Benchmarking Ab Initio Computational Methods for the Quantitative Prediction of Sunlight-Driven Pollutant Degradation in Aquatic Environments

Trerayapiwat K.* and Eustis S.*

Department of Chemistry, Bowdoin College, Brunswick, ME

E-mail: ktreraya@bowdoin.edu; seustis@bowdoin.edu

Abstract

Understanding excitation from ground state to the singlet excited state through simulating absorption spectra of a molecule is essential to predicting the rate of the photoreaction. Excitation energies and oscillator strengths were calculated using different theories and methods. Among all theories, a new approach was selected to model the photon absorption: Molecular DynamicsTime Dependent Density Functional Theory (MD-TDDFT). An aniline molecule equilibrates in the presence of a number of water molecules at room temperature using 6-311++G** basis set. Excitation energy and oscillator strength of aniline geometries in equilibrium are then calculated using TDDFT with w-B97X-D, CAMB3LYP, or M06-2X functional. As a theoretical benchmark, OEMCCSD calculation was carried out with optimized geometry from using using 6-311++G** basis set and implicit water model implemented using Polarizable Continuum Model (PCM). The computed physical properties from MD-TDDFT and OEMCCSD were then compared with data from experimental absorption spectra to evaluate the accuracy of the two methods. Absorption spectras underlying modified Gaussian functions were decomposed and integrated to calculate experimental oscillator strength at a certain excitation energy using an R code written by Peter Cohen. The more accurate method would be applied to triclosan and other water contaminants to predict the rate of their photodegradations in the environment.

Introduction

Environmental Photochemistry and Excited State Energies calculation

Micropollutants are pollutants whose concentrations are individually low, but combined have an effect that is hard to predict. The studies of particular micropollutant specie are hard to conduct because other species interfere in analytical reactions. Triclosan, a micropollutant, has been used as an anti-bacterial agent in household soap and health care products. Under sunlight, Triclosan decomposes to Dioxins and PCBs, well-known carcinogens.[1] Previously, computational studies of Triclosan in the excited states were carried out by Soren N. Eustis[2] and Nathan Ricke.[3] While excited states are important to understanding photochemical reaction, excitation from ground states by photons to exited states is equally important to understand complete reaction mechanism. There are currently no prior studies on how to quantitatively calculate excited state energies of organic molecule in water allowing a systematic approach to develop computational model to be studied. After calculation of excited state energies and the oscillator strength, computational results will be compared with experimental UV-VIS spectrum to evaluate accuracy of the models used.

Solvent Models

Despite recent advent of growth in computer speed and burgeoning interest in incorporating computational models to further understand the nature world, large systems such as solvation models remains a big challenge.[4] In modeling effects of solvent molecules on solute, implicit solvation models were previously implemented because it allows for acceptable results calculation while maintaining good speed (low computational cost). Most famous of all implicit models is Potential Continuum Model (PCM).[5] Instead of explicitly handling each solvent molecules quantum mechanically, PCM expresses their bulk effects on solute molecule in means of dielectric continuum field surrounding molecule of interest. Its downfall is that, however, its accuracy falls short of static and dynamic contribution of excited states properties.[6] Furthermore, implicit solvent model also neglects hydrogen-bonding as it assumes implicit implementation in dispersion forces and electrostatics.[7] Especially in calculating excited state energies, an accurate solvent model should be used.[8] In explicit solvent model, one recent notable method Effective Fragment Potentials (EFP) can be used to model explicit solvents with non-bonded van der Waals interactions, hydrogen bonding using Coulomb interactions, polarization, and exchange repulsion without high computational expense of explicit models.[9,10] This model is chosen to implement explicit solvent in calculating excited states energies.

In modeling organic solute in aquatic environment, the solute, the appropriate number of water molecules to be included as EFP in the model has never been evaluated. Too many water means expensive computational cost. Too few water will not fully model solvating shells around the solute. Binary system will be used to model how many water molecule is needed to fully solvate the solute molecules: 2, 4, 8, 16... Once excited energies for each system is calculated, the results will be compared with experimental value to evaluate how many water is needed before determining on which functional out of three choices should be chosen to achieve the most accurate computational model.

Computational Models: Theories, Basis Sets, and Functionals

Among all current theories, Time-Dependent Density Functional Theory (TDDFT) is the most promising with its high accuracy when used with appropriate functionals and low computational cost.[11] Implementing EFP solvent model, TDDFT can be used to accurately calculate excited state energy of acetone in water.[10] Typically in Implicit solvent model, geometry optimization of solute molecule is carried out with PCM, followed by calculation of excited state energies, also with PCM. This static ground state molecule however does not accurately represent solute in water.[12] Instead, Molecular Dynamics (MD) of solute and solvent fragments can be used to obtain a range of equilibrated structures for excited state energies calculation. Mark Gordon averaged the calculated energies of each excited state to arrive at a final excited states energy.[12]

According to previous basis set studies, wile having roughly the same computational cost, an average-sized basis set 6-311++(2d,p) performs better than aug-cc-pVDZ (ACCD).[13,14] For example, transition energies calculated of CN molecule as calculated by ACCD deviates 1117-1669 cm^{-1} from experimental value while those by 6-311++(2d,p) only deviate 220-470 cm^{-1}. Hoping to most accurately calculate the excited energies, 6-311++(2d,p) basis set is chosen to run TDDFT after MD run. In running MD, a smaller basis set 6-31+(2d,p) will be used in order to cut computational cost. The decision comes after weeks of waiting for computational results when determining the number of water molecule in the model. Two best-performing DFT functionals out of all examined in previous study are explored: CAMB3LYP, M06-2X.[14] PBE0 will also be used.

Method

Molecules - aniline - then para-methoxy m-methoxyacetophenone... Triclosan

discuss # of water

Preliminary Results and Discussions

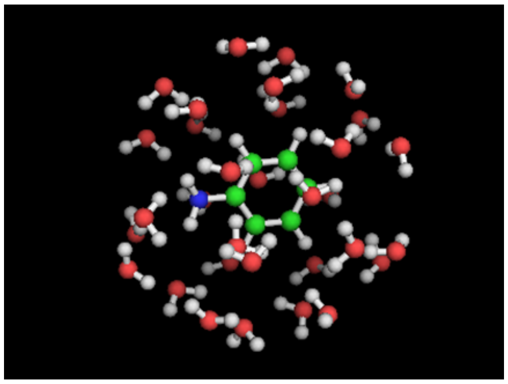
Determining the number of water

TDDFT calculation for aniline with 32, 64, 128, 256, 512 surrounding water molecules were performed with CAMB3LYP basis set. Firstly, for 32 water molecules, the equilibrium were chosen to start from 15 ps and the stopping point of calculation was 25 ps; 1000 jobs for every 10 fs. Determination of equilibrium was determined by eyeballing a plot of the solvent solute system's potential energy over time for a stable period as shown in the figure 1d. The consistently low fluctuation indicates the start of equilibrium at 15000 fs. 1000 frames or 10000 fs of MD geometries were used to calculate the excitation energies in TDDFT run. Geometry of the system though challenge the accuracy of 32-water model. Aniline molecule surrounded in 32 water molecules is unfortunately most stable not being fully solvated. Aniline can be seen outside of the water cluster at the time of equilibrium. This is in contrast to expected 32 water as the first solvation shell for aniline.[15]

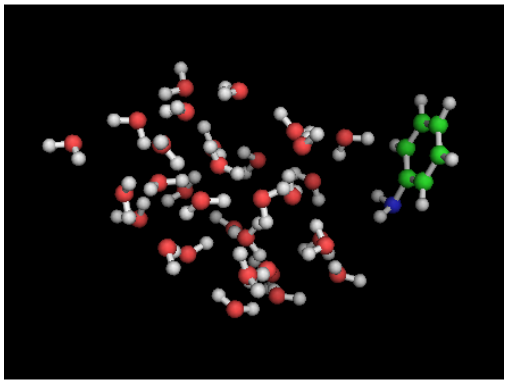
Table 1: Wavelength and Oscillator Strength from MD-TDDFT calculation of aniline in 32 water molecules.

Wavelength (nm)	Oscillator Strength
173.00	0.165685
180.20	0.364739
184.99	0.339029
214.30	0.143915
246.26	0.0383513

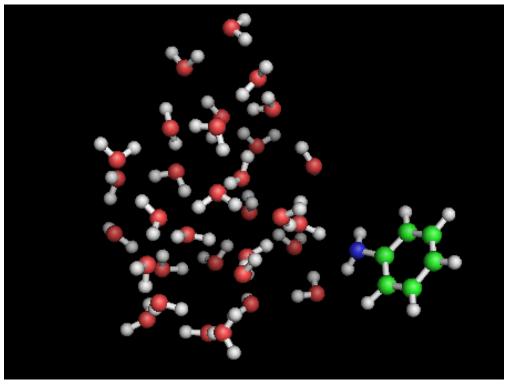
The excited state energies and its oscillator strength are tabulated in table 1. When compared with aniline's UVVIS spectra, as in figure 2, there are several problems. Firstly, the calculated value at 246 nm does not appropriately capture the peak at 230 nm and there is no calculated excitation energy at 280 nm, where the experimental peak is. The problem is probably due to aniline not being fully solvated.



