```python
"""
Helper classes and functions for molecular properties requiring
solution of CPHF equations.
"""

__authors__   =  "Daniel G. A. Smith"
__credits__   = ["Daniel G. A. Smith", "Eric J. Berquist"]

__copyright__ = "(c) 2014-2017, The Psi4NumPy Developers"
__license__   = "BSD-3-Clause"
__date__      = "2017-8-30"

import time
import numpy as np
np.set_printoptions(precision=5, linewidth=200, suppress=True)
import psi4

import os.path
import sys
dirname = os.path.dirname(os.path.abspath(__file__))
sys.path.append(os.path.join(dirname, '../../Self-Consistent-Field'))
from helper_HF import DIIS_helper


class helper_CPHF(object):

    def __init__(self, mol, numpy_memory=2):

        self.mol = mol
        self.numpy_memory = numpy_memory

        # Compute the reference wavefunction and CPHF using Psi
        scf_e, self.scf_wfn = psi4.energy('SCF', return_wfn=True)

        self.C = self.scf_wfn.Ca()
        self.Co = self.scf_wfn.Ca_subset("AO", "OCC")
        self.Cv = self.scf_wfn.Ca_subset("AO", "VIR")
        self.epsilon = np.asarray(self.scf_wfn.epsilon_a())

        self.nbf = self.scf_wfn.nmo()
        self.nocc = self.scf_wfn.nalpha()
        self.nvir = self.nbf - self.nocc

        # Integral generation from Psi4's MintsHelper
        self.mints = psi4.core.MintsHelper(self.scf_wfn.basisset())

        # Get nbf and ndocc for closed shell molecules
        print('\nNumber of occupied orbitals: %d' % self.nocc)
        print('Number of basis functions: %d' % self.nbf)

        # Grab perturbation tensors in MO basis
        nCo = np.asarray(self.Co)
        nCv = np.asarray(self.Cv)
        self.tmp_dipoles = self.mints.so_dipole()
        self.dipoles_xyz = []
        for num in range(3):
            Fso = np.asarray(self.tmp_dipoles[num])
            Fia = (nCo.T).dot(Fso).dot(nCv)
            Fia *= -2
            self.dipoles_xyz.append(Fia)

        self.x = None
        self.rhsvecs = None

    def run(self, method='direct', omega=None):
        self.method = method
        if self.method == 'direct':
```

```
68              if not omega:
69                  self.solve_static_direct()
70              else:
71                  self.solve_dynamic_direct(omega=omega)
72          elif self.method == 'iterative':
73              if not omega:
74                  self.solve_static_iterative()
75              else:
76                  self.solve_dynamic_iterative(omega=omega)
77          else:
78              raise Exception("Method %s is not recognized" % self.method)
79          self.form_polarizability()
80
81      def solve_static_direct(self):
82          # Run a quick check to make sure everything will fit into memory
83          I_Size = (self.nbf ** 4) * 8.e-9
84          oNNN_Size = (self.nocc * self.nbf ** 3) * 8.e-9
85          ovov_Size = (self.nocc * self.nocc * self.nvir * self.nvir) * 8.e-9
86          print("\nTensor sizes:")
87          print("ERI tensor           %4.2f GB." % I_Size)
88          print("oNNN MO tensor       %4.2f GB." % oNNN_Size)
89          print("ovov Hessian tensor  %4.2f GB." % ovov_Size)
90
91          # Estimate memory usage
92          memory_footprint = I_Size * 1.5
93          if I_Size > self.numpy_memory:
94              psi4.core.clean()
95              raise Exception("Estimated memory utilization (%4.2f GB) exceeds numpy_memory \
96                               limit of %4.2f GB." % (memory_footprint, self.numpy_memory))
97
98          # Compute electronic Hessian
99          print('\nForming Hessian...')
100         t = time.time()
101         docc = np.diag(np.ones(self.nocc))
102         dvir = np.diag(np.ones(self.nvir))
103         eps_diag = self.epsilon[self.nocc:].reshape(-1, 1) - self.epsilon[:self.nocc]
104
105         # Form [o,N,N,N] MO tensor, oN^4 cost
106         MO = np.asarray(self.mints.mo_eri(self.Co, self.C, self.C, self.C))
107
108         H = np.einsum('ai,ij,ab->iajb', eps_diag, docc, dvir)
109         H += 4 * MO[:, self.nocc:, :self.nocc, self.nocc:]
110         H -= MO[:, self.nocc:, :self.nocc, self.nocc:].swapaxes(0, 2)
111
112
113         H -= MO[:, :self.nocc, self.nocc:, self.nocc:].swapaxes(1, 2)
114
115         print('...formed Hessian in %.3f seconds.' % (time.time() - t))
116
117         # Invert Hessian (o^3v^3)
118         print('\nInverting Hessian...')
