Assignment 3

Calculating Electronic Structures of the
Helium atom and Hydrogen Molecule using
Hartree-Fock and Roothaan Equations

Due: 2022/03/04

1 Directions

This is a direct follow-up to assignment 2. We are interested in solving the same two systems (hydrogen molecule and helium atom) as in that assignment and comparing the results with what we obtained there.

In this assignment, use the Roothaan equation. As basis functions, use two s-type, normalized Gaussian functions (with different values of α) centered on each atom. Note that you can choose whatever values of α you find most suitable for the two Gaussian functions on each atom. Play around with a few different α values to see which ones give you the most reasonable answers. If you are ambitious, include additional basis functions.

(a) (4 points) Describe your approach and show the steps of your derivation all the way to obtaining the matrix equation.

(b) (4 points) Write a computer code to implement what you built in part a.

(c) (4 points) Plot several molecular orbitals and give their associated energies for the helium atom. Also calculate the total energy of the system. Discuss your results.

(d) (4 points) Plot several molecular orbitals and give their associated energies for the hydrogen molecule. Also calculate the total energy of the system (not including the nucleus-nucleus interaction). Discuss your results.

(e) (4 points) Compare your results of both the above exercise and what you obtained in assignment 2 together and also with values available from the literature, and what you obtain using a commercial program such as Gaussian (run through ABACUS, for example).

2 Derivation of Solution

As with the previous assignment, since both subjects for this assignment have an even number of electrons that close shells, we can use the restricted Hartree-Fock equation for closed shell systems to numerically calculate the resulting orbitals. The equation is as follows:

$$\hat{F}(\vec{r})\psi_n(\vec{r}) = \epsilon_n\psi_n(\vec{r})$$

Where $\hat{F}(\vec{r})$ is the Fock operator which is defined as follows:

$$\hat{F}(\vec{r}) = \hat{H}_{core}(\vec{r}) + \sum_{n=1}^{N/2} [2J_n(\vec{r}) - K_n(\vec{r})]$$

In the previous assignment, the wave equation $\psi_n(\vec{r})$ was obtained by discretizing the solution of the wave equation into a three-dimensional grid partition of size $N \times N \times N$ where each dimension was partitioned into N elements. The Hartree-Fock equation was manipulated to create a matrix eigenvector/eigenvalue problem out of the wave equation to be solved iteratively by a computer program over this space. The largest computational load in the previous assignment's program turned out to be the integration that needed to be performed for every iteration. The exchange term has an integral to calculate over the $N \times N \times N$ space of the matrix. This meant that for every element in the $N \times N \times N$ space, the integration had to be performed over a space of $N \times N \times N$. This integral alone resulted in N^6 calculations during every iteration of the Hartree-Fock algorithm. Multiprocessing helped alleviate some of this load by spreading it across multple processing units, but the problem was still very taxing to a modern computer with a modest number of spatial partitions.

2.1 Introducing Basis through the Roothaan Equations

This assignment aims to simplify the Hartree-Fock procedure by introducing the concept of basis functions. The following derivations are presented in Szabo and Ostlund (referenced at the end of this assignment) and are summarized in the following paragraphs. Known as the Roothaan equations, the Hartree-Fock equation is augmented by using a combination of non-orthonormal spatial basis functions. The Hartree-Fock procedure is now modified to converge on the coefficients of these basis functions which linearly combine to approximate the electron orbitals. These functions are chosen very carefully to approximate the orbital that the electrons should exist in, which allows a small combination of them to simulate a real system with high accuracy. Likely more accurate and more stable than what was implemented in the previous assignment. Since there are only a few coefficients to solve for in this basis set of functions, the Hartree-Fock procedure is greatly simplified when put into a matrix eigenvector/eigenvalue problem. The basis functions can be described as follows:

$$\psi_n(\vec{r}) = \sum_{\mu=1}^{K} C_{\mu n}\phi_\mu(\vec{r}) \qquad n = 1, 2, ..., K$$

A complete set of functions ϕ_μ could exactly represent a system; however, for computational practicality much less can be used, as is evident by this assignment only asking for two or more of these functions. Specifically, normalized gaussian functions are to be used in this assignment. When combined, two or more gaussian functions can better approximate the shape of an atomic orbital as a single gaussian function alone does not match the shape of an atomic orbital very well. A real orbital function will have a pointed asymptotic shape in the middle where the nucleus is present, wheras a gaussian function has a rounded shape at its maximum. By adding a combination of gaussian functions together, the pointed shape is better approximated. An advantage of using a combination of gaussian functions is that they are easily integrated. Since integration is a common operation in Hartree-Fock, this lends well to a rapid calculations towards a convergence.

By substituting the linear combination of functions of ϕ_μ in the place of ψ_n in the Hartree-Fock equation, we obtain the following:

$$\hat{F}(\vec{r})\sum_{\mu=1}^{K} C_{\mu n}\phi_{\mu}(\vec{r}) = \epsilon_n \sum_{\mu=1}^{K} C_{\mu n}\phi_{\mu}(\vec{r})$$

We can then multiply both sides by $\phi_v^*(\vec{r})$ and integrate to obtain the following result:

$$\sum_{\mu=1}^{K} C_{\mu n}\int_{-\infty}^{\infty}\phi_v^*(\vec{r})\hat{F}(\vec{r})\phi_{\mu}(\vec{r})d\vec{r} = \epsilon_n \sum_{\mu=1}^{K} C_{\mu n}\int_{-\infty}^{\infty}\phi_v^*(\vec{r})\phi_{\mu}(\vec{r})d\vec{r}$$

This allows the Hartree-Fock equation to be turned into a matrix equation containing the overlap matrix $\boldsymbol{S}$ containing $K \times K$ elements $S_{v\mu}$:

$$S_{v\mu} = \int_{-\infty}^{\infty}\phi_v^*(\vec{r})\phi_{\mu}(\vec{r})d\vec{r}$$

and the Fock matrix $\boldsymbol{F}$ containing $K \times K$ elements $F_{v\mu}$:

$$F_{v\mu} = \int_{-\infty}^{\infty}\phi_v^*(\vec{r})\hat{F}(\vec{r})\phi_{\mu}(\vec{r})d\vec{r}$$

Bringing these back into the Hartree-Fock equation, we obtain the following equation, known as the Roothan equations:

$$\sum_{\mu=1}^{K} F_{v\mu}C_{\mu n} = \epsilon_n \sum_{\mu=1}^{K} S_{v\mu}C_{\mu n}$$

or more concisely:

$$\boldsymbol{FC} = \boldsymbol{SC\epsilon}$$

The matrix $\boldsymbol{C}$ is a $K \times K$ square matrix of the expansion coefficients $C_{\mu n}$:

$$\boldsymbol{C} = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1K} \\ C_{21} & C_{22} & \dots & C_{2K} \\ \vdots & \vdots & & \vdots \\ C_{K1} & C_{K2} & \dots & C_{KK} \end{bmatrix}$$

and $\boldsymbol{\epsilon}$ is a diagonal matrix of the orbital energies ϵ_n:

$$\boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_1 & & & \\ & \epsilon_2 & & \\ & & \ddots & \\ & & & \epsilon_K \end{bmatrix}$$

2.2 Charge Density Matrix

In order to calculate the Fock matrix $\boldsymbol{F}$, we have to introduce the concept of a charge density matrix. This matrix describes the probability distribution of finding an electron in a given space described by the wave function $\psi_n(\vec{r})$. In a closed-shell system described by a single wave function occupied by two electrons (as in this assignment), the total charge density $\rho(\vec{r})$ is equal to:

$$\rho(\vec{r}) = 2\sum_{n}^{N/2} |\psi_n(\vec{r})|^2$$

We can then substitue the basis function expansion performed above to derive what is known as the density matrix $\boldsymbol{P}$:

$$\begin{aligned}
\rho(\vec{r}) &= 2\sum_{n}^{N/2} \psi_n^*(\vec{r})\psi_n(\vec{r}) \\
\rho(\vec{r}) &= 2\sum_{n}^{N/2}\sum_{v=1}^{K} C_{vn}^*\phi_v^*(\vec{r})\sum_{\mu=1}^{K} C_{\mu n}\phi_\mu(\vec{r}) \\
\rho(\vec{r}) &= \sum_{v=1}^{K}\sum_{\mu=1}^{K}\left[2\sum_{n}^{N/2} C_{vn}^*C_{\mu n}\right]\phi_v^*(\vec{r})\phi_\mu(\vec{r}) \\
\rho(\vec{r}) &= \sum_{v=1}^{K}\sum_{\mu=1}^{K} P_{v\mu}\phi_v^*(\vec{r})\phi_\mu(\vec{r}) \\
P_{v\mu} &= 2\sum_{n}^{N/2} C_{vn}^*C_{\mu n}
\end{aligned}$$

The density matrix is composed of the coefficients of the basis functions, and it describes the charge density $\rho(\vec{r})$ for the electrons in the system.

