Contents

Restricted Hartee-Fock Self Consistent Field Report

Joeri Van Meerssche

April 2020

1 Acknowledgement

I would like to thank Aaron De Clercq and John De Vos for their help during the entire project.

2 Introduction

Prior to starting my bachelor project in the Ghent Quantum Chemistry Group, a bootcamp containing three programming exercises were foreseen to introduce me to quantumchemical computational programming. The latter of these exercises is the subject for this report. A great overview is summarized here[1].

3 Theory and code

3.1 Packages

The following packages were imported: psi4 (as ps), numpy (as np) and unittest.

3.2 The initial Fock matrix

Before starting the iterative procedure an initial Fock matrix is required. The first step consists of constructing the core Hamiltonian. The core Hamiltonian is defined as follows.

$$H_{\mu\nu}^{\text{core}} = T_{\mu\nu} + V_{\mu\nu}$$

Where T and V are the kinetic-energy and nuclear-attraction matrices respectively. Their basis consists of one-electron integrals of wavefunctions built around the geometry input of the molecule e.g. water and a chosen atom orbital basis set e.g. sto—3g. The distances are expressed in Bohr radii (5.29×10-11 m).

```
ps.set_options({'basis': 'sto-3g'})
mol = ps.geometry("""
O  0.000000000000 -0.143225816552 0.000000000000
H  1.638036840407 1.136548822547 -0.000000000000
H -1.638036840407 1.136548822547 -0.000000000000
units bohr
""")
mol.update_geometry()
# Nuclear Repulsion Energy calculation
nre = mol.nuclear_repulsion_energy()
# Initialization Atom Orbital basis set
wave = ps.core.Wavefunction.build(mol, ps.core.get_global_option('basis'))
mints = ps.core.MintsHelper(wave.basisset())
# One-electron integrals
S = np.asarray(mints.ao_overlap())
T = np.asarray(mints.ao_kinetic())
V = np.asarray(mints.ao_potential())
# Initialization of the core Hamiltonian
Hcore = T + V
```

Once the core Hamiltonian is constructed a transformation is required to form the initial Fock matrix. This is done via the symmetric orthogonalization matrix (SOM). The SOM is built starting from the AO-basis overlap matrix S, also consisting of one electron integrals (see code above).

The overlap matrix is then diagonalized and transformed via the following equations.

$$\mathbf{S}\mathbf{L}_S = \mathbf{L}_S\Lambda_S$$

where $\mathbf{L}_S$ is the matrix of eigenvectors and Λ_S the diagonal matrix of corresponding eigenvalues.

$$\mathbf{S}^{-1/2} \equiv \mathbf{L}_S\Lambda^{-1/2}\tilde{\mathbf{L}}_S$$

```
# Building the Orthogonalization Matrix
def OrthoS(eigenval, eigenvec):
    return eigenvec @ (np.linalg.inv(eigenval) ** 0.5) @ np.transpose(eigenvec)
def Diag(mat):
    eigenval = np.diag(np.linalg.eigh(mat)[0])
    eigenvec = np.linalg.eigh(mat)[1]
    return eigenval, eigenvec
Sval, Svec = Diag(S)
Smin = OrthoS(Sval, Svec)
```

Now all the tools are available for constructing the initial Fock matrix.

$$\mathbf{F}'_0 \equiv \tilde{\mathbf{S}}^{-1/2}\mathbf{H}^{\text{core}}\mathbf{S}^{-1/2}$$

$$\mathbf{F}'_0\mathbf{C}'_0 = \mathbf{C}'_0\epsilon_0$$

```
def OrthoF(eigenval, eigenvec):
    return np.transpose(eigenvec) @ eigenval @ eigenvec
# Initial Guess Density Matrix (AO)
Fz = OrthoF(Hcore, Smin)
Fz_val, Fz_vec = Diag(Fz)
```

3.3 The initial Density matrix and SCF energies

A density matrix is constructed by summation over the occupied molecular orbitals, which are in turn (transformed) eigenvectors of the (intitial) Fock matrix, and is defined as follows.

$$\mathbf{C}_0 = \mathbf{S}^{-1/2}\mathbf{C}'_0$$

$$D^0_{\mu\nu} = \sum_m^{\text{occ.}} (\mathbf{C}_0)^m_\mu \, (\mathbf{C}_0)^m_\nu$$

```
# Eigenvector transformation from orthonormal to non-orthogonal (original) AO basis
Cz = Smin @ Fz_vec
occ = wave.nalpha()
Dens_z = np.einsum('ik,jk->ij',Cz[:, :occ], Cz[:, :occ])
```

