Electronic Supplementary Material (ESI) for ChemComm.

ATP Synthase:
A Moonlighting Enzyme with Unprecedented Functions

Jean-Nicolas Vigneau, Peyman Fahimi, Maximilian Ebert, Youji Cheng, Connor Tannahill, Paul Muir, Thanh-Tung Nguyen-Dang, Chérif F. Matta

SUPPORTING INFORMATION

Technical Details:

The experimental structures used in this work are:

- *Paracoccus denitrificans* (PDB# 5DN6) [1].
- *Bacillus sp.* (strain PS3) (PDB# 6N2Y) [2].
- *Saccharomyces cerevisiae* (PDB# 6CP6) [3].
- *Yarrowia lipolytica* (PDB# 5FL7) [4].
- *Sus scrofa* (PDB# 6J5I) [5].

The above-mentioned structures were then uploaded on the PDB2PQR and Adaptive Poisson-Boltzmann Solver (APBS) server [6] to calculate their respective molecular electrostatic potentials (MESPs). The summary of the principal parameters in the calculation is:

- pH = 7.0.
- T = 298 K (25 ^{o}C).
- Solvent dielectric constant = 78.5.
- Protein interior dielectric constant = 6.
- Monovalent ion/counterion concentrations were set to 150 mM [7].
- AMBER [8-10] force-field atomic point charges are used to approximate the charge density of the protein.

Preparation and Cleaning of the Protein structures and Calculation of MESPs:

The PDB2PQR prepares the raw structures by adding the missing atoms, hydrogen atoms, and by assigning force-field atomic charges and radii. The programme then generates an output PQR file which is cleaned and includes the force field charges of every atom in the protein. Normally, the PQR is then the input for APBS which then calculates the electrostatic potential in an orthorhombic grid around the protein. However, in this case, and in order to have a consistent orientation of the five different (cleaned/prepared) experimental structures, each of which being oriented in its own arbitrary Cartesian coordinate system, an in-

house **Python utility programme (listed in the Appendix)** was used to reorient them all so that their long axis coincides with the *z*-coordinate.

The alignment algorithm is fed the atomic positions and then uses Principal Component Analysis (PCA) to detect the long axis of the protein. A simple rotation matrix is then applied to reorient the protein along its long *z*-axis. Similarly, the second longest axis of the protein is made to coincide with the *y*-axis.

Every structure was treated/cleaned using the same process before the calculation of the electrostatic potential.

An initial step of atomic charge and radius attribution is done using the PDB2PQR software using the AMBER force-field at neutral pH (7.0). The program then "*debumps*" atoms by displacing those with unreasonably short interatomic distances and adjusts the positions of the hydrogen atoms. Any explicit water molecules and unspecified amino acid residues are removed by PDB2PQR.

The protein's electrostatic potential is then calculated using the APBS server using the linearized Poisson-Boltzmann equation with a Single Debye-Hückel boundary condition. The protein and solvent dielectric constants were taken to be, respectively, 6.0 and 78.54. For every structure, the entire ATP synthase molecule has been solvated in its entirety in water.

For every protein, the molecule is centred within a grid the dimensions of which were set to 170 x 190 x 280 Å (with 193 x 193 x 289 = 10,764,961 points). The "*solvent spheres*" were taken to have a radius of 1.4 Å, while the ionic radii were 2.0 Å for cations and 1.8 Å for the anions, both with a concentration of 0.15 M [7] (as mentioned above). From these parameters, the platform provides an estimate of the bulk ion accessibility.

All other PDB2PQR and APBS default parameters were used [11,12].

References

[1] Morales-Rios E, Montgomery MG, Leslie AG, Walker JE. Structure of ATP synthase from Paracoccus denitrificans determined by X-ray crystallography at 4.0 Å resolution. *Proc. Natl. Acad. Sci. USA* 112, 13231-13236 (2015).

[2] Guo H, Suzuki T, Rubinstein JL. Structure of a bacterial ATP synthase. *Elife* 8, Article # e43128 (2019).

[3] Srivastava AP, Luo M, Zhou W, Symersky J, Bai D, Chambers MG, Faraldo-Gómez JD, Liao M, Mueller DM. High-resolution cryo-EM analysis of the yeast ATP synthase in a lipid membrane. *Science* 360, Article # eaas9699 (2018).

[4] Hahn A, Parey K, Bublitz M, Mills DJ, Zickermann V, Vonck J, Kühlbrandt W, Meier T. Structure of a complete ATP synthase dimer reveals the molecular basis of inner mitochondrial membrane morphology. *Mol. Cell* 63, 445-456 (2016).

[5] Gu J, Zhang L, Zong S, Guo R, Liu T, Yi J, Wang P, Zhuo W, Yang M. Cryo-EM structure of the mammalian ATP synthase tetramer bound with inhibitory protein IF1. *Science* 364, 1068-1075 (2019).

[6] Jurrus E, Engel D, Star K, Monson K, Brandi J, Felberg LE, Brookes DH, Wilson L, Chen J, Liles K, Chun M, Li P, Gohara DW, Dolinsky T, Konecny R, Koes DR, Nielsen JE, Head-Gordon T, Geng W, Krasny R, Wei G-W, Holst MJ, McCammon JA, Baker NA. APBS & PDB2PQR: Software for biomolecular electrostatics and solvation. *https://www.poissonboltzmann.org/* (2021).

[7] Baker NA, Sept D, Joseph S, Holst MJ, McCammon JA. Electrostatics of nanosystems: Application to microtubules and the ribosome. *Proc. Natl. Acad. Sci. USA* 98, 10037-10041 (2001).

[8] Salomon-Ferrer R, Case DA, Walker RC. An overview of the Amber biomolecular simulation package. *WIREs Comput. Mol. Sci.* 3, 198-210 (2013).

[9] Case DA, Cheatham ITE, Darden T, Gohlke H, Luo R, Merz JrKM, Onufriev A, Simmerling C, Wang B, Woods R. The Amber biomolecular simulation programs. *J. Computat. Chem.* 26, 1668-1688 (2005).

[10] Ponder JW , Case DA. Force fields for protein simulations. *Adv. Prot. Chem.* 66, 27-85 (2003).

[11] https://pdb2pqr.readthedocs.io/en/latest/

[12] https://apbs.readthedocs.io/en/latest/

APPENDIX:

Python Utility Program to Analyze ATP Synthase's MESP using PDB2PQR and APBS

The following programme does the following principal functions:

- Fetch PDB file from the Protein database (RCSB).
- Translate protein to center it in the chosen box.
- Run PDB2PQR.
- Use the cleaned Cartesian atomic coordinates to perform a principal component analysis (PCA) to identify the principal three long axes.
- Reorient the protein based on the PCA to align the long axis with a chosen Cartesian axis (here the z-axis).
- Generate the final atomic coordinates.
- Manage/run APBS automatically.
- Read the electrostatic potential calculated from APBS.
- Specify how far and how close to the protein the electrostatic potential is to be calculated.
- Plot two dimensional color-coded slices in the electrostatic potential ("CT-scan").
- Calculate the projection of the protein's electric field on any chosen axis. In this paper we calculate the projection on the *z*-axis.
- Calculate the average molecular electrostatic potential in a chosen plane (or within an angular sector of a plane).
- Calculate average electric field in a chosen plane.