Now we have all the pieces to expand the Fock operator into its new basis function matrix form:

$$
\begin{aligned}
F_{v\mu} &= \int_{-\infty}^{\infty} \phi_v^*(\vec{r})\hat{F}(\vec{r})\phi_\mu(\vec{r})d\vec{r} \\
F_{v\mu} &= \int_{-\infty}^{\infty} \phi_v^*(\vec{r})\hat{H}(\vec{r})\phi_\mu(\vec{r})d\vec{r} + \sum_{n=1}^{N/2}\int_{-\infty}^{\infty} \phi_v^*(\vec{r})\left[2J_n(\vec{r}) - K_n(\vec{r})\right]\phi_\mu(\vec{r})d\vec{r} \\
F_{v\mu} &= H_{v\mu}^{core} + \sum_{n=1}^{N/2} 2(v\mu|nn) - (vn|\mu n)
\end{aligned}
$$

The core Hamiltonian matrix element $H_{v\mu}^{core}$ can be further expanded into the kinetic energy integrals $T_{v\mu}$ and nuclear attraction integrals $V_{v\mu}$:

$$
\begin{aligned}
T_{v\mu} &= \int_{-\infty}^{\infty} \phi_v^*(\vec{r})\left[-\frac{1}{2}\nabla^2\right]\phi_\mu(\vec{r})d\vec{r} \\
V_{v\mu} &= \int_{-\infty}^{\infty} \phi_v^*(\vec{r})\left[-\sum_{A=1}^{M}\frac{Z_A}{\vec{r}-\vec{R}_A}\right]\phi_\mu(\vec{r})d\vec{r} \\
H_{v\mu}^{core} &= T_{v\mu} + V_{v\mu}
\end{aligned}
$$

Interesting to note here is that the Hamiltonian matrix does not depend on the coefficients, which are the elements being iteratively solved in the algorithm. Therefore, the Hamiltonian matrix only needs to be evaluated once during the process. This is unlike the previous assignment, where the Hamiltonian needed to be evaluated at every iteration.

The last piece of the puzzle is to introduce the linear expansion into the two-electron integral using the work derived above for the density matrix $\boldsymbol{P}$:

$$
\begin{aligned}
F_{v\mu} &= H_{v\mu}^{core} + \sum_{n=1}^{N/2}\sum_{\lambda=1}^{K}\sum_{\sigma=1}^{K} C_{\lambda n}^* C_{\sigma n}\left[2(v\mu|\sigma\lambda) - (v\lambda|\sigma\mu)\right] \\
P_{\lambda\sigma} &= 2\sum_{n}^{N/2} C_{\lambda n}^* C_{\sigma n} \\
F_{v\mu} &= H_{v\mu}^{core} + \sum_{\lambda=1}^{K}\sum_{\sigma=1}^{K} P_{\lambda\sigma}\left[(v\mu|\sigma\lambda) - \frac{1}{2}(v\lambda|\sigma\mu)\right] \\
F_{v\mu} &= H_{v\mu}^{core} + G_{v\mu}
\end{aligned}
$$

The two-electron intergrals expand as follows:

$$
(v\mu|\sigma\lambda) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \phi_v^*(\vec{r}_1)\phi_\mu(\vec{r}_1)\frac{1}{\|\vec{r}_{12}\|}\phi_\sigma^*(\vec{r}_2)\phi_\lambda(\vec{r}_2)d\vec{r}_1 d\vec{r}_2
$$

For every iteration of the Hartree-Fock algorithm, the matrix $\boldsymbol{G}$ will need to be recalculated which is reliant on the density matrix $\boldsymbol{P}$.

2.3 Orthogonalization of the Basis

In order to solve an eigenvalue problem on every iteration, it will be necessary to reconfigure the matrix Fock equation from:

$$\boldsymbol{FC} = \boldsymbol{SC\epsilon}$$

to:

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

Which is a form where the modified Fock matrix $\boldsymbol{F'}$ can be diagonalized to solve for $\boldsymbol{C'}$ which in turn is converted back into $\boldsymbol{C}$. In this modified equation, the overlap matrix $\boldsymbol{S}$ disappears as a transformation is applied to orthonormalize the basis functions. Note that the overlap matrix $\boldsymbol{S}$ is present because the basis functions chosen are not normally orthonormal.

A transformation matrix $\boldsymbol{X}$ needs to be found to satisfy the following condition:

$$\boldsymbol{X}^\dagger \boldsymbol{SX} = 1$$

This assignment will using symmetric orthogonolization, which uses the inverse square root of $\boldsymbol{S}$ as $\boldsymbol{X}$:

$$\boldsymbol{X} = \boldsymbol{S}^{-\frac{1}{2}} = \boldsymbol{U}s^{-\frac{1}{2}}\boldsymbol{U}^\dagger$$

To obtain the matrix $\boldsymbol{S}^{-\frac{1}{2}}$, the overlap matrix is diagonalized, and the resulting eigenvalues are inverse square rooted and form a diagonal matrix $s^{-\frac{1}{2}}$. It is then undiagonalized by multiplying the unitary matrices as shown in the previous equation.

The expansion coefficient matrix $\boldsymbol{C}$ can then be transformed to and from its modified state using $\boldsymbol{X}$ as follows:

$$\boldsymbol{C'} = \boldsymbol{X}^{-1}\boldsymbol{C} \qquad \boldsymbol{C} = \boldsymbol{XC'}$$

Similarly, the modified Fock matrix $\boldsymbol{F'}$ can be obtained as follows:

$$\boldsymbol{F'} = \boldsymbol{X}^\dagger \boldsymbol{FX}$$

2.4 Choosing the Basis

For this assignment, the STO-1G basis sets will be used for the basis functions. STO-1G are Gaussian approximations of the more realistic Slater functions for orbitals (hence the namesake Slater-Type Orbital:

STO) [1]. These basis sets represent a 1s atomic orbital approximated by a single Gaussian function each and have the following generic form:

$$\phi = ae^{-b\vec{r}^2}$$

Tuned STO-1G basis functions were found in [2, p. 214][3] for both the 1s orbital of the Hydrogen atom as well as the Helium atom with values for a and b as follows:

$$\phi_H = 0.3696e^{-0.4166|\vec{r}-\vec{R}|^2}$$
$$\phi_{He} = 0.5881e^{-0.7739|\vec{r}-\vec{R}|^2}$$

For the Helium atom, two of the Helium 1s orbital STO-1G basis functions will be centered on the nucleus, which exists at the origin of the problem. For the Hydrogen molecule, two of the the Hydrogen 1s orbital STO-1G basis functions will be center on each nucleus, for a total of four basis functions. As calculated in the previous assignment, the distance between the two Hydrogen nuclei is $\frac{0.74\times0.1\times10^{-9}}{5.29177210903\times10^{-11}} = 1.39839733222307$ atomic units of length. The Hydrogen molecule nuclei will be placed on the x axis.

2.5 Precalculating the Integrals

An advantage of using basis functions is that many of the integrals required for the resulting matrices can be pre-computed as they do not alter between iterations. For the one-electron integrals, one must calculate the kinetic energy integral for every entry in the kinetic energy matrix and the potential energy integral for the potential energy matrix. These two matrices make up the core Hamiltonian matrix which is composed of one-electron integrals. Similarly, one must calculate the two-electron integrals, however this piece is more complicated and more computationally involved. The result of these two-electron integrals will make up the $\boldsymbol{G}$ matrix mentioned previously, which consists of the density matrix $\boldsymbol{P}$ multiplied by the Coulomb repulsion two-electron integral and the exchange operator two-electron integral. Only the density matrix will be modified, so the two-electron integrals that the density matrix elements will be multiplying can be precomputed.