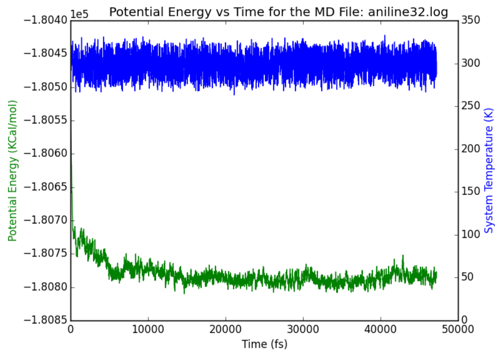
(a)



(b)



(c)



(d)

Figure 1: Molecular Dynamics run of aniline in 32 explicit solvating water molecules. Notice that at equilibrium, aniline molecule comes outside of the water sphere. Albeit hydrogen bond being clearly established, lack of total submersion in water means 32-water does not fully solvate the aniline molecule and suggests that 64-water will give more accurate results. (a) starting geometry of MD run created by packmol. (b) geometry after 15000 fs. Notice the hydrogen bond between the amino group and water cluster. (c) geometry after 25000 fs. The amino group is pointing in the water sphere, as it continues to through out the whole MD run. (d) A plot of potential Energy of the system vs time. At 15000 fs, equilibrium starts as evident by decrease in energy fluctuation.

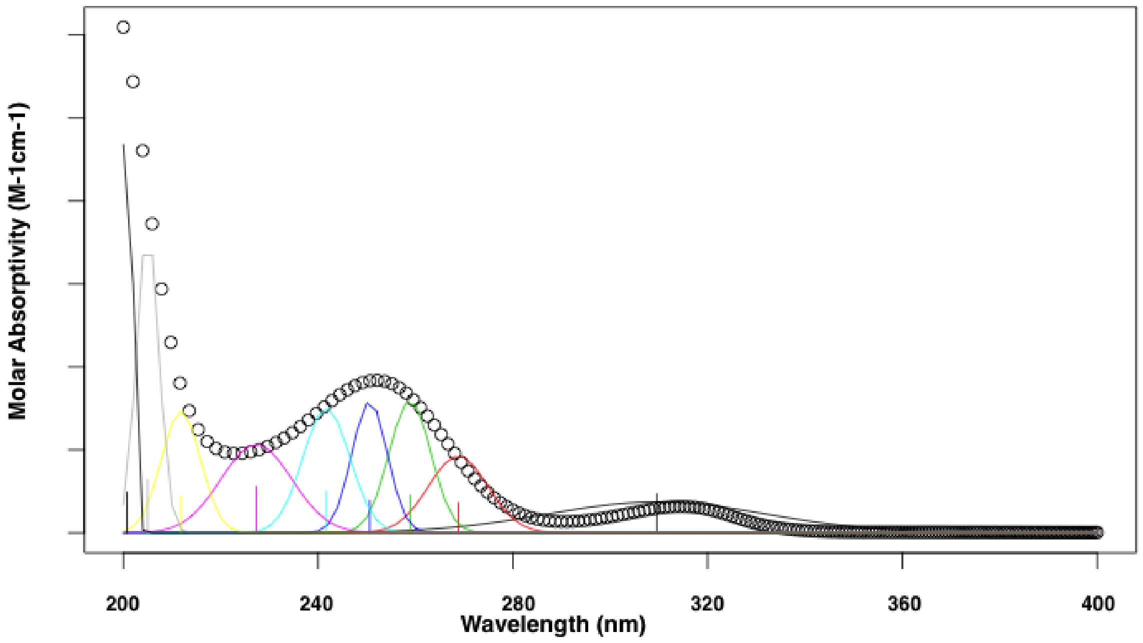


Figure 2: Experimental UVVIS spectra. Gaussian plots are fitted under the curve to find oscillator strength underlying the curve. Note here that data starts from 200 nm to 400 nm. Wavelength oscillator strength of underlying gaussians are reported in table.

Table 2: Wavelength and Oscillator Strength calculated from experimental UV-VIS spectrum using Bayesian probability (see appendix for R code).

Wavelength (nm)	Oscillator Strength
241.53	1808.0
234.52	2251.3
228.95	1939.4
223.28	2506.5
214.67	2795.1
206.10	2208.4
202.53	3202.8
200.39	2447.5

Appendix

Python Scripts

In order to automatically generate input files and cultivate output data from output files, many python scripts are written from scratch. Since scripts are specific to each GAMESS run, there is a limited number of scripts available on the internet (virtually none for this project). Log files obtained from GAMESS contains both valuable experimental data and useless text strings. Python scripts play an important role in both data collection and smoothing up the process between each computational steps. For example, even though WEBMO can generate sets of latest geometry in MD run, but retrieving geometry from each MD step requires one to manually open the log file and copy-paste the geometry into input files of the next step one by one. The python script postMDDataPull2.py is designed to pull thousands of geometries and generate GAMESS input files for TDDFT energy calculation within seconds. Generating these python scripts will also allow unified program to be developed in order to automate the whole project without any manual input.