LISTING OF THE PROGRAMME (Python 3.9)

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import os
import sys
import time
import math
import copy



#FUNCTION: FETCH PDB FILE
def pdb_fetch(nam,start_time):
    print('\nDownloading PDB file')
    start_time2=time.time()

    os.system('mkdir '+nam)
    notdone=True
    while(notdone):
        if(os.path.exists(nam)):
            notdone=False
    os.system('curl
http://files.rcsb.org/download/'+nam[:4].upper()+'.pdb -o
'+nam+'/'+nam[:4]+'.pdb')
    notdone=True
    while(notdone):
        if(os.path.exists(nam+'/'+nam[:4]+'.pdb')):
            notdone=False

    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return



#FUNCTION: TRANSLATE PROTEIN
def translate(nam,start_time):
    print('\nTranslating protein')
    start_time2=time.time()

    xa=[]
    ya=[]
    za=[]
    fr=open(nam+'/'+nam[:4]+'.pdb','r')
    l=fr.readlines()
    fr.close()
    os.system('mv '+nam+'/'+nam[:4]+'.pdb '+nam+'/'+nam[:4]+'_ref.pdb')
    for i in range(len(l)):
        if((l[i][:4]=='ATOM')or(l[i][:6]=='HETATM')):
            xa.append(float(l[i][30:38]))
            ya.append(float(l[i][38:46]))
            za.append(float(l[i][46:54]))
    xa=np.array(xa)
    ya=np.array(ya)
```

```
    za=np.array(za)
    xa=xa-np.min(xa)
    ya=ya-np.min(ya)
    za=za-np.min(za)
    ai=0
    fw=open(nam+'/'+nam[:4]+'.pdb','w')
    for i in range(len(l)):
        if((l[i][:4]=='ATOM')or(l[i][:6]=='HETATM')):

fw.write(l[i][:30]+'%8.3f%8.3f%8.3f'%(xa[ai],ya[ai],za[ai])+l[i][54:])
            ai=ai+1
        else:
            fw.write(l[i])
    fw.close()

    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return



#FUNCTION: RUN PDB2PQR
def pdb2pqr(nam,modit,start_time):
    print('\nInitializing PDB2PQR')
    start_time2=time.time()

    if(os.path.exists(nam+'/pdb2pqr.log')):
        os.system('rm '+nam+'/pdb2pqr.log')
    if(modit=='n'):
        pdb_options='--ff=AMBER --keep-chain --assign-only --titration-
state-method=propka'
    else:
        pdb_options='--ff=AMBER --keep-chain --titration-state-
method=propka'
    os.system('stdbuf -oL pdb2pqr30 '+pdb_options+'
'+nam+'/'+nam[:4]+'.pdb '+nam+'/'+nam[:4]+'.pqr 1>> '+nam+'/pdb2pqr.log
2>> '+nam+'/pdb2pqr.log')
    notdone=True
    while(notdone):
        if(os.path.exists(nam+'/'+nam[:4]+'.pqr')):
            notdone=False
        elif((time.time()-
start_time2)>(np.array([3.0e1,1.0e3])[modit!='n'])):
            notdone=False
            sys.exit('\nError with PDB2PQR. Consult '+nam+'/pdb2pqr.log
for further informations.')

    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return



#FUNCTION: REORIENT ACCORDING TO PRINCIPAL COMPONENT ANALYSIS
def pca(nam,start_time):
    print('\nApplying PCA')
    start_time2=time.time()
```

```
    print('Reading atoms')
    xa=[]
    ya=[]
    za=[]
    if(os.path.exists(nam+'/'+nam[:4]+'_ref.pqr')):
        os.system('cp '+nam+'/'+nam[:4]+'_ref.pqr
'+nam+'/'+nam[:4]+'.pqr')
    fr=open(nam+'/'+nam[:4]+'.pqr','r')
    l=fr.readlines()
    fr.close()
    for i in range(len(l)):
        if((l[i][:4]=='ATOM')or(l[i][:6]=='HETATM')):
            xa.append(float(l[i][30:38]))
            ya.append(float(l[i][38:46]))
            za.append(float(l[i][46:54]))
    xa=np.array(xa)
    ya=np.array(ya)
    za=np.array(za)

    print('Centering protein')
    xa=xa-np.average(xa)
    ya=ya-np.average(ya)
    za=za-np.average(za)

    print('Generating covariance matrix')

coordmat=np.array([copy.deepcopy(xa),copy.deepcopy(ya),copy.deepcopy(za
)])

covmat=np.cov(np.array([copy.deepcopy(xa),copy.deepcopy(ya),copy.deepco
py(za)]))

    print('Finding eigenvalues and eigenvectors')
    eigvals,eigvecs=np.linalg.eig(covmat)
    goodord=np.argsort(eigvals)

    print('Orienting protein')
    eigvecs=np.transpose(eigvecs)
    coordmat=np.dot(eigvecs,coordmat)
    xa=copy.deepcopy(coordmat[0])
    ya=copy.deepcopy(coordmat[1])
    za=copy.deepcopy(coordmat[2])

minmax=np.array([np.absolute(np.min(xa)),np.max(xa),np.absolute(np.min(
ya)),np.max(ya),np.absolute(np.min(za)),np.max(za)])
    alp=0.0e0
    bet=0.0e0
    gam=0.0e0
    if((goodord[2]==0)and(minmax[0]>minmax[1])):
        bet=-np.pi/2.0e0
        if((goodord[1]==1)and(minmax[2]>minmax[3])):
            alp=np.pi
        elif((goodord[1]==2)and(minmax[4]>minmax[5])):
            alp=np.pi/2.0e0
        elif((goodord[1]==2)and(minmax[4]<minmax[5])):
            alp=-np.pi/2.0e0
```