In order to ease the burden of integrating the two-electron integrals, they will be simplified by invoking the Gaussian product rule since Gaussian functions are used as the basis functions. This transforms the integrand from four centers ($\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}$) to two ($\boldsymbol{P}$, $\boldsymbol{Q}$). The integral is further simplified by replacing the factors of the resulting integrand with their Fourier transforms, which eventually collapses into a single integration. The full derivation can be found in [4, p. 153-155,157]. This will simplify the two-electron integral to the following equivalent:

$$\begin{aligned}(v\mu|\sigma\lambda) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\phi_v^*(\vec{r}_1)\phi_\mu(\vec{r}_1)\frac{1}{\|\vec{r}_{12}\|}\phi_\sigma^*(\vec{r}_2)\phi_\lambda(\vec{r}_2)d\vec{r}_1 d\vec{r}_2 \\ &= G_{AB}G_{CD}\frac{2\pi^{\frac{5}{2}}}{\zeta\eta(\zeta+\eta)^{\frac{1}{2}}}\int_0^1 e^{-Tu^2}du\end{aligned}$$

This single variable integral is evaluated very quickly, saving a lot of time in the two-electron integral step. If time permitted, the same simplifications would have been applied to the other one-electron integrals as well.

2.6 Calculating the Total Energy

The total energy of the system can be obtained through the following equation [5, p. 150]:

$$E_o = \frac{1}{2}\sum_{v=1}^{K}\sum_{\mu=1}^{K} P_{v\mu}\left[H_{v\mu}^{core} + F_{v\mu}\right]$$

This will be calculated at every step to give an indicator of how the algorithm is converging.

2.7 Final Recipe

We now have all of the background required to implement our solution. The following steps outline what is required for this simulation to run:

1. Prepare $K \times K$ empty matrices where K corresponds to the number of basis functions
2. Pre-calculate one and two-electron integrals using the specified basis set chosen
   - Overlap matrix $\boldsymbol{S}$ containing one electron integrals $S_{v\mu}$
   - Kinetic $\boldsymbol{T}$ energy matrix one electron integrals $T_{v\mu}$
   - Nuclear attraction $\boldsymbol{V}$ matrix one electron integrals $V_{v\mu}$
   - $\boldsymbol{G}$ matrix Coulomb repulsion and exchange two-electron integrals
3. Fill in $\boldsymbol{T}$ and $\boldsymbol{V}$ matrices using one-electron integrals
4. Obtain the transformation matrix $\boldsymbol{X}$ by obtaining $\boldsymbol{X} = \boldsymbol{S}^{-\frac{1}{2}}$
   - Diagonalize $\boldsymbol{S}$ to find its eigenvalues
   - Create a new matrix with the eigenvalues set to the negative half power on the diagonal: $\boldsymbol{s}^{-\frac{1}{2}}$
   - Form the transformation matrix $\boldsymbol{X}$ by calculating $\boldsymbol{U}\boldsymbol{s}^{-\frac{1}{2}}\boldsymbol{U}^{\dagger}$
5. Calculate the $\boldsymbol{G}$ matrix using the current density matrix $\boldsymbol{P}$ (zero on the first iteration) and the two-electron integrals
6. Calculate the $\boldsymbol{F}$ Fock matrix using $\boldsymbol{T} + \boldsymbol{V} + \boldsymbol{G}$
7. Apply the transformation matrix $\boldsymbol{X}$ to obtain the orthonormalized Fock matrix $\boldsymbol{F}' = \boldsymbol{X}^{\dagger}\boldsymbol{F}\boldsymbol{X}$:
8. Diagonalize the orthonormalized Fock matrix $\boldsymbol{F}'$ to obtain the orthonormalized coefficient matrix $\boldsymbol{C}'$
9. Convert the orthonormalized coefficient matrix $\boldsymbol{C}'$ to the regular coefficient matrix $\boldsymbol{C}$: $\boldsymbol{C} = \boldsymbol{X}\boldsymbol{C}'$
10. Extract the coefficients from $\boldsymbol{C}$ to obtain a new version of the $\boldsymbol{P}$ matrix
11. Compare this new density matrix $\boldsymbol{P}$ with the one we started with, if the difference between the two is not within a specified threshold, repeat the process from 4. If the difference between both $\boldsymbol{P}$ is within the threshold, the algorithm is complete.

3 Program Implementation Details

- On startup, the program searches for two JSON files containing pre-calculated integrals for both the Helium atom and the Hydrogen molecule. These files include all of the combinations that produce a unique integration result provided that the basis functions are different. If the files are not found, then the program pre-calculates the integrals and saved the result in these files. Multiprocessing is used to calculate an integral per available CPU core, helping to parallelize the workload.
- The combinations for the one electron and two electron integrals attempt to reduce the amount of integrations performed by the program. For the one-electron integrals, the combinations will exclude those whose flipped indices are already accounted for as they produce the same results. For the two electron integrals, a similar process is performed on the pairs of indices on either side of the repulsion term where flipped indices are discarded. The symmetry where the two functions are swapped at the same time on either side is also respeted. This can be summarized as follows: $(rs|t\mu) = (rs|\mu t) = (sr|t\mu) = (sr|\mu t) = (t\mu|rs) = (t\mu|sr) = (\mu t|rs) = (\mu t|sr)$ [3].
- Classes are defined for each basis function which provides the ability to extract the two coefficients of the basis function, define where the basis function is centered, and evaluate the function as a member function of the class. Since the program uses both naive and optimized equations for the integration, the naive version uses the full function defined in the class and the optimized version uses the coefficients of the Gaussian functions.
- The `sympy` library is used to symbolically define the functions, which is helpful in perform operations such as the Laplacian in the kinetic energy integrand. The resulting symbolic representation of each naive integrand is then converted into a Python lambda function to be numerically evaluated through the `scipy` package.
- The main loop of the program generally follows what was written in the final recipe section. The difference between the previous P matrix and the new P matrix is used as the convergence criteria to know when the program has ended. The difference is calculated as the mean percent difference of all P matrix entries.

4 Discussion

- As submitted, the current program does not work as expected. After only two iterations, the program thinks it obtains a convergence because the $\boldsymbol{P}$ matrices barely changes or doesn't change at all after the first iteration. As a result, the $\boldsymbol{G}$ matrix remains the same between the two iterations producing an identical result. Looking at the resulting coefficient matrix $\boldsymbol{C}$, the coefficients appear to barely change, only the sign appears to change. Could this be because two exact basis functions were chosen for each atom?
- With the current program, the total energy calculated for the Helium atom is -3.045430 Hartrees and for the Hydrogen molecule is -24.154613 Hartrees.
- Using Abacus, Hartree-Fock was run using the STO-3G basis set to calculate the molecular orbitals for both the Helium atom and the Hydrogen molecule. The Helium atom's first orbital is calculated to have an energy of -636.746 eV or -23.3999729 Hartrees and the Hydrogen molecule first orbital is calculated to have an energy of -480.416 eV or -17.6549541 Hartrees. Unfortunately, the results obtained in this program do not appear to be correct.
- Another round of integration was performed with basis functions that have slightly different alpha values. However, this did not affect the algorithm in a meaningful way. While many calculation errors were found in followup revisions of the code and theory, none of the fixes made any difference to the solution. There is very likely a logical error somewhere in the code or the theoretical interpretation that is causing the program to behave improperly. Other Hartree-Fock implementations using Roothaan-Hall equations were studied online, but no major deviations were found between them and this program's implementation.
- If time permitted, the simplified integrals outlined in [4] would have been applied to the other, one-electron integrals in this assignment as well. The amount of time saved is phenomenal. Unfortunately, the resource was discovered too late in the game.
- If the program had worked, the resulting linear combinations of wave functions would have been plotted in a similar manner to the previous assignment as a comparison.