Preparing MD Input Files

This script does two things. First (line 35-84), it calculates appropriate radius for solvent boundary potential. Some time and effort were spent on figuring out what the radius should be without emperically guess it. A simple model is proposed: At most solute will rotate around its outmost solute atom. This radius, in the code, is called solute radius. The other radius is solvent radius, its the distance between the outmost solvent atom to the solute's CG. These two radius plus an extra 2-3 Angstrom gives ssbp radius for MD input file. Second (line 87-155), the script parses xyz file's geometry data into MD input file. Slight format change is required for GAMESS input files, so this python code automate that change. The output file is MD file which can be run on GAMESS. Output of this script can be seen below in MD Input File section.

```
############################################################
###Create inp for MD run from xyz file from packmol      ###
############################################################

import sys
import csv
import os
import string


#for asking what the input in terminal should be
try:
        if str(sys.argv[1])=='?':
                print '\nCall function as: prepareMD.py input.xyz
                 ↪  numberOfSoluteAtoms numberofSolventAtoms
                 ↪  numberOfSolventMolecules \n'
                sys.exit()
except IndexError:
    print '\n!!!Input command Error. Call function as: prepareMD.py input.xyz
     ↪  numberOfSoluteAtoms numberofSolventAtoms numberOfSolventMolecules \n'
    sys.exit()
#for assigning received input from terminal
try:
    input=str(sys.argv[1])
    numberofSoluteAtoms=int(sys.argv[2])
    numberofSoventAtoms=int(sys.argv[3])
    numberOfSolventMolecules=int(sys.argv[4])
except IndexError:
    print '\n!!!Input command Error. Call function as: prepareMD.py input.xyz
     ↪  numberOfSoluteAtoms numberofSolventAtoms numberOfSolventMolecules \n'
    sys.exit()
#generate output name
if input.endswith('.xyz'):
    output = input[:-4]+'.inp'
#for safety - at worst the output will not overwrite the input
else:
    output=input+'.inp'

#part one
#This part is for finding ssbp radius for inout file
#enumerate gets data in line - line and line index - n
radiusInSolute=0.0
radiusInSolvent=0.0
avgX=0.0
avgY=0.0
avgZ=0.0
X=[]
```

```
Y=[]
Z=[]

lineNumber=0
#open input
f2=open(input)
for line in f2:
        lineNumber+=1
        #first two line does not contain useful info - x y z start on the third
         ↪  line
        if lineNumber>2:
                #x y z
                lineSplit=line.split()
                X.append(float(lineSplit[1]))
                Y.append(float(lineSplit[2]))
                Z.append(float(lineSplit[3]))
#for looping through array below
size=len(X)
#find a CG for solute atoms
avgX=sum(X[:numberofSoluteAtoms-1])/numberofSoluteAtoms
avgY=sum(Y[:numberofSoluteAtoms-1])/numberofSoluteAtoms
avgZ=sum(Z[:numberofSoluteAtoms-1])/numberofSoluteAtoms
#looping to find radius of each atoms in relative to solute's CG
#also find the maximum value of them
for i in range(0,size):
                d=((X[i]-avgX)**2+(Y[i]-avgY)**2+(Z[i]-avgZ)**2)**0.5
                if i<numberofSoluteAtoms:
                        if radiusInSolute<d:
                                radiusInSolute=d
                else:
                        if radiusInSolvent<d:
                                radiusInSolvent=d

#radius should be a little bit larger than the two combined - 3 Angstrom larger -
 ↪  this does not need to be super accurate
radiusInSolute=radiusInSolute
radiusInSolvent=radiusInSolvent
ssbpRadius=radiusInSolute+radiusInSolvent+3
print '\n'
print 'Radius in solute is:\t'+str(radiusInSolute)
print 'Radius in solvent is:\t'+str(radiusInSolvent)
print 'ssbp Radius should be:\t'+str(ssbpRadius)
print '\n'
###############################

#Part two - this is where geometry data is taken from xyz, change into GAMESS
 ↪  input's format + other input
```

```
numberOfAllSolventsAtoms=numberofSoventAtoms*numberOfSolventMolecules
fragmentNumber=1;
atomLabel=1

#this dict is for generating atomic number from Acronym
atomicNumber={’LV’: 116.0, ’BE’: 4.0, ’FR’: 87.0, ’BA’: 56.0, ’BH’: 107.0, ’BI’:
 ↪  83.0, ’BK’: 97.0, ’EU’: 63.0, ’FE’: 26.0, ’BR’: 35.0, ’ES’: 99.0, ’FL’:
 ↪  114.0, ’FM’: 100.0, ’RG’: 111.0, ’RU’: 44.0, ’NO’: 102.0, ’NA’: 11.0, ’NB’:
 ↪  41.0, ’ND’: 60.0, ’NE’: 10.0, ’RE’: 75.0, ’RF’: 104.0, ’LU’: 71.0, ’RA’:
 ↪  88.0, ’RB’: 37.0, ’NP’: 93.0, ’RN’: 86.0, ’RH’: 45.0, ’B’: 5.0, ’CO’: 27.0,
 ↪  ’TH’: 90.0, ’CM’: 96.0, ’CL’: 17.0, ’H’: 1.0, ’CA’: 20.0, ’CF’: 98.0, ’CE’:
 ↪  58.0, ’N’: 7.0, ’CN’: 112.0, ’P’: 15.0, ’GE’: 32.0, ’GD’: 64.0, ’GA’: 31.0,
 ↪  ’V’: 23.0, ’CS’: 55.0, ’CR’: 24.0, ’DS’: 110.0, ’CU’: 29.0, ’SR’: 38.0,
 ↪  ’UUP’: 115.0, ’UUS’: 117.0, ’TC’: 43.0, ’KR’: 36.0, ’SI’: 14.0, ’SN’: 50.0,
 ↪  ’SM’: 62.0, ’UUT’: 113.0, ’SC’: 21.0, ’SB’: 51.0, ’TA’: 73.0, ’OS’: 76.0,
 ↪  ’PU’: 94.0, ’SE’: 34.0, ’AC’: 89.0, ’HS’: 108.0, ’YB’: 70.0, ’DB’: 105.0,
 ↪  ’C’: 6.0, ’HO’: 67.0, ’DY’: 66.0, ’HF’: 72.0, ’HG’: 80.0, ’HE’: 2.0, ’PR’:
 ↪  59.0, ’PT’: 78.0, ’LA’: 57.0, ’F’: 9.0, ’UUO’: 118.0, ’LI’: 3.0, ’PB’: 82.0,
 ↪  ’TL’: 81.0, ’TM’: 69.0, ’LR’: 103.0, ’PD’: 46.0, ’TI’: 22.0, ’TE’: 52.0,
 ↪  ’TB’: 65.0, ’PO’: 84.0, ’PM’: 61.0, ’ZN’: 30.0, ’AG’: 47.0, ’NI’: 28.0, ’I’:
 ↪  53.0, ’K’: 19.0, ’IR’: 77.0, ’AM’: 95.0, ’AL’: 13.0, ’O’: 8.0, ’S’: 16.0,
 ↪  ’AR’: 18.0, ’AU’: 79.0, ’AT’: 85.0, ’W’: 74.0, ’IN’: 49.0, ’Y’: 39.0, ’CD’:
 ↪  48.0, ’ZR’: 40.0, ’ER’: 68.0, ’MD’: 101.0, ’MG’: 12.0, ’PA’: 91.0, ’SG’:
 ↪  106.0, ’MO’: 42.0, ’MN’: 25.0, ’AS’: 33.0, ’MT’: 109.0, ’U’: 92.0, ’XE’:
 ↪  54.0}

#write out put the headers - all the commands for GAMESS + ssbp
#functional = M06-2X - DFTTYP=M06-2X
f = open(output, ’w’);
f.write(’’’ $CONTRL SCFTYP=RHF RUNTYP=MD COORD=UNIQUE
    DFTTYP=M06-2X MAXIT=200 ICHARG=0 MULT=1 $END
 $MD KEVERY=10 PROD=.T. NVTNH=2 MBT=.T. MBR=.T.
    BATHT=298 RSTEMP=.T. DTEMP=25 NSTEPS=50000
    SSBP=.T. SFORCE=1.0 DROFF=’’’+str(ssbpRadius)+’’’ $END
 $DFT DC=.F. $END
 $SYSTEM MWORDS=1000 MEMDDI=1000 $END
 $SCF DIRSCF=.T. $END
 $BASIS GBASIS=N31 NGAUSS=6 NDFUNC=2 NPFUNC=1
    DIFFS=.TRUE. POLAR=POPN311 $END
 $DATA\n’’’+ ’MD INPUT for’ +input+’\nC1 1\n’’’)

#geometry
with open(input) as f1:
    #read by line
    #readlines if okay to use bc xyz is not too big
    lines = f1.readlines()
    #enumerate gets data in line - line and line index - n
    for n, line in enumerate(lines):
        #take all solute molecules (in range of 2 (line 3 where packmol starts)
         ↪  to num+2)
        #it's num+2 bc the range will go to num+1
        if n == 2:
            print 'Now Writing Solute:\n'
        if n in range(2,numberofSoluteAtoms+2):
            lineSplit=line.split();
            lineSplit.insert(1,str(atomicNumber[lineSplit[0]]))
            #convert coordinates to 10 decimals (add zeros if need be)
            for index in [2,3,4]:
                lineSplit[index]=float(lineSplit[index])
                lineSplit[index]=format(lineSplit[index],'.10f')
                grandString=lineSplit[0]+'\t'+lineSplit[1]+'\t'+lineSplit[2] +
                 ↪  '\t'+lineSplit[3]+'\t'+lineSplit[4]+'\n';
            f.write(grandString)
            print grandString
        if n == numberofSoluteAtoms+2:
            f.write(' $END\n\n $EFRAG\nCOORD=CART  POSITION=OPTIMIZE\n')
            print 'Now Writing Solvent:\n'
        #now start doing solvent - (need to add fragment number and atom labels)
        startPointOfSolvent=numberofSoluteAtoms+2
        if n in range(startPointOfSolvent,
         ↪  startPointOfSolvent+numberOfAllSolventsAtoms+1):
            #atomlabel = O1, H2, H3 from O, H, H
            if atomLabel%numberofSoventAtoms==1:
                grandString='FRAGNAME=H2ODFT ! '+str(fragmentNumber)+'\n'
                f.write(grandString)
                print grandString
                fragmentNumber+=1;
                atomLabel%=numberofSoventAtoms
            lineSplit=line.split();
            lineSplit.insert(1,str(atomLabel))
            atomLabel+=1
            #convert coordinates to 10 decimals (add zeros if need be)
            for index in [2,3,4]:
                lineSplit[index]=float(lineSplit[index])
                lineSplit[index]=format(lineSplit[index],'.10f')
            grandString=' '+lineSplit[0]+lineSplit[1]+'\t'+lineSplit[2] +
             ↪  '\t'+lineSplit[3]+'\t'+lineSplit[4]+'\n';
            f.write(grandString)
            print grandString
    #close the inp with £END
    f.write(' $END\n')
################################################################
```

