Supporting information: "Excitonic couplings on Solubilised Light Harvesting Complex II: Changeling the ideal dipole coupling approximation from TDDFT calculations"

June 29, 2017

P. López-Tarifa[a], Nicoletta Liguori[b], Naudin van den Heuvel[c], Roberta Croce[b], and Lucas Visscher[a]

1 DFT assessment

In order to select the best exchange and correlation (XC) functionals for reproducing the spectroscopic properties of Chla, Chlb and Lut, different XC functionals implemented in the ADF[1,2] code are assessed. Our final choice is based on the overall performance among all chromophores.

For Chla, results are shown in Fig.(1) where spectra of optimised gas-phase Chla using different XC functionals are compared to the experimental/theoretical work of Milne et al.[3]. In the picture, the range-separated CAMY-B3LYP functional shows best performance compared to the hybrid functional B3LYP and PBE0. This functional predicts a Qy band centred at 605 nm (2.05eV) in good agreement with the one predicted by the theoretical CAM-B3LYP[4] calculations of Milne et al. (608 nm, 2.04eV).

For Lut, XC functional assessment is made according to the performance in predicting the S0→S2 transition. Results are compared to the high-level DFT/MRCI values of reference[5]. For these calculations, we use a Lut structure that is geometrically optimised at the B3LYP/DZP level as is done in the reference. Results for different XC functionals are summarized in Fig.(2) . It is worth to mention that, as expected, all functionals fail at describing Lut transitions other than S0→S2 due to their multiple excitation character. Among the tested functionals, CAMY-B3LYP predicts an excitation energy of 2.5831 eV with an error smaller than 0.05 eV. Considering this result and also good performance in prediction of the Chla spectrum, this functional is used throughout the study.

[a] *Amsterdam Center for Multiscale Modeling, Dep. Theoretical Chemistry, Faculty of Sciences, VU University Amsterdam, De Boelelaan 1083, 1081 HV Amsterdam, The Netherlands; E-mail: p.l.t.lopeztarifa@vu.nl*

[b] *Laboratory of Biophysics of Photosynthesis, Dep. Physics and Astronomy, Faculty of Sciences, VU University Amsterdam, De Boelelaan 1083, 1081 HV Amsterdam, The Netherlands*

[c] *University of Amsterdam, van't Hoff Institute for Molecular Sciences, Science Park 904, 1098 XH Amsterdam, The Netherlands*