119         t = time.time()
120         Hinv = np.linalg.inv(H.reshape(self.nocc * self.nvir, -1)).reshape(self.nocc, self.nvir,
            ↪  self.nocc, self.nvir)
121         print('...inverted Hessian in %.3f seconds.' % (time.time() - t))
122
123         # Form perturbation response vector for each dipole component
124         self.x = []
125         for numx in range(3):
126             xcomp = np.einsum('iajb,ia->jb', Hinv, self.dipoles_xyz[numx])
127             self.x.append(xcomp.reshape(-1))
128
129         self.rhsvecs = []
130         for numx in range(3):
131             rhsvec = self.dipoles_xyz[numx].reshape(-1)
132             self.rhsvecs.append(rhsvec)
133
134     def solve_dynamic_direct(self, omega=0.0):
```

```
        # Adapted completely from TDHF.py

        eps_v = self.epsilon[self.nocc:]
        eps_o = self.epsilon[:self.nocc]

        t = time.time()
        I = self.mints.ao_eri()
        v_ijab = np.asarray(self.mints.mo_transform(I, self.Co, self.Co, self.Cv, self.Cv))
        v_iajb = np.asarray(self.mints.mo_transform(I, self.Co, self.Cv, self.Co, self.Cv))
        print('Integral transform took %.3f seconds\n' % (time.time() - t))

        # Since we are time dependent we need to build the full Hessian:
        # | A B |      | D  S | |  x |   |  b |
        # | B A |  - w | S -D | | -x | = | -b |

        # Build A and B blocks
        t = time.time()
        A11  = np.einsum('ab,ij->iajb', np.diag(eps_v), np.diag(np.ones(self.nocc)))
        A11 -= np.einsum('ij,ab->iajb', np.diag(eps_o), np.diag(np.ones(self.nvir)))
        A11 += 2 * v_iajb
        A11 -= v_ijab.swapaxes(1, 2)
        A11 *= 2

        B11  = -2 * v_iajb
        B11 += v_iajb.swapaxes(0, 2)
        B11 *= 2

        # Reshape and jam it together
        nov = self.nocc * self.nvir
        A11.shape = (nov, nov)
        B11.shape = (nov, nov)

        Hess1 = np.hstack((A11, B11))
        Hess2 = np.hstack((B11, A11))
        Hess = np.vstack((Hess1, Hess2))

        S11 = np.zeros_like(A11)
        D11 = np.zeros_like(B11)
        S11[np.diag_indices_from(S11)] = 2

        S1 = np.hstack((S11, D11))
        S2 = np.hstack((D11, -S11))
        S = np.vstack((S1, S2))
        S *= omega
        print('Hessian formation took %.3f seconds\n' % (time.time() - t))

        t = time.time()
        Hinv = np.linalg.inv(Hess - S)
        print('Hessian inversion took %.3f seconds\n' % (time.time() - t))

        self.x = []
        self.rhsvecs = []
        for numx in range(3):
            rhsvec = self.dipoles_xyz[numx].reshape(-1)
            rhsvec = np.concatenate((rhsvec, -rhsvec))
            xcomp = Hinv.dot(rhsvec)
            self.rhsvecs.append(rhsvec)
            self.x.append(xcomp)

    def solve_static_iterative(self, maxiter=20, conv=1.e-9, use_diis=True):

        # Init JK object
        jk = psi4.core.JK.build(self.scf_wfn.basisset())
        jk.initialize()

        # Add blank matrices to the jk object and numpy hooks to C_right
        npC_right = []
        for xyz in range(3):
```

```
            jk.C_left_add(self.Co)
            mC = psi4.core.Matrix(self.nbf, self.nocc)
            npC_right.append(np.asarray(mC))
            jk.C_right_add(mC)

        # Build initial guess, previous vectors, diis object, and C_left updates
        self.x = []
        x_old = []
        diis = []
        ia_denom = - self.epsilon[:self.nocc].reshape(-1, 1) + self.epsilon[self.nocc:]
        for xyz in range(3):
            self.x.append(self.dipoles_xyz[xyz] / ia_denom)
            x_old.append(np.zeros(ia_denom.shape))
            diis.append(DIIS_helper())

        # Convert Co and Cv to numpy arrays
        Co = np.asarray(self.Co)
        Cv = np.asarray(self.Cv)

        print('\nStarting CPHF iterations:')
        t = time.time()
        for CPHF_ITER in range(1, maxiter + 1):

            # Update jk's C_right
            for xyz in range(3):