MD Geometries extraction

One of the reasons, an MD run might fail is if solute molecule is pushed out of the water sphere. 3dExtract4.py allows geometries to be extracted into a xyz-movie file. xyz files, capable of containing more than one frame of geometries, allows one to follow MD through a combination of screenshot (each frame is 10 femtosecond - in the current MD input file - see MD Input File section).

```
##########################################################
### 3dExtract pulls out geometries from MD run and make ###
### an xyz-movie file for inspection MD        progress             ###
##########################################################

import os as os
import sys

#for asking what the input in terminal should be
try:
        if str(sys.argv[1])=='?':
                print '\nCall function as: 3dExtract.py input.log
                 ↪  numberOfSoluteAtoms numberofSolventAtoms
                 ↪  numberOfSolventMolecules   \n'
                sys.exit()
except IndexError:
    print '\n!!!Input command Error. Call function as: 3dExtract.py input.log
     ↪  numberOfSoluteAtoms numberofSolventAtoms numberOfSolventMolecules   \n'
    sys.exit()

#Call as 3dExtract.py inputfile #ofsoluteAtom #ofsolventAtom #ofsoluteMolecules
try:
    input=str(sys.argv[1])
    numberofSoluteAtoms=int(sys.argv[2])
    numberofSoventAtoms=int(sys.argv[3])
    numberOfSolventMolecules=int(sys.argv[4])
except IndexError:
    print '\n!!!Input command Error. Call function as: 3dExtract.py input.log
     ↪  numberOfSoluteAtoms numberofSolventAtoms numberOfSolventMolecules \n'
    sys.exit()
if input.endswith('.log'):
    output = str(input[:-4])+'.xyz'
else:
    output=str(input)

numberOfAllSolventsAtoms=numberofSoventAtoms*numberOfSolventMolecules
#This is for comparing files to be written
```

```
previousGrandString=''
collectionStarted=False
time=''
#1 is for cartesian line (useless), then 1 in 3(n+1) is for fragment H2O line
 ↪  (also useless)
numberOfLinesToBecollected=numberofSoluteAtoms+1+numberOfSolventMolecules*(numberofSoventAtoms


#number of molecules so far
timeCount=0
#total number of atoms (solute + solvent) - used later in checking if file is
 ↪  complete
atomCount=0
#define functions here
lineSinceTimeIsFound=0;
#do an input of solvent, solute atoms
molList=[]
#for printing time
def printTime (thisLine):
    lineComponents=thisLine.split();
    timeString=str(lineComponents[3]);
    print "Analyzing t = "+timeString+" fsec\n"
#to determine if line should be collected -
def shouldCollect():
    #only check if collection is in progress - if it is, then continue to finish
     ↪  collecting the lines
    #collectionStarted is determined when ' QM ATOM COORDINATES (ANG)'  is found
    if collectionStarted:
        #from first solute atom to the last fragment atom
        if (atomCount>=0 and atomCount<numberOfLinesToBecollected):
            return True;
        else:
            return False;
    else:
        return False;
# only write when atomCount==numberOfLinesToBecollected
def shouldWrite():
    #only check if collection is in progress
    if collectionStarted:
        #solute
        if (atomCount==numberOfLinesToBecollected):
            return True;
        else:
            return False;
    else:
        return False;

```

```
#Even now I still don't understand why GAMESS duplicate system geometry for a
↪  step twice in the log file
#This is written to prevent duplication of geometry in the xyz-movie file
def moleculeIsNotADuplication(currentMoleculeToBeWritten):
        # to compare previously stored geometry and a new one is tricky bc each
        ↪  string has different lengths
        # there must be a better of doing this - note for possible place for
        ↪  improvement
        #current the speed is quite slow probably due to this step
    halfSize=int(len(previousGrandString)/2)
    threeQuartersSize=int(len(previousGrandString)*3/4)
    if previousGrandString=='':
        return True;
    if (currentMoleculeToBeWritten[halfSize:threeQuartersSize] not in
    ↪  previousGrandString[halfSize-1:threeQuartersSize+1]):
        return True;
    else:
        return False;

#clear output.xyz
f = open(output, 'w');
f.write('')
#open input
f1=open(input)
#enumerate gets data in line - line and line index - n
#readlines() is eliminated because it creates a huge array and python cannot
↪  handle it when log file get very large
#using for line in... alleviate the burden on memory and actually speed up the
↪  process
for line in f1:
    #this keyword is usually before coordinate
    grandString=''
    #find out if checking for collectionStarted is needed
    if shouldCollect():
        #split line
        lineSplit=line.split()
        atomCount+=1;
        #append to molList
        molList.append(lineSplit)
    # if this then start collecting
    elif ' QM ATOM COORDINATES (ANG)' in line:
        collectionStarted=True
    #lastly, if none of the above, then find and print time
    elif  ' *** AT T=' in line:
        time=str(line)
        printTime(line);

```

```python
    if (shouldWrite()):
        atomCount=atomCount-(numberOfSolventMolecules+1);
        #for loop through a ***COPY*** of molList and delete some element from molList!
        #if you don't realize six asterisk then you should go back up - we do this so we can remove element along the way without messing up the index
        for line in list(molList):
            #if line has 4 elements then it's a coordinate from solvent fragment - we have to drop number behind atom - O1 to O
            if len (line) == 4:
                #store string
                oldString =  line[0]
                #replacement string
                newString=''
                #loop to check if it's a alphabet or not
                for character in range(len(oldString)):
                    #do substring of 1 character
                    subString = oldString[character:character+1]
                    #check if it's an alphabet - yes? then add to newString
                    if subString.isalpha():
                        newString = newString + subString
                #replace 'O1' with 'O'
                line[0]=newString
            #if it's 5 then it's solute coordinate - we have to get rid of atomic number behind atomic representation
            elif len(line) == 5:
                # 'N 7.0 ...' will become 'N ...'
                del line[1]
            #the rest are crap - just remove it out of the line
            else:
                #there's a reason why this is remove - not del - since we are iterating if we delete using index we are gonna be screwed
                molList.remove(line)
        #this is for if we have an incomplete file or inconsistant number of atoms we should only use the one before and break for loop without appending to grandString
        if len(molList) != atomCount:
            print 'error'
        #xyz file has a format that we need atomCount at the top followed by snapshot number(timeCount) on the next line before adding any coordinates
        grandString=grandString+str(atomCount)+'\n'+str(timeCount)+'\n'
        #loop tho molList to add data - molList = [['N','1','1','1'],['C'...],...] And element = ['N','1','1','1']
        for element in molList:
            #loop through element in molList data = 'N','1','1','1'
```

```
                for data in element:
                    #add to grandString and don't forget tab, return
                    grandString=grandString+data+'\t'
                #end one screenshot with a return
                grandString=grandString+'\n'
            #open animate.xyz for writing
            if moleculeIsNotADuplication(grandString):
                with open(output, 'a') as f:
                    f.write(grandString)
                    #add one to timeCount because we already write grandString
                    timeCount=timeCount+1
            #reset all values after writing
            atomCount=0;
            molList=[]
            collectionStarted=False;
            previousGrandString=str(grandString)
f.close()
#sanity check
print 'Done. Extract ' + str(timeCount) + ' snapshots total.'

```

Plot Potential Energy of MD run

plotEnergyMD6.py script is used to extract potential energy and temperature of each MD frame to determine the if the system has equilibrated. This and 3dExtract are very essential to the first stage of the project: they determine whether MD has failed or reached equilibrium based on the geometry and potential energy of the system. Many versions of this code has been developed and this is the most refined piece of code for its purpose. Future work can be done on plotting the plot on Matlab instead of obviously inferior python counterpart - matplotlib.

```
############################################################
### Use this to plot energy vs time to see if MD has    ###
### run its course. Generates: csv of PE and            ###
### temperature vs time, pdf of the plot                ###
############################################################

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import csv
import sys
import string

#call as plotEnergyMD3.py inputfile
#for asking what the input in terminal should be
try:
        if str(sys.argv[1])=='?':
                print '\nCall function as: plotEnergyMD.py input.log   \n'
                sys.exit()
        else:
                input=str(sys.argv[1])
except IndexError:
    print '\n!!!Input command Error. Call function as: plotEnergyMD.py input.log
     ↪  \n'
    sys.exit()
output=str(input) + '_energies.csv'

#initiate variables
lineBwTimeAndEnergy=0;
lineBwTimeAndTemp=0;
lineCountFromTime=0;
collectionStarted=False
foundTime=False
foundPE=False
foundTemp=False
firstTime=True
grandString=''

###############################
#functions

#check if time is in line
def shouldCollectTime(line):
    #only check if collection is in progress
    if ' *** AT T=' in line:
        return True;
    else:
        return False;

# for printing time so one can keep track of the progress
def printTime (thisLine):
    lineComponents=thisLine.split();
    timeString=str(lineComponents[3]);
    print "Analyzing t = "+timeString+" fsec\n"

#Are we currently looking potential energy?
def shouldCollectPE(line,reference,currentLine):
    #check if line bw time and energy is known - this is written as reference
```

```python
    if reference>0:
        if currentLine==reference:
            return True
        else:
            return False
    #if reference is not known, then it needs to be found by finding string POT EN...
    elif reference==0:
        if '     POT  ENERGY' in line:
            reference=int(currentLine)
            return True;
        else:
            return False;
#Are we currently looking Temp?
def shouldCollectTemp(line,reference,currentLine):
    #check if line bw time and energy is known - reference
    if reference>0:
        if currentLine==reference:
            return True
        else:
            return False
    #if not then it needs to be found by searching for the string TEMPER...
    elif reference==0:
        if '     TEMPER' in line:
            reference=int(currentLine)
            return True;
        else:
            return False;

#once everything is found, we should write down before moving on to the next snapshot
def shouldWrite():
    #only check if collection is in progress
    if foundTime:
        if foundPE:
            if foundTemp:
                return True;
            else:
                return False
        else:
            return False;
    else:
        return False;

################################
#open csv and prepare for writing
f = open(output, 'w');
```

```python
f.write('')

#open input
f1=open(input)
print 'finding patterns...'
#avoid using readlines() so there'll be no problem with large files
for line in f1:
    #time keyword is before PE, PE is before Temp so the search should be in this
    ↪  order in order to be most efficient
    #find out if collection is needed
    if not foundTime:
        if shouldCollectTime(line):
            printTime(line)
            #split line
            lineComponents=line.split();
            #append time (split) to string
            #split line using space - sample(*** AT T=         10.00 FSEC, THIS
            ↪  RUN'S STEP NO.=      10)
            #this will be split to ['***', 'AT', 'T=', '10.00'...] -time =
            ↪  element 4
            grandString=grandString+str(lineComponents[3]);
            foundTime=True;
    elif not foundPE:
        if shouldCollectPE(line,lineBwTimeAndEnergy,lineCountFromTime):
            #append time (split) to string
            #split line using space - sample(     POT  ENERGY      =
            ↪  -1.804578585E+05 KCAL/MOL)
            #this will be split to [..., '=', '-1.804578585E+05'...] -time =
            ↪  element 4
            lineComponents=line.split();
            grandString=grandString+','+str(lineComponents[3])
            foundPE=True
    elif not foundTemp:
        if not firstTime:
            if shouldCollectTemp(line,lineBwTimeAndTemp,lineCountFromTime):
                #append time (split) to string
                #split line using space - sample(     TEMPERATURE      =
                ↪  349.98666547 K)
                #this will be split to [..., '=', '-349.98666547'...] -time =
                ↪  element 3
                lineComponents=line.split();
                grandString=grandString+', '+str(lineComponents[2])+'\n';
                foundTemp=True
        #problem with this is - the first snapshot's temperature is not given in
        ↪  the log file
        #set Temp to 0 to indicate the beginning
        if firstTime:
            grandString=grandString+',0\n';
            #once append, turn off the boolean
            firstTime=False
            foundTemp=True
    #write after all data is collected for one snapshot
    if shouldWrite():
        with open(output, 'a') as f:
            f.write(grandString)
        #reset the variables
        lineCountFromTime=0;
        collectionStarted=False
        foundTime=False
        foundPE=False
        foundTemp=False
        grandString=''
    lineCountFromTime+=1;
#finish writing csv
f.close()

#plot
#pull out CSV
#use csv.reader bc csv has ',' and this automate the formatting
f = csv.reader(open(output))
#convert column to array using zip (a built in function)
Time, Energy, Temp = zip(*f)
#convert string to float
Time = map(float, Time)
Energy = map(float, Energy)
Temp = map(float, Temp)


#plot
x = Time
y1 = Energy
y2 = Temp

fig, ax1 = plt.subplots()

ax2 = ax1.twinx()
ax1.plot(x, y1, 'g-')
ax2.plot(x, y2, 'b-')

ax1.set_xlabel('Time (fs)')
ax1.set_ylabel('Potential Energy (KCal/mol)', color='g')
ax1.ticklabel_format(axis='y', style='sci', scilimits=(-2,2), useOffset=False)
ax2.set_ylabel('System Temperature (K)', color='b')
plt.title(r'      Potential Energy vs Time for the MD File: ' + str(input))
```

```
#Saving to pdf gives better resolution - picture is saved to vector
#there is a room for improvement especially these energy plots which look very
↪  mediocre
plt.savefig(str(input) + '_EnergyPlot.pdf', format='pdf')
```