5 Code Listings and Data

5.1 Python Code Listing

The following is the code written in Python to generate the solutions and plots used in this report.

```
"""
MIT License

Copyright (c) [2022] [Michel Kakulphimp]

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
"""

import argparse
import matplotlib
import matplotlib.pyplot as plt
import numpy
import scipy
import scipy.sparse
import scipy.sparse.linalg
import scipy.integrate
import sympy
import sympy.vector
import math
import functools
import itertools
import time
import pickle
import datetime # for timestamping
import multiprocessing # for multiprocessing (MP) of matrix generation
import tqdm # progress bar for MP
import sys
import json
import signal
import copy

numpy.set_printoptions(precision=None, suppress=None, edgeitems=30, linewidth=100000,
    formatter=dict(float=lambda x: '%.3g' % x))

# Matplotlib export settings
if False:
    matplotlib.use('pgf')
    matplotlib.rcParams.update({
        'pgf.texsystem': 'pdflatex',
        'font.size': 10 ,
        'font.family': 'serif',  # use serif/main font for text elements
        'text.usetex': True,     # use inline math for ticks
        'pgf.rcfonts': False     # don't setup fonts from rc parameters
    })
```

```
# program constants
NUM_ELECTRONS = 2
H2_BOND_LENGTH_ATOMIC_UNITS = 1.39839733222307
TINY_NUMBER = 1e-9
DATETIME_STR_FORMAT = '[%Y/%m/%d-%H:%M:%S]'
HE_NUM_BASIS_FUNCTIONS = 2
H2_NUM_BASIS_FUNCTIONS = 4
HE_INTEGRALS_FILENAME = 'he_integrals.json'
H2_INTEGRALS_FILENAME = 'h2_integrals.json'
PROGRAM_VERBOSITY = 1
# dictionary keys
OVERLAP = 'overlap'
KINETIC = 'kinetic'
ATTRACTION = 'attraction'
EXCHANGE = 'exchange'

#
# This is the main function
#
def main(cmd_args):

    # read from file or pre-calculate one-electron and two-electron integrals
    he_integrals, h2_integrals = precalculate_integrals()

    # do HF for He
    do_hartree(HE_NUM_BASIS_FUNCTIONS, he_integrals)
    # do HF for H2
    do_hartree(H2_NUM_BASIS_FUNCTIONS, h2_integrals)


def do_hartree(num_basis_functions, integrals):

    # prepare matrices P, S, T, V, G, F
    p_matrix = numpy.empty((num_basis_functions,num_basis_functions))
    new_p_matrix = numpy.empty((num_basis_functions,num_basis_functions))
    s_matrix = numpy.empty((num_basis_functions,num_basis_functions))
    t_matrix = numpy.empty((num_basis_functions,num_basis_functions))
    v_matrix = numpy.empty((num_basis_functions,num_basis_functions))
    g_matrix = numpy.empty((num_basis_functions,num_basis_functions))
    f_matrix = numpy.empty((num_basis_functions,num_basis_functions))
    f_prime_matrix = numpy.empty((num_basis_functions,num_basis_functions))

    # fill up S matrix
    for v in range(num_basis_functions):
        for u in range(num_basis_functions):
            # sort the combination to obtain the unique integral result
            combination = tuple(sorted([v,u]))
            s_matrix[v,u] = integrals[OVERLAP][combination]

    console_print(1, 'S␣matrix:')
    console_print(1, str(s_matrix))

    # fill up T matrix
    for v in range(num_basis_functions):
        for u in range(num_basis_functions):
            # sort the combination to obtain the unique integral result
            combination = tuple(sorted([v,u]))
            t_matrix[v,u] = integrals[KINETIC][combination]

    console_print(1, 'T␣matrix:')
    console_print(1, str(t_matrix))

    # fill up V matrix
    for v in range(num_basis_functions):
        for u in range(num_basis_functions):
            # sort the combination to obtain the unique integral result
            combination = tuple(sorted([v,u]))
            v_matrix[v,u] = integrals[ATTRACTION][combination]

    console_print(1, 'V␣matrix:')
    console_print(1, str(v_matrix))

    # obtain transformation matrix X through S^-0.5
    # obtain eigenvalues
    s_eigenvalues, s_eigenvectors = numpy.linalg.eigh(s_matrix)
    # sort eigenthings
    sorted_indices = numpy.argsort(s_eigenvalues)
    s_eigenvalues = s_eigenvalues[sorted_indices]
    s_eigenvectors = s_eigenvectors[:,sorted_indices]
    console_print(1, 'S␣eigenvalues:')
    console_print(1, str(s_eigenvalues))
    console_print(1, 'S␣eigenvectors:')
    console_print(1, str(s_eigenvectors))
    # inverse square root the resulting eigenvalues and put them in a diagonal matrix
    # add a TINY_NUMBER to avoid div by zero
    s_eigenvalues = numpy.array([eigenvalue + TINY_NUMBER for eigenvalue in s_eigenvalues])
    s_eigenvalues_inverse_square_root = numpy.diag(s_eigenvalues**-0.5)
    # form the transformation matrix X by undiagonalizing the previous matrix
    x_matrix = s_eigenvectors @ s_eigenvalues_inverse_square_root @ numpy.transpose(
        s_eigenvectors)

    # MAIN HF LOOP
    iteration = 0
    while True:

        console_print(0, '\n␣****␣ITERATION␣%d␣****␣\n' % iteration)

        # calculate G matrix using density matrix P and two-electron integrals
        for v in range(num_basis_functions):
            for u in range(num_basis_functions):
                for lambda_ in range(num_basis_functions):
                    for sigma in range(num_basis_functions):
                        # get index
                        # Coulomb attraction two-electron integral
                        coulomb_combination = [v,u,sigma,lambda_]
                        # Exchange two-electron integral
                        exchange_combination = [v,lambda_,sigma,u]
                        # Avoid calculating identical integrals
                        # (rs|tu) = (rs|ut) = (sr|tu) = (sr|ut) = (tu|rs) = (tu|sr) = (ut|rs
                            ) = (ut|sr)
                        coulomb_combination = tuple(sorted(coulomb_combination[0:2]) +
                            sorted(coulomb_combination[2:4]))
                        exchange_combination = tuple(sorted(exchange_combination[0:2]) +
                            sorted(exchange_combination[2:4]))
                        coulomb_combination_swapped = tuple(coulomb_combination[2:4] +
                            coulomb_combination[0:2])
                        exchange_combination_swapped = tuple(exchange_combination[2:4] +
                            exchange_combination[0:2])

                        # integral is going to exist in dictionary as _combination or
                            _combination_swapped
                        # TODO: there's got to be a better way!
                        try:
                            coulomb_term = integrals[EXCHANGE][coulomb_combination]
                        except:
                            coulomb_term = integrals[EXCHANGE][coulomb_combination_swapped]

                        try:
                            exchange_term = integrals[EXCHANGE][exchange_combination]
                        except:
                            exchange_term = integrals[EXCHANGE][exchange_combination_swapped
                                ]

                        g_matrix[v,u] = p_matrix[lambda_,sigma]*(coulomb_term - 0.5*
                            exchange_term)
        console_print(1, 'G␣matrix:')
        console_print(1, str(g_matrix))

        # calculate F = T + V + G
        f_matrix = t_matrix + v_matrix + g_matrix

        console_print(1, 'F␣matrix:')
        console_print(1, str(f_matrix))

        # apply transform X to obtain F'
        f_prime_matrix = numpy.transpose(x_matrix) @ f_matrix @ x_matrix

        console_print(1, 'F\'␣matrix:')
        console_print(1, str(f_prime_matrix))

        # diagonalize F' to get C'
        f_prime_eigenvalues, c_prime_matrix = numpy.linalg.eigh(f_prime_matrix)
        # sort eigenthings
        sorted_indices = numpy.argsort(f_prime_eigenvalues)
        f_prime_eigenvalues = s_eigenvalues[sorted_indices]
        c_prime_matrix = c_prime_matrix[:,sorted_indices]
        console_print(1, 'F\'␣eigenvalues:')
        console_print(1, str(f_prime_eigenvalues))
        console_print(1, 'C\'␣matrix:')
        console_print(1, str(c_prime_matrix))

        # convert C' to C to get a new P
        c_matrix = x_matrix @ c_prime_matrix

        console_print(1, 'C␣matrix:')
        console_print(1, str(c_matrix))

        console_print(1, 'old␣P␣matrix:')
        console_print(1, str(p_matrix))

        # calculate the new P matrix
        for v in range(num_basis_functions):