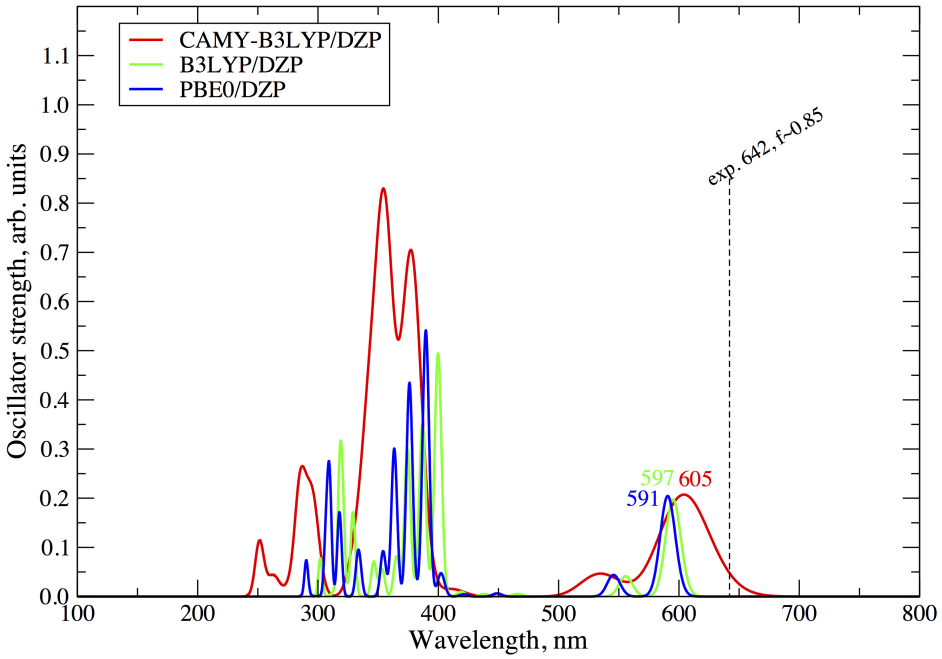

Figure 1 XC functional assessment for optimised gas-phase Chla using a DZP basis set. The dashed line at 642 nm refers to the combined experimental and theoretical value of reference[3].

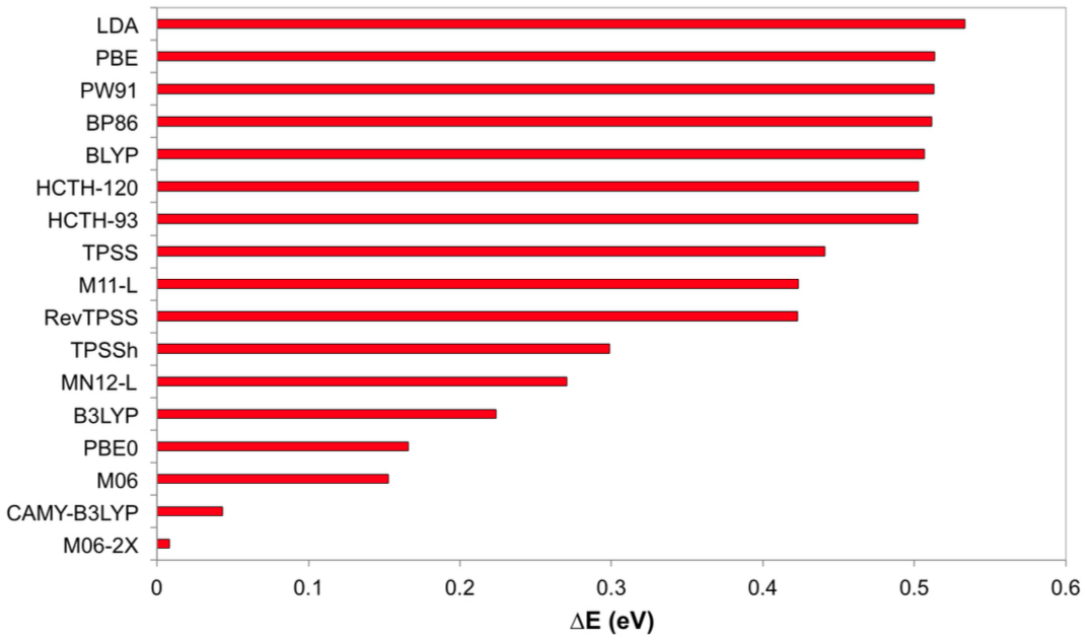

Figure 2 Calculated S0 → S2 transition energy as compared to high level results of reference[5].

2 Phytol chain effect in the Chla electronic spectrum

In agreement with reference[6], no meaningful difference is found in the gas-phase Chla electronic spectrum calculation if the pythol chain is not considered. The chain contains carbons from C2 to C20, according to Liu et al.[7] notation, and C1 is replaced by a methyl group. Fig. (3) shows the comparison for the Chla612 geometry coming from the 1RWT crystal structure[7] using the CAMY-B3LYP XC functional combined with a DZP basis set.