Find The Most Equilibrated Period

There are currently no consensus as to when MD has reached the equilibrium. In the past, plotEnergyMD (previous script) was used to indicate whether the potential energy of the system(solute and solvent) has stabilized. Arbitrariness in deciding whether the equilibrium is reached falls in the hands of users. findEquilibrium.py is designed to solve this subjectivity. With a list of potential energies at different time from plotEnergyMD, linear fit can be done in a fix interval to evaluate the rise or fall in energy. Currently, the limit value is taken, still empirically, from 15000 to 25000 fs interval in CAMB3LYP aniline32.log. Further improvement can be done to find the bottom slope limit as a variable with molecule input.

```
############################################################
### This is used to determined the equilibrium using    ###
### linear regression and an upper limit for the slope        ###
############################################################

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import csv
import sys
import numpy as np

#for asking what the input in terminal should be
try:
    if str(sys.argv[1])=='?':
        print '\nCall function as: findEquilibrium.py input.log   \n'
        sys.exit()
    else:
        input=str(sys.argv[1])
except IndexError:
    print '\n!!!Input command Error. Call function as: findEquilibrium.py
    ↪  input.log \n'
    sys.exit()
output=str(input) + '_energies.csv'
#This portion is the same as in plotEnergyMD6
```

```python
#pull out CSV
f = csv.reader(open(output))
#convert column to array using zip (a built in function)
Time, Energy, Temp = zip(*f)
#convert string to float
Time = map(float, Time)
Energy = map(float, Energy)
Temp = map(float, Temp)

#find Equilibrium using linear regression
#this number control the range of time to be used in energy fluctuation calculation
minNumberOfStep=1000
#This is the limit above which the script will report no equilibrium is found
#this is from aniline32.log - 15000 to 25000
maxSlope=1e-4
#for plotting
slope=[]
print 'Finding equilibrium using minimum number of steps = '+str(minNumberOfStep)+' and top limit of acceptable slope = '+str(maxSlope)
try:
        #for looping
    size=len(Time)
    x=Time[0:minNumberOfStep]
    y=Energy[0:minNumberOfStep]
    #for using in loop
    #set thio a high value - it can be any number bc we will replace it with the lowest slope value found in loop
    lowestSlopeValue=1e5
    #Same - this will be replaced
    indexOfLowestSlope=-1
    for i in range(0,size-1-minNumberOfStep):
        print 'Finding equilibrium from t= '+str(Time[i])+' to '+str(Time[i+minNumberOfStep-1])
        #poly fit is basically a linear fit - m=slope, b=y_intersect
        m,b = np.polyfit(x, y, 1)
        #for plotting - append to array of existing slope values
        slope.append(m)
        print 'slope = ' +str(m)
        #take absolute value and see which interval does not fluctuate the least
        if abs(m)<=abs(lowestSlopeValue):
            lowestSlopeValue=float(m)
            indexOfLowestSlope=int(i)
        #this is similar to queue structure - room for improvement is to make x and y arrays into actual queues
        #else remove the head and add next tail
        del x[0]
        x.append(Time[minNumberOfStep+i])
        del y[0]
        y.append(Energy[minNumberOfStep+i])
#if x or y does not have enough element (minNumberOfStep) then report error
#room for improvement - move this up top instead of having a long try
except IndexError:
        print 'There is not enough data to determine the equilibrium'
#if lowestSlopeValue pass the top limit then report
if abs(lowestSlopeValue)<=abs(maxSlope):
    print 'Found best equilibrium starting from '+str(Time[indexOfLowestSlope])+'
     ↪  to '+str(Time[indexOfLowestSlope+minNumberOfStep-1])+' with slope =
     ↪  '+str(lowestSlopeValue)
#if not then say so
else:
        print 'Equilibrium is not yet reach.' '
        print 'The current limit is at '+str(maxSlope) +' kcal/mol/fs and the
         ↪  lowest value of slope = '+str(lowestSlopeValue)

#plot slope vs time
x=slope
#align time with slope
y=Time[0:size-1-minNumberOfStep]
plt.plot(y,x)
plt.xlabel('Starting Time (fs)')
plt.ylabel('Slope (KCal/mol/fs)')
plt.ticklabel_format(axis='y', style='sci', scilimits=(-2,2), useOffset=False)
plt.title('Slope vs Time for the MD File: ' + str(input))
plt.savefig(str(input) + '_SlopePlot.pdf', format='pdf')
```

Prepare TDDFT input

After equilibrium is determined, fincut2.py can be used to create TDDFT input files from xyz-movie file. a Text file containing gmssub commands especially for Bowdoin hpc grid is created. The script was created by Nathan Ricke for this work, but many improvement has been made. The new script works faster and more efficient, even though it still has outdated syntax and methods.

```
###########################################################
### This script create a folder of inp for TDDFT        ###
###########################################################

import os
import sys