                npC_right[xyz][:] = Cv.dot(self.x[xyz].T)

            # Compute JK objects
            jk.compute()

            # Update amplitudes
            for xyz in range(3):
                # Build J and K objects
                J = np.asarray(jk.J()[xyz])
                K = np.asarray(jk.K()[xyz])

                # Bulid new guess
                X = self.dipoles_xyz[xyz].copy()
                X -= (Co.T).dot(4 * J - K.T - K).dot(Cv)
                X /= ia_denom

                # DIIS for good measure
                if use_diis:
                    diis[xyz].add(X, X - x_old[xyz])
                    X = diis[xyz].extrapolate()
                self.x[xyz] = X.copy()

            # Check for convergence
            rms = []
            for xyz in range(3):
                rms.append(np.max((self.x[xyz] - x_old[xyz]) ** 2))
                x_old[xyz] = self.x[xyz]

            avg_RMS = sum(rms) / 3
            max_RMS = max(rms)

            if max_RMS < conv:
                print('CPHF converged in %d iterations and %.2f seconds.' % (CPHF_ITER,
                ↪  time.time() - t))
                self.rhsvecs = []
                for numx in range(3):
                    rhsvec = self.dipoles_xyz[numx].reshape(-1)
                    self.rhsvecs.append(rhsvec)
                    self.x[numx] = self.x[numx].reshape(-1)
                break

            print('CPHF Iteration %3d: Average RMS = %3.8f  Maximum RMS = %3.8f' %
                  (CPHF_ITER, avg_RMS, max_RMS))
```

```
270
271     def solve_dynamic_iterative(self, omega=0.0, maxiter=20, conv=1.e-9, use_diis=True):
272
273         # Init JK object
274         jk = psi4.core.JK.build(self.scf_wfn.basisset())
275         jk.initialize()
276
277         # Add blank matrices to the JK object and NumPy hooks to
278         # C_right; there are 6 sets of matrices to account for X and Y
279         # vectors separately.
280         npC_right = []
281         for xyz in range(6):
282             jk.C_left_add(self.Co)
283             mC = psi4.core.Matrix(self.nbf, self.nocc)
284             npC_right.append(np.asarray(mC))
285             jk.C_right_add(mC)
286
287         # Build initial guess, previous vectors, diis object, and C_left updates
288         x_l, x_r = [], []
289         x_l_old, x_r_old = [], []
290         diis_l, diis_r = [], []
291         ia_denom_l = self.epsilon[self.nocc:] - self.epsilon[:self.nocc].reshape(-1, 1) - omega
292         ia_denom_r = self.epsilon[self.nocc:] - self.epsilon[:self.nocc].reshape(-1, 1) + omega
293         for xyz in range(3):
294             x_l.append(self.dipoles_xyz[xyz] / ia_denom_l)
295             x_r.append(self.dipoles_xyz[xyz] / ia_denom_r)
296             x_l_old.append(np.zeros(ia_denom_l.shape))
297             x_r_old.append(np.zeros(ia_denom_r.shape))
298             diis_l.append(DIIS_helper())
299             diis_r.append(DIIS_helper())
300
301         # Convert Co and Cv to numpy arrays
302         Co = np.asarray(self.Co)
303         Cv = np.asarray(self.Cv)
304
305         print('\nStarting CPHF iterations:')
306         t = time.time()
307         for CPHF_ITER in range(1, maxiter + 1):
308
309             # Update jk's C_right; ordering is Xx, Xy, Xz, Yx, Yy, Yz
310             for xyz in range(3):
311                 npC_right[xyz][:] = Cv.dot(x_l[xyz].T)
312                 npC_right[xyz + 3][:] = Cv.dot(x_r[xyz].T)
313
314             # Perform generalized J/K build
315             jk.compute()
316
317             # Update amplitudes
318             for xyz in range(3):
319                 # Build J and K objects
320                 J_l = np.asarray(jk.J()[xyz])
321                 K_l = np.asarray(jk.K()[xyz])
322                 J_r = np.asarray(jk.J()[xyz + 3])
323                 K_r = np.asarray(jk.K()[xyz + 3])
324
325                 # Bulid new guess
326                 X_l = self.dipoles_xyz[xyz].copy()