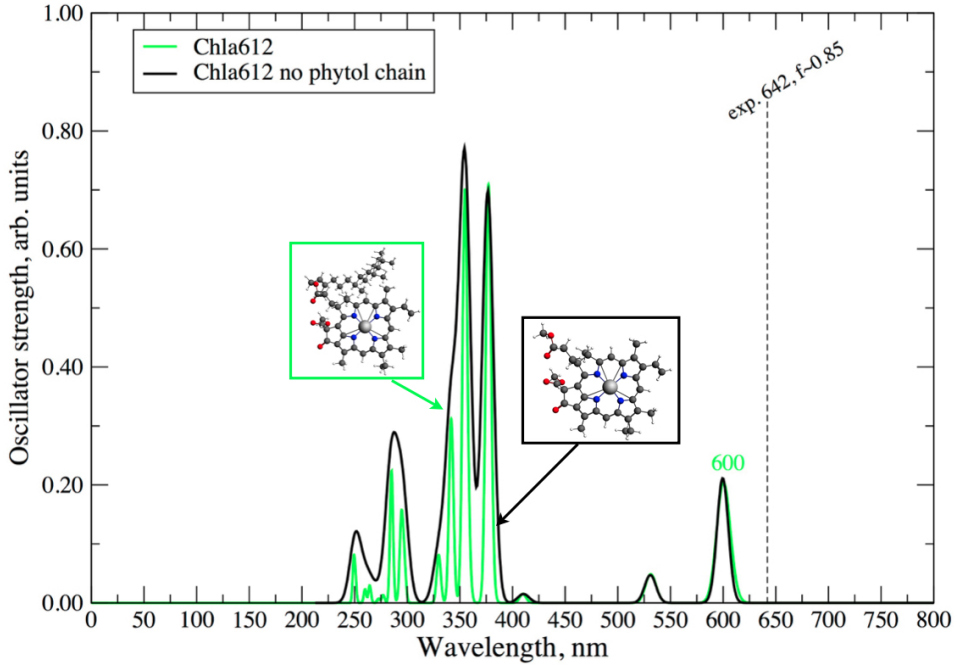

Figure 3 Two superimposed electronic spectra at the CAMY-B3LYP/DZP level of theory of gas-phase Chla612 and gas-phase Chla612 without the phytol chain.