Einsum is used to sum over the columns since it allows for faster calculations compared to looping. The amount of occupied orbitals is given by the amount of alpha or beta spins (or their average) since this calculation involves Restricted Hartree Fock where only double occupied MOs are present.
Next, the initial SFC energies are calculated.

$$E^0_{\text{elec}} = 2\sum_{\mu\nu}^{\text{AO}} D^0_{\mu\nu}H^{\text{core}}_{\mu\nu}$$

$$E^0_{\text{total}} = E^0_{\text{elec}} + E_{\text{nuc}}$$

```
# Compute the Initial SCF Energy
Ez_el = np.einsum('ij,ij->', Dens_z, 2*Hcore)
Ez_tot = Ez_el + nre
```

Again, einsum is used for faster summation. As can be seen, the summation over the occupied MOs is accomplished by slicing the eigenvector matrices to their appropiate dimensions. As for the initial SCF electronic energy, $F_{\mu v}$ can be equated to $H^{core}_{\mu v}$.

3.4 Self Consistent Field procedure

And now the fun part. The goal is to lower the initial SCF total energy by an iterative process called the Self Consistent Field. The first step consists of the construction of a new Fock matrix using the previously calculated (initial) density matrix and the 2-electron integrals (represented as Mulliken symbols). Note that these 2-electron-repulsion integrals are stored as a rank 4 tensor matrix called 'ERI'. These are calculated outside of the loop because they are invariant under the SFC procedure. Also the according new SCF energies are calculated using the same density matrix. The Fock matrix can now not be equated to the core Hamiltonian.

$$F_{\mu\nu} = H_{\mu\nu}^{\text{core}} + \sum_{\lambda\sigma}^{\text{AO}} D_{\lambda\sigma}^{i} \left[2(\mu\nu|\lambda\sigma) - (\mu\lambda|\nu\sigma)\right]$$

$$E_{\text{elec}}^{i} = \sum_{\mu\nu}^{\text{AO}} D_{\mu\nu}^{i} \left(H_{\mu\nu}^{\text{core}} + F_{\mu\nu}\right)$$

$$E_{\text{total}}^{i} = E_{\text{elec}}^{i} + E_{\text{nuc}}$$

```
# Two-Electron Integrals
ERI = np.asarray(mints.ao_eri())
# Compute New Fock Matrix
def fock(D, Hcore, ERI, dimension):
    F = np.zeros((dimension, dimension))
    for i in range(dimension):
        for j in range(dimension):
            sum = 0
            F[i][j] += Hcore[i][j]
            for k in range(dimension):
                for l in range(dimension):
                    sum += D[k][l] * ((2 * ERI[i][j][k][l]) - ERI[i][k][j][l])
            F[i][j] += sum
    return F
def ElEnergy(D, Hcore, Fock):
    sum = Hcore + Fock
    return np.einsum('ij,ij->', D, sum)
Fock = fock(Dens_p, Hcore, ERI, Cz.shape[0])
# Compute the New SCF Energy
    E_el = ElEnergy(Dens_p, Hcore, Fock)
    E_tot = E_el + nre
```

Where 'Densp' denotes the previous calculated density and i the current iteration. After calculation, orthogonalization and diagonalization follow.

$$\mathbf{F}' \equiv \tilde{\mathbf{S}}^{-1/2}\mathbf{F}\mathbf{S}^{-1/2}$$

$$\mathbf{F}'\mathbf{C}' = \mathbf{C}'\epsilon$$

```
Facc = OrthoF(Fock, Smin)
Facc_val, Facc_vec = Diag(Facc)
```

The new density matrix is then built via backtransformation of the Fock eigenvectors.

$$\mathbf{C} = \mathbf{S}^{-1/2}\mathbf{C}'$$

$$D_{\mu\nu}^{i} = \sum_{m}^{\text{occ.}} (\mathbf{C})_{\mu}^{m} (\mathbf{C})_{\nu}^{m}$$

```
# Build the New Density Matrix
C = Smin @ Facc_vec
Dens_old = Dens_p
Dens_p = np.einsum('ik,jk->ij', C[:, :occ], C[:, :occ])
```

The amount of iterations of the SCF-procedure solely depend on the user's requirements. This is quantified in a convergence test. In this case, the difference in the newly and previously calculated SCF total energy. Once it's value is below a chosen tress hold e.g. 1e-12, the iterative process stops. This is imposed in the while loop's statement.

$$\Delta E = E^{i}_{\text{elec}} - E^{i-1}_{\text{elec}} < \delta_1$$

```
lamda = 1e-12
while abs(DE) > lamda
    ...
    DE = E_el - Ep_el
    Ep_tot, Ep_el = E_tot, E_el
```

This concludes the section concerning the theory and the corresponding code. A plot is given below for a h2o molecule in cc-pvdz AO basis with it's geometry defined above. As can be seen, the SCF total energy of the next iteration is lower than the previous one and converges to a limit value. Unittests are also included for a water molecule test case.

