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 sol-

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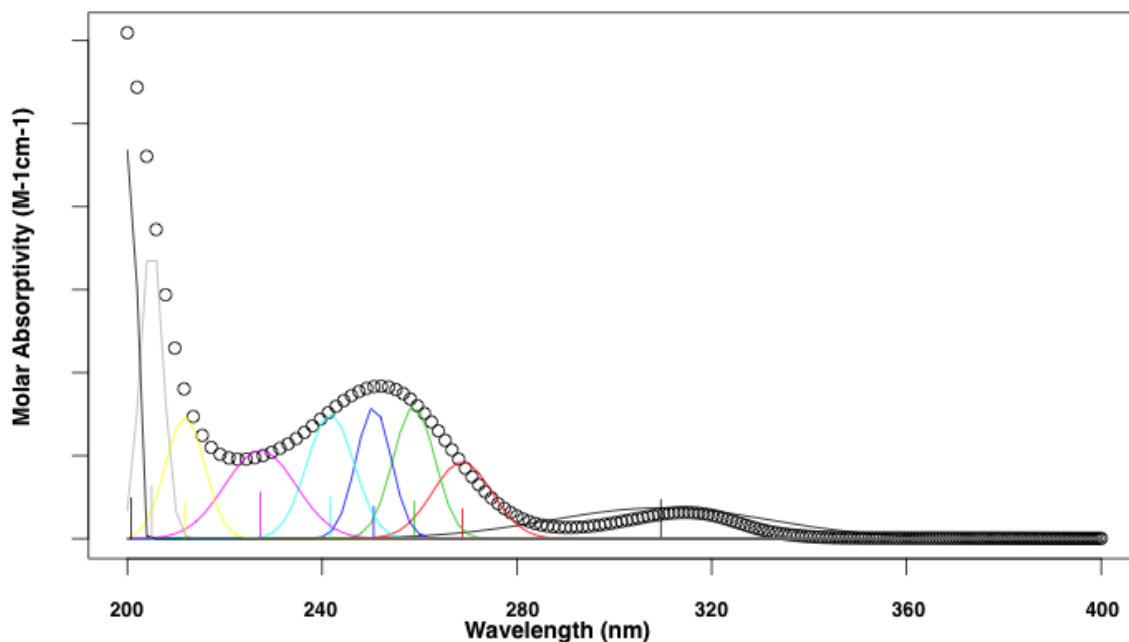
Table 1: Wavelength and Oscillator Strength from MD-TDDFT calculation of aniline in 32 water molecules.