327                 X_r = self.dipoles_xyz[xyz].copy()
328                 X_l -= (Co.T).dot(2 * J_l - K_l).dot(Cv)
329                 X_r -= (Co.T).dot(2 * J_r - K_r).dot(Cv)
330                 X_l /= ia_denom_l
331                 X_r /= ia_denom_r
332
333                 # DIIS for good measure
334                 if use_diis:
335                     diis_l[xyz].add(X_l, X_l - x_l_old[xyz])
336                     X_l = diis_l[xyz].extrapolate()
337                     diis_r[xyz].add(X_r, X_r - x_r_old[xyz])
```

```
                    X_r = diis_r[xyz].extrapolate()
                x_l[xyz] = X_l.copy()
                x_r[xyz] = X_r.copy()

            # Check for convergence
            rms = []
            for xyz in range(3):
                rms_l = np.max((x_l[xyz] - x_l_old[xyz]) ** 2)
                rms_r = np.max((x_r[xyz] - x_r_old[xyz]) ** 2)
                rms.append(max(rms_l, rms_r))
                x_l_old[xyz] = x_l[xyz]
                x_r_old[xyz] = x_r[xyz]

            avg_RMS = sum(rms) / 3
            max_RMS = max(rms)

            if max_RMS < conv:
                print('CPHF converged in %d iterations and %.2f seconds.' % (CPHF_ITER,
                ↪  time.time() - t))
                self.rhsvecs = []
                for numx in range(3):
                    rhsvec = self.dipoles_xyz[numx].reshape(-1)
                    self.rhsvecs.append(np.concatenate((rhsvec, -rhsvec)))
                    self.x.append(np.concatenate((x_l[numx].reshape(-1),
                                                  x_r[numx].reshape(-1))))
                break

            print('CPHF Iteration %3d: Average RMS = %3.8f  Maximum RMS = %3.8f' %
                  (CPHF_ITER, avg_RMS, max_RMS))

    def form_polarizability(self):
        self.polar = np.empty((3, 3))
        for numx in range(3):
            for numf in range(3):
                self.polar[numx, numf] = self.x[numx].dot(self.rhsvecs[numf])

if __name__ == '__main__':
    print('\n')
    print('@test_CPHF running CPHF.py')

    from CPHF import *

    from helper_CPHF import helper_CPHF

    helper = helper_CPHF(mol)

    print('\n')
    print('@test_CPHF running solve_static_direct')

    helper.solve_static_direct()
    helper.form_polarizability()
    assert np.allclose(polar, helper.polar, rtol=0, atol=1.e-5)

    print('\n')
    print('@test_CPHF running solve_static_iterative')

    helper.solve_static_iterative()
    helper.form_polarizability()
    assert np.allclose(polar, helper.polar, rtol=0, atol=1.e-5)

    f = 0.0

    print('\n')
    print('@test_CPHF running solve_dynamic_direct ({})'.format(f))

    helper.solve_dynamic_direct(omega=f)
    helper.form_polarizability()
    assert np.allclose(polar, helper.polar, rtol=0, atol=1.e-5)
```

```
405
406     print('\n')
407     print('@test_CPHF running solve_dynamic_iterative ({})'.format(f))
408
409     helper.solve_dynamic_iterative(omega=f)
410     helper.form_polarizability()
411     assert np.allclose(polar, helper.polar, rtol=0, atol=1.e-5)
412
413     f = 0.0773178
414     ref = np.array([
415         [8.19440121,  0.00000000,  0.00000000],
416         [0.00000000, 12.75967150,  0.00000000],
417         [0.00000000,  0.00000000, 10.25213939]
418     ])
419
420     print('\n')
421     print('@test_CPHF running solve_dynamic_direct ({})'.format(f))
422
423     helper.solve_dynamic_direct(omega=f)
424     helper.form_polarizability()
425     assert np.allclose(ref, helper.polar, rtol=0, atol=1.e-5)
426
427     print('\n')
428     print('@test_CPHF running solve_dynamic_iterative ({})'.format(f))
429
430     helper.solve_dynamic_iterative(omega=f)
431     helper.form_polarizability()
432     assert np.allclose(ref, helper.polar, rtol=0, atol=1.e-5)
```

```
1   """
2   A reference implementation to compute the first dipole
3   hyperpolarizability $\beta$ from a restricted HF reference using the
4   $2n+1$ rule from perturbation theory.
