

Computer Methods in Engineering

Exercise 6

In this exercise we will work on using the conjugate gradient method to solve an elliptic equation. More specifically, we will solve the Laplace equation we encountered in Exercise 3 . Recall that Exercise 3 was considering flow in porous media as governed by the Darcy equation:

$$\vec{q} = -\frac{k}{\mu} \nabla p \quad ,$$

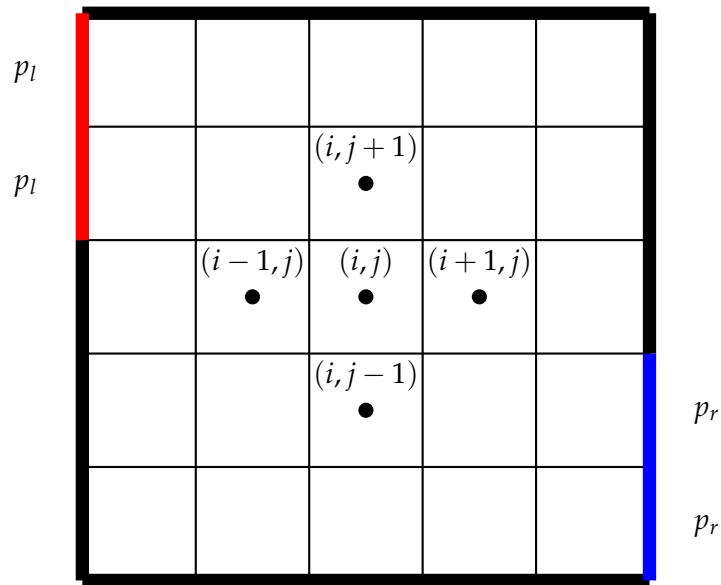
where q is the volumetric flow rate, k is the permeability (a measure for how well the porous medium allows for transport of fluids), μ is the viscosity of the fluid, and p is the fluid pressure. At steady state this gives the Laplace equation $\nabla^2 p = 0$. We considered a two-dimensional model, thus

$$\nabla = \left(\frac{\partial}{\partial x'}, \frac{\partial}{\partial y} \right)$$

Assume a sand-body connecting two fluid reservoirs at different pressure. The left reservoir has a pressure $p_l = 1 \times 10^5$ Pa, while the right reservoir has a pressure $p_r = 2 \times 10^5$ Pa. The sand-body has a shape between the two reservoirs as outlined in Fig. 1, where the grid cell size is $100 \text{ m} \times 100 \text{ m}$. Further, assume a viscosity of 1×10^{-3} Pa s, a permeability of $1 \times 10^{-10} \text{ m}^2$, and assume a sand body thickness of 10 m.

We saw in Exercise 3 that this gave the matrix representation for the pressure field as $A\vec{P} = \vec{b}$, where the A matrix was given as

[illegible]



$$\vec{b} = \begin{bmatrix} -100000 \\ 0 \\ 0 \\ 0 \\ -100000 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -200000 \\ 0 \\ 0 \\ 0 \\ 0 \\ -200000 \end{bmatrix}$$

Write a Pyth

Solution:

```
import numpy as np
import matplotlib.pyplot as plt
from math import *
import time

def steepdesc(A,b,x,maxiter,eps) :
    success = False
```

```
#Iterate over estimates of x
for i in range(0,maxiter):
    #Find search direction:
    r = b-np.dot(A,x)
    res = sqrt(np.dot(r,r))
    fval = 0.5*np.dot(x,np.dot(A,x))-np.dot(b,x)
    #print("Iteration: ",i,"f value: ",fval,"Error: ",res,"Position: ...
        ",x,"Direction: ",r)
    if(res < eps):
        success = True
        break

    #Compute alfa
    alpha = np.dot(r,r)/np.dot(r,np.dot(A,r))

    #update solution
    x = x+alpha*r
print('Steepest decent number of iterations:',i)
return [x,success]

def cg(A,b,x,maxiter,eps) :
    success = False
    r = b-np.dot(A,x)
    d = r
    #Iterate over estimates of x
    for i in range(0,maxiter):
        #Evaluate error:
        rr = b-np.dot(A,x)
        res = sqrt(np.dot(rr,rr))
        #print("Iteration: ",i,"Error: ",res,"Position: ",x,"Direction: ",d)
        if(res < eps):
            success = True
            break
        #Compute alfa
        tmpr=np.dot(r,r) #store temporarily inner-product of r at previous step
        alpha = tmpr/np.dot(r,np.dot(A,d))
        #Update position
        x = x+alpha*d
        #Update r:
        r = r-alpha*np.dot(A,d)
        #Compute beta
        beta = np.dot(r,r)/tmpr
        d = r + beta*d

    return [x,success]

#Grid size
innSide=5
#innCells=(innSide,innSide)

#Boundary conditions
pl=1E5
pr=2E5

A=np.zeros((innSide**2,innSide**2))
b=np.zeros(innSide**2)

#Set up matrix A for all internal cells
```

```
for jj in range(0,innSide):
    for ii in range(0,innSide):
        cellNum=ii+innSide*jj
        if ii>0:
            A[cellNum,cellNum]-=1
            A[cellNum,cellNum-1]+=1
        if ii<innSide-1:
            A[cellNum,cellNum]-=1
            A[cellNum,cellNum+1]+=1
        if jj>0:
            A[cellNum,cellNum]-=1
            A[cellNum,ii+innSide*(jj-1)]+=1
        if jj<innSide-1:
            A[cellNum,cellNum]-=1
            A[cellNum,ii+innSide*(jj+1)]+=1

#Add boundaries to matrix A
#and to vector b
A[0,0]==1
b[0]=-p1
A[5,5]==1
b[5]=-p1
A[19,19]==1
b[19]=-pr
A[24,24]==1
b[24]=-pr

eps=1.0e-08
maxiter = 10000
x0=np.zeros(25)

x,success = steepdesc(A,b,x0,maxiter,eps)
print(x,success)

x0=np.zeros(25)
tic=time.time()
x,success = cg(A,b,x0,maxiter,eps)
toc=time.time()
print('Time of CG: ',toc-tic)
print(x,success)

tic=time.time()
x=np.dot(np.linalg.inv(A),b)
toc=time.time()
print('Time of numpy: ',toc-tic)

print(x)
```

Need 1191 iterations.

Problem 2

Write a Python code to solve for the pressure \vec{P} using the conjugate gradient method. How does your solution compare with the solution you obtained in Exercise 3 . Compare the time used by your own python code for the conjugate gradient method to the numpy library `linalg.inv`.

Solution:

The solutions are exactly the same.

On my computer the CG used 4.8E-4s, while numpy used 3.0E-4s. This is quite comparable.