```

```python
#call as fincut.py input.log startTime stopTime timePerFrame
#this will run from starting startTime+timePerFrame to stopTime
#for example 15010-25000 if input is 15000, 25000

#this dict is for generating atomic number from Acronym
atomicNumber={'LV': 116.0, 'BE': 4.0, 'FR': 87.0, 'BA': 56.0, 'BH': 107.0, 'BI': 83.0, 'BK': 97.0, 'EU': 63.0, 'FE': 26.0, 'BR': 35.0, 'ES': 99.0, 'FL': 114.0, 'FM': 100.0, 'RG': 111.0, 'RU': 44.0, 'NO': 102.0, 'NA': 11.0, 'NB': 41.0, 'ND': 60.0, 'NE': 10.0, 'RE': 75.0, 'RF': 104.0, 'LU': 71.0, 'RA': 88.0, 'RB': 37.0, 'NP': 93.0, 'RN': 86.0, 'RH': 45.0, 'B': 5.0, 'CO': 27.0, 'TH': 90.0, 'CM': 96.0, 'CL': 17.0, 'H': 1.0, 'CA': 20.0, 'CF': 98.0, 'CE': 58.0, 'N': 7.0, 'CN': 112.0, 'P': 15.0, 'GE': 32.0, 'GD': 64.0, 'GA': 31.0, 'V': 23.0, 'CS': 55.0, 'CR': 24.0, 'DS': 110.0, 'CU': 29.0, 'SR': 38.0, 'UUP': 115.0, 'UUS': 117.0, 'TC': 43.0, 'KR': 36.0, 'SI': 14.0, 'SN': 50.0, 'SM': 62.0, 'UUT': 113.0, 'SC': 21.0, 'SB': 51.0, 'TA': 73.0, 'OS': 76.0, 'PU': 94.0, 'SE': 34.0, 'AC': 89.0, 'HS': 108.0, 'YB': 70.0, 'DB': 105.0, 'C': 6.0, 'HO': 67.0, 'DY': 66.0, 'HF': 72.0, 'HG': 80.0, 'HE': 2.0, 'PR': 59.0, 'PT': 78.0, 'LA': 57.0, 'F': 9.0, 'UUO': 118.0, 'LI': 3.0, 'PB': 82.0, 'TL': 81.0, 'TM': 69.0, 'LR': 103.0, 'PD': 46.0, 'TI': 22.0, 'TE': 52.0, 'TB': 65.0, 'PO': 84.0, 'PM': 61.0, 'ZN': 30.0, 'AG': 47.0, 'NI': 28.0, 'I': 53.0, 'K': 19.0, 'IR': 77.0, 'AM': 95.0, 'AL': 13.0, 'O': 8.0, 'S': 16.0, 'AR': 18.0, 'AU': 79.0, 'AT': 85.0, 'W': 74.0, 'IN': 49.0, 'Y': 39.0, 'CD': 48.0, 'ZR': 40.0, 'ER': 68.0, 'MD': 101.0, 'MG': 12.0, 'PA': 91.0, 'SG': 106.0, 'MO': 42.0, 'MN': 25.0, 'AS': 33.0, 'MT': 109.0, 'U': 92.0, 'XE': 54.0}

#if not sure use ? to ask
try:
        if str(sys.argv[1])=='?':
                print '\nCall function as: fincut.py input.log numberOfSoluteAtoms numberofSolventAtoms numberOfSolventMolecules startTime stopTime timePerFrame\n'
                sys.exit()
except IndexError:
    print '\n!!!Input command Error. Call function as: fincut.py input.log numberOfSoluteAtoms numberofSolventAtoms numberOfSolventMolecules startTime stopTime timePerFrame\n'
    sys.exit()

#Call as fincut.py input.log numberOfSoluteAtoms numberofSolventAtoms numberOfSolventMolecules startTime stopTime timePerFrame
try:
    input=str(sys.argv[1])
    numberOfSoluteAtoms=int(sys.argv[2])
    numberOfSolventAtoms=int(sys.argv[3])
    numberOfSolventMolecules=int(sys.argv[4])
    startTime =int(sys.argv[5])
    stopTime =int(sys.argv[6])
    timePerFrame=int(sys.argv[7])
except IndexError:
    print '\n!!!Input command Error. Call function as: fincut.py input.log numberOfSoluteAtoms numberofSolventAtoms numberOfSolventMolecules startTime stopTime timePerFrame\n'
    sys.exit()

#should collect data when 1) time within interval specified
#2) they are x y z line - not line 1 and 2
def shouldCollect(numberOfLine):
    frame=int(numberOfLine/totalNumberOfLineInOneSet)
    time=frame*timePerFrame
    #if within range
    if time>startTime and time<=stopTime:
        #0 and 1 (line 1 and 2) in each frame in xyz are not useful
        if numberOfLine%totalNumberOfLineInOneSet!=0 and numberOfLine%totalNumberOfLineInOneSet!=1:
            return True
        else:
            return False
    else:
        return False

#condition for writing to each inp
def shouldWrite(numberOfLine):
    #should write when last line of a frame is read
    if numberOfLine%totalNumberOfLineInOneSet==totalNumberOfLineInOneSet-1 :
        return True
    else:
        return False

#write FRAGNAME=H2ODFT ! 1
def shouldWriteFragmentHeader(numberOfLineInSolvent):
    #should write before writing geometry of solvent molecules
    if numberOfLineInSolvent%numberOfSolventAtoms==0 :
        return True
    else:
        return False

#create format of GAMESS inp
def insertAtomicNumberInto(line):
                lineSplit=line.split()
                lineSplit.insert(1,str(atomicNumber[lineSplit[0]]))
                return lineSplit[0]+'\t'+lineSplit[1]+'\t'+lineSplit[2]+'\t'+lineSplit[3]+'\t'+li
#create format of GAMESS inp O -> O1, H-> H2...
```

```
def insertCountsInto(line, count):
                lineSplit=line.split()
                lineSplit[0]=lineSplit[0]+str(count)
                return
                 ↪  lineSplit[0]+'\t'+lineSplit[1]+'\t'+lineSplit[2]+'\t'+lineSplit[3]+'\n'

#even tho input is received written in .log - it will ultimately use .xyz created
 ↪  by 3dExtract for efficiency
#this can be confusing - room for improvement
if input.endswith('.log'):
    input = input[:-4]

#Path for storing input files
path = os.getcwd()
inputPath=path+'/'+input+'InputFiles'
#create folders if not already done
if not os.path.exists(inputPath): os.makedirs(inputPath)

#initiate variables
grandString=[]
lineNumber=0
numberOfAllSolventsAtoms=numberOfSolventAtoms*numberOfSolventMolecules
totalNumberOfAtoms=numberOfSoluteAtoms+numberOfAllSolventsAtoms
totalNumberOfLineInOneSet=totalNumberOfAtoms+2

#if no xyz in folder - ask for it
    #if no 3dExtract.py - print...
    #if there is, call it from here + print s'th
#else create one?
#if there is, then proceed
#this is very confusing - if there is xyz but it's not updated when it's of no
 ↪  use - room for improvement

#open input.xyz
try:
    f=open(input+'.xyz')
except IOError:
    print '\nThere is currently no xyz file named '+input
    print 'trying to call 3dExtract'
    try:
        os.system('python 3dExtract4.py'+' '+input+'.log'+'
         ↪  '+str(numberOfSoluteAtoms)+' '+str(numberOfSolventAtoms)+'
         ↪  '+str(numberOfSolventMolecules))
    except IOError:
        print '\n  Error. There is no 3dExtract to call. Try copying 3dExtract
         ↪  here. \n'
        print 'Process terminated abnormally'
        sys.exit()
    f=open(input+'.xyz')

#since we are reading from xyz - using readlines() is okay
numberOfAllAtoms=int(f.readline().strip())
#enumerate gets data in line - line and line index - n
for line in f:
    lineNumber+=1
    #check if the line should be writen
    if shouldCollect(lineNumber):
        grandString.append(line)
        #if about to write -create file with this name
        if shouldWrite(lineNumber):
            frame=int(lineNumber/totalNumberOfLineInOneSet)
            time=frame*timePerFrame
            f1 = open(inputPath+'/'+input+'_'+str(time)+'.inp','w')
            #write header
            headerString=""" $CONTRL SCFTYP=RHF TDDFT=EXCITE DFTTYP=CAMB3LYP
            ↪  RUNTYP=ENERGY
       ICHARG=0 MULT=1 COORD=UNIQUE MAXIT=200 $END
 !TDDFT requires lots of memory space
 $SYSTEM MWORDS=200 MEMDDI=250 $END
 $SCF DIRSCF=.T. $END
 $TDDFT NSTATE=5 TPA=.f. $END
 $BASIS GBASIS=N311 NGAUSS=6 NDFUNC=2 NPFUNC=1
       DIFFSP=.TRUE. DIFFS=.TRUE. POLAR=POPN311 $END
 $DATA\n"""+input+' at t= '+str(time)+'\nC1 1\n'
            print 'Making input file for 'input+' at t= '+str(time)
            #write the header
            f1.write(headerString)
            #write one line by one - before writing we need to add atomic number
            ↪  in using the function defined above
            for eachLine in grandString[:numberOfSoluteAtoms]:
                eachLine=insertAtomicNumberInto(eachLine)
                f1.write (eachLine)
            #end solute and go to solvent
            stringBwSoluteAndSolvent=""" $END\n\n $EFRAG\nCOORD=CART
            ↪  POSITION=OPTIMIZE \n"""
            f1.write(stringBwSoluteAndSolvent)
            #write fragments
            #for iteration
            i=0
            for numberOfLineWithInSolvents, eachLine in
            ↪  enumerate(grandString[numberOfSoluteAtoms:]):
                    #is header needed - if so write
                if shouldWriteFragmentHeader(numberOfLineWithInSolvents):

                    ↪  fragmentsNumber=numberOfLineWithInSolvents/numberOfSolventAtoms+1
```

```
                        f1.write("FRAGNAME=H2ODFT ! "+str(fragmentsNumber)+'\n')
                    #write each line in fragments
                    atomNumberInFragments=i%numberOfSolventAtoms+1
                    eachLine=insertCountsInto(eachLine, atomNumberInFragments)
                    f1.write (eachLine)
                    i+=1
                #close and reset
                f1.write (" $END\n")
                grandString=[]
                f1.close()

#create megaio for gmssub input
allFiles = os.listdir(inputPath)
program = 'gmssub'
processors = '32'

#create the write file - like gmssub aniline32.inp aniline32.log -l 10g=true
f = open('megaio_'+input+'.txt','w')
#write each item
for item in allFiles:
        #for GAMESS output naming
    outputName=str(item)[:-4]+'.log'
    f.write(program + ' %s '%item + processors +' '+outputName+' -l 10g=true\n')
f.close()
```