            for u in range(num_basis_functions):
                new_p_matrix[v,u] = 0
                for n in range(int(NUM_ELECTRONS/2)):
                    new_p_matrix[v,u] = new_p_matrix[v,u] + 2*c_matrix[v,n]*c_matrix[u,n]
                    console_print(2, 'new_p_matrix[%d,%d]␣=␣%f' % (v,u,new_p_matrix[v,u]))
                    console_print(2, 'c_matrix[%d,%d]␣=␣%f' % (v,n,c_matrix[v,n]))
                    console_print(2, 'c_matrix[%d,%d]␣=␣%f' % (u,n,c_matrix[u,n]))
                    console_print(2, 'new_p_matrix[%d,%d]␣=␣new_p_matrix[%d,%d]␣+␣2*c_matrix
                        [%d,%d]*c_matrix[%d,%d]␣=␣%f' % (v,u,v,u,v,n,u,n,new_p_matrix[v,u]))

        console_print(1, 'new␣P␣matrix:')
        console_print(1, str(new_p_matrix))

        # compare old and new P matrix
        # get the average percent difference of all elements
        delta = 0
        for v in range(num_basis_functions):
            for u in range(num_basis_functions):
                console_print(2, 'v=%d␣u=%d␣error:␣%f' % (v, u, abs(new_p_matrix[v,u] -
                    p_matrix[v,u])/abs((new_p_matrix[v,u] + p_matrix[v,u])/2)))
                delta = delta + (p_matrix[v,u] - new_p_matrix[v,u])**2
        delta = (delta / num_basis_functions)**0.5

        # set old p matrix to new
        p_matrix = new_p_matrix

        # calculate total energy, check for convergence
        total_energy_sum = 0
        for v in range(num_basis_functions):
            for u in range(num_basis_functions):
                total_energy_sum = total_energy_sum + p_matrix[v,u]*(t_matrix[v,u] +
                    v_matrix[v,u] + f_matrix[v,u])
        total_energy = 0.5*total_energy_sum

        console_print(0, '\t\tTotal␣energy:␣%f' % total_energy)
        console_print(0, '\t\tDelta␣between␣P␣matrices:␣%f' % delta)

        # increment iteration
        iteration = iteration + 1

        # check for end condition
        if delta < 0.00001 or iteration > 10:
            break
#
# Initializer for child processes to respect SIGINT
#
def initializer():
    #Ignore SIGINT in child workers
    signal.signal(signal.SIGINT, signal.SIG_IGN)

#
# Console formatter
#
def console_print(verbose_level=0, string='', end='\n'):

    if verbose_level > PROGRAM_VERBOSITY:
        return

    # get str representation
    if not isinstance(string, str):
        string = str(string)

    # split string at new lines
    string = string.split('\n')

    # write out line by line
    for string_line in string:
        datetime_now = datetime.datetime.now()
        print(datetime_now.strftime(DATETIME_STR_FORMAT) + '␣' + string_line, end=end)

#
# Helium STO-1G basis function class
#
class sto_1g_helium_class:

    k = 0.5881
    alpha = 0.7739

    def __init__(self, center=(0,0,0), modifier=1.0):
        self.center = center
        self.alpha = self.alpha*modifier

    def eval(self, z, y, x):
        return self.k*sympy.exp(-self.alpha*(sympy.sqrt((x - self.center[0])**2 + (y - self.
            center[1])**2 + (z - self.center[2])**2)**2))

#
# Hydrogen STO-1G basis function class
#
class sto_1g_hydrogen_class:

    k = 0.3696
    alpha = 0.4166

    def __init__(self, center=(0,0,0), modifier=1.0):
        self.center = center
        self.alpha = self.alpha*modifier
    def eval(self, z, y, x):
        return self.k*sympy.exp(-self.alpha*(sympy.sqrt((x - self.center[0])**2 + (y - self.
            center[1])**2 + (z - self.center[2])**2)**2))

#
# Calculate the overlap integrals given the function lookup table and the desired
    combinations
#
def calculate_overlap_integrals_naive(funcs):

    # sympy regular symbolic variables
    x, y, z = sympy.symbols('x␣y␣z')

    overlap_ints = {}

    # symbolic version of the integrand
    overlap_intgd_sym = sympy.simplify(funcs[0](z, y, x)*funcs[1](z, y, x))
    # numerical version of the integrand
    overlap_intgd_num = sympy.lambdify([z, y, x], overlap_intgd_sym, 'scipy')
    # integrate (first index of tuple contains result)
    overlap_int_val = scipy.integrate.nquad(overlap_intgd_num, [[-scipy.inf, scipy.inf], [-
        scipy.inf, scipy.inf], [-scipy.inf, scipy.inf]])[0]

    return overlap_int_val

#
# Calculate the overlap integrals given the function lookup table and the desired
    combinations
#
def calculate_overlap_integrals_optimized(func_objs):

    # combine the outer K coefficients into a single one that will multiply the final result
    k_all = func_objs[0].k * func_objs[1].k

    # get the four different function alphas
    alpha = func_objs[0].alpha
    gamma = func_objs[1].alpha

    # get the two different centers, A, C
    center_a = scipy.array(func_objs[0].center)
    center_c = scipy.array(func_objs[1].center)

    overlap_int_val = k_all*((scipy.pi/(alpha + gamma))**(3.0/2.0))*scipy.exp(-alpha*gamma*(
        calculate_distance(center_a, center_c)**2)/(alpha + gamma))

    return overlap_int_val

#
# Calculate the kinetic energy integrals given the function lookup table and the desired
    combinations
#
def calculate_kinetic_energy_integrals(funcs):

    # sympy vector x,y,z coordinate system
    R = sympy.vector.CoordSys3D('R')

    # symbolic version of the integrand
    kinetic_energy_intgd_sym = sympy.simplify(funcs[0](R.z, R.y, R.x)*(-1/2)*sympy.vector.
        Laplacian(funcs[1](R.z, R.y, R.x)).doit())
    # numerical version of the integrand
    kinetic_energy_intgd_num = sympy.lambdify([R.z, R.y, R.x], kinetic_energy_intgd_sym, '
        scipy')
    # integrate (first index of tuple contains result)
    kinetic_energy_int_val = scipy.integrate.nquad(kinetic_energy_intgd_num, [[-scipy.inf,
        scipy.inf], [-scipy.inf, scipy.inf], [-scipy.inf, scipy.inf]])[0]

    return kinetic_energy_int_val
```

```
#
# Calculate the nuclear attraction integrals given the function lookup table
# and the desired combinations.  This is the naive version with no
# optimizations to the integrand.
#
def calculate_nuclear_attraction_integrals_naive(subject, funcs):

    # sympy regular symbolic variables
    x, y, z = sympy.symbols('x␣y␣z')

    # symbolic version of the integrand
    if subject == 'he':
        nuclear_attraction_intgd_sym = sympy.simplify(funcs[0](z, y, x) * (-2/sympy.sqrt(x
            **2 + y**2 + z**2)) * funcs[1](z, y, x))
    elif subject == 'h2':
        nuclear_attraction_intgd_sym = sympy.simplify(funcs[0](z, y, x) * (-1/sympy.sqrt((x
            - (-H2_BOND_LENGTH_ATOMIC_UNITS/2.0))**2 + y**2 + z**2)) + (-1/sympy.sqrt((x - (
            H2_BOND_LENGTH_ATOMIC_UNITS/2.0))**2 + y**2 + z**2)) * funcs[1](z, y, x))
    # numerical version of the integrand
    nuclear_attraction_intgd_num = sympy.lambdify([z, y, x], nuclear_attraction_intgd_sym, '
        scipy')
    # integrate (first index of tuple contains result)
    nuclear_attraction_int_val = scipy.integrate.nquad(nuclear_attraction_intgd_num, [[-
        scipy.inf, scipy.inf], [-scipy.inf, scipy.inf], [-scipy.inf, scipy.inf]])[0]

    return nuclear_attraction_int_val

#
# Calculate the nuclear attraction integrals given the function lookup table
# and the desired combinations. This is the optimized version to help speed
# up calculations. Note that this optmized version of the nuclear attraction
# integral optimizes into the same form as the exchange integral optimization.
#
def calculate_nuclear_attraction_integrals_optimized(subject, func_objs):

    # sympy regular symbolic variables
    u = sympy.symbols('u')

    # combine the outer K coefficients into a single one that will multiply the final result
    k_all = func_objs[0].k * func_objs[1].k

    # get the four different function alphas
    alpha = func_objs[0].alpha
    beta = func_objs[1].alpha
    gamma = 1e9 # large number to avoid infinity
    delta = 0.0

    # get the four different centers, A, B, C, D
    center_a = scipy.array(func_objs[0].center)
    center_b = scipy.array(func_objs[1].center)
    # C becomes the nuclear center
    center_c_he = scipy.array((0,0,0))
    center_c_h2_0 = scipy.array(((-H2_BOND_LENGTH_ATOMIC_UNITS/2.0),0,0))
    center_c_h2_1 = scipy.array(((H2_BOND_LENGTH_ATOMIC_UNITS/2.0),0,0))
    # unused
    center_d = scipy.array((0,0,0))

    # calculate the G coefficients
    g_ab = numpy.exp(((-alpha * beta)/(alpha + beta))*(calculate_distance(center_a, center_b
        )**2))