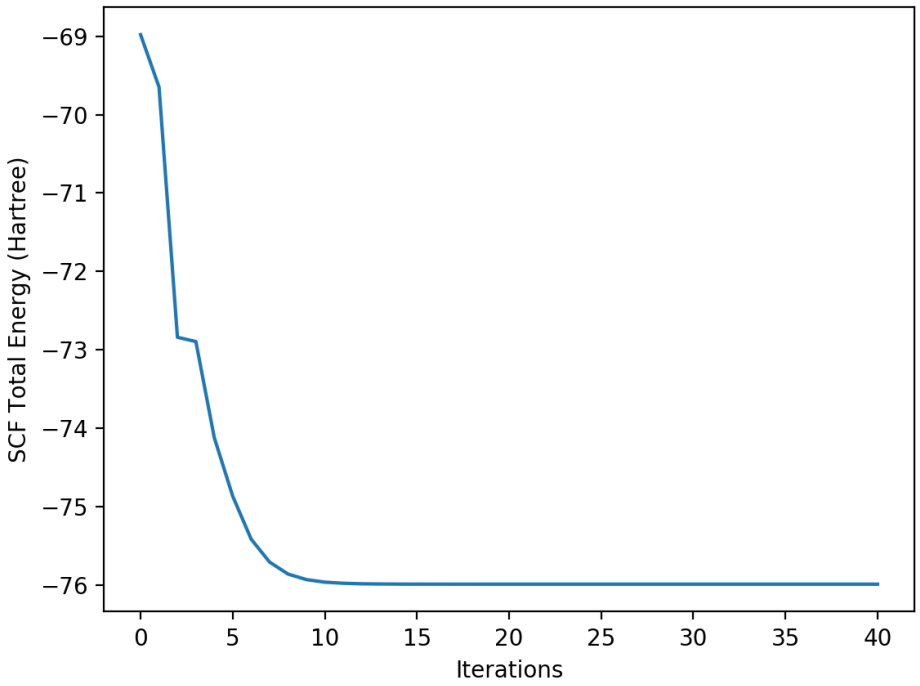

Figure 1: *SCF total energy in function of the amount of iterations for a water molecule in cc-pvdz AO basis and maximal iterative energy difference 1e-12.*