5
6   References:
7   Equations taken from [Karna:1991:487], http://dx.doi.org/10.1002/jcc.540120409
8   """
9
10  __authors__   =  "Eric J. Berquist"
11  __credits__   = ["Eric J. Berquist"]
12
13  __copyright__ = "(c) 2014-2017, The Psi4NumPy Developers"
14  __license__   = "BSD-3-Clause"
15  __date__      = "2017-08-26"
16
17  from itertools import permutations, product
18
19  import numpy as np
20  np.set_printoptions(precision=5, linewidth=200, suppress=True)
21  import psi4
22  from helper_CPHF import helper_CPHF
23
24  # Memory for Psi4 in GB
25  psi4.set_memory('2 GB')
26  psi4.core.set_output_file("output.dat", False)
27
28  mol = psi4.geometry("""
29  O
30  H 1 1.1
31  H 1 1.1 2 104
32  symmetry c1
33  """)
34
35  # Set options for CPHF
36  psi4.set_options({"basis": "aug-cc-pvdz",
37                    "scf_type": "direct",
38                    "df_scf_guess": False,
```

```
                   "e_convergence": 1e-9,
                   "d_convergence": 1e-9})

# Compute the (first) hyperpolarizability corresponding to static
# fields, beta(0;0,0), eqns. (IV-2a) and (VII-4).

helper = helper_CPHF(mol)
# For the 2n+1 rule, the quadratic response starting quantities must
# come from linear response.
helper.run()

na = np.newaxis
moenergies = helper.epsilon
C = np.asarray(helper.C)
Co = helper.Co
Cv = helper.Cv
nbf, norb = C.shape
nocc = Co.shape[1]
nvir = norb - nocc
nov = nocc * nvir
x = np.asarray(helper.x)
ncomp = x.shape[0]
integrals_ao = np.asarray([np.asarray(dipole_ao_component)
                           for dipole_ao_component in helper.tmp_dipoles])

# form full MO-basis dipole integrals
integrals_mo = np.empty(shape=(ncomp, norb, norb))
for i in range(ncomp):
    integrals_mo[i] = (C.T).dot(integrals_ao[i]).dot(C)

# repack response vectors to [norb, norb]; 1/2 is due to X + Y
U = np.zeros_like(integrals_mo)
for i in range(ncomp):
    U[i, :nocc, nocc:] = 0.5 * x[i].reshape(nocc, nvir)
    U[i, nocc:, :nocc] = -0.5 * x[i].reshape(nocc, nvir).T

# form G matrices from perturbation and generalized Fock matrices; do
# one more Fock build for each response vector
jk = psi4.core.JK.build(helper.scf_wfn.basisset())
jk.initialize()
G = np.empty_like(U)
R = psi4.core.Matrix(nbf, nocc)
npR = np.asarray(R)
for i in range(ncomp):
    V = integrals_mo[i]

    # eqn. (III-1b) Note: this simplified handling of the response
    # vector transformation for the Fock build is insufficient for
    # frequency-dependent response.
    jk.C_clear()
    # Psi4's JK builders don't take a density, but a left set of
    # coefficients with shape [nbf, nocc] and a right set of
    # coefficents with shape [nbf, nocc]. Because the response vector
    # describes occ -> vir transitions, we perform ([nocc, nvir] *
    # [nbf, nvir]^T)^T.
    L = Co
    npR[:] = x[i].reshape(nocc, nvir).dot(np.asarray(Cv).T).T
    jk.C_left_add(L)
    jk.C_right_add(R)
    jk.compute()
    # 1/2 is due to X + Y
    J = 0.5 * np.asarray(jk.J()[0])
    K = 0.5 * np.asarray(jk.K()[0])

    # eqn. (21b)
    F = (C.T).dot(4 * J - K.T - K).dot(C)
    G[i] = V + F

```

```
# form epsilon matrices, eqn. (34)
E = G.copy()
omega = 0
for i in range(ncomp):
    eoU = (moenergies[..., na] + omega) * U[i]
    Ue = U[i] * moenergies[na]
    E[i] += (eoU - Ue)