```
    if((goodord[2]==0)and(minmax[0]<minmax[1])):
        bet=np.pi/2.0e0
        if((goodord[1]==1)and(minmax[2]>minmax[3])):
            alp=np.pi
        elif((goodord[1]==2)and(minmax[4]>minmax[5])):
            alp=-np.pi/2.0e0
        elif((goodord[1]==2)and(minmax[4]<minmax[5])):
            alp=np.pi/2.0e0
    if((goodord[2]==1)and(minmax[2]>minmax[3])):
        gam=np.pi/2.0e0
        if((goodord[1]==0)and(minmax[0]>minmax[1])):
            alp=-np.pi/2.0e0
        elif((goodord[1]==0)and(minmax[0]<minmax[1])):
            alp=np.pi/2.0e0
        elif((goodord[1]==2)and(minmax[4]<minmax[5])):
            alp=np.pi
    if((goodord[2]==1)and(minmax[2]<minmax[3])):
        gam=-np.pi/2.0e0
        if((goodord[1]==0)and(minmax[0]>minmax[1])):
            alp=-np.pi/2.0e0
        elif((goodord[1]==0)and(minmax[0]<minmax[1])):
            alp=np.pi/2.0e0
        elif((goodord[1]==2)and(minmax[4]>minmax[5])):
            alp=np.pi
    if((goodord[2]==2)and(minmax[4]>minmax[5])):
        if((goodord[1]==0)and(minmax[0]>minmax[1])):
            alp=-np.pi/2.0e0
        elif((goodord[1]==0)and(minmax[0]<minmax[1])):
            alp=np.pi/2.0e0
        elif((goodord[1]==1)and(minmax[2]>minmax[3])):
            alp=np.pi
    if((goodord[2]==2)and(minmax[4]<minmax[5])):
        bet=np.pi
        if((goodord[1]==0)and(minmax[0]>minmax[1])):
            alp=np.pi/2.0e0
        elif((goodord[1]==0)and(minmax[0]<minmax[1])):
            alp=-np.pi/2.0e0
        elif((goodord[1]==1)and(minmax[2]>minmax[3])):
            alp=np.pi
    rotmat=np.array([[np.cos(alp)*np.cos(bet),np.sin(alp)*np.cos(bet),-
np.sin(bet)],[np.cos(alp)*np.sin(bet)*np.sin(gam)-
np.sin(alp)*np.cos(gam),np.sin(alp)*np.sin(bet)*np.sin(gam)+np.cos(alp)
*np.cos(gam),np.cos(bet)*np.sin(gam)],[np.cos(alp)*np.sin(bet)*np.cos(g
am)+np.sin(alp)*np.sin(gam),np.sin(alp)*np.sin(bet)*np.cos(gam)-
np.cos(alp)*np.sin(gam),np.cos(bet)*np.cos(gam)]])
    coordmat=np.dot(np.transpose(rotmat),coordmat)
    xa=copy.deepcopy(coordmat[0])
    ya=copy.deepcopy(coordmat[1])
    za=copy.deepcopy(coordmat[2])

    print('Reorienting PQR file')
    xo=np.min(copy.deepcopy(xa))
    yo=np.min(copy.deepcopy(ya))
    zo=np.min(copy.deepcopy(za))
    xa=xa-copy.deepcopy(xo)
    ya=ya-copy.deepcopy(yo)
    za=za-copy.deepcopy(zo)
```

```
    totprotlen=[]
    totprotlen.append(np.max(xa))
    totprotlen.append(np.max(ya))
    totprotlen.append(np.max(za))
    totprotlen=np.array(totprotlen)
    os.system('mv '+nam+'/'+nam[:4]+'.pqr '+nam+'/'+nam[:4]+'_ref.pqr')
    fw=open(nam+'/'+nam[:4]+'.pqr','w')
    ai=0
    for i in range(len(l)):
        if((l[i][:4]=='ATOM')or(l[i][:6]=='HETATM')):

fw.write(l[i][:30]+'%8.3f%8.3f%8.3f'%(xa[ai],ya[ai],za[ai])+l[i][54:])
            ai=ai+1
        else:
            fw.write(l[i])
    fw.close()

    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return totprotlen



#FUNCTION: DELETE CHAINS
def chain_deletion(nam,start_time):
    print('\nDeleting side chains')
    fr=open(nam+'/'+nam[:4]+'.pqr','r')
    l=fr.readlines()
    fr.close()
    os.system('mv '+nam+'/'+nam[:4]+'.pqr '+nam+'/'+nam[:4]+'_tot.pqr')
    fw1=open(nam+'/'+nam[:4]+'_del.pqr','w')
    fw2=open(nam+'/'+nam[:4]+'.pqr','w')
    ai=0
    for i in range(len(l)):

if(((l[i][:4]=='ATOM')or(l[i][:6]=='HETATM'))and((delchains.find(l[i][2
0:22]))!=(-1))):
            ai=ai+1
            continue
        elif(l[i][:3]=='TER'):
            if(((delchains.find(l[i-1][20:22]))!=(-1))):
                continue
            else:
                fw1.write(l[i])
                fw2.write(l[i])
        elif((l[i][:4]=='ATOM')or(l[i][:6]=='HETATM')):
            fw1.write(l[i])
            fw2.write(l[i][:20]+'  '+l[i][22:])
            ai=ai+1
        else:
            fw1.write(l[i])
            fw2.write(l[i])
    fw1.close()
    fw2.close()

    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
```

```
    return



#FUNCTION: GET ATOMS COORDINATES AND PROTEIN DIMENSIONS
def prot_dim(nam,start_time):
    print('\nGetting protein dimensions')
    start_time2=time.time()

    xa=[]
    ya=[]
    za=[]
    xch=[]
    ych=[]
    zch=[]
    zch_min=[]
    zch_max=[]
    lench=[]
    ch_nam=[]

new_chain=['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o',
'p','q','r','s','t','u','v','w','x','y','z']
    new_chain_it=0
    fr=open(nam+'/'+nam[:4]+'.pqr','r')
    l=fr.readlines()
    fr.close()
    fw=open(nam+'/'+nam[:4]+'_dim.dat','w')
    fw2=open(nam+'/'+nam[:4]+'_apbs.pqr','w')
    for i in range(len(l)):
        if((l[i][:4]=='ATOM')or(l[i][:6]=='HETATM')):
            ch_nam.append(l[i][20:22])
            xa.append(float(l[i][30:38]))
            ya.append(float(l[i][38:46]))
            za.append(float(l[i][46:54]))
            xch.append(float(l[i][30:38]))
            ych.append(float(l[i][38:46]))
            zch.append(float(l[i][46:54]))
            if(new_chain_it<=25):
                fw2.write(l[i][:20]+'
%s'%(new_chain[new_chain_it])+l[i][22:])
            else:
                fw2.write(l[i][:20]+'
%s'%(new_chain[new_chain_it%26].upper())+l[i][22:])
        elif(l[i][:3]=='TER'):
            xch=np.array(xch)
            ych=np.array(ych)
            zch=np.array(zch)
            zch_min.append(np.min(zch))
            zch_max.append(np.max(zch))
            lench.append(len(xch))
            if(new_chain_it<=25):
                fw.write('%s ->
%s%10i%30.15f%30.15f%30.15f%30.15f%30.15f%30.15f\n'%(ch_nam[len(ch_nam)
-
1],new_chain[new_chain_it],len(xch),np.min(xch),np.max(xch),np.min(ych)
,np.max(ych),np.min(zch),np.max(zch)))
            else:
```