3 Automatisation of input generation using qmworks

The Python-based library qmworks[8] is used to automatise the input generation for DFT single point, FDE and FDEc calculations using the ADF quantum chemistry code. As example, a workflow for the Chlb606-Chlb607 pair is listed below. The starting point is a pdb file generated by the VMD package[9] on a selected xtc GROMACS classical trajectory[10] reported by Liguori et al.[11]. In the generation of the pdb file only the two Chlb chromophores are saved in a total of 100 frames, 50 at the beginning and 50 at the end of the MM simulation. Then the python workflow chopps the file into individual frames and removes the phytol chains of both chlorophylls (using *create_monomers* and *fragment_molecule functions*). Calling the RDKit library[12] by means of the *rdkit_saturate* function, the smiles template given in lines 13,14 is applied to saturated bonds with missed hydrogen atoms. After that, the function *create_jobs* generates all the job dependencies between the isolated monomer (lines 55,56), embedded (lines 59-62), excitation (lines 65-68) and subsystem coupling (line 72) calculations.

```
 1 from functools import partial
 2 from itertools import chain
 3 from noodles import (gather, schedule)
 4 from os.path import join
 5 from plams import Molecule
 6 from qmworks import (Settings, templates, run)
 7 from qmworks import molkit
 8 from qmworks.packages.SCM import adf
 9 from rdkit import Chem
10
11 import argparse
12
13 smile = 'CCC1=C(C2=NC1=CC3=C(C4=C([N-]3)C(=C5[C@H]([C@@H](C(=N5)\
14 C=C6C(=C(C(=C2)[N-]6)C=C)C)C)CCC(=O)OC)[C@H](C4=O)C(=O)OC)C)C=O.[Mg+2]'
15
16 template = Chem.MolFromSmiles(smile)
17
18 def main(pdb_path, splitAt, work_dir):
19
20     monomers = create_monomers(pdb_path, splitAt, work_dir)
21
22     settings = Settings()
23     settings.basis = "DZP"
24     settings.specific.adf.basis.core = "None"
25     settings.specific.adf.symmetry = "Nosym"
26     settings.specific.adf.xc.hybrid = "CAMY-B3LYP"
27     settings.specific.adf.xc.xcfun = ""
28
29     fde_settings = Settings()
30     fde_settings.specific.adf.xc.hybrid = "CAMY-B3LYP"
31     fde_settings.specific.adf.xc.xcfun = ""
32     fde_settings.basis = "DZP"
33     fde_settings.specific.adf.symmetry = "Nosym"
34     fde_settings.specific.adf.fde.PW91k = ""
35     fde_settings.specific.adf.fde.fullgrid = ""
36     fde_settings.specific.adf.allow = "Partialsuperfrags"
37     fde_settings.specific.adf.fde.GGAPOTXFD = "Becke"
38     fde_settings.specific.adf.fde.GGAPOTCFD = "LYP"
39
40     # Read input from molfiles
41     jobs = chain(*list(map(lambda xs: create_jobs(settings, fde_settings, *xs),
42                            enumerate(monomers))))
43
44     run(gather(*jobs), n_processes=2)
45
46 def create_jobs(settings, fde_settings, i, mols):
47     """ Create all the jobs associated to the dimer/monomers"""
48     mol1, mol2 = mols
49     name1 = 'frag1_frame_{}'.format(i)
50     name2 = 'frag2_frame_{}'.format(i)
51     name3 = 'frame_{}'.format(i)
52
53     # Prepare isolated fragments
54     temp_overlay = templates.singlepoint.overlay
55     iso_frag1 = adf(temp_overlay(settings), mol1, job_name='iso_' + name1)
56     iso_frag2 = adf(temp_overlay(settings), mol2, job_name='iso_' + name2)
57
58     # Prepare embedded fragments
59     emb_frag1 = embed_job(fde_settings, iso_frag1, iso_frag2, 'emb_' + name1,
60                           switch=False)
61     emb_frag2 = embed_job(fde_settings, iso_frag2, iso_frag1, 'emb_' + name2,
62                           switch=True)
63
64     # Prepare excitation calculations
65     exc_frag1 = excitations_job(fde_settings, iso_frag1, iso_frag2, emb_frag1, emb_frag2,
66                                 'exc_' + name1, switch=False)
67     exc_frag2 = excitations_job(fde_settings, iso_frag1, iso_frag2, emb_frag2, emb_frag1,
68                                 'exc_' + name2, switch=True)
```

```
69
70      # Run all dependencies and final subsystem calculation