Pull Excited State Energies from TDDFT Log Files

postMDDataPull2.py pulls out excited state energies and dipole moments. Energy output is in the format of time, S1, S2... Dipole output is in the format of time, X1, Y1, Z1, X2...

```
############################################################
### Use this pull out excited state energies and dipole ###
### moments from all the files in specified folder      ###
### Output is time, S1, S2... or time, X1, Y1, Z1, X2...###
############################################################

import os as os
import numpy as numpy
import matplotlib.pyplot as plt
import sys

#if not sure use ? to ask
try:
        if str(sys.argv[1])=='?':
                print '\nCall function as: postMDDataPull2.py inputDirectory
                 ↪  NumberOfExcitedStateEnergies\n'
                sys.exit()
except IndexError:
    print '\n!!!Input command Error. Call function as: postMDDataPull2.py
     ↪  inputDirectory NumberOfExcitedStateEnergies\n'
    sys.exit()

#Call as fincut.py input.log numberOfSoluteAtoms numberofSolventAtoms
 ↪  numberOfSolventMolecules startTime stopTime timePerFrame
try:
    input=str(sys.argv[1])
        numberOfExcitedStates=int(sys.argv[2])
except IndexError:
    print '\n!!!Input command Error. Call function as: postMDDataPull2.py
     ↪  inputDirectory NumberOfExcitedStateEnergies\n'
    sys.exit()
# to prevent / at the end of the input file if used terminal autofill
if input.endswith('/'):
    input = input[:-1]

#make input path - folder containing the log files or out files
path = os.getcwd()
inputPath=path+'/'+input

#arrays for storing and manipulating the data extracted
outputList=[]
fileList=os.listdir(inputPath)
listAllEAndf=[]
energies=[]
oscillatorStrengths=[]
timeList=[]
energyFinal=numpy.array(['Energy(eV)'])
oscillatorStrengthFinal=numpy.array(['Oscillator Strength'])

#get all the file names
for fileName in fileList:
    if (".out" in fileName or ".log" in fileName):
        outputList.append(fileName)

#open 2 output files
f1=open(input+'MD_data.csv','w')
f2=open(input+'MD_dipole.csv','w')
#now read each file and collect data
for fileName in outputList:
        eachFile=inputPath+'/'+fileName
        f=open(eachFile,'r')
        lines=f.readlines()
        startingLine=0
```

```python
    fileIsComplete=False
    #find if the file is complete
    # use enumerate and reverse
    for i, line in reversed(list(enumerate(lines))):
        if ' STATE #    1  ENERGY =     ' in line:
            startingLine=i
            fileIsComplete=True
            break
    # now start collecting data
    if fileIsComplete:
        print 'Harvesting data from file named: '+fileName
        counter=0
        #loop
        for n, line in enumerate(lines):
            #time
            if line.startswith('     RUN TITLE'):
                nextLine=lines[n+2]
                time=str(nextLine.split()[3])
                f1.write(time)
                f2.write(time)
            #energy
            if line.startswith(' STATE # '):
                E=line.split()[5]
                nextLine=lines[n+1]
                f=nextLine.split()[3]
                f1.write(','+E+','+f)
                counter+=1
            #stop collecting
            if (counter==numberOfExcitedStates):
                f1.write('\n')
                counter=0
            #dipole
            if line.startswith('                          SUMMARY OF TDDFT RESULTS'):
                #5 lines are from 'SUMMARY...' to ' 1  A ...' which starts to contain dipole moments
                for linesAhead in range(5,5+numberOfExcitedStates):
                    lineContainingDipoles=lines[n+linesAhead]
                    lineSplit=lineContainingDipoles.split()
                    #locations of dipole in line
                    [X,Y,Z]=[lineSplit[4], lineSplit[5], lineSplit[6]]
                    f2.write(','+X+','+Y+','+Z)
                f2.write('\n')
```

GAMESS inputs

MD Input File

MD run is core to modeling explicit solvent. The MD run is simulated every femtosecond but only record every 10 femtoseconds. The bath temperature is 25 ± 25 degree Celsius. Solvent boundary potential is also actiavetd using default Sforce value, but with estimate ssbp radius. ########n########## are for restarting MD in case the calculation abruptly ends (see next section). In this version, dispersion correction is not turned on. Basis set = 6-31+(2d,p)

```
 # run type = MD, with functional = CAMB3LYP, COORD = UNIQUE is important
 $CONTRL SCFTYP=RHF RUNTYP=MD COORD=UNIQUE
    DFTTYP=CAMB3LYP MAXIT=200 ICHARG=0 MULT=1 $END
 # MD is recording every 10 frames with default 1 frame =10 fs
 # 25 degree celcius, RSTEMP is on for keeping the temp ~ +/-25
 # ssbp is on with default SForce value and radius estimated from prepareMD2.py
 $MD KEVERY=10 PROD=.T. NVTNH=2 MBT=.T. MBR=.T.
    BATHT=298 RSTEMP=.T. DTEMP=25 NSTEPS=50000
    SSBP=.T. SFORCE=1.0 DROFF=12.0632116659 $END
 ########1#########
 ##################
 # dispersion correction is off
 $DFT DC=.F. $END
 # memory requested at each node =1000 million words
 # memory reserved for communication = 1000 million words
 $SYSTEM MWORDS=1000 MEMDDI=1000 $END
 $SCF DIRSCF=.T. $END
 # Basis set = 6-31+(2d,p)
 $BASIS GBASIS=N31 NGAUSS=6 NDFUNC=2 NPFUNC=1
    DIFFS=.TRUE. POLAR=POPN31 $END
 # solute geometry - C1 1 = symmetry data
 $DATA
MD INPUT for aniline32
C1 1
 ########2#########
N        7.0       -2.3128100000       -0.0046000000       -0.0894530000
C        6.0       -0.9197160000       -0.0031280000       -0.0360090000
C        6.0       -0.2076150000       1.2004070000       -0.0355880000
.
.
H        1.0       -2.7544730000       0.8437050000       0.2369240000
H        1.0       -2.7570450000       -0.8248640000       0.2993670000
 ##################
 $END

 # solvent geometry in EFP1 (EFP2 is still not available)
 ########3#########
 $EFRAG
COORD=CART  POSITION=OPTIMIZE
FRAGNAME=H2ODFT ! 1
 O1         1.7760990000        4.8390610000        -2.1049530000
 H2         0.9224740000        4.4173920000        -2.2628490000
 H3         2.4128200000        4.1148590000        -2.1441810000
FRAGNAME=H2ODFT ! 2
 O1         3.6783070000        3.8351060000        0.8717800000
 H2         3.6020030000        4.1116030000        1.7932670000
 H3         3.4138960000        4.6126330000        0.3648680000
.
.
FRAGNAME=H2ODFT ! 32
 O1         3.7691430000        -1.4091250000        -4.0319510000
 H2         2.8756120000        -1.5562470000        -4.3656710000
 H3         3.6686840000        -1.3995500000        -3.0721400000
 $END
 ###################
```