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)
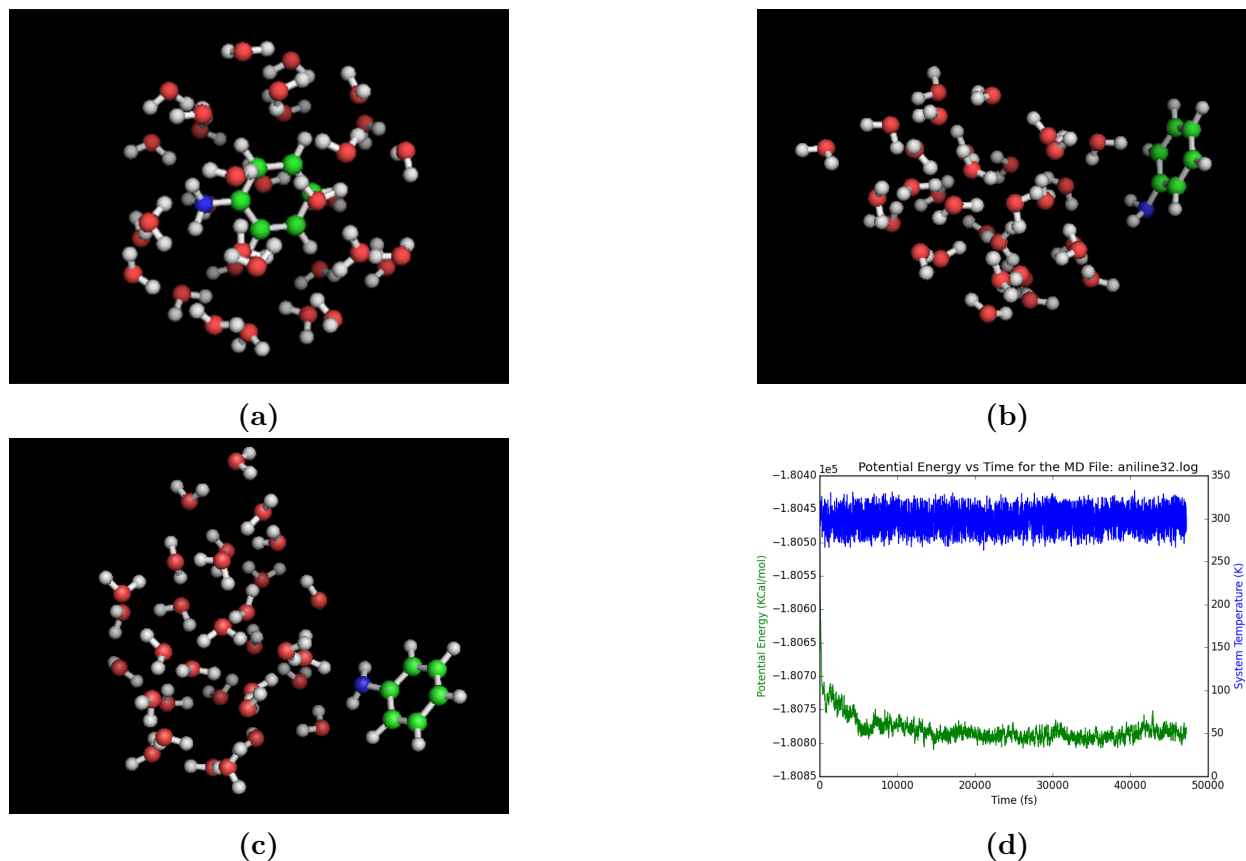
Figure 1: Molecular Dynamics run of aniline in 32 explicit solvating water molecules. Notice that at equilibrium, aniline molecule comes outside of the water sphere. Albeit hydrogen bond being clearly established, lack of total submersion in water means 32-water does not fully solvate the aniline molecule 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.

```
1    ###########################################################
2    ###Create inp for MD run from xyz file from packmol      ###
3    ###########################################################
4
5    import sys
6    import csv
7    import os
8    import string
9
10
11   #for asking what the input in terminal should be
12   try:
13           if str(sys.argv[1])=='?':
14                   print '\nCall function as: prepareMD.py input.xyz
                     ↪   numberOfSoluteAtoms numberofSolventAtoms
                     ↪   numberOfSolventMolecules \n'
15                   sys.exit()
16   except IndexError:
17       print '\n!!!Input command Error. Call function as: prepareMD.py input.xyz
             ↪   numberOfSoluteAtoms numberofSolventAtoms numberOfSolventMolecules \n'
18       sys.exit()
19   #for assigning received input from terminal
20   try:
21       input=str(sys.argv[1])
22       numberofSoluteAtoms=int(sys.argv[2])
23       numberofSoventAtoms=int(sys.argv[3])
24       numberOfSolventMolecules=int(sys.argv[4])
25   except IndexError:
26       print '\n!!!Input command Error. Call function as: prepareMD.py input.xyz
             ↪   numberOfSoluteAtoms numberofSolventAtoms numberOfSolventMolecules \n'
27       sys.exit()
28   #generate output name
29   if input.endswith('.xyz'):
30       output = input[:-4]+'.inp'
31   #for safety - at worst the output will not overwrite the input
32   else:
33       output=input+'.inp'
34
35   #part one
36   #This part is for finding ssbp radius for inout file
37   #enumerate gets data in line - line and line index - n
38   radiusInSolute=0.0
39   radiusInSolvent=0.0
40   avgX=0.0
41   avgY=0.0
42   avgZ=0.0
43   X=[]
```

```python
44    Y=[]
45    Z=[]
46
47    lineNumber=0
48    #open input
49    f2=open(input)
50    for line in f2:
51            lineNumber+=1
52            #first two line does not contain useful info - x y z start on the third
              ↪   line
53            if lineNumber>2:
54                    #x y z
55                    lineSplit=line.split()
56                    X.append(float(lineSplit[1]))
57                    Y.append(float(lineSplit[2]))
58                    Z.append(float(lineSplit[3]))
59    #for looping through array below
60    size=len(X)
61    #find a CG for solute atoms
62    avgX=sum(X[:numberofSoluteAtoms-1])/numberofSoluteAtoms
63    avgY=sum(Y[:numberofSoluteAtoms-1])/numberofSoluteAtoms
64    avgZ=sum(Z[:numberofSoluteAtoms-1])/numberofSoluteAtoms
65    #looping to find radius of each atoms in relative to solute's CG
66    #also find the maximum value of them
67    for i in range(0,size):
68                    d=((X[i]-avgX)**2+(Y[i]-avgY)**2+(Z[i]-avgZ)**2)**0.5
69                    if i<numberofSoluteAtoms:
70                            if radiusInSolute<d:
