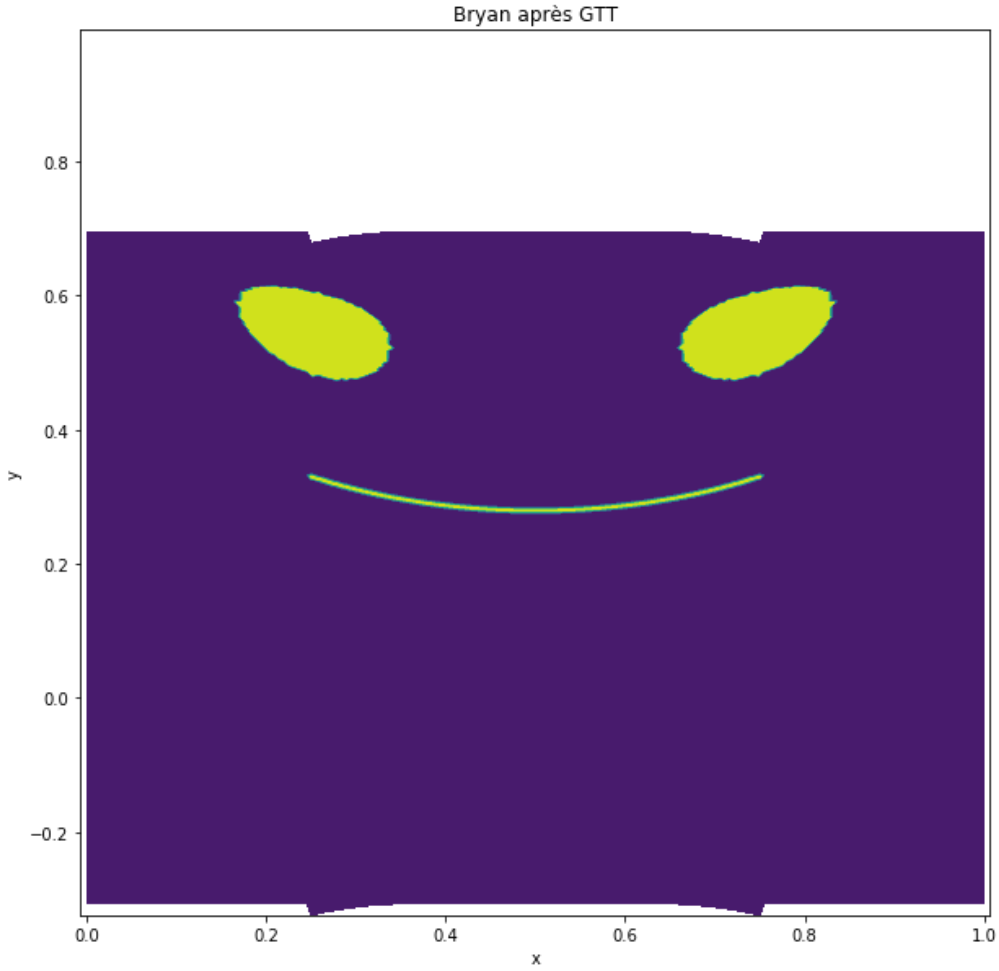
In [39]:
```
def deformation(N,mu,lamb):

    taille = (N+1)*(N+1)

    x = np.linspace(0,1,N+1)
    y = np.linspace(0,1,N+1)


    [U1,U2] = resolution(N,mu,lamb)

    U1 = U1.reshape(N+1,N+1)
    U2 = U2.reshape(N+1,N+1)

    Bryan = BRYAN(N)

    X,Y = np.meshgrid(x,y)

    fig2 = plt.figure(figsize = [10,10])

    ax = fig2.add_subplot(111)
    ax.contourf(X,Y,Bryan)
    plt.title("Bryan avant GTT")
    plt.xlabel("x")
    plt.ylabel("y")

    fig1 = plt.figure(figsize = [10,10])

    ax = fig1.add_subplot(111)

    X1 =  X + U1
    Y1 =  Y + U2

    ax.contourf(X1,Y1,Bryan)
    plt.title("Bryan quand y a pas GTT")
    plt.xlabel("x")
    plt.ylabel("y")


    fig3 = plt.figure(figsize = [10,10])
    ax = fig3.add_subplot(111)

    X2 =  X - U1
    Y2 =  Y - U2

    ax.contourf(X2,Y2,Bryan)
    plt.title("Bryan après GTT")
    plt.xlabel("x")
    plt.ylabel("y")

    plt.show()
```

In [40]:
```
deformation(192,1,0)
```

```
/home/matthieu/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:72
: RuntimeWarning: divide by zero encountered in log
```




Bryan quand y a pas GTT
1.2
1.0
0.8
0.6
0.4
0.2
y
0.0
0.2
0.4
0.6
0.8
1.0
x


Bryan après GTT
0.8
0.6
0.4
0.2
0.0
−0.2
y
0.0
0.2
0.4
0.6
0.8
1.0
x