MD Restart

########n######### are for restarting MD. For example, if MD stops from errors at t= 39000 fs, a restart geometry and $MD should be obtained from t= 38960 in the run's trj file. ########1######### from trj goes to ########1######### in MD input file and so on with 2 and 3.

```
.
.
 #time = 38960 fs
===== MD DATA PACKET =====
NAT=      14 NFRG=      32 NQMMM=       0
TTOTAL=    38960.00 FS    TOT. E=      -180710.543716 KCAL/MOL
POT. E=       -180783.917894 KCAL/MOL  BATHT=           298.000000
KIN. E=            73.374179  TRANS KE=  44.012966  ROT KE=  29.361213 KCAL/MOL
----- QM PARTICLE COORDINATES FOR $DATA GROUP -----
 ########2#########
N           7.0        3.8554852781       -1.5814882196       -3.7440432660
C           6.0        2.9445799802       -1.6872198850       -4.7513152920
.
.
```

```
H           1.0        3.8827088558       -0.7337182877       -3.1704671239
 ###################
----- EFP PARTICLE COORDINATES FOR $EFRAG GROUP -----
 #########3#########
 $EFRAG
COORD=CART POSITION=OPTIMIZE
FRAGNAME=H2ODFT  !   1
O1                     0.4899097683        5.6815060052        1.3332597175
H2                    -0.1774491436        5.6953737679        2.0005813955
H3
.
.
FRAGNAME=H2ODFT  !  32
O1                     1.8397452656       -1.0633683830        1.4432099580
H2                     2.6999380304       -1.1499324935        1.8219625372
H3                     1.3989608369       -0.3812464183        1.9241435987
 $END
 ###################
     GRADIENT DATA (NOT USED BY RESTARTS)...
FRAGMENT #     1  H2ODFT
.
.
----- RESTART VELOCITIES FOR $MD GROUP -----

 #########1#########
 $MD READ=.TRUE. MBT=.FALSE. MBR=.FALSE. TTOTAL=    3.90E-11
 MDINT= VVERLET     DT= 0.10E-14 NVTNH= 2  NSTEPS=   11040
 RSTEMP=.T. DTEMP=    25.00 LEVERY= 50000
 RSRAND=.F. NRAND=  1000 NVTOFF=  0 JEVERY=    10
 PROD=.T.   KEVERY=    10 DELR=   0.020
Batht(1)=298.00
 SSBP=.T.   SFORCE=   1.0 DROFF= 12.1*****
TVELQM(1)=     ! QM ATOM TRANS. VELOCITIES (BOHR/PS) !
  -1.436664655E+00  -1.014912632E+00   1.731488079E+01
.
.
  -4.714191860E+00  -3.065697154E+00   1.306355299E+01
TVEL(1)=     ! EFP TRANSLATIONAL VELOCITIES (BOHR/PS) !
  -9.879998843E+00  -1.286351783E+01  -1.625361337E-01
.
.
   4.573922683E+00   9.602851076E+00   6.875495095E+00
QUAT(1)=     ! EFP QUATERNIONS !
   6.658493597E-01  -7.408965451E-01   1.545516153E-02   8.647587874E-02
.
.
   8.096376474E-01   4.753557554E-01   1.423717372E-01   3.134550593E-01
```

```
RVEL(1)=     ! EFP ANGULAR VELOCITY (RAD/PS) !
   1.392338986E+01  -1.145718732E+01  -1.039863492E+01
.
.
  -1.217796312E+01   2.229922184E+00   2.270460047E+01
QUAT1D(1)=     ! EFP QUATERNION 1ST DERIV. !
   4.237107071E+12   4.149735474E+12   8.424472510E+12   1.422917746E+12
.
.
   5.909828315E+12  -1.125106858E+13  -8.408659179E+11   2.179439222E+12
QUAT2D(1)=     ! EFP QUATERNION 2ND DERIV. !
  -6.427981808E+26  -4.203604186E+26  -7.365308723E+25   1.102236012E+26
.
.
   8.379527677E+26   4.204660564E+26  -1.448608909E+27  -2.676740098E+27
 $END
 ###################

 #time = 38970 fs
===== MD DATA PACKET =====
NAT=      14 NFRG=      32 NQMMM=       0
TTOTAL=    38970.00 FS    TOT. E=      -180715.284973 KCAL/MOL
POT. E=       -180783.048242 KCAL/MOL  BATHT=           298.000000
KIN. E=            67.763269  TRANS KE=  41.754223  ROT KE=  26.009046 KCAL/MOL
----- QM PARTICLE COORDINATES FOR $DATA GROUP -----
N           7.0        3.8518322659       -1.6019379232       -3.6477010146
.
.
```

TDDFT Input File

Excited state energies are calculated using TDDFT. Direct SCF calculation is turned on.

Basis set = 6-311++(2d,p)

```
 # run type = [excitation] energy, with functional = CAMB3LYP, and TDDFT
 $CONTRL SCFTYP=RHF TDDFT=EXCITE DFTTYP=CAMB3LYP RUNTYP=ENERGY
       ICHARG=0 MULT=1 COORD=UNIQUE MAXIT=200 $END
 #TDDFT requires lots of memory space
 # memory requested at each node =1000 million words
 # memory reserved for communication = 1000 million words
  ↪
 $SYSTEM MWORDS=200 MEMDDI=250 $END
 #activate direct SCF calculation
 $SCF DIRSCF=.T. $END
 # find 5 excited states - the current setting is purely driven by its lower cost
 # Previous experience shows that 10 states gives only a few strong peak.
 $TDDFT NSTATE=5 TPA=.f. $END
 # Basis set = 6-311++(2d,p)
 $BASIS GBASIS=N311 NGAUSS=6 NDFUNC=2 NPFUNC=1
       DIFFSP=.TRUE. DIFFS=.TRUE. POLAR=POPN311 $END
 # solute geometry - C1 1 = symmetry data
 $DATA
aniline32 at t= 15010
C1 1
N        7.0        2.4008547653        5.9114221893        -1.1412310058
C        6.0        1.9371475177        5.9223533811        -2.4157851823
C        6.0        0.6209366009        6.2361805033        -2.7812041720
.
.
H        1.0        1.7323956480        5.6459040114        -0.4329519914
H        1.0        3.3564486514        5.6102990678        -1.0242941608
 $END

 # solvent geometry in EFP1 (EFP2 is still not available)
 $EFRAG
COORD=CART  POSITION=OPTIMIZE
FRAGNAME=H2ODFT ! 1
O1         2.335993939511        3.751856628604        1.427418842826
H2         1.439322965739        3.833168541986        1.710699607266
H3         2.854351938959        3.613582975500        2.203990690907
FRAGNAME=H2ODFT ! 2
O1         3.266753260182        2.613267395174        3.808546039622
H2         3.514552920456        1.768271678250        3.468757839439
H3         3.822697636667        2.780022240614        4.552855871622
.
.
FRAGNAME=H2ODFT ! 32
O1         -0.041331901955        -3.906598195206        1.282021099515
H2         -0.790424236756        -4.406259423929        1.565001283174
H3         -0.374908444181        -3.111491773717        0.898079798177
 $END
```

Acknowledgement

Sed aliquam euismod nunc nec consectetur. Fusce eget dui id tortor tristique luctus. Pellentesque elit eros, molestie et molestie vitae, laoreet in risus. Nullam ligula lectus, pulvinar eget sagittis sed, cursus ac magna. . . .

Supporting Information Available

Ut volutpat, felis sit amet malesuada blandit, arcu sapien feugiat libero, vel interdum ipsum dolor et dolor. Fusce tortor sapien, pharetra sit amet posuere ac, viverra mollis est. Maecenas auctor ultrices quam a pharetra. Aenean ornare dictum libero vitae gravida. Mauris auctor sapien at purus accumsan lacinia.

References

(1) Bedoux, G.; Roig, B.; Thomas, O.; Dupont, V.; Le Bot, B. *Environmental Science and Pollution Research* **2012**, *19*, 1044–1065.

(2) Kliegman, S.; Eustis, S. N.; Arnold, W. a.; McNeill, K. *Environmental science & technology* **2013**, *47*, 6756–63.

(3) Ricke, N. D. Application of the Landau-Zener Model and Fermi ' s Golden Rule to Estimate Triplet Quantum Yield for Organic Molecules. Ph.D. thesis, Bowdoin College, 2014.

(4) Lin, H.; Truhlar, D. G. *Theoretical Chemistry Accounts* **2007**, *117*, 185–199.

(5) Cossi, M.; Barone, V. *The Journal of Chemical Physics* **2000**, *112*, 2427.

(6) Barone, V.; Polimeno, A. *Chem. Soc. Rev.* **2007**, *36*, 1724–1731.

(7) Li, J.; Cramer, C. J.; Truhlar, D. G. *International Journal of Quantum Chemistry* **1999**, *77*, 264–280.

(8) Tomasi, J.; Mennucci, B.; Cammi, R. *Chemical Reviews* **2005**, *105*, 2999–3093.

(9) Day, P. N.; Jensen, J. H.; Gordon, M. S.; Webb, S. P.; Stevens, W. J.; Krauss, M.; Garmer, D.; Basch, H.; Cohen, D. *The Journal of Chemical Physics* **1996**, *105*, 1968–1986.

(10) Yoo, S.; Zahariev, F.; Sok, S.; Gordon, M. S. *Journal of Chemical Physics* **2008**, *129*, 1–8.

(11) Magyar, R. J.; Tretiak, S. *Journal of Chemical Theory and Computation* **2007**, *3*, 976–987.

(12) Defusco, A.; Minezawa, N.; Slipchenko, L. V.; Zahariev, F.; Gordon, M. S. *J. Phys. Chem. Lett* **2011**, *2*, 2184–2192.

(13) Wiberg, K. B. *Journal of Computational Chemistry* **2004**, *25*.

(14) Barnes, L.; Abdul-Al, S.; Allouche, A.-R. *The Journal of Physical Chemistry A* **2014**, *118*, 11033–11046.

(15) Plugatyr, A.; Svishchev, I. M. *The Journal of chemical physics* **2009**, *130*, 114509.