71      name = 'subexc_' + name3
72      subexc = subexc_job(fde_settings, iso_frag1, iso_frag2, exc_frag1, exc_frag2, name, switch=False)
73
74      return [iso_frag1, iso_frag2, emb_frag1, emb_frag2, exc_frag1, exc_frag2, subexc]
75
76  def create_monomers(pdb_path, splitAt, work_dir):
77      """
78      Read the dimers from the pDB, clean, add hydrogens and split into monomers.
79      """
80      # Clean PDB and read it with RDkit
81      mols = split_and_clean_pdb(pdb_path)
82      # Remove Water and Atoms from C2 to C20
83      alkyl_chain = ['C{}'.format(i) for i in range(2, 21)]
84      water = ['OW', 'HW1', 'HW2']
85      names = alkyl_chain + water
86
87      # Read alkyl chain and water from molecules
88      clean_mols = map(partial(remove_atoms, names), mols)
89
90      # Create fragments from the molecules
91      fragments = map(partial(fragment_molecule, splitAt), clean_mols)
92
93      # saturate the monomers
94      fun = lambda ts: (rdkit_saturate(ts[0], ts[1][0], name='monomer1_Hs'),
95                        rdkit_saturate(ts[0], ts[1][1], name='monomer2_Hs'))
96
97      saturated_monomers = map(fun, enumerate(fragments))
98
99      return list(saturated_monomers)
100
101 def fragment_molecule(splitAt, mol):
102     """
103     split a dimer in fragment using an integer index `splitAt`.
104     Note: It assumes that the number of atoms in each monomer is half of
105     the dimers.
106     """
107     emol1 = Chem.EditableMol(mol)
108     emol2 = Chem.EditableMol(mol)
109
110     for x in range(splitAt):
111         emol1.RemoveAtom(0)
112         emol2.RemoveAtom(splitAt)
113
114     return emol1.GetMol(), emol2.GetMol()
115
116 def rdkit_saturate(i, mol, name='Hs'):
117     bs = []
118     for atom in mol.GetAtoms():
119         if atom.GetSymbol() == "Mg":
120             atom.SetNoImplicit(True)
121             for bond in atom.GetBonds():
122                 bs.append([bond.GetBeginAtomIdx(), bond.GetEndAtomIdx()])
123     emol = Chem.EditableMol(mol)
124     for b in bs:
125         emol.RemoveBond(b[0], b[1])
126     newmol = emol.GetMol()
127
128     name_pdb = '{}_{}.pdb'.format(name, i)
129     Chem.MolToPDBFile(newmol, name_pdb)
130
131     newmol = Chem.AllChem.AssignBondOrdersFromTemplate(template, newmol)
132     molHs = molkit.add_prot_Hs(newmol)
133
134     return molHs
135
136 def remove_atoms(names, mol):
137     """
138     Remove `names` from pdb
139     """
140     def index_to_remove(i, at):
141         info = at.GetPDBResidueInfo()
142         name = info.GetName()
143         split_name = name.split()[0]
144
145         return split_name in names
146
147     indexes_atoms = enumerate(mol.GetAtoms())
148     # filter only the atoms to remove
149     index_atoms_to_remove = list(filter(lambda ts: index_to_remove(*ts),
150                                         indexes_atoms))
151     # extract only the index and discard the atoms
152     index_to_delete = list(zip(*index_atoms_to_remove))[0]
153
154     # Editable molecule
155     emol = Chem.EditableMol(mol)
156
157     # the accumulator is mandatory because the index of an atom
158     # changed when other atoms in the molecule are deleted
159     for acc, i in enumerate(index_to_delete):
```

```
160         emol.RemoveAtom(i - acc)
161
162     return emol.GetMol()
163
164 def split_and_clean_pdb(path_pdb):
165     """
166     Clean Pdb then split then in frames and read them using rdkit
167     """
168     with open(path_pdb) as f:
169         xss = f.read()
170
171     # Filter block containing more than 1 element
172     pdbs = filter(lambda x: len(x) > 1, xss.split('END'))
173     frames = map(partial(replace_element_name, ('MG', 'Mg')), pdbs)
174     mols = map(lambda frame: Chem.rdmolfiles.MolFromPDBBlock(frame, sanitize=False), frames)
175     return mols
176
177 def replace_element_name(names, st):
178     """
179     Change the name of the element in the PDB
180     """
181     data = ''