71                                    radiusInSolute=d
72                    else:
73                            if radiusInSolvent<d:
74                                    radiusInSolvent=d
75
76    #radius should be a little bit larger than the two combined - 3 Angstrom larger -
      ↪   this does not need to be super accurate
77    radiusInSolute=radiusInSolute
78    radiusInSolvent=radiusInSolvent
79    ssbpRadius=radiusInSolute+radiusInSolvent+3
80    print '\n'
81    print 'Radius in solute is:\t'+str(radiusInSolute)
82    print 'Radius in solvent is:\t'+str(radiusInSolvent)
83    print 'ssbp Radius should be:\t'+str(ssbpRadius)
84    print '\n'
85    ############################
86
87    #Part two - this is where geometry data is taken from xyz, change into GAMESS
      ↪   input's format + other input
```

```python
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
```

```
116    for n, line in enumerate(lines):
117        #take all solute molecules (in range of 2 (line 3 where packmol starts)
           ↪    to num+2)
118        #it's num+2 bc the range will go to num+1
119        if n == 2:
120            print 'Now Writing Solute:\n'
121        if n in range(2,numberofSoluteAtoms+2):
122            lineSplit=line.split();
123            lineSplit.insert(1,str(atomicNumber[lineSplit[0]]))
124            #convert coordinates to 10 decimals (add zeros if need be)
125            for index in [2,3,4]:
126                lineSplit[index]=float(lineSplit[index])
127                lineSplit[index]=format(lineSplit[index],'.10f')
128                grandString=lineSplit[0]+'\t'+lineSplit[1]+'\t'+lineSplit[2] +
                   ↪    '\t'+lineSplit[3]+'\t'+lineSplit[4]+'\n';
129            f.write(grandString)
130            print grandString
131        if n == numberofSoluteAtoms+2:
132            f.write(' $END\n\n $EFRAG\nCOORD=CART  POSITION=OPTIMIZE\n')
133            print 'Now Writing Solvent:\n'
134        #now start doing solvent - (need to add fragment number and atom labels)
135        startPointOfSolvent=numberofSoluteAtoms+2
136        if n in range(startPointOfSolvent,
           ↪    startPointOfSolvent+numberOfAllSolventsAtoms+1):
137            #atomlabel = O1, H2, H3 from O, H, H
138            if atomLabel%numberofSoventAtoms==1:
139                grandString='FRAGNAME=H2ODFT ! '+str(fragmentNumber)+'\n'
140                f.write(grandString)
141                print grandString
142                fragmentNumber+=1;
143                atomLabel%=numberofSoventAtoms
144            lineSplit=line.split()
145            lineSplit.insert(1,str(atomLabel))
146            atomLabel+=1
147            #convert coordinates to 10 decimals (add zeros if need be)
148            for index in [2,3,4]:
149                lineSplit[index]=float(lineSplit[index])
150                lineSplit[index]=format(lineSplit[index],'.10f')
151            grandString=' '+lineSplit[0]+lineSplit[1]+'\t'+lineSplit[2] +
                   ↪    '\t'+lineSplit[3]+'\t'+lineSplit[4]+'\n';
152            f.write(grandString)
153            print grandString
154    #close the inp with £END
155    f.write(' $END\n')
156    ###############################################################
```