    g_cd = 1.0 # numpy.exp(((-gamma * delta)/(gamma + delta))*(calculate_distance(center_c,
        center_d)**2)) # this turns into 1.0 since delta = 0.0

    # calculate zeta and eta
    zeta = alpha + beta
    eta = gamma + delta

    # calculate new centers Q and P
    center_p = (((alpha)/(alpha + beta)) * center_a) + (((beta)/(alpha + beta)) * center_b)
    center_q_he = (((gamma)/(gamma + delta)) * center_c_he) + (((delta)/(gamma + delta)) *
        center_d)
    center_q_h2_0 = (((gamma)/(gamma + delta)) * center_c_h2_0) + (((delta)/(gamma + delta))
         * center_d)
    center_q_h2_1 = (((gamma)/(gamma + delta)) * center_c_h2_1) + (((delta)/(gamma + delta))
         * center_d)

    # calculate fancy v**2
    v_squared = ((zeta*eta)/(zeta + eta))

    # calculate T
    t_he = v_squared * calculate_distance(center_q_he, center_p)
    t_h2_0 = v_squared * calculate_distance(center_q_h2_0, center_p)
    t_h2_1 = v_squared * calculate_distance(center_q_h2_1, center_p)

    # combined constant
    coeff = (k_all*g_ab*g_cd)*((2*scipy.pi**(5/2))/(zeta*eta*scipy.sqrt(zeta + eta)))

    # symbolic version of the integrand
    fundamental_electron_repulsion_he_intgd_sym = coeff*sympy.exp(-t_he*u**2)
    fundamental_electron_repulsion_h2_0_intgd_sym = coeff*sympy.exp(-t_h2_0*u**2)
    fundamental_electron_repulsion_h2_1_intgd_sym = coeff*sympy.exp(-t_h2_1*u**2)
    # numerical version of the integrand
    if subject == 'he':
        fundamental_electron_repulsion_intgd_num = sympy.lambdify([u],
            fundamental_electron_repulsion_he_intgd_sym, 'scipy')
        # integrate (first index of tuple contains result)
        fundamental_electron_repulsion_int_val = scipy.integrate.nquad(
            fundamental_electron_repulsion_intgd_num, [[0, 1]])
    elif subject == 'h2':
        fundamental_electron_repulsion_intgd_num_0 = sympy.lambdify([u],
            fundamental_electron_repulsion_h2_0_intgd_sym, 'scipy')
        fundamental_electron_repulsion_intgd_num_1 = sympy.lambdify([u],
            fundamental_electron_repulsion_h2_1_intgd_sym, 'scipy')
        # integrate (first index of tuple contains result)
        fundamental_electron_repulsion_int_val_0 = scipy.integrate.nquad(
            fundamental_electron_repulsion_intgd_num_0, [[0, 1]])
        # integrate (first index of tuple contains result)
        fundamental_electron_repulsion_int_val_1 = scipy.integrate.nquad(
            fundamental_electron_repulsion_intgd_num_1, [[0, 1]])

    if subject == 'he':
        result = 2*fundamental_electron_repulsion_int_val[0]
    elif subject == 'h2':
        result = fundamental_electron_repulsion_int_val_0[0] +
            fundamental_electron_repulsion_int_val_1[0]

    return result


#
# Calculate the Coulomb repulsion and exchange integrals given the function
# lookup table and the desired combinations. This is the naive version with
# no optimizations to the integrand.
#
def calculate_coulomb_repulsion_and_exchange_integrals_naive(funcs):

    # sympy regular symbolic variables
    x, y, z = sympy.symbols('x␣y␣z')
    u, v, w = sympy.symbols('u␣v␣w')

    # nquad parameters
    limits = 1e3
    err = 1e-3

    # symbolic version of the integrand
    coulomb_repulsion_and_exchange_intgd_sym = sympy.simplify(funcs[0](z, y, x) * funcs[1](z
        , y, x) * (1/sympy.sqrt(TINY_NUMBER + (u-x)**2 + (v-y)**2 + (w-z)**2)) * funcs[2](w,
         v, u) * funcs[3](w, v, u))
    # numerical version of the integrand
    coulomb_repulsion_and_exchange_intgd_num = sympy.lambdify([z, y, x, w, v, u],
        coulomb_repulsion_and_exchange_intgd_sym, 'scipy')
    # integrate (first index of tuple contains result)
    coulomb_repulsion_and_exchange_int_val = scipy.integrate.nquad(
        coulomb_repulsion_and_exchange_intgd_num, [[-limits, limits]]*6, opts={'epsabs':err,
         'epsrel':err}, full_output=True)

    return coulomb_repulsion_and_exchange_int_val[0]

#
# Calculate the Coulomb repulsion and exchange integrals given the function
# lookup table and the desired combinations. This is the optimized version to
# help speed up calculations.
#
def calculate_coulomb_repulsion_and_exchange_integrals_optimized(func_objs):

    # sympy regular symbolic variables
    u = sympy.symbols('u')

    # combine the outer K coefficients into a single one that will multiply the final result
    k_all = func_objs[0].k * func_objs[1].k * func_objs[2].k * func_objs[3].k

    # get the four different function alphas
    alpha = func_objs[0].alpha
    beta = func_objs[1].alpha
    gamma = func_objs[2].alpha
    delta = func_objs[3].alpha

    # get the four different centers, A, B, C, D
    center_a = scipy.array(func_objs[0].center)
    center_b = scipy.array(func_objs[1].center)
    center_c = scipy.array(func_objs[2].center)
    center_d = scipy.array(func_objs[3].center)

    # calculate the G coefficients
    g_ab = numpy.exp(((-alpha * beta)/(alpha + beta))*(calculate_distance(center_a, center_b
        )**2))
    g_cd = numpy.exp(((-gamma * delta)/(gamma + delta))*(calculate_distance(center_c,
        center_d)**2))

    # calculate zeta and eta
    zeta = alpha + beta
    eta = gamma + delta

    # calculate new centers Q and P
    center_p = (((alpha)/(alpha + beta)) * center_a) + (((beta)/(alpha + beta)) * center_b)
    center_q = (((gamma)/(gamma + delta)) * center_c) + (((delta)/(gamma + delta)) *
        center_d)

    # calculate fancy v**2
    v_squared = ((zeta*eta)/(zeta + eta))

    # calculate T
    t = v_squared * calculate_distance(center_q, center_p)

    # combined constant
    coeff = (k_all*g_ab*g_cd)*((2*scipy.pi**(5/2))/(zeta*eta*scipy.sqrt(zeta + eta)))

    # symbolic version of the integrand
    coulomb_repulsion_and_exchange_intgd_sym = coeff*sympy.exp(-t*u**2)
    # numerical version of the integrand
    coulomb_repulsion_and_exchange_intgd_num = sympy.lambdify([u],
        coulomb_repulsion_and_exchange_intgd_sym, 'scipy')
    # integrate (first index of tuple contains result)
    coulomb_repulsion_and_exchange_int_val = scipy.integrate.nquad(
        coulomb_repulsion_and_exchange_intgd_num, [[0, 1]])

    return coulomb_repulsion_and_exchange_int_val[0]

#
# Generate one-electron integral combinations
#
def get_one_electron_combinations(num_basis_functions):

    combinations = list(itertools.combinations_with_replacement(list(range(
        num_basis_functions)),2))
    console_print(0, '␣␣One-Electron␣Integral␣Combinations␣(total=%d):' % len(combinations))
    for combination in combinations:
        console_print(0, '␣␣␣␣(%d,␣%d)' % (combination[0], combination[1]))

    return combinations

#
# Generate two-electron integral combinations
#
def get_two_electron_combinations(num_basis_functions):
    # generate combinations
    # G_{row/v,col/u}
    combinations = []
    for v in range(num_basis_functions):
        for mu in range(num_basis_functions):
            for lambda_ in range(num_basis_functions):
                for sigma in range(num_basis_functions):
                    # Coulomb attraction two-electron integral
                    coulomb = [v,mu,sigma,lambda_]
                    # Exchange two-electron integral
                    exchange = [v,lambda_,sigma,mu]
                    # Avoid calculating identical integrals
                    # (rs|tu) = (rs|ut) = (sr|tu) = (sr|ut) = (tu|rs) = (tu|sr) = (ut|rs) =
                        (ut|sr)
                    coulomb = tuple(sorted(coulomb[0:2]) + sorted(coulomb[2:4]))
                    exchange = tuple(sorted(exchange[0:2]) + sorted(exchange[2:4]))
                    coulomb_swapped = tuple(coulomb[2:4] + coulomb[0:2])
                    exchange_swapped = tuple(exchange[2:4] + coulomb[0:2])
                    if coulomb not in combinations and coulomb_swapped not in combinations:
                        combinations.append(coulomb)
                    if exchange not in combinations and exchange_swapped not in combinations
                        :
                        combinations.append(exchange)

    console_print(0, '␣␣Two-Electron␣Integral␣Combinations␣(total=%d):' % len(combinations))
    for combination in combinations:
        console_print(0, '␣␣␣␣(%d,␣%d,␣%d,␣%d)' % (combination[0], combination[1],
            combination[2], combination[3]))

    return combinations

#
# Distance between two 3D vectors
#
def calculate_distance(vector_0, vector_1):