182     old, new = names
183     for l in st.splitlines():
184         if old in l:
185             head, tail = l.split(old)
186             data += (head + new + tail + '\n')
187         else:
188             data += (l + '\n')
189
190     return data
191
192 def add_fragments(job1, job2, switch=False):
193     print("path1:␣", job1.kf.path)
194     print("path2:␣", job2.kf.path)
195     mol_1 = job1.molecule.copy()
196     mol_2 = job2.molecule.copy()
197     for a in mol_1:
198         if not switch:
199             a.fragment = 'frag1'
200         else:
201             a.fragment = 'frag2'
202     for a in mol_2:
203         if not switch:
204             a.fragment = 'frag2'
205         else:
206             a.fragment = 'frag1'
207     m_tot = Molecule()
208     if not switch:
209        m_tot += mol_1 + mol_2
210     else:
211        m_tot += mol_2 + mol_1
212     return m_tot
213
214 @schedule
215 def embed_job(settings, emb_frag, frozen_frag, jobname, switch):
216     """
217     Define different jobs
218     """
219     frag_settings = Settings()
220     m_tot = add_fragments(emb_frag, frozen_frag, switch)
221     if not switch:
222         frag_settings.specific.adf.fragments['frag1'] = emb_frag.kf.path + '␣subfrag=active'
223         frag_settings.specific.adf.fragments['frag2'] = frozen_frag.kf.path + '␣subfrag=active␣type=FDE'
224     else:
225         frag_settings.specific.adf.fragments['frag2'] = emb_frag.kf.path + '␣subfrag=active'
226         frag_settings.specific.adf.fragments['frag1'] = frozen_frag.kf.path + '␣subfrag=active␣type=FDE'
227     return adf(settings.overlay(frag_settings), m_tot, job_name=jobname)
228
229 @schedule
230 def excitations_job(settings, iso1, iso2, emb_frag, frozen_frag, jobname, switch):
231     exc_settings = Settings()
232     if not switch:
233         m_tot = add_fragments(iso1, iso2, switch)
234     else:
235         m_tot = add_fragments(iso2, iso1, switch)
236     s_frag = exc_settings.specific.adf.fragments
237     if not switch:
238         s_frag['frag1'] = emb_frag.kf.path + '␣subfrag=active'
239         s_frag['frag2'] = frozen_frag.kf.path + '␣subfrag=active␣type=FDE'
240     else:
241         s_frag['frag2'] = emb_frag.kf.path + '␣subfrag=active'
242         s_frag['frag1'] = frozen_frag.kf.path + '␣subfrag=active␣type=FDE'
243     exc_settings.specific.adf.excitations.onlysing = ""
244     exc_settings.specific.adf.excitations.lowest = "5"
245     return adf(settings.overlay(exc_settings), m_tot, job_name=jobname)
246
247
248 @schedule
249 def subexc_job(settings, iso_frag1, iso_frag2, emb_frag, frozen_frag, jobname, switch):
250     subexc_settings = Settings()
```

```
    m_tot = add_fragments(iso_frag1, iso_frag2, switch)
    s_frag = subexc_settings.specific.adf.fragments
    s_frag['frag1'] = emb_frag.kf.path + ' subfrag=active'
    s_frag['frag2'] = frozen_frag.kf.path + ' subfrag=active type=FDE'
    subexc_settings.specific.adf.excitations.onlysing = ""
    subexc_settings.specific.adf.excitations.lowest = "5"
    subexc_settings.specific.adf.subexci.cthres = "10000.00"
    subexc_settings.specific.adf.subexci.sfthres = "0.00010000"
    subexc_settings.specific.adf.subexci.couplblock = ""
    subexc_settings.specific.adf.subexci.cicoupl = ""
    subexc_settings.specific.adf.subexci.tda = ""
    return adf(settings.overlay(subexc_settings), m_tot, job_name=jobname)

def read_cmd_line(parser):
    """
    Parse Command line options.
    """
    args = parser.parse_args()

    work_dir = args.w if args.w is not None else '.'

    return args.pdb, args.s, work_dir

if __name__ == "__main__":
    msg = " fde_qmworks -pdb <path/to/pdb>"

    parser = argparse.ArgumentParser(description=msg)
    parser.add_argument('-pdb', required=True, help='path to PDB')
    parser.add_argument('-w', help='path to PDB')
    parser.add_argument('-s', help='index to split dimer', required=True,
                        type=int)

    main(*read_cmd_line(parser))
```