# Assume some symmetry and calculate only part of the tensor.
# eqn. (VII-4)
hyperpolarizability = np.zeros(shape=(6, 3))
off1 = [0, 1, 2, 0, 0, 1]
off2 = [0, 1, 2, 1, 2, 2]
for r in range(6):
    b = off1[r]
    c = off2[r]
    for a in range(3):
        tl1 = 2 * np.trace(U[a].dot(G[b]).dot(U[c])[:nocc, :nocc])
        tl2 = 2 * np.trace(U[a].dot(G[c]).dot(U[b])[:nocc, :nocc])
        tl3 = 2 * np.trace(U[c].dot(G[a]).dot(U[b])[:nocc, :nocc])
        tr1 = np.trace(U[c].dot(U[b]).dot(E[a])[:nocc, :nocc])
        tr2 = np.trace(U[b].dot(U[c]).dot(E[a])[:nocc, :nocc])
        tr3 = np.trace(U[c].dot(U[a]).dot(E[b])[:nocc, :nocc])
        tr4 = np.trace(U[a].dot(U[c]).dot(E[b])[:nocc, :nocc])
        tr5 = np.trace(U[b].dot(U[a]).dot(E[c])[:nocc, :nocc])
        tr6 = np.trace(U[a].dot(U[b]).dot(E[c])[:nocc, :nocc])
        tl = tl1 + tl2 + tl3
        tr = tr1 + tr2 + tr3 + tr4 + tr5 + tr6
        hyperpolarizability[r, a] = -2 * (tl - tr)

ref_static = np.array([
    [ 0.00000001,   0.00000000,   0.22843772],
    [ 0.00000000,   0.00000000, -25.35476040],
    [ 0.00000000,   0.00000000, -10.84023375],
    [ 0.00000000,   0.00000000,   0.00000000],
    [ 0.22843772,   0.00000000,   0.00000000],
    [ 0.00000000, -25.35476040,   0.00000000]
])
assert np.allclose(ref_static, hyperpolarizability, rtol=0.0, atol=1.0e-3)
print('\nFirst dipole hyperpolarizability (static):')
print(hyperpolarizability)

# Compute the (first) hyperpolarizability corresponding to
# second-harmonic generation, beta(-2w;w,w), eqns. (IV-2c) and
# (VII-1). Because two different frequencies are involved, the linear
# response equations must be solved twice.

print('Setting up for second-harmonic generation (SHG) calculation...')
# In SHG, the first frequency is doubled to obtain the second
# frequency. All variables containing '1' correspond to the first
# (set) frequency, and all variables containing '2' correspond to the
# second (doubled) frequency.
f1 = 0.0773178
f2 = 2 * f1

print('\nForming response vectors for {} a.u.'.format(f1))
helper1 = helper_CPHF(mol)
helper1.solve_dynamic_direct(omega=f1)
helper1.form_polarizability()
print(helper1.polar)
print('\nForming response vectors for {} a.u.'.format(f2))
helper2 = helper_CPHF(mol)
helper2.solve_dynamic_direct(omega=f2)
helper2.form_polarizability()
print(helper2.polar)

rspvecs1 = helper1.x
rspvecs2 = helper2.x
```

```
175
176 # repack response vectors to [norb, norb]
177 U1 = np.zeros_like(integrals_mo)
178 U2 = np.zeros_like(integrals_mo)
179 for i in range(ncomp):
180     U1[i, :nocc, nocc:] = rspvecs1[i][nov:].reshape(nocc, nvir)
181     U1[i, nocc:, :nocc] = rspvecs1[i][:nov].reshape(nocc, nvir).T
182     U2[i, :nocc, nocc:] = rspvecs2[i][nov:].reshape(nocc, nvir)
183     U2[i, nocc:, :nocc] = rspvecs2[i][:nov].reshape(nocc, nvir).T
184
185 G1 = np.empty_like(U1)
186 G2 = np.empty_like(U2)
187 R1_l = psi4.core.Matrix(nbf, nocc)
188 R1_r = psi4.core.Matrix(nbf, nocc)
189 R2_l = psi4.core.Matrix(nbf, nocc)
190 R2_r = psi4.core.Matrix(nbf, nocc)
191 npR1_l = np.asarray(R1_l)
192 npR1_r = np.asarray(R1_r)
193 npR2_l = np.asarray(R2_l)
194 npR2_r = np.asarray(R2_r)
195 jk.C_clear()
196 jk.C_left_add(Co)
197 jk.C_right_add(R1_l)
198 jk.C_left_add(Co)