MD Geometries extraction

One of the reasons, an MD run might fail is if solute molecule is pushed out of the water sphere. This script 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).

```python
#############################################################
### 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
```

```python
34  previousGrandString=''
35  collectionStarted=False
36  time=''
37  #1 is for cartesian line (useless), then 1 in 3(n+1) is for fragment H2O line
    ↪  (also useless)
38  numberOfLinesToBecollected=numberofSoluteAtoms+1+numberOfSolventMolecules*(numberofSoventAtoms
39
40
41  #number of molecules so far
42  timeCount=0
43  #total number of atoms (solute + solvent) - used later in checking if file is
    ↪  complete
44  atomCount=0
45  #define functions here
46  lineSinceTimeIsFound=0;
47  #do an input of solvent, solute atoms
48  molList=[]
49  #for printing time
50  def printTime (thisLine):
51      lineComponents=thisLine.split();
52      timeString=str(lineComponents[3]);
53      print "Analyzing t = "+timeString+" fsec\n"
54  #to determine if line should be collected -
55  def shouldCollect():
56      #only check if collection is in progress - if it is, then continue to finish
        ↪  collecting the lines
57      #collectionStarted is determined when ' QM ATOM COORDINATES (ANG)'  is found
58      if collectionStarted:
59          #from first solute atom to the last fragment atom
60          if (atomCount>=0 and atomCount<numberOfLinesToBecollected):
61              return True;
62          else:
63              return False;
64      else:
65          return False;
66  # only write when atomCount==numberOfLinesToBecollected
67  def shouldWrite():
68      #only check if collection is in progress
69      if collectionStarted:
70          #solute
71          if (atomCount==numberOfLinesToBecollected):
72              return True;
73          else:
74              return False;
75      else:
76          return False;
77
```

```python
78   #Even now I still don't understand why GAMESS duplicate system geometry for a
     ↪    step twice in the log file
79   #This is written to prevent duplication of geometry in the xyz-movie file
80   def moleculeIsNotADuplication(currentMoleculeToBeWritten):
81           # to compare previously stored geometry and a new one is tricky bc each
                 ↪    string has different lengths
82           # there must be a better of doing this - note for possible place for
                 ↪    improvement
83           #current the speed is quite slow probably due to this step
84       halfSize=int(len(previousGrandString)/2)
85       threeQuartersSize=int(len(previousGrandString)*3/4)
86       if previousGrandString=='':
87           return True;
88       if (currentMoleculeToBeWritten[halfSize:threeQuartersSize] not in
         ↪    previousGrandString[halfSize-1:threeQuartersSize+1]):
89           return True;
90       else:
91           return False;
92
93   #clear output.xyz
94   f = open(output, 'w');
95   f.write('')
96   #open input
97   f1=open(input)
98   #enumerate gets data in line - line and line index - n
99   #readlines() is eliminated because it creates a huge array and python cannot
     ↪    handle it when log file get very large
100  #using for line in... alleviate the burden on memory and actually speed up the
     ↪    process
101  for line in f1:
102      #this keyword is usually before coordinate
103      grandString=''
104      #find out if checking for collectionStarted is needed
105      if shouldCollect():
106          #split line
107          lineSplit=line.split()
108          atomCount+=1;
109          #append to molList
110          molList.append(lineSplit)
111      # if this then start collecting
112      elif ' QM ATOM COORDINATES (ANG)' in line:
113          collectionStarted=True
114      #lastly, if none of the above, then find and print time
115      elif  ' *** AT T=' in line:
116          time=str(line)
117          printTime(line);
118
```

```python
119    if (shouldWrite()):
120        atomCount=atomCount-(numberOfSolventMolecules+1);
121        #for loop through a ***COPY*** of molList and delete some element from
           ↪  molList!
122        #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
123        for line in list(molList):
124            #if line has 4 elements then it's a coordinate from solvent fragment
               ↪  - we have to drop number behind atom - O1 to O
125            if len (line) == 4:
126                #store string
127                oldString =  line[0]
128                #replacement string
129                newString=''
130                #loop to check if it's a alphabet or not
131                for character in range(len(oldString)):
132                    #do substring of 1 character
133                    subString = oldString[character:character+1]
134                    #check if it's an alphabet - yes? then add to newString
135                    if subString.isalpha():
136                        newString = newString + subString
137                #replace 'O1' with 'O'
138                line[0]=newString
139            #if it's 5 then it's solute coordinate - we have to get rid of atomic
               ↪  number behind atomic representation
140            elif len(line) == 5:
141                # 'N 7.0 ...' will become 'N ...'
142                del line[1]
143            #the rest are crap - just remove it out of the line
144            else:
145                #there's a reason why this is remove - not del - since we are
                   ↪  iterating if we delete using index we are gonna be screwed
146                molList.remove(line)
147        #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
148        if len(molList) != atomCount:
149            print 'error'
150        #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
151        grandString=grandString+str(atomCount)+'\n'+str(timeCount)+'\n'
152        #loop tho molList to add data - molList = [['N','1','1','1'],['C'...],...
           ↪  ] And element = ['N','1','1','1']
153        for element in molList:
154            #loop through element in molList data = 'N','1','1','1'
```

```
155             for data in element:
156                 #add to grandString and don't forget tab, return
157                 grandString=grandString+data+'\t'
158             #end one screenshot with a return
159             grandString=grandString+'\n'
160         #open animate.xyz for writing
161         if moleculeIsNotADuplication(grandString):
162             with open(output, 'a') as f:
163                 f.write(grandString)
164                 #add one to timeCount because we already write grandString
165                 timeCount=timeCount+1
166         #reset all values after writing
167         atomCount=0;
168         molList=[]
169         collectionStarted=False;
170         previousGrandString=str(grandString)
171 f.close()
172 #sanity check
173 print 'Done. Extract ' + str(timeCount) + ' snapshots total.'
174
```

