```
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize # module phyton de minimisation

def ADRS(NX,xcontrol,Target):
  """ La fonction est une de controle qui prend comme parametres:
      NX=le nombre de point, xcontrol= la contrainte à mettre sur le système
      et Target= comme objectif visé (comme entrer).
      """

    #u,t = -V u,x + k u,xx  -lamda u + f

    # PHYSICAL PARAMETERS
    K = 0.1     #Diffusion coefficient
    L = 1.0     #Domain size
    Time = 20.  #Integration time

    V=1
    lamda=1

    # NUMERICAL PARAMETERS
    NT = 10000   #Number of time steps max
    ifre=1000000  #plot every ifre time iterations
    eps=0.001     #relative convergence ratio


    dx = L/(NX-1)                 #Grid step (space)
    dt = dx**2/(V*dx+K+dx**2)   #Grid step (time)  condition CFL de stabilite
10.4.5
    #print(dx,dt)

    ### MAIN PROGRAM ###

    # Initialisation
    x = np.linspace(0.0,1.0,NX)
    T = np.zeros((NX)) #np.sin(2*np.pi*x)
    F = np.zeros((NX))
    rest = []
    RHS = np.zeros((NX))

    for j in range (1,NX-1):
        for ic in range(len(xcontrol)):
            F[j]+=xcontrol[ic]*np.exp(-100*(x[j]-L/(ic+1))**2)# le terme source
sur le quel on impose le controle


    dt = dx**2/(V*dx+2*K+abs(np.max(F))*dx**2)   #Grid step (time)  condition CFL
de stabilite 10.4.5

    plt.figure(1)
```

```
    # Main loop en temps
    #for n in range(0,NT):
    n=0
    res=1
    res0=1
    while(n<NT and res/res0>eps):
        n+=1
    #discretization of the advection/diffusion/reaction/source equation
        res=0
        for j in range (1, NX-1):
            xnu=K+0.5*dx*abs(V)
            Tx=(T[j+1]-T[j-1])/(2*dx)
            Txx=(T[j-1]-2*T[j]+T[j+1])/(dx**2)
            RHS[j] = dt*(-V*Tx+xnu*Txx-lamda*T[j]+F[j])
            res+=abs(RHS[j])

        for j in range (1, NX-1):
            T[j] += RHS[j]
            RHS[j]=0


        if (n == 1 ):
            res0=res

        rest.append(res)
    #Plot every ifre time steps
        if (n%ifre == 0 or (res/res0)<eps):
            #print(n,res)
            plotlabel = "t = %1.2f" %(n * dt)
            plt.plot(x,T, label=plotlabel,color = plt.get_cmap('copper')
(float(n)/NT))


    plt.plot(x,T)
    plt.plot(x,Target)
    plt.show()
    cost=np.dot(T-Target,T-Target)*dx # calcul du cout

    return cost,T
"""La fonction ADRS résout le problème d'advection-diffusion-reaction et
retourne le cout et la solution
     """

#%%

nbc=4
NX=3

#define admissible solution for inverse problem
```

```
# Target=np.zeros(NX)
# xcible=[1,2,3,4]
# cost,Target=ADRS(NX,xcible,Target)
# plt.plot(Target)
# plt.show()
""" cette partie du code itère sur une boucle pour raffiner progressivement
 le maillage  et calculer la valeur optimale qui correspont à la donnée"""
nb_iter_refine=10 # nombre de points d'itération pour le raffinément
cost_tab=np.zeros(nb_iter_refine)
NX_tab=np.zeros(nb_iter_refine) # NX_tab résoit un tableau de taille
nb_iter_refine

for irefine in range(nb_iter_refine):

    NX+=5
    NX_tab[irefine]=NX

    Target=np.zeros(NX) # Target comme un tableau de taille NX
    # boucle qui nous donne la valeur de la donnée pour chaque itération
    for i in range(NX):
        Target[i]=2+np.sin(4*np.pi/(i+1))

    xcontrol=np.zeros(nbc)
    cost,T0=ADRS(NX,xcontrol,Target) # appel de la fonction ADRS avec ces
paramètres definies plus haut

    plt.plot(T0) # figure de T0 qui résoit ADRS
    plt.show() # affichage de la figure

    A=np.zeros((nbc,nbc)) #initialisation d'une matrice carré de taille nbc
    B=np.zeros(nbc) #initialisation d'un vecteur de taille nbc
    """Dans le cadre du calcul de la valeur optimale, on calcul à chaque itération
    la matrice A et le vecteur B en fonction des données(Target) et du controle"""
    for ic in range(nbc):
        xic=np.zeros(nbc)
        xic[ic]=1
        cost,Tic=ADRS(NX,xic,Target)
        B[ic]=np.dot((Target-T0),Tic)/(NX-1) # B[]=le produit de deux vecteurs/NX-
1
        for jc in range(0,ic+1):
            xjc=np.zeros(nbc)
            xjc[jc]=1
            cost,Tjc=ADRS(NX,xjc,Target)
            A[ic,jc]=np.dot(Tic,Tjc)/(NX-1)

    for ic in range(nbc):          # A est une matrice symétrique
        for jc in range(ic,nbc):
            A[ic,jc]=A[jc,ic]

    print("A=",A)
    print("B=",B)
```

```
    xopt=np.linalg.solve(A, B)""" résolution du problème linéaire matricielle
    pour l'obtention de la valeur du controle optimal"""

    print("Xopt=",xopt)
    cost_opt,T=ADRS(NX,xopt,Target)
    print("cost_opt=",cost_opt)
    cost_tab[irefine]=cost_opt

    # plt.figure()
    # plt.plot(Target)
    # plt.plot(T)

plt.plot(NX_tab,cost_tab) #figure de convergence du cout par rapport au nombre de
maillages
plt.show() # affichage de la figure.

#%%
"""Autre manière de résoudre le meme problème avec une méthode plus direct qui
la fonction minimize de la biblio Scipy"""
#Using python optimizer

# def functional(x):
#     nbc=4
#     NX=100
#     Target=np.zeros(NX)
#     xcible=[1,2,3,4]
#     cost,Target=ADRS(NX,xcible,Target)
#     cost,T=ADRS(NX,x,Target)
#     return cost

# #use python minimizer
# nbc=4
# x0=np.zeros((nbc))
# options = { "maxiter": 100, 'xatol': 1e-3, 'disp': True}
# res = minimize(functional, x0, options=options)

# print(res.x)
```