4 Geometrical parameters of the LHCII crystal structure assessment

Table 1 Geometrical parameters of the Chla611-Chla612, Chlb606-Chlb607 and Chla612-Lut620 chromophore pairs of the LHCII crystal structure calculated by FDE and FIX-IDA approximations. Distance is given in Å and angles in degrees.

Pair	r	$\alpha^{FDE}_{\vec{\mu}_{611},\vec{\mu}_{612}}$	$\alpha^{FIX}_{\vec{\mu}_{611},\vec{\mu}_{612}}$	$\beta^{FDE}_{\vec{\mu}_{611},\vec{r}}$	$\beta^{FIX}_{\vec{\mu}_{611},\vec{r}}$	$\gamma^{FDE}_{\vec{\mu}_{612},\vec{r}}$	$\gamma^{FIX}_{\vec{\mu}_{612},\vec{r}}$	$\lvert k_{FDE}\rvert$	$\lvert k_{FIX}\rvert$
Chla611-Chla612	9.65	133.14	142.92	144.28	143.77	24.80	18.26	1.54	1.50
	r	$\alpha^{FDE}_{\vec{\mu}_{606},\vec{\mu}_{607}}$	$\alpha^{FIX}_{\vec{\mu}_{606},\vec{\mu}_{607}}$	$\beta^{FDE}_{\vec{\mu}_{606},\vec{r}}$	$\beta^{FIX}_{\vec{\mu}_{606},\vec{r}}$	$\gamma^{FDE}_{\vec{\mu}_{607},\vec{r}}$	$\gamma^{FIX}_{\vec{\mu}_{607},\vec{r}}$	$\lvert k_{FDE}\rvert$	$\lvert k_{FIX}\rvert$
Chlb606-Chlb607	9.46	42.14	27.64	53.77	63.16	95.15	87.56	0.90	0.83
	r	$\alpha^{FDE}_{\vec{\mu}_{612},\vec{\mu}_{620}}$	$\alpha^{FIX}_{\vec{\mu}_{612},\vec{\mu}_{620}}$	$\beta^{FDE}_{\vec{\mu}_{612},\vec{r}}$	$\beta^{FIX}_{\vec{\mu}_{612},\vec{r}}$	$\gamma^{FDE}_{\vec{\mu}_{620},\vec{r}}$	$\gamma^{FIX}_{\vec{\mu}_{620},\vec{r}}$	$\lvert k_{FDE}\rvert$	$\lvert k_{FIX}\rvert$
Chla612-Lut620	6.02	101.67	121.88	31.90	32.96	98.45	108.59	0.17	0.27

References

[1] Amsterdam density functional program. Theoretical Chemistry, Vrije Universiteit, Amsterdam, http://www.scm.com.

[2] G. T. Velde, F. M. Bickelhaupt, E. J. Baerends, C. F. Guerra, S. J. A. van Gisbergen, J. Snijders and T. Ziegler, *J. Comput. Chem.*, 2001, **22**, 931.

[3] B. F. Milne, Y. Toker, A. Rubio and S. B. Nielsen, *Angew. Chem. Int. Ed.*, 2015, **54**, 2170.

[4] T. Yanai, D. P. Tew and N. C. Handy, *Chem. Phys. Lett.*, 2004, **393**, 51.

[5] O.Andreussi, S. Knecht, C. Marian, J. Kongsted and B. Mennucci, *J. Chem. Theor. Comput.*, 2015, **11**, 655.

[6] J. Chmeliov, W. P. Bricker, C. Lo, E. Jouin, L. Valkunas, A. V. Ruband and C. D. P. Duffy, *Phys.Chem.Chem.Phys.*, 2015, **17**, 15857.

[7] Z. Liu, H. Yan, K. Wang, T. Kuang, J. Zhang, L. Gui, X. An and W. Chang, *Nature*, 2004, **428**, 287–292.

[8] *qmworks: Open-source python library for workflow automatisation in quantum chemistry codes*, http://www.github.com/SCM-NV/qmworks.

[9] W. Humphrey, A. Dalke and K. Schulten, *J. Molec. Graphics,*, 1996, **14**, 33.

[10] V. D. Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark and H. J. Berendsen, *J. Comput. Chem.*, 2005, **26**, 1701.

[11] N. Liguori, X. Periole, S. J. Marrink and R. Croce, *Scientific Reports*, 2015, **5**, vol 1.

[12] *RDKit: Open-source cheminformatics*, http://www.rdkit.org.