```
                fw.write('%s ->
%s%10i%30.15f%30.15f%30.15f%30.15f%30.15f%30.15f\n'%(ch_nam[len(ch_nam)
-
1],new_chain[new_chain_it%26].upper(),len(xch),np.min(xch),np.max(xch),
np.min(ych),np.max(ych),np.min(zch),np.max(zch)))
            xch=[]
            ych=[]
            zch=[]
            new_chain_it=new_chain_it+1
    xa=np.array(xa)
    ya=np.array(ya)
    za=np.array(za)
    zch_min=np.array(zch_min)
    zch_max=np.array(zch_max)
    lench=np.array(lench)
    ch_nam=np.array(ch_nam)
    find_fo=0
    max_fo=np.array([0,0,0])
    for i in range(1,len(lench)):
        if((lench[i-1]>(lench[i]-50))and(lench[i-
1]<(lench[i]+50))and(zch_min[i]<(1.e-1*np.max(za)))):
            find_fo=find_fo+1
        else:
            find_fo=0
        if(find_fo>max_fo[0]):
            max_fo=np.array([find_fo+1,i-find_fo+1,i+1])
    if((max_fo[0]==0)and(max_fo[1]==0)and(max_fo[2]==0)):
        fo_pt=zch_max[np.where(zch_min==np.min(zch_min))[0][0]]
    else:
        fo_pt=np.max(zch_max[max_fo[1]-1:max_fo[2]])
    fw.write('\n-PROTEIN- X len: %30.15f, Y len: %30.15f, Z len:
%30.15f\n-F0 UNIT- Z len: %30.15f, Z0: %30.15f'%(np.max(xa)-
np.min(xa),np.max(ya)-np.min(ya),np.max(za)-np.min(za),fo_pt-
np.min(za),fo_pt))
    fw.close()
    fw2.close()

    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return xa,ya,za,ch_nam,fo_pt



#FUNCTION: RUN APBS
def apbs(nam,totprotlen,start_time):
    print('\nInitializing APBS')
    start_time2=time.time()

    print('Writing APBS input file')
    fw=open(nam+'/'+nam[:4]+'.in','w')
    fw.write('read\n')
    #fw.write('    mol pqr '+nam+'/'+nam[:4]+'_apbs.pqr\n')
    fw.write('    mol pqr '+nam[:4]+'_apbs.pqr\n')
    fw.write('end\n')
    fw.write('elec \n')
    fw.write('    mg-auto\n')
    fw.write('    dime %3d %3d %3d\n'%(193,193,289))
```

```
    fw.write('    cglen %8.3f %8.3f
%8.3f\n'%(1.5e0*1.7e2,1.5e0*1.9e2,1.5e0*2.8e2))#(1.5e0*(totprotlen[0]+4
.0e1),1.5e0*(totprotlen[1]+4.0e1),1.5e0*(totprotlen[2]+4.0e1)))
    fw.write('    fglen %8.3f %8.3f
%8.3f\n'%(1.7e2,1.9e2,2.8e2))#(totprotlen[0]+4.0e1,totprotlen[1]+4.0e1,
totprotlen[2]+4.0e1))
    fw.write('    cgcent %8.3f %8.3f
%8.3f\n'%(totprotlen[0]/2.e0,totprotlen[1]/2.e0,totprotlen[2]/2.e0))
    fw.write('    fgcent %8.3f %8.3f
%8.3f\n'%(totprotlen[0]/2.e0,totprotlen[1]/2.e0,totprotlen[2]/2.e0))
    #fw.write('    cgcent mol 1\n')
    #fw.write('    fgcent mol 1\n')
    fw.write('    mol 1\n')
    fw.write('    lpbe\n')
    fw.write('    bcfl sdh\n')
    fw.write('    pdie 6.0000\n')
    fw.write('    sdie 78.5400\n')
    fw.write('    srfm smol\n')
    fw.write('    chgm spl2\n')
    fw.write('    sdens 10.00\n')
    fw.write('    srad 1.40\n')
    fw.write('    swin 0.30\n')
    fw.write('    temp 298.15\n')
    fw.write('    ion charge 1 conc 0.15 radius 2.0\n')
    fw.write('    ion charge -1 conc 0.15 radius 1.8\n')
    fw.write('    calcenergy total\n')
    fw.write('    calcforce no\n')
    #fw.write('    write pot dx '+nam+'/'+nam[:4]+'\n')
    fw.write('    write pot dx '+nam[:4]+'\n')
    fw.write('end\n')
    fw.write('print elecEnergy 1 end\n')
    fw.write('quit\n')
    fw.close()

    print('Executing APBS')
    if(os.path.exists(nam+'/apbs.log')):
        os.system('rm '+nam+'/apbs.log')
    os.system('apbs '+nam+'/'+nam[:4]+'.in 1>> '+nam+'/apbs.log 2>>
'+nam+'/apbs.log')
    notdone=True
    while(notdone):
        if(os.path.exists(nam+'/'+nam[:4]+'.dx')):
            notdone=False
        elif((time.time()-start_time2)>4.2e2):
            notdone=False
            sys.exit('\nError with APBS. Consult '+nam+'/apbs.log for
further informations.')
    time.sleep(30)
    os.system('mv io.mc '+nam+'/.')

    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return



#FUNCTION: READ APBS POTENTIAL
```

```
def read_pot(nam,start_time):
    print('\nReading potential grid')
    start_time2=time.time()

    fr=open(nam+'/'+nam[:4]+'.dx','r')
    l=fr.readlines()
    fr.close()
    nx=int(l[4][35:39])
    ny=int(l[4][39:43])
    nz=int(l[4][43:47])
    if(l[5][7]=='-'):
        ox=float(l[5][6:20])
        if(l[5][21]=='-'):
            oy=float(l[5][20:34])
            if(l[5][35]=='-'):
                oz=float(l[5][34:48])
            else:
                oz=float(l[5][34:47])
        else:
            oy=float(l[5][20:33])
            if(l[5][34]=='-'):
                oz=float(l[5][33:47])
            else:
                oz=float(l[5][33:46])
    else:
        ox=float(l[5][6:19])
        if(l[5][20]=='-'):
            oy=float(l[5][19:33])
            if(l[5][34]=='-'):
                oz=float(l[5][33:47])
            else:
                oz=float(l[5][33:46])
        else:
            oy=float(l[5][19:32])
            if(l[5][33]=='-'):
                oz=float(l[5][32:46])
            else:
                oz=float(l[5][32:45])
    dx=float(l[6][5:18])
    dy=float(l[7][18:31])
    dz=float(l[8][31:44])
    xp=np.arange(ox,ox+(dx*(nx-5.e-1)),dx)
    yp=np.arange(oy,oy+(dy*(ny-5.e-1)),dy)
    zp=np.arange(oz,oz+(dz*(nz-5.e-1)),dz)
    x2d=np.repeat(xp,ny)
    x2d=np.reshape(x2d,(nx,ny))
    x2d=np.transpose(x2d)
    y2d=np.repeat(yp,nx)
    y2d=np.reshape(y2d,(ny,nx))
    p=[]
    for i in range(int(nx*ny*nz/3)):
        if(l[i+11][0:1]=='-'):
            p.append(float(l[i+11][0:13]))
            if(l[i+11][14:15]=='-'):
                p.append(float(l[i+11][13:27]))
                if(l[i+11][28:29]=='-'):
                    p.append(float(l[i+11][27:41]))
```