Plot Potential Energy of MD run

This 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.

```
1  ##############################################################
2  ### Use this to plot energy vs time to see if MD has     ###
3  ### run its course. Generates: csv of PE and             ###
4  ### temperature vs time, pdf of the plot                 ###
5  ##############################################################
6
7  import matplotlib
8  matplotlib.use('Agg')
9  import matplotlib.pyplot as plt
10 import csv
11 import sys
12 import string
13
```

```python
14   #call as plotEnergyMD3.py inputfile
15   #for asking what the input in terminal should be
16   try:
17           if str(sys.argv[1])=='?':
18                   print '\nCall function as: plotEnergyMD.py input.log   \n'
19                   sys.exit()
20           else:
21                   input=str(sys.argv[1])
22   except IndexError:
23       print '\n!!!Input command Error. Call function as: plotEnergyMD.py input.log
         ↪   \n'
24       sys.exit()
25   output=str(input) + '_energies.csv'
26
27   #initiate variables
28   lineBwTimeAndEnergy=0;
29   lineBwTimeAndTemp=0;
30   lineCountFromTime=0;
31   collectionStarted=False
32   foundTime=False
33   foundPE=False
34   foundTemp=False
35   firstTime=True
36   grandString=''
37
38   #############################
39   #functions
40
41   #check if time is in line
42   def shouldCollectTime(line):
43       #only check if collection is in progress
44       if ' *** AT T=' in line:
45           return True;
46       else:
47           return False;
48
49   # for printing time so one can keep track of the progress
50   def printTime (thisLine):
51       lineComponents=thisLine.split();
52       timeString=str(lineComponents[3]);
53       print "Analyzing t = "+timeString+" fsec\n"
54
55   #Are we currently looking potential energy?
56   def shouldCollectPE(line,reference,currentLine):
57       #check if line bw time and energy is known - this is written as reference
58       if reference>0:
59           if currentLine==reference:
```

```python
60              return True
61          else:
62              return False
63      #if reference is not known, then it needs to be found by finding string POT
            ↪   EN...
64      elif reference==0:
65          if '    POT  ENERGY' in line:
66              reference=int(currentLine)
67              return True;
68          else:
69              return False;
70  #Are we currently looking Temp?
71  def shouldCollectTemp(line,reference,currentLine):
72      #check if line bw time and energy is known - reference
73      if reference>0:
74          if currentLine==reference:
75              return True
76          else:
77              return False
78      #if not then it needs to be found by searching for the string TEMPER...
79      elif reference==0:
80          if '    TEMPER' in line:
81              reference=int(currentLine)
82              return True;
83          else:
84              return False;
85
86  #once everything is found, we should write down before moving on to the next
        ↪   snapshot
87  def shouldWrite():
88      #only check if collection is in progress
89      if foundTime:
90          if foundPE:
91              if foundTemp:
92                  return True;
93              else:
94                  return False
95          else:
96              return False;
97      else:
98          return False;
99
100 ############################
101 #open csv and prepare for writing
102 f = open(output, 'w');
103 f.write('')
104
```

```python
105  #open input
106  f1=open(input)
107  print 'finding patterns...'
108  #avoid using readlines() so there'll be no problem with large files
109  for line in f1:
110      #time keyword is before PE, PE is before Temp so the search should be in this
         ↪   order in order to be most efficient
111      #find out if collection is needed
112      if not foundTime:
113          if shouldCollectTime(line):
114              printTime(line)
115              #split line
116              lineComponents=line.split();
117              #append time (split) to string
118              #split line using space - sample(*** AT T=         10.00 FSEC, THIS
                 ↪   RUN'S STEP NO.=      10)
119              #this will be split to ['***', 'AT', 'T=', '10.00'...] -time =
                 ↪   element 4
120              grandString=grandString+str(lineComponents[3]);
121              foundTime=True;
122      elif not foundPE:
123          if shouldCollectPE(line,lineBwTimeAndEnergy,lineCountFromTime):
124              #append time (split) to string
125              #split line using space - sample(    POT   ENERGY      =
                 ↪   -1.804578585E+05 KCAL/MOL)
126              #this will be split to [..., '=', '-1.804578585E+05'...] -time =
                 ↪   element 4
127              lineComponents=line.split();
128              grandString=grandString+','+str(lineComponents[3])
129              foundPE=True
130      elif not foundTemp:
131          if not firstTime:
132              if shouldCollectTemp(line,lineBwTimeAndTemp,lineCountFromTime):
133                  #append time (split) to string
134                  #split line using space - sample(    TEMPERATURE      =
                     ↪   349.98666547 K)
135                  #this will be split to [..., '=', '-349.98666547'...] -time =
                     ↪   element 3
136                  lineComponents=line.split();
137                  grandString=grandString+', '+str(lineComponents[2])+'\n';
138                  foundTemp=True
139          #problem with this is - the first snapshot's temperature is not given in
             ↪   the log file
140          #set Temp to 0 to indicate the beginning
141          if firstTime:
142              grandString=grandString+',0\n';
143              #once append, turn off the boolean
```

```python
144                firstTime=False
145                foundTemp=True
146        #write after all data is collected for one snapshot
147        if shouldWrite():
148            with open(output, 'a') as f:
149                f.write(grandString)
150            #reset the variables
151            lineCountFromTime=0;
152            collectionStarted=False
153            foundTime=False
154            foundPE=False
155            foundTemp=False
156            grandString=''
157        lineCountFromTime+=1;
158    #finish writing csv
159    f.close()


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



173    #plot
174    x = Time
175    y1 = Energy
176    y2 = Temp

178    fig, ax1 = plt.subplots()

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

184    ax1.set_xlabel('Time (fs)')
185    ax1.set_ylabel('Potential Energy (KCal/mol)', color='g')
186    ax1.ticklabel_format(axis='y', style='sci', scilimits=(-2,2), useOffset=False)
187    ax2.set_ylabel('System Temperature (K)', color='b')
188    plt.title(r'      Potential Energy vs Time for the MD File: ' + str(input))
189    #Saving to pdf gives better resolution - picture is saved to vector
190    #there is a room for improvement especially these energy plots which look very
        ↪    mediocre
```

```
191  plt.savefig(str(input) + '_EnergyPlot.pdf', format='pdf')
```