4 Code Appendix

```
import psi4 as ps
import numpy as np
import unittest
from HFSCF import Diag, OrthoS, OrthoF, fock, ElEnergy
ps.core.set_output_file('output.dat', True)
ps.set_memory(int(5e8))
np_memory = 2

# Initialization Molecular Geometry
ps.set_options({'basis': 'cc-pvdz'})
mol = ps.geometry("""
O  0.000000000000 -0.143225816552 0.000000000000
H  1.638036840407 1.136548822547 -0.000000000000
H -1.638036840407 1.136548822547 -0.000000000000
units bohr
""")

mol.update_geometry()

# Nuclear Repulsion Energy calculation
nre = mol.nuclear_repulsion_energy()

# Initialization Atom Orbital basis set
wave = ps.core.Wavefunction.build(mol, ps.core.get_global_option('basis'))
mints = ps.core.MintsHelper(wave.basisset())

# One-electron integrals
S = np.asarray(mints.ao_overlap())
T = np.asarray(mints.ao_kinetic())
V = np.asarray(mints.ao_potential())

# Initialization of the core Hamiltonian
Hcore = T + V

# Two-Electron Integrals
ERI = np.asarray(mints.ao_eri())

# Building the Orthogonalization Matrix
Sval, Svec = Diag(S)
Smin = OrthoS(Sval, Svec)

# Initial Guess Density Matrix (AO)
Fz = OrthoF(Hcore, Smin)
Fz_val, Fz_vec = Diag(Fz)

# Eigenvector transformation from orthonormal to non-orthogonal (original) AO basis
Cz = Smin @ Fz_vec
occ = wave.nalpha()
Dens_z = np.einsum('ik,jk->ij',Cz[:, :occ], Cz[:, :occ])

# Compute the Initial SCF Energy
Ez_el = np.einsum('ij,ij->', Dens_z, 2*Hcore)
Ez_tot = Ez_el + nre
```

```
class TestHF(unittest.TestCase):
    def test_Hcore(self):
        Hcore_h2o = [[-32.5773954, -7.5788328, -0.0144738, 0.0,  0.0, -1.2401023, -1.2401023],
                     [-7.5788328, -9.2009433, -0.1768902, 0.0,  0.0, -2.9067098, -2.9067098],
                     [-0.0144738, -0.1768902, -7.4153118, 0.0,  0.0, -1.3568683, -1.3568683],
                     [0.0, 0.0, 0.0, -7.4588193, 0.0, -1.6751501, 1.6751501],
                     [0.0, 0.0, 0.0, 0.0, -7.3471449, 0.0, 0.0],
                     [-1.2401023, -2.9067098, -1.3568683, -1.6751501, 0.0, -4.5401711, -1.0711459],
                     [-1.2401023, -2.9067098, -1.3568683, 1.6751501, 0.0, -1.0711459, -4.5401711]]
        self.assertIsNone(np.testing.assert_array_almost_equal(Hcore, Hcore_h2o))
    def test_Smin(self):
        Smin_h2o = [[1.0236346, -0.1368547, -0.0074873, -0.0,  -0.0, 0.0190279, 0.0190279],
                    [-0.1368547, 1.1578632, 0.0721601, 0.0,  0.0, -0.2223326, -0.2223326],
                    [-0.0074873, 0.0721601, 1.038305, 0.0,  0.0, -0.1184626, -0.1184626],
                    [-0.0, 0.0, 0.0, 1.0733148, -0.0, -0.1757583, 0.1757583],
                    [-0.0, 0.0, -0.0, 0.0, 1.0, -0.0, -0.0],
                    [0.0190279, -0.2223326, -0.1184626, -0.1757583, -0.0, 1.1297234, -0.0625975],
                    [0.0190279, -0.2223326, -0.1184626, 0.1757583, -0.0, -0.0625975, 1.1297234]]
        self.assertIsNone(np.testing.assert_array_almost_equal(Smin, Smin_h2o))
    def test_InitialElEn(self):
        self.assertAlmostEqual(Ez_el, -125.842077437699)
    def test_InitialFock(self):
        InitialFock = [[-32.2545866, -2.7914909, 0.008611, -0.0,  0.0, -0.1812967, -0.1812967],
                       [-2.7914909, -8.2368891, -0.2282926, -0.0,  0.0, -0.3857987, -0.3857987],
                       [0.008611, -0.2282926, -7.4570295, -0.0, -0.0, -0.1102196, -0.1102196],
                       [-0.0, -0.0, -0.0, -7.542889, 0.0, -0.1132121, 0.1132121],
                       [0.0, 0.0, 0.0, -0.0, -7.3471449, 0.0, 0.0],
                       [-0.1812967, -0.3857987, -0.1102196, -0.1132121, 0.0, -4.0329547, -0.0446466],
                       [-0.1812967, -0.3857987, -0.1102196, 0.1132121, 0.0, -0.0446466, -4.0329547]]
        self.assertIsNone(np.testing.assert_array_almost_equal(InitialFock, Fz))
# LOOP
Dens_p, Ep_tot, Ep_el = Dens_z, Ez_tot, Ez_el
DE = -Ez_tot
lamda = 1e-12
iteration = 0
energies = []

total_time = time.time()
while abs(DE) > lamda:
    DE = 0
    iteration += 1
    # Compute New Fock Matrix
    Fock = fock(Dens_p, Hcore, ERI, Cz.shape[0])
    # Compute the New SCF Energy
    E_el = ElEnergy(Dens_p, Hcore, Fock)
    E_tot = E_el + nre
    # Build the New Density Matrix
    Facc = OrthoF(Fock, Smin)
    Facc_val, Facc_vec = Diag(Facc)
    C = Smin @ Facc_vec
    Dens_old = Dens_p
    Dens_p = np.einsum('ik,jk->ij', C[:, :occ], C[:, :occ])
    # Test for Convergence
    print('Energy: {}\tIteration: {}'.format(E_tot, iteration))
    energies.append(E_tot)
    DE = E_el - Ep_el
    Ep_tot, Ep_el = E_tot, E_el
print('Loop time: {}s'.format(time.time() - total_time))

if __name__ == '__main__':
    unittest.main()
```

References

[1] C. David Sherrill. "An Introduction to Hartree-Fock Molecular Orbital Theory." In: *School of Chemistry and Biochemistry, Georgia Institute of Technology* (2000), http://vergil.chemistry.gatech.edu/notes/hf-intro/hf–intro.pdf.