```
            else:
                p.append(float(l[i+11][27:40]))
        else:
            p.append(float(l[i+11][13:26]))
            if(l[i+11][27:28]=='-'):
                p.append(float(l[i+11][26:40]))
            else:
                p.append(float(l[i+11][26:39]))
    else:
        p.append(float(l[i+11][0:12]))
        if(l[i+11][13:14]=='-'):
            p.append(float(l[i+11][12:26]))
            if(l[i+11][27:28]=='-'):
                p.append(float(l[i+11][26:40]))
            else:
                p.append(float(l[i+11][26:39]))
        else:
            p.append(float(l[i+11][12:25]))
            if(l[i+11][26:27]=='-'):
                p.append(float(l[i+11][25:39]))
            else:
                p.append(float(l[i+11][25:38]))
if(((nx*ny*nz)%3)==2):
    if(l[int(nx*ny*nz/3)+11][0:1]=='-'):
        p.append(float(l[int(nx*ny*nz/3)+11][0:13]))
        if(l[int(nx*ny*nz/3)+11][14:15]=='-'):
            p.append(float(l[int(nx*ny*nz/3)+11][13:27]))
        else:
            p.append(float(l[int(nx*ny*nz/3)+11][13:26]))
    else:
        p.append(float(l[int(nx*ny*nz/3)+11][0:12]))
        if(l[int(nx*ny*nz/3)+11][13:14]=='-'):
            p.append(float(l[int(nx*ny*nz/3)+11][12:26]))
        else:
            p.append(float(l[int(nx*ny*nz/3)+11][12:25]))
elif(((nx*ny*nz)%3)==1):
    if(l[int(nx*ny*nz/3)+11][0:1]=='-'):
        p.append(float(l[int(nx*ny*nz/3)+11][0:13]))
    else:
        p.append(float(l[int(nx*ny*nz/3)+11][0:12]))
p=np.array(p)
p=2.57e1*p
#pmin=np.min(p)
#pmax=np.max(p)
#plogmax_neg=np.log10(-pmin)
#plogmax_pos=np.log10(pmax)
#plogmin=np.min(np.array([plogmax_pos,plogmax_neg]))-5.0e0
#plog_pos=np.log10(p)
#plog_neg=np.log10(-p)
p=np.reshape(p,(nx,ny,nz))
#plog_pos=np.reshape(plog_pos,(nx,ny,nz))
#plog_neg=np.reshape(plog_neg,(nx,ny,nz))
p=np.transpose(p)
#plog_pos=np.transpose(plog_pos)
#plog_neg=np.transpose(plog_neg)
```

```
    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return nx,ny,nz,xp,yp,zp,p,x2d,y2d



#FUNCTION: KEEP POTENTIAL WHITHIN DISTANCE
def
mask_pot(nam,nx,ny,nz,xa,ya,za,xp,yp,zp,distmin,distmax,start_time):
    print('\nKeeping proper potentials')
    start_time2=time.time()

    if(os.path.exists(nam+'/'+nam[:4]+'_mask.dat')):
        mp=[]
        fr=open(nam+'/'+nam[:4]+'_mask.dat','r')
        l=fr.readlines()
        fr.close()
        for i in range(int(nx*ny*nz/100)):
            for j in range(100):
                mp.append(l[i][j])
        for i in range((nz*ny*nx)%100):
            mp.append(l[int(nx*ny*nz/100)][i])
        mp=np.array(mp,dtype=int).reshape(nz,ny,nx)
        mp=np.array(mp,dtype=bool)
    else:
        mp=''
        for i in range(nz):
            xa_z=xa[(np.absolute(za-zp[i])<=distmax)]
            ya_z=ya[(np.absolute(za-zp[i])<=distmax)]
            za_z=za[(np.absolute(za-zp[i])<=distmax)]
            if(xa_z.size>0):
                for j in range(ny):
                    xa_y=xa_z[(np.sqrt(((za_z-zp[i])**2.e0)+((ya_z-
yp[j])**2.e0))<distmax)]
                    ya_y=ya_z[(np.sqrt(((za_z-zp[i])**2.e0)+((ya_z-
yp[j])**2.e0))<distmax)]
                    za_y=za_z[(np.sqrt(((za_z-zp[i])**2.e0)+((ya_z-
yp[j])**2.e0))<distmax)]
                    if(xa_y.size>0):
                        for k in range(nx):
                            x_comp=xa_y-xp[k]
                            y_comp=ya_y-yp[j]
                            z_comp=za_y-zp[i]

comp=(np.sqrt((x_comp**2.e0)+(y_comp**2.e0)+(z_comp**2.e0)))
                            comp=np.min(comp)
                            if((comp>distmin)and(comp<distmax)):
                                mp=mp+'1'
                            else:
                                mp=mp+'0'
                    else:
                        mp=mp+''.zfill(nx)
            else:
                mp=mp+''.zfill(ny*nx)

if((int(100*(i+1)/nz)!=int(100*(i)/nz))and((int(100*(i+1)/nz)%5)==0)):
                print('%3d%%'%(int(100*(i+1)/nz)))
```

```
        print('Writing data')
        fw=open(nam+'/'+nam[:4]+'_mask.dat','w')
        for i in range(int(nz*ny*nx/100)):
            fw.write(mp[i*100:(i+1)*100]+'\n')

fw.write(mp[int(nz*ny*nx/100)*100:int(nz*ny*nx/100)*100+((nz*ny*nx)%100
)])
        fw.close()
        mp=np.array(list(mp),dtype=int).reshape(nz,ny,nx)
        mp=np.array(mp,dtype=bool)

    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return mp



#FUNCTION: PLOT POTENTIAL SURFACES ALONG Z AXIS (CT-SCAN)
def ct_scan(nam,p_type,xp,yp,zp,p,x2d,y2d,mp,ch_nam,start_time):
    print('\nPlotting entire surfaces')
    start_time2=time.time()

    if(os.path.exists(nam+'/'+p_type+'_ct_scan')):
        os.system('rm '+nam+'/'+p_type+'_ct_scan/*.png')
    else:
        os.system('mkdir '+nam+'/'+p_type+'_ct_scan')
        notdone=True
        while(notdone):
            if(os.path.exists(nam+'/'+p_type+'_ct_scan')):
                notdone=False
    #nlvl=25
    #plogmin=np.log10(np.absolute(pmin))
    #plogmax=np.log10(pmax)
    plogmax=np.log10(np.amax(np.absolute(p)))
    #lvl=np.concatenate((-1.e1**np.arange(plogmin,-2.e1,-
(plogmin+2.e1)/np.float(nlvl-1)),0.e0,1.e1**np.arange(-
2.e1+(plogmax+2.e1)/np.float(nlvl-
1),plogmax+(plogmax+2.e1)/np.float(2*(nlvl-
1)),(plogmax+2.e1)/np.float(nlvl-1))),axis=None)
    #lvl=np.concatenate((-1.e1**np.arange(plogmax,-5.e0,-
(plogmax+5.e0)/np.float(nlvl-1)),0.e0,1.e1**np.arange(-
5.e0+(plogmax+5.e0)/np.float(nlvl-
1),plogmax+(plogmax+5.e0)/np.float(2*(nlvl-
1)),(plogmax+5.e0)/np.float(nlvl-1))),axis=None)
    lvl=np.concatenate((-1.e1**np.arange(math.ceil(plogmax),-5.e0,-
1.e0),0.e0,1.e1**np.arange(-4.e0,math.ceil(plogmax)+5.e-
1,1.e0)),axis=None)
    #lvl=1.e1**np.arange(-np.log10(pmax),np.log10(pmax)+dpmax,dpmax)
    #lvl=plt.MaxNLocator(nbins=100).tick_values(-1000,1000)#(-
pmax,pmax)