Find The Most Equilibrated Period

Prepare TDDFT input

Pull Excited State Energies from TDDFT Log Files

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)

```
1   # run type = MD, with functional = CAMB3LYP, COORD = UNIQUE is important
2   $CONTRL SCFTYP=RHF RUNTYP=MD COORD=UNIQUE
3      DFTTYP=CAMB3LYP MAXIT=200 ICHARG=0 MULT=1 $END
4   # MD is recording every 10 frames with default 1 frame =10 fs
5   # 25 degree celcius, RSTEMP is on for keeping the temp ~ +/-25
6   # ssbp is on with default SForce value and radius estimated from prepareMD2.py
7   $MD KEVERY=10 PROD=.T. NVTNH=2 MBT=.T. MBR=.T.
8      BATHT=298 RSTEMP=.T. DTEMP=25 NSTEPS=50000
9      SSBP=.T. SFORCE=1.0 DROFF=12.0632116659 $END
10  #########1#########
11  ##################
12  # dispersion correction is off
13  $DFT DC=.F. $END
14  # memory requested at each node =1000 million words
15  # memory reserved for communication = 1000 million words
16  $SYSTEM MWORDS=1000 MEMDDI=1000 $END
17  $SCF DIRSCF=.T. $END
18  # Basis set = 6-31+(2d,p)
19  $BASIS GBASIS=N31 NGAUSS=6 NDFUNC=2 NPFUNC=1
20      DIFFS=.TRUE. POLAR=POPN31 $END
21  # solute geometry - C1 1 = symmetry data
22  $DATA
```

```
23   MD INPUT for aniline32
24   C1 1
25     #########2#########
26   N        7.0         -2.3128100000        -0.0046000000        -0.0894530000
27   C        6.0         -0.9197160000        -0.0031280000        -0.0360090000
28   C        6.0         -0.2076150000         1.2004070000        -0.0355880000
29   C        6.0          1.1800230000         1.1968760000        -0.0402490000
30   C        6.0          1.8885400000         0.0002120000        -0.0451380000
31   C        6.0          1.1826030000        -1.1982380000        -0.0439250000
32   C        6.0         -0.2049580000        -1.2053990000        -0.0404060000
33   H        1.0         -0.7444290000        -2.1464580000        -0.0434230000
34   H        1.0          1.7166170000        -2.1417620000        -0.0464910000
35   H        1.0          2.9715860000         0.0014800000        -0.0479970000
36   H        1.0          1.7114190000         2.1418420000        -0.0377940000
37   H        1.0         -0.7497420000         2.1399280000        -0.0298190000
38   H        1.0         -2.7544730000         0.8437050000         0.2369240000
39   H        1.0         -2.7570450000        -0.8248640000         0.2993670000
40     ##################
41     $END
42
43     # solvent geometry in EFP1 (EFP2 is still not available)
44     #########3#########
45     $EFRAG
46   COORD=CART   POSITION=OPTIMIZE
47   FRAGNAME=H2ODFT ! 1
48     O1          1.7760990000         4.8390610000        -2.1049530000
49     H2          0.9224740000         4.4173920000        -2.2628490000
50     H3          2.4128200000         4.1148590000        -2.1441810000
51   FRAGNAME=H2ODFT ! 2
52     O1          3.6783070000         3.8351060000         0.8717800000
53     H2          3.6020030000         4.1116030000         1.7932670000
54     H3          3.4138960000         4.6126330000         0.3648680000
55     .
56     .
57     .
58   FRAGNAME=H2ODFT ! 32
59     O1          3.7691430000        -1.4091250000        -4.0319510000
60     H2          2.8756120000        -1.5562470000        -4.3656710000
61     H3          3.6686840000        -1.3995500000        -3.0721400000
62     $END
63     ##################
```