    return scipy.sqrt((vector_1[0] - vector_0[0])**2 + (vector_1[1] - vector_0[1])**2 + (
        vector_1[2] - vector_0[2])**2)

#
# Integral calculator wrapper
#
def do_integrals(subject_name, basis_function_objs):

    integrals = {}

    # generate combinations for the one and two-electron integral calculations
    # for the one-electron integrals, we use naive functions and scipy to evaluate the
        integrals as they are because they're simple
    one_electron_combinations = get_one_electron_combinations(len(basis_function_objs))
    one_electron_function_combinations = [(basis_function_objs[combination[0]].eval,
        basis_function_objs[combination[1]].eval) for combination in
        one_electron_combinations]
    one_electron_function_combinations_obj = [(basis_function_objs[combination[0]],
        basis_function_objs[combination[1]]) for combination in one_electron_combinations]

    # for the two-electron integrals, we use optimized functions which only have a single
        integral to evaluate as they are too complex to evaluate naively
    two_electron_combinations = get_two_electron_combinations(len(basis_function_objs))
    two_electron_function_combinations = [(basis_function_objs[combination[0]],
        basis_function_objs[combination[1]], basis_function_objs[combination[2]],
        basis_function_objs[combination[3]]) for combination in two_electron_combinations]

    calculate_overlap_integrals_optimized(one_electron_function_combinations_obj[0])

    with multiprocessing.Pool(processes = multiprocessing.cpu_count()-2, initializer=
        initializer) as pool:

        console_print(0, '**␣Calculating␣%s␣overlap␣integrals...' % subject_name)
        results = list(tqdm.tqdm(pool.imap(calculate_overlap_integrals_optimized,
            one_electron_function_combinations_obj), total=len(
            one_electron_function_combinations_obj), ascii=True))
        integrals[OVERLAP] = dict(zip(one_electron_combinations, results))

        console_print(0, '**␣Calculating␣%s␣kinetic␣energy␣integrals...' % subject_name)
        results = list(tqdm.tqdm(pool.imap(calculate_kinetic_energy_integrals,
            one_electron_function_combinations), total=len(
            one_electron_function_combinations), ascii=True))
        integrals[KINETIC] = dict(zip(one_electron_combinations, results))

        console_print(0, '**␣Calculating␣%s␣nuclear␣attraction␣integrals...' % subject_name)
        func = functools.partial(calculate_nuclear_attraction_integrals_naive, subject_name.
            lower())
        results = list(tqdm.tqdm(pool.imap(func, one_electron_function_combinations), total=
            len(one_electron_function_combinations), ascii=True))
        integrals[ATTRACTION] = dict(zip(one_electron_combinations, results))

        console_print(0, '**␣Calculating␣%s␣repulsion␣and␣exchange␣integrals...' %
            subject_name)
        results = list(tqdm.tqdm(pool.imap(
            calculate_coulomb_repulsion_and_exchange_integrals_optimized,
            two_electron_function_combinations), total=len(
            two_electron_function_combinations), ascii=True))
        integrals[EXCHANGE] = dict(zip(two_electron_combinations, results))

    console_print(0, '**␣Finished␣calculating␣%s␣atom␣integrals!' % subject_name)

    return integrals

#
# This functions returns whether or not the two matrices are within a specified tolerance
#
def is_symmetric(A, B, tol=0.1):
    return scipy.sparse.linalg.norm(A-B, scipy.inf) < tol;

#
# Process the integral dictionary for jsonification by converting combination tuples to
    strings
#
def process_dict_for_json(integral_dict_from_program):

    integral_dict_json_copy = copy.deepcopy(integral_dict_from_program)
    for integral_type_key in list(integral_dict_json_copy.keys()):
        for combination_key in list(integral_dict_json_copy[integral_type_key]):
            integral_dict_json_copy[integral_type_key][str(combination_key)] =
                integral_dict_json_copy[integral_type_key].pop(combination_key)

    return integral_dict_json_copy

#
# Process the integral dictionary for the program by converting combination string to tuples
#
def process_dict_for_program(integral_dict_from_json):

    for integral_type_key in list(integral_dict_from_json.keys()):
        for combination_key in list(integral_dict_from_json[integral_type_key]):
            combination_key_tuple = tuple([int(str_number) for str_number in combination_key
                .strip('(').strip(')').split(',')])
            integral_dict_from_json[integral_type_key][combination_key_tuple] =
                integral_dict_from_json[integral_type_key].pop(combination_key)

    return integral_dict_from_json

#
# Pre-calculate integrals
#
def precalculate_integrals():

    do_he_integrals = False
    do_h2_integrals = False

    try:
        with open(HE_INTEGRALS_FILENAME) as he_integrals_json_file:
            he_integrals = process_dict_for_program(json.load(he_integrals_json_file))
            console_print(0, '**␣HE␣integrals␣loaded␣from␣%s' % (HE_INTEGRALS_FILENAME))
    except:
        console_print(0, '**␣Unable␣to␣load␣He␣integrals␣file,␣will␣recalculate')
        do_he_integrals = True

    try:
        with open(H2_INTEGRALS_FILENAME) as h2_integrals_json_file:
            h2_integrals = process_dict_for_program(json.load(h2_integrals_json_file))
            console_print(0, '**␣H2␣integrals␣loaded␣from␣%s' % (H2_INTEGRALS_FILENAME))
    except:
        console_print(0, '**␣Unable␣to␣load␣H2␣integrals␣file,␣will␣recalculate')
        do_h2_integrals = True

    # instantiate basis function classes for integral calculations
    sto_1g_helium_0_obj = sto_1g_helium_class((0,0,0))
    sto_1g_helium_1_obj = sto_1g_helium_class((0,0,0))
    sto_1g_hydrogen_00_obj = sto_1g_hydrogen_class(((-H2_BOND_LENGTH_ATOMIC_UNITS/2.0),0,0))
    sto_1g_hydrogen_01_obj = sto_1g_hydrogen_class(((-H2_BOND_LENGTH_ATOMIC_UNITS/2.0),0,0))
    sto_1g_hydrogen_10_obj = sto_1g_hydrogen_class(((H2_BOND_LENGTH_ATOMIC_UNITS/2.0),0,0))
    sto_1g_hydrogen_11_obj = sto_1g_hydrogen_class(((H2_BOND_LENGTH_ATOMIC_UNITS/2.0),0,0))

    # ******************
    # *** HELIUM ATOM ***
    # ******************

    if do_he_integrals:

        console_print(0, '**␣Starting␣He␣atom␣integral␣calculations')

        # He uses the same basis function twice, so the integrals end up being
        # identical four all four entries 11 = 12 = 21 = 22

        # basis function lookup table
        he_basis_func_lut = (sto_1g_helium_0_obj, sto_1g_helium_1_obj)

        he_integrals = do_integrals('He', he_basis_func_lut)

        # write out integrals
        console_print(0, '**␣Saving␣Helium␣atom␣integrals␣to␣%s...' % (HE_INTEGRALS_FILENAME
            ))
        with open(HE_INTEGRALS_FILENAME, 'w') as he_integrals_json_file:
            json.dump(process_dict_for_json(he_integrals), he_integrals_json_file, indent=4)

    # ************************
    # *** HYDROGEN MOLECULE ***
    # ************************

    if do_h2_integrals:

        console_print(0, '**␣Starting␣H2␣molecule␣integral␣calculations')

        # H2 molecule has two basis functions centered on each nuclei, for a
        # total of four basis function (only two are unique). They will need to
        # be combined to form the matrices, so we'll have to find unique
        # combinations

        # basis function lookup table
        h2_basis_func_lut = (sto_1g_hydrogen_00_obj, sto_1g_hydrogen_01_obj,
            sto_1g_hydrogen_10_obj, sto_1g_hydrogen_11_obj)

        h2_integrals = do_integrals('H2', h2_basis_func_lut)

        # write out integrals
        console_print(0, '**␣Saving␣H2␣molecule␣integrals␣to␣%s...' % (H2_INTEGRALS_FILENAME
            ))
        with open(H2_INTEGRALS_FILENAME, 'w') as h2_integrals_json_file:
            json.dump(process_dict_for_json(h2_integrals), h2_integrals_json_file,  indent
                =4)

        console_print(0, '**␣Finished␣calculating␣H2␣molecule␣integrals!')

    return (he_integrals, h2_integrals)

if __name__ == '__main__':
    # the following sets up the argument parser for the program
    parser = argparse.ArgumentParser(description='Exact␣Hartree-Fock␣simulator')

    args = parser.parse_args()

    main(args)
```