199 jk.C_right_add(R1_r)
200 jk.C_left_add(Co)
201 jk.C_right_add(R2_l)
202 jk.C_left_add(Co)
203 jk.C_right_add(R2_r)
204 nCo = np.asarray(Co)
205 # Do 4 Fock builds at a time: X/Y vectors for both frequencies; loop
206 # over operator components
207 for i in range(3):
208     V = integrals_mo[i]
209
210     x1 = U1[i, :nocc, :]
211     y1 = U1[i, :, :nocc]
212     x2 = U2[i, :nocc, :]
213     y2 = U2[i, :, :nocc]
214     npR1_l[:] = C.dot(x1.T)
215     npR1_r[:] = C.dot(y1)
216     npR2_l[:] = C.dot(x2.T)
217     npR2_r[:] = C.dot(y2)
218
219     jk.compute()
220
221     J1_l = -np.asarray(jk.J()[0])
222     K1_l = -np.asarray(jk.K()[0])
223     J1_r = np.asarray(jk.J()[1])
224     K1_r = np.asarray(jk.K()[1])
225     J2_l = -np.asarray(jk.J()[2])
226     K2_l = -np.asarray(jk.K()[2])
227     J2_r = np.asarray(jk.J()[3])
228     K2_r = np.asarray(jk.K()[3])
229     J1 = J1_l + J1_r
230     J2 = J2_l + J2_r
231     K1 = K1_l + K1_r.T
232     K2 = K2_l + K2_r.T
233
234     F1 = (C.T).dot(2 * J1 - K1).dot(C)
235     F2 = (C.T).dot(2 * J2 - K2).dot(C)
236     G1[i, ...] = V + F1
237     G2[i, ...] = V + F2
238
239 # form epsilon matrices, eqn. (34), one for each frequency
240 E1 = G1.copy()
241 E2 = G2.copy()
242 for i in range(ncomp):
```

```
    eoU1 = (moenergies[..., na] + f1) * U1[i]
    Ue1 = U1[i] * moenergies[na]
    E1[i] += (eoU1 - Ue1)
    eoU2 = (moenergies[..., na] + f2) * U2[i]
    Ue2 = U2[i] * moenergies[na]
    E2[i] += (eoU2 - Ue2)

# Assume some symmetry and calculate only part of the tensor.

hyperpolarizability = np.zeros(shape=(6, 3))
for r in range(6):
    b = off1[r]
    c = off2[r]
    for a in range(3):
        tl1 = np.trace(U2[a].T.dot(G1[b]).dot(U1[c])[:nocc, :nocc])
        tl2 = np.trace(U1[c].dot(G1[b]).dot(U2[a].T)[:nocc, :nocc])
        tl3 = np.trace(U2[a].T.dot(G1[c]).dot(U1[b])[:nocc, :nocc])
        tl4 = np.trace(U1[b].dot(G1[c]).dot(U2[a].T)[:nocc, :nocc])
        tl5 = np.trace(U1[c].dot(-G2[a].T).dot(U1[b])[:nocc, :nocc])
        tl6 = np.trace(U1[b].dot(-G2[a].T).dot(U1[c])[:nocc, :nocc])
        tr1 = np.trace(U1[c].dot(U1[b]).dot(-E2[a].T)[:nocc, :nocc])
        tr2 = np.trace(U1[b].dot(U1[c]).dot(-E2[a].T)[:nocc, :nocc])
        tr3 = np.trace(U1[c].dot(U2[a].T).dot(E1[b])[:nocc, :nocc])
        tr4 = np.trace(U2[a].T.dot(U1[c]).dot(E1[b])[:nocc, :nocc])
        tr5 = np.trace(U1[b].dot(U2[a].T).dot(E1[c])[:nocc, :nocc])
        tr6 = np.trace(U2[a].T.dot(U1[b]).dot(E1[c])[:nocc, :nocc])
        tl = tl1 + tl2 + tl3 + tl4 + tl5 + tl6
        tr = tr1 + tr2 + tr3 + tr4 + tr5 + tr6
        hyperpolarizability[r, a] = 2 * (tl - tr)

# pylint: disable=C0326
ref = np.array([
    [ 0.00000000,   0.00000000,   1.92505358],
    [ 0.00000000,   0.00000000, -31.33652886],
    [ 0.00000000,   0.00000000, -13.92830863],
    [ 0.00000000,   0.00000000,   0.00000000],
    [-1.80626084,   0.00000000,   0.00000000],
    [ 0.00000000, -31.13504192,   0.00000000]
])
ref_avgs = np.array([0.00000000, 0.00000000, 45.69300223])
ref_avg = 45.69300223

thresh = 1.0e-2
# assert np.all(np.abs(ref - hyperpolarizability) < thresh)

print('hyperpolarizability: SHG, (-{}; {}, {}), symmetry-unique components'.format(f2, f1, f1))
print(hyperpolarizability)
print('ref')
print(ref)