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.

```
 1    .
 2    .
 3    .
 4     #time = 38960 fs
 5    ===== MD DATA PACKET =====
 6    NAT=        14 NFRG=        32 NQMMM=          0
 7    TTOTAL=      38960.00 FS      TOT. E=        -180710.543716 KCAL/MOL
 8    POT. E=          -180783.917894 KCAL/MOL   BATHT=              298.000000
 9    KIN. E=               73.374179   TRANS KE=  44.012966   ROT KE=   29.361213 KCAL/MOL
10    ----- QM PARTICLE COORDINATES FOR $DATA GROUP -----
11     #########2#########
12    N          7.0        3.8554852781        -1.5814882196        -3.7440432660
13    C          6.0        2.9445799802        -1.6872198850        -4.7513152920
14    .
15    .
16    .
17    H          1.0        3.8827088558        -0.7337182877        -3.1704671239
18     #################
19    ----- EFP PARTICLE COORDINATES FOR $EFRAG GROUP -----
20     #########3#########
21     $EFRAG
22    COORD=CART POSITION=OPTIMIZE
23    FRAGNAME=H2ODFT   !    1
24    O1                     0.4899097683         5.6815060052         1.3332597175
25    H2                    -0.1774491436         5.6953737679         2.0005813955
26    H3
27    .
28    .
29    .
30    FRAGNAME=H2ODFT   !   32
31    O1                     1.8397452656        -1.0633683830         1.4432099580
32    H2                     2.6999380304        -1.1499324935         1.8219625372
33    H3                     1.3989608369        -0.3812464183         1.9241435987
34     $END
35     #################
36        GRADIENT DATA (NOT USED BY RESTARTS)...
37    FRAGMENT #      1   H2ODFT
38    .
39    .
```

```
40    .
41    ----- RESTART VELOCITIES FOR $MD GROUP -----
42
43     #########1#########
44     $MD READ=.TRUE. MBT=.FALSE. MBR=.FALSE. TTOTAL=    3.90E-11
45     MDINT= VVERLET     DT= 0.10E-14 NVTNH= 2  NSTEPS=    11040
46     RSTEMP=.T. DTEMP=     25.00 LEVERY= 50000
47     RSRAND=.F. NRAND=  1000 NVTOFF=  0 JEVERY=      10
48     PROD=.T.   KEVERY=     10 DELR=   0.020
49    Batht(1)=298.00
50     SSBP=.T.    SFORCE=   1.0 DROFF= 12.1*****
51    TVELQM(1)=      ! QM ATOM TRANS. VELOCITIES (BOHR/PS) !
52      -1.436664655E+00  -1.014912632E+00   1.731488079E+01
53    .
54    .
55    .
56      -4.714191860E+00  -3.065697154E+00   1.306355299E+01
57    TVEL(1)=      ! EFP TRANSLATIONAL VELOCITIES (BOHR/PS) !
58      -9.879998843E+00  -1.286351783E+01  -1.625361337E-01
59    .
60    .
61    .
62       4.573922683E+00   9.602851076E+00   6.875495095E+00
63    QUAT(1)=      ! EFP QUATERNIONS !
64       6.658493597E-01  -7.408965451E-01   1.545516153E-02   8.647587874E-02
65    .
66    .
67    .
68       8.096376474E-01   4.753557554E-01   1.423717372E-01   3.134550593E-01
69    RVEL(1)=      ! EFP ANGULAR VELOCITY (RAD/PS) !
70       1.392338986E+01  -1.145718732E+01  -1.039863492E+01
71    .
72    .
73    .
74      -1.217796312E+01   2.229922184E+00   2.270460047E+01
75    QUAT1D(1)=       ! EFP QUATERNION 1ST DERIV. !
76       4.237107071E+12   4.149735474E+12   8.424472510E+12   1.422917746E+12
77    .
78    .
79    .
80       5.909828315E+12  -1.125106858E+13  -8.408659179E+11   2.179439222E+12
81    QUAT2D(1)=       ! EFP QUATERNION 2ND DERIV. !
82      -6.427981808E+26  -4.203604186E+26  -7.365308723E+25   1.102236012E+26
83    .
84    .
85    .
86       8.379527677E+26   4.204660564E+26  -1.448608909E+27  -2.676740098E+27
```

```
87   $END
88   ##################
89
90   #time = 38970 fs
91   ===== MD DATA PACKET =====
92   NAT=       14 NFRG=       32 NQMMM=       0
93   TTOTAL=    38970.00 FS    TOT. E=      -180715.284973 KCAL/MOL
94   POT. E=        -180783.048242 KCAL/MOL  BATHT=              298.000000
95   KIN. E=             67.763269  TRANS KE=  41.754223  ROT KE=  26.009046 KCAL/MOL
96   ----- QM PARTICLE COORDINATES FOR $DATA GROUP -----
97   N           7.0         3.8518322659        -1.6019379232        -3.6477010146
98   C           6.0         2.9622328474        -1.6965581751        -4.7374485057
99   C           6.0         2.3084030907        -2.8973077207        -5.0455517014
100  C           6.0         1.5620995753        -2.9996398440        -6.2552765267
101  .
102  .
103  .
```