5.2 Helium Atom Integrals

The following is the JSON file content for the Helium atom integrals.

```
{
    "overlap": {
        "(0,␣0)": 1.0001264299519068,
        "(0,␣1)": 1.0001264299519068,
        "(1,␣1)": 1.0001264299519068
    },
    "kinetic": {
        "(0,␣0)": 1.160996766209679,
        "(0,␣1)": 1.160996766209679,
        "(1,␣1)": 1.160996766209679
    },
    "attraction": {
        "(0,␣0)": -2.8080017912774053,
        "(0,␣1)": -2.8080017912774053,
        "(1,␣1)": -2.8080017912774053
    },
    "exchange": {
        "(0,␣0,␣0,␣0)": 0.9929040710429119,
        "(0,␣0,␣0,␣1)": 0.9929040710429119,
        "(0,␣0,␣1,␣1)": 0.9929040710429119,
        "(0,␣1,␣0,␣1)": 0.9929040710429119,
        "(0,␣1,␣1,␣1)": 0.9929040710429119,
        "(1,␣1,␣1,␣1)": 0.9929040710429119
    }
}
```

5.3 Hydrogen Molecule Integrals

The following is the JSON file content for the Hydrogen atom integrals.

```
{
    "overlap": {
        "(0,␣0)": 1.000149315884204,
        "(0,␣1)": 1.000149315884204,
        "(0,␣2)": 0.6655213967087238,
        "(0,␣3)": 0.6655213967087238,
        "(1,␣1)": 1.000149315884204,
        "(1,␣2)": 0.6655213967087238,
        "(1,␣3)": 0.6655213967087238,
        "(2,␣2)": 1.000149315884204,
        "(2,␣3)": 1.000149315884204,
        "(3,␣3)": 1.000149315884204
    },
    "kinetic": {
        "(0,␣0)": 0.6249933075865092,
        "(0,␣1)": 0.6249933075865092,
        "(0,␣2)": 0.30294849506084853,
        "(0,␣3)": 0.30294849506084853,
        "(1,␣1)": 0.6249933075865092,
        "(1,␣2)": 0.30294849506084853,
        "(1,␣3)": 0.30294849506084853,
        "(2,␣2)": 0.6249933075865092,
        "(2,␣3)": 0.6249933075865092,
        "(3,␣3)": 0.6249933075865092
    },
    "attraction": {
        "(0,␣0)": -9.943124080849893,
        "(0,␣1)": -9.943124080849893,
        "(0,␣2)": -11.148657174705974,
        "(0,␣3)": -11.148657174705974,
        "(1,␣1)": -9.943124080849893,
        "(1,␣2)": -11.148657174705974,
        "(1,␣3)": -11.148657174705974,
        "(2,␣2)": -9.943124080849893,
        "(2,␣3)": -9.943124080849893,
        "(3,␣3)": -9.943124080849893
    },
    "exchange": {
        "(0,␣0,␣0,␣0)": 0.728524860763678,
        "(0,␣0,␣0,␣1)": 0.728524860763678,
        "(0,␣0,␣0,␣2)": 0.441550299931593,
        "(0,␣0,␣0,␣3)": 0.441550299931593,
        "(0,␣0,␣1,␣1)": 0.728524860763678,
        "(0,␣1,␣0,␣1)": 0.728524860763678,
        "(0,␣0,␣1,␣2)": 0.441550299931593,
        "(0,␣1,␣0,␣2)": 0.441550299931593,
        "(0,␣0,␣1,␣3)": 0.441550299931593,
        "(0,␣1,␣0,␣3)": 0.441550299931593,
        "(0,␣2,␣0,␣1)": 0.441550299931593,
        "(0,␣0,␣2,␣2)": 0.608702186405423,
        "(0,␣2,␣0,␣2)": 0.322580965605786,
        "(0,␣0,␣2,␣3)": 0.608702186405423,
        "(0,␣2,␣0,␣3)": 0.322580965605786,
        "(0,␣3,␣0,␣1)": 0.441550299931593,
        "(0,␣3,␣0,␣2)": 0.322580965605786,
        "(0,␣0,␣3,␣3)": 0.608702186405423,
        "(0,␣3,␣0,␣3)": 0.322580965605786,
        "(0,␣1,␣1,␣1)": 0.728524860763678,
        "(0,␣1,␣1,␣2)": 0.441550299931593,
        "(0,␣1,␣1,␣3)": 0.441550299931593,
        "(0,␣2,␣1,␣1)": 0.441550299931593,
        "(0,␣1,␣2,␣2)": 0.608702186405423,
        "(0,␣2,␣1,␣2)": 0.322580965605786,
        "(0,␣1,␣2,␣3)": 0.608702186405423,
        "(0,␣2,␣1,␣3)": 0.322580965605786,
        "(0,␣3,␣1,␣1)": 0.441550299931593,
        "(0,␣3,␣1,␣2)": 0.322580965605786,
        "(0,␣1,␣3,␣3)": 0.608702186405423,
        "(0,␣3,␣1,␣3)": 0.322580965605786,
        "(0,␣2,␣2,␣2)": 0.441550299931593,
        "(0,␣2,␣2,␣3)": 0.441550299931593,
        "(0,␣3,␣2,␣2)": 0.441550299931593,
        "(0,␣2,␣3,␣3)": 0.441550299931593,
        "(0,␣3,␣2,␣3)": 0.441550299931593,
        "(0,␣3,␣3,␣3)": 0.441550299931593,
        "(1,␣1,␣1,␣1)": 0.728524860763678,
        "(1,␣1,␣1,␣2)": 0.441550299931593,
        "(1,␣1,␣1,␣3)": 0.441550299931593,
        "(1,␣1,␣2,␣2)": 0.608702186405423,
        "(1,␣2,␣1,␣2)": 0.322580965605786,
        "(1,␣1,␣2,␣3)": 0.608702186405423,
        "(1,␣2,␣1,␣3)": 0.322580965605786,
        "(1,␣3,␣1,␣2)": 0.322580965605786,
        "(1,␣1,␣3,␣3)": 0.608702186405423,
        "(1,␣3,␣1,␣3)": 0.322580965605786,
        "(1,␣2,␣2,␣2)": 0.441550299931593,
        "(1,␣2,␣2,␣3)": 0.441550299931593,
        "(1,␣3,␣2,␣2)": 0.441550299931593,
        "(1,␣2,␣3,␣3)": 0.441550299931593,
        "(1,␣3,␣2,␣3)": 0.441550299931593,
        "(1,␣3,␣3,␣3)": 0.441550299931593,
        "(2,␣2,␣2,␣2)": 0.728524860763678,
        "(2,␣2,␣2,␣3)": 0.728524860763678,
        "(2,␣2,␣3,␣3)": 0.728524860763678,
        "(2,␣3,␣2,␣3)": 0.728524860763678,
        "(2,␣3,␣3,␣3)": 0.728524860763678,
        "(3,␣3,␣3,␣3)": 0.728524860763678
    }
}
```

References

[1] (2020) Chemistry LibreText: Gaussian Basis Sets. [Online]. Available: https://chem.libretexts.org/Courses/Pacific_Union_College/Quantum_Chemistry/11%3A_Computational_Quantum_Chemistry/11.02%3A_Gaussian_Basis_Sets

[2] E. Lewars, S. ebooks Chemistry, and M. Science, *Computational chemistry: introduction to the theory and applications of molecular and quantum mechanics*, 2nd ed. London;New York;Dordrecht [Netherlands];: Springer, 2011;2016;2010;.

[3] (2020) T >T: Hartree Fock Theory in 100 Lines. [Online]. Available: https://adambaskerville.github.io/posts/HartreeFockGuide/

[4] P. M. W. Gill, *Molecular integrals Over Gaussian Basis Functions*, ser. Advances in Quantum Chemistry. Elsevier Science & Technology, 1994, vol. 25, pp. 141–205.

[5] A. Szabo and N. S. Ostlund, *Modern quantum chemistry: introduction to advanced electronic structure theory*, 1st ed. New York: McGraw-Hill, 1989.