#lvl_pos=plt.MaxNLocator(nbins=100).tick_values(plogmin,plogmax_pos)#(-
pmax,pmax)

#lvl_neg=plt.MaxNLocator(nbins=100).tick_values(plogmin,plogmax_neg)#(-
pmax,pmax)
```

```
    #lvl_contour=np.array([-200,-100,-50,-25,-10,-
5,0,5,10,25,50,100,200])#plt.MaxNLocator(nbins=11).tick_values(-
200,200)#(-pmax,pmax)
    lvl_contour=np.array([0])#np.array([-500,-100,-25,-
5,0,5,25,100,500])

#lvl_ccolors=np.array(['darkblue','blue','darkcyan','cyan','darkgreen',
'green','black','yellow','gold','orange','peru','red','darkred'])

#lvl_contourp=np.array([int(plogmin),int(plogmin)+1,int(plogmin)+2,int(
plogmin)+4,int(plogmin)+6])#np.arange(int(plogmin),int(plogmax_pos))#np
.arange(int(-np.sqrt(-plogmin)),int(np.sqrt(plogmax_pos)))

#lvl_contourn=np.array([int(plogmin),int(plogmin)+1,int(plogmin)+2,int(
plogmin)+4,int(plogmin)+6])#np.arange(int(plogmin),int(plogmax_neg))#np
.arange(int(-np.sqrt(-plogmin)),int(np.sqrt(plogmax_neg)))
    #lvl_contourp=lvl_contourp**2.0e0
    #lvl_contourn=lvl_contourn**2.0e0
    for i in range(len(zp)):
        fig=plt.figure()
        if((len(x2d[mp[i]])!=0)and(len(y2d[mp[i]])!=0)):
            if((np.min(x2d[mp[i]])-distmax)>np.amin(xp)):
                xmin=np.min(x2d[mp[i]])-distmax
            else:
                xmin=np.amin(xp)
            if((np.max(x2d[mp[i]])+distmax)<np.amax(xp)):
                xmax=np.max(x2d[mp[i]])+distmax
            else:
                xmax=np.amax(xp)
            if((np.min(y2d[mp[i]])-distmax)>np.amin(yp)):
                ymin=np.min(y2d[mp[i]])-distmax
            else:
                ymin=np.amin(yp)
            if((np.max(y2d[mp[i]])+distmax)<np.amax(yp)):
                ymax=np.max(y2d[mp[i]])+distmax
            else:
                ymax=np.amax(yp)
            plt.xlim(xmin,xmax)
            plt.ylim(ymin,ymax)
        #cf=plt.contourf(xp,yp,p[i],levels=lvl,cmap=seisJN)

cf=plt.contourf(xp,yp,p[i],levels=lvl,cmap='bwr',norm=mpl.colors.SymLog
Norm(linthresh=0.001))
        #cf=plt.contourf(xp,yp,p[i],levels=lvl,cmap='seismic')

#cf_pos=plt.contourf(xp,yp,plog_pos[i],levels=lvl_pos,cmap='Reds')

#cf_neg=plt.contourf(xp,yp,plog_neg[i],levels=lvl_neg,cmap='Blues')

cf_contour=plt.contour(xp,yp,p[i],levels=lvl_contour,colors='black')#lv
l_ccolors)

#cf_contourp=plt.contour(xp,yp,plog_pos[i],levels=lvl_contourp,colors='
black')

#cf_contourn=plt.contour(xp,yp,plog_neg[i],levels=lvl_contourn,colors='
black')
```

```
        cbar=fig.colorbar(cf,format='%.1e')
        #cbarp=fig.colorbar(cf_neg)
        #cbarn=fig.colorbar(cf_pos)
        #cbar.add_lines(cf_contour)
        #plt.clabel(cf_contour, inline=1,fontsize=10)

#plt.clabel(cf_contourp,inline=1,fontsize=10)#,fmt='%1d')#!JN:Label
contours

#plt.clabel(cf_contourn,inline=1,fontsize=10)#,fmt='%1d')#!JN:Label
contours
        dz=np.average(zp[1:]-zp[:len(zp)-1])
        dclose=5.e-1
        xclose=xa[(za>(zp[i]-(dclose*dz)))*(za<(zp[i]+(dclose*dz)))]
        if(len(xclose)!=0):
            yclose=ya[(za>(zp[i]-
(dclose*dz)))*(za<(zp[i]+(dclose*dz)))]
            chclose=ch_nam[(za>(zp[i]-
(dclose*dz)))*(za<(zp[i]+(dclose*dz)))]
            lastchain=0
            for j in range(len(chclose)):
                if((j>0)and(chclose[j]!=chclose[j-1])):
                    plt.scatter(xclose[lastchain:j-
1],yclose[lastchain:j-1],s=2,label=chclose[j-1])
                    lastchain=j
            plt.scatter(xclose[lastchain:len(chclose)-
1],yclose[lastchain:len(chclose)-1],s=1,label=chclose[len(chclose)-1])
            #plt.legend()
        plt.scatter(np.average(xa),np.average(ya),s=1,c='black')
        plt.title('Potential (mV) CT-scan at z=%.3f,
avg=%.3f'%(zp[i],np.average(p[i])))
        plt.xlabel('x (Å)')
        plt.ylabel('y (Å)')
        plt.tight_layout()
        plt.savefig(nam+'/'+p_type+'_ct_scan/%05d.png'%(i+1))
        plt.close()
        if((10*int(10*(i+1)/len(zp)))!=(10*int(10*(i)/len(zp)))):
            print('%3d%%'%(10*int(10*(i+1)/len(zp))))

    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return



#FUNCTION: CALCULATE ELECTRIC FIELD Z COMPONENT
def efield(nam,zp,p,start_time):
    print('\nCalculating electric field z component')
    start_time2=time.time()

    ef=copy.deepcopy(p)
    ef[1:len(ef)-1]=-(p[2:]-p[:len(p)-2])/np.average(zp[2:]-
zp[:len(zp)-2])
    ef[0]=-(p[1]-p[0])/np.average(zp[1]-zp[0])
    ef[len(ef)-1]=-(p[len(ef)-1]-p[len(ef)-2])/np.average(zp[len(zp)-
1]-zp[len(zp)-2])
```