TDDFT Input File

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

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

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

```
22  C          6.0        0.6209366009        6.2361805033        -2.7812041720
23  C          6.0        0.1647443916        6.2506143410        -4.1120467466
24  C          6.0        1.0257966066        6.0302470214        -5.1619968306
25  C          6.0        2.3206871944        5.6013436130        -4.8191336787
26  C          6.0        2.7816701943        5.6319869564        -3.5109071785
27  H          1.0        3.7174658997        5.1792167910        -3.3730537148
28  H          1.0        3.0442819748        5.3626012085        -5.5970784678
29  H          1.0        0.6964441939        6.0626184504        -6.2108207757
30  H          1.0       -0.8806713295        6.5494661522        -4.2694627361
31  H          1.0       -0.0805914586        6.4688905479        -2.0333995184
32  H          1.0        1.7323956480        5.6459040114        -0.4329519914
33  H          1.0        3.3564486514        5.6102990678        -1.0242941608
34   $END
35
36   # solvent geometry in EFP1 (EFP2 is still not available)
37   $EFRAG
38  COORD=CART  POSITION=OPTIMIZE
39  FRAGNAME=H2ODFT ! 1
40  O1         2.335993939511      3.751856628604      1.427418842826
41  H2         1.439322965739      3.833168541986      1.710699607266
42  H3         2.854351938959      3.613582975500      2.203990690907
43  FRAGNAME=H2ODFT ! 2
44  O1         3.266753260182      2.613267395174      3.808546039622
45  H2         3.514552920456      1.768271678250      3.468757839439
46  H3         3.822697636667      2.780022240614      4.552855871622
47  .
48  .
49  .
50  FRAGNAME=H2ODFT ! 32
51  O1        -0.041331901955     -3.906598195206      1.282021099515
52  H2        -0.790424236756     -4.406259423929      1.565001283174
53  H3        -0.374908444181     -3.111491773717      0.898079798177
54   $END
```

Acknowledgement

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.

This material is available free of charge via the Internet at `http://pubs.acs.org/`.