# Transpose all frequency-doubled quantities (+2w) to get -2w.

for i in range(ncomp):
    U2[i] = U2[i].T
    G2[i] = -G2[i].T
    E2[i] = -E2[i].T

# Assume some symmetry and calculate only part of the tensor. This
# time, work with the in-place manipulated quantities (this tests
# their correctness).

mU = (U2, U1)
mG = (G2, G1)
me = (E2, E1)

hyperpolarizability = np.zeros(shape=(6, 3))
off1 = [0, 1, 2, 0, 0, 1]
off2 = [0, 1, 2, 1, 2, 2]
```

```
for r in range(6):
    b = off1[r]
    c = off2[r]
    for a in range(3):
        tl1 = np.trace(mU[0][a].dot(mG[1][b]).dot(mU[1][c])[:nocc, :nocc])
        tl2 = np.trace(mU[1][c].dot(mG[1][b]).dot(mU[0][a])[:nocc, :nocc])
        tl3 = np.trace(mU[0][a].dot(mG[1][c]).dot(mU[1][b])[:nocc, :nocc])
        tl4 = np.trace(mU[1][b].dot(mG[1][c]).dot(mU[0][a])[:nocc, :nocc])
        tl5 = np.trace(mU[1][c].dot(mG[0][a]).dot(mU[1][b])[:nocc, :nocc])
        tl6 = np.trace(mU[1][b].dot(mG[0][a]).dot(mU[1][c])[:nocc, :nocc])
        tr1 = np.trace(mU[1][c].dot(mU[1][b]).dot(me[0][a])[:nocc, :nocc])
        tr2 = np.trace(mU[1][b].dot(mU[1][c]).dot(me[0][a])[:nocc, :nocc])
        tr3 = np.trace(mU[1][c].dot(mU[0][a]).dot(me[1][b])[:nocc, :nocc])
        tr4 = np.trace(mU[0][a].dot(mU[1][c]).dot(me[1][b])[:nocc, :nocc])
        tr5 = np.trace(mU[1][b].dot(mU[0][a]).dot(me[1][c])[:nocc, :nocc])
        tr6 = np.trace(mU[0][a].dot(mU[1][b]).dot(me[1][c])[:nocc, :nocc])
        tl = [tl1, tl2, tl3, tl4, tl5, tl6]
        tr = [tr1, tr2, tr3, tr4, tr5, tr6]
        hyperpolarizability[r, a] = 2 * (sum(tl) - sum(tr))

assert np.all(np.abs(ref - hyperpolarizability) < thresh)

# Assume no symmetry and calculate the full tensor.

hyperpolarizability_full = np.zeros(shape=(3, 3, 3))

# components x, y, z
for ip, p in enumerate(list(product(range(3), range(3), range(3)))):
    a, b, c = p
    tl, tr = [], []
    # 1st tuple -> index a, b, c (*not* x, y, z!)
    # 2nd tuple -> index frequency (0 -> -2w, 1 -> +w)
    for iq, q in enumerate(list(permutations(zip(p, (0, 1, 1)), 3))):
        d, e, f = q
        tlp = (mU[d[1]][d[0]]).dot(mG[e[1]][e[0]]).dot(mU[f[1]][f[0]])
        tle = np.trace(tlp[:nocc, :nocc])
        tl.append(tle)
        trp = (mU[d[1]][d[0]]).dot(mU[e[1]][e[0]]).dot(me[f[1]][f[0]])
        tre = np.trace(trp[:nocc, :nocc])
        tr.append(tre)
    hyperpolarizability_full[a, b, c] = 2 * (sum(tl) - sum(tr))
print('hyperpolarizability: SHG, (-{}; {}, {}), full tensor'.format(f2, f1, f1))
print(hyperpolarizability_full)

# Check that the elements of the reduced and full tensors are
# equivalent.

for r in range(6):
    b = off1[r]
    c = off2[r]
    for a in range(3):
        diff = hyperpolarizability[r, a] - hyperpolarizability_full[a, b, c]
        assert abs(diff) < 1.0e-14
```