```
    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return ef



#FUNCTION: ANGULAR AVERAGES
def
ang_avg(nam,p_type,p_name,p_symb,xa,ya,x2d,y2d,zp,p,mp,fo_pt,start_time
):
    print('\nCalculating angular averages')
    start_time2=time.time()

    if(os.path.exists(nam+'/'+p_type+'_avg')):
        os.system('rm '+nam+'/'+p_type+'_avg/*.png')
    else:
        os.system('mkdir '+nam+'/'+p_type+'_avg')
    xpc=x2d-np.average(xa)
    ypc=y2d-np.average(xa)
    zavg=[]
    pavg=[]
    stddev=[]
    minpt=[]
    maxpt=[]
    #nang=[36,8]
    nang=[20,10]
    fw=open(nam+'/'+p_type+'_avg/'+nam[:4]+'_avg.dat','w')
    for i in range(len(nang)):
        if(i==0):
            print('Calculating fine angles')
        elif(i==1):
            print('Calculating coarse angles')
        ang=np.arctan(-1.e0*xpc/ypc)
        ang=ang+(np.pi*(ypc-np.absolute(ypc))/(2.e0*ypc))
        ang=ang+(np.pi/float(nang[i]))
        ang=ang+(np.pi*(ang-np.absolute(ang))/ang)
        ang=np.array(float(nang[i])*ang/(2.e0*np.pi),dtype=int)
        zavg_ang=[]
        pavg_ang=[]
        for j in range(nang[i]):
            zavg_ang.append([])
            pavg_ang.append([])
        for j in range(len(zp)):
            pts=copy.deepcopy(p[j][mp[j]])
            if(len(pts)!=0):
                if(i==0):
                    zavg.append(zp[j])
                    pavg.append(np.average(pts))
                    stddev.append(np.std(pts))
                    minpt.append(np.min(pts))
                    maxpt.append(np.max(pts))

fw.write('%30.15e%30.15e%30.15e%30.15e%30.15e\n'%(zp[j],np.average(pts)
,np.std(pts),np.min(pts),np.max(pts)))
                for k in range(nang[i]):
                    mask_ang=mp[j]*((ang-k)==(-1.e0*(ang-k)))
                    pts_ang=copy.deepcopy(p[j][mask_ang])
```

```
                    if(len(pts_ang)!=0):
                        zavg_ang[k].append(zp[j])
                        pavg_ang[k].append(np.average(pts_ang))
            if((10*int(10*(j+1)/len(zp)))!=(10*int(10*(j)/len(zp)))):
                print('%3d%%'%(10*int(10*(j+1)/len(zp))))
        if(i==0):
            zavg=np.array(zavg)
            pavg=np.array(pavg)
            stddev=np.array(stddev)
            minpt=np.array(minpt)
            maxpt=np.array(maxpt)
        zavg_ang=np.array(zavg_ang)
        pavg_ang=np.array(pavg_ang)
        print('Plotting figures')
        if(i==0):
            for j in range(nang[i]):
                if(len(zavg_ang[j])!=0):

plt.plot(zavg_ang[j],pavg_ang[j],label='Average',color='C%01d'%(j%10))
                    plt.title('Average %s at angles
[%.1f°,%.1f°['%(p_name,(float(j)-
0.5)*360.0/float(nang[i]),(float(j)+0.5)*360.0/float(nang[i])))
                    plt.xlabel(r'$z\ (Å)$')
                    plt.ylabel(r'$%s$'%(p_symb))
                    plt.xlim(np.min(zavg_ang[j]),np.max(zavg_ang[j]))
                    plt.legend()
                    plt.grid()
                    plt.tight_layout()

plt.savefig(nam+'/'+p_type+'_avg/'+nam[:4]+'_avg_ang%02d.png'%(j))
                    plt.close()
        elif(i==1):
            for j in range(nang[i]):
                if(len(zavg_ang)!=0):

plt.plot(zavg_ang[j],pavg_ang[j],label=r'$\theta=[%.1f°,%.1f°[$'%((floa
t(j)-
0.5)*360.0/float(nang[i]),(float(j)+0.5)*360.0/float(nang[i])),color='C
%01d'%(j%10))
            plt.title('Average %s for different angles'%(p_name))
            plt.xlabel(r'$z\ (Å)$')
            plt.ylabel(r'$%s$'%(p_symb))
            plt.xlim(np.min(zavg),np.max(zavg))
            plt.legend(fontsize=8,ncol=2)
            plt.grid()
            plt.tight_layout()

plt.savefig(nam+'/'+p_type+'_avg/'+nam[:4]+'_avg_cangs.png')
            plt.close()
    plt.plot(zavg,pavg,label='Average')
    plt.title('Average %s'%(p_name))
    plt.xlabel(r'$z\ (Å)$')
    plt.ylabel(r'$%s$'%(p_symb))
    plt.xlim(np.min(zavg),np.max(zavg))
    plt.legend()
    plt.grid()
    plt.tight_layout()
```

```
    plt.savefig(nam+'/'+p_type+'_avg/'+nam[:4]+'_avg.png')
    plt.close()
    plt.plot(zavg,pavg,label='Average')
    plt.plot(zavg,stddev,label='Standard Deviation')
    plt.title(r'Average %s and standard deviation'%(p_name))
    plt.xlabel(r'$z\ (Å)$')
    plt.ylabel(r'$%s$'%(p_symb))
    plt.xlim(np.min(zavg),np.max(zavg))
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.savefig(nam+'/'+p_type+'_avg/'+nam[:4]+'_avg_stddev.png')
    plt.close()
    plt.plot(zavg,pavg,label='Average')
    plt.plot(zavg,minpt,label='Minimum Point')
    plt.plot(zavg,maxpt,label='Maximum Point')
    plt.title(r'Average %s, minima and maxima'%(p_name))
    plt.xlabel(r'$z\ (Å)$')
    plt.ylabel(r'$%s$'%(p_symb))
    plt.xlim(np.min(zavg),np.max(zavg))
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.savefig(nam+'/'+p_type+'_avg/'+nam[:4]+'_avg_minmax.png')
    plt.close()
    fw.close()

    print('Calculating top-mid-bot differences')
    midnotfound=True
    incr=0
    while(midnotfound):
        incr=incr+1
        if(zavg[incr]>fo_pt):
            avgmid=(pavg[incr]*(fo_pt-zavg[incr-1])+pavg[incr-
1]*(zavg[incr]-fo_pt))/(zavg[incr]-zavg[incr-1])
            midnotfound=False
    notinprot=True
    incr=0
    delta=[]
    topbot=[]
    topmid=[]
    midbot=[]
    while(notinprot):
        if(zavg[incr]<=0):
            delta.append(np.absolute(zavg[incr]))
            topbot.append(np.absolute(pavg[incr]-pavg[len(pavg)-1-
incr]))
            topmid.append(np.absolute(pavg[incr]-avgmid))
            midbot.append(np.absolute(avgmid-pavg[len(pavg)-1-incr]))
            incr=incr+1
        else:
            notinprot=False
    delta=np.array(delta)
    topbot=np.array(topbot)
    topmid=np.array(topmid)
    midbot=np.array(midbot)
    plt.plot(delta,topbot,label='|Top-Bot|')
```

```
    if(nam[len(nam)-2:]!='fo'):
        plt.plot(delta,topmid,label='|Top-Mid|')
        plt.plot(delta,midbot,label='|Mid-Bot|')
    plt.title(r'Average %s differences'%(p_name))
    plt.xlabel(r'$\delta\ (Å)$')
    plt.ylabel(r'$\Delta %s$'%(p_symb))
    plt.xlim(0,np.max(delta))
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.savefig(nam+'/'+p_type+'_avg/'+nam[:4]+'_avg_diff.png')
    plt.close()

    print('Done. Execution time: %f s (%f s)'%(time.time()-
start_time2,time.time()-start_time))
    return zavg,pavg



#MAIN
print('\n**************************************************')
print('**************************************************\n')
print('          Program ATP Synthase Analyser')
print('                   Version 1.0')
print('                      (2021)\n')
print('Wraparound program for PDB2PQR and APBS softwares')
print('      (https://server.poissonboltzmann.org/)')
print('    to treat and analyse ATP synthase proteins')
print('  fetched from RCSB PDB (https://www.rcsb.org/)\n')
print('                     Written by')
print('               Jean-Nicolas Vigneau')
print('        (jean-nicolas.vigneau.1@ulaval.ca)\n')
print('**************************************************')
print('**************************************************\n')

multit=input('Process multiple files? [y/(n)]\n')
if(multit=='y'):
    fr=open('scheduler.dat','r')
    sched=fr.readlines()
    sched=sched[1:]
    sched_dim=len(sched)
    fr.close()
else:
    sched_dim=1

z_pot_avg_all=[]
p_pot_avg_all=[]
z_ef_avg_all=[]
p_ef_avg_all=[]
nam_all=[]
for run_num in range(sched_dim):
    if(multit=='y'):
        sched_dat=np.array(sched[run_num][:len(sched[run_num])-
1].split(','))
        fetchit=sched_dat[0]
        modit=sched_dat[1]
        nam=sched_dat[2]
```

```
        distmin=sched_dat[3]
        distmax=sched_dat[4]
        delit=sched_dat[5]
        if(delit!='n'):
            delchains=sched_dat[6]
        else:
            delchains=' '
    else:
        fetchit=input('Fetch PDB file? [y/(n)]\n')
        modit=input('Apply PDB2PQR modifications? [(y)/n]\n')
        nam=input('File repertory?\n')
        distmin=input('Minimum distance of potential from atoms? [Keep
empty for no minimum, Å]\n')
        distmax=input('Maximum distance of potential from atoms? [Keep
empty for no maximum, Å]\n')
        delit=input('Delete side chains? [y/(n)]\n')
        if(delit!='n'):
            delchains=input('Chains to delete?\n')

    nam_all.append(nam[:4].upper())

    print('\n************')
    print('*** '+nam[:4].upper()+' ***')
    print('************')

    if(distmin==''):
        distmin=0.0e0
    else:
        distmin=float(distmin)
    if(distmax==''):
        distmax=1.0e99
    else:
        distmax=float(distmax)

    start_time=time.time()
    print('\n*** STEP 1: PREPARING PROTEIN ***')

    #FETCH PDB FILE
    if(fetchit=='y'):
        pdb_fetch(nam,start_time)

    #TRANSLATE PROTEIN
    translate(nam,start_time)

    #RUN PDB2PQR
    pdb2pqr(nam,modit,start_time)

    #REORIENT ACCORDING TO PRINCIPAL COMPONENT ANALYSIS
    totprotlen=pca(nam,start_time)

    #DELETE CHAINS
    if(delit=='y'):
        chain_deletion(nam,start_time)

    #GET ATOMS COORDINATES AND PROTEIN DIMENSIONS
    xa,ya,za,ch_nam,fo_pt=prot_dim(nam,start_time)
```

```
    print('\n*** STEP 2: APBS POTENTIAL ***')
    #RUN APBS
    apbs(nam,totprotlen,start_time)

    #READ APBS POTENTIAL
    nx,ny,nz,xp,yp,zp,apbs_p,x2d,y2d=read_pot(nam,start_time)

    #KEEP POTENTIAL WHITHIN DISTANCE

mp=mask_pot(nam,nx,ny,nz,xa,ya,za,xp,yp,zp,distmin,distmax,start_time)

    #PLOT POTENTIAL SURFACES ALONG Z AXIS (CT-SCAN)
    ct_scan(nam,'apbs',xp,yp,zp,apbs_p,x2d,y2d,mp,ch_nam,start_time)

    #CALCULATE ELECTRIC FIELD Z COMPONENT
    apbs_ef=efield(nam,zp,apbs_p,start_time)

    #ANGULAR AVERAGES OF POTENTIAL
    zavg,pavg=ang_avg(nam,'apbs','potential','\Phi\
(mV)',xa,ya,x2d,y2d,zp,apbs_p,mp,fo_pt,start_time)
    z_pot_avg_all.append(zavg)
    p_pot_avg_all.append(pavg)

    #ANGULAR AVERAGES OF ELECTRIC FIELD
    zavg,pavg=ang_avg(nam,'apbs_ef','electric field','E\
(mV/Å)',xa,ya,x2d,y2d,zp,apbs_ef,mp,fo_pt,start_time)
    z_ef_avg_all.append(zavg)
    p_ef_avg_all.append(pavg)

    print('\nAnalysis done.')

zmin=0.E0
zmax=0.E0

#PLOT POTENTIAL AVERAGE OF ALL PROTEINS
for i in range(sched_dim):
    plt.plot(z_pot_avg_all[i],p_pot_avg_all[i],label='%s'%(nam_all[i]))
    if(np.min(z_pot_avg_all[i])<zmin):
        zmin=np.min(z_pot_avg_all[i])
    if(np.max(z_pot_avg_all[i])>zmax):
        zmax=np.max(z_pot_avg_all[i])
plt.xlim(zmin,zmax)
plt.grid()
plt.legend()
plt.title(r'Average potential comparison')
plt.xlabel(r'$z\ (Å)$')
plt.ylabel(r'$\Phi\ (mV)$')
plt.tight_layout()
plt.savefig('comp_pot_avg.png')
plt.close()

#PLOT ELECTRIC FIELD AVERAGE OF ALL PROTEINS
for i in range(sched_dim):
    plt.plot(z_ef_avg_all[i],p_ef_avg_all[i],label='%s'%(nam_all[i]))
    if(np.min(z_ef_avg_all[i])<zmin):
        zmin=np.min(z_ef_avg_all[i])
    if(np.max(z_ef_avg_all[i])>zmax):
```

```
        zmax=np.max(z_ef_avg_all[i])
plt.xlim(zmin,zmax)
plt.grid()
plt.legend()
plt.title(r'Average electric field comparison')
plt.xlabel(r'$z\ (Å)$')
plt.ylabel(r'$E\ (mV/Å)$')
plt.tight_layout()
plt.savefig('comp_ef_avg.png')
plt.close()

print('\n**************************************************\n')
```