Assignment 2

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

Due: 2022/02/11

1 Directions

The goal in this assignment is to calculate the electronic structure of the helium atom and the hydrogen molecule (two separate systems). The distance between the two hydrogen atoms (the bond length) is 0.74 Å. In this assignment, do not perform geometry optimization, and instead use that fixed bond length in order to calculate the molecular orbitals (both energies and wave functions). Also, note that the problem is in full 3-dimensional space. The goal is to solve the Hartree-Fock equation for these two systems.

Note that you are solving an iterative problem. The Fock operator depends on the orbitals, which you do not have. So, start by making a guess for the orbital shapes, form the Fock operator, solve for the orbitals, and keep iterating until convergence. Use the convergence of the total energy of the system as your convergence criterion.

In this assignment, the calculations must be done directly on a real-space grid, over which the orbitals (wave functions) are defined. So, do not use the Roothaan equations (which we will see soon).

(a) (7 points) Directly discretize the Fock operator to put it into matrix form. Describe your approach and show the steps of your derivation all the way to obtaining the matrix equation.

(b) (5 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 (not including the nucleus-nucleus interaction). 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.

2 Derivation of Solution

2.1 Hartree-Fock Matrix Equation Derivation

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:

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

The Fock operator is composed of the core Hamiltonian operator $\hat{H}(\vec{r})$, the Coulomb operator $\hat{J}(\vec{r})$, and the exchange operator $\hat{K}(\vec{r})$. P_2 is a transposition operator which switches the indices of the electron indices, something required by the exchange operator.

$$\hat{H}_{core}(\vec{r}) = -\frac{1}{2}\nabla_1^2 - \sum_A \frac{Z_A}{\|\vec{r}_{1A}\|}$$
$$\hat{J}_j(\vec{r}_1)\psi_i(\vec{r}_1) = \psi_i(\vec{r}_1)\int_{-\infty}^{\infty}|\psi_i(\vec{r}_2)|^2\frac{1}{\|\vec{r}_{12}\|}d\vec{r}_2$$
$$\hat{K}_j(\vec{r}_1)\psi_i(\vec{r}_1) = \psi_j(\vec{r}_1)\int_{-\infty}^{\infty}\frac{\psi_j^*(\vec{r}_2)\psi_i(\vec{r}_2)}{\|\vec{r}_{12}\|}d\vec{r}_2$$

Expanded, the Fock operator takes the following form:

$$\hat{F}(\vec{r}) = -\frac{1}{2}\nabla_1^2 - \sum_A \frac{Z_A}{\|\vec{r}_{1A}\|} + \sum_{n=1}^{N/2}\left[2\int_{-\infty}^{\infty}|\psi_n(\vec{r}_2)|^2\frac{1}{\|\vec{r}_{12}\|}d\vec{r}_2 - \int_{-\infty}^{\infty}\frac{\psi_n^*(\vec{r}_2)P_2\psi_n(\vec{r}_2)}{\|\vec{r}_{12}\|}d\vec{r}_2\right]$$

For the Helium atom, the Fock operator takes the following form:

$$\hat{F}(\vec{r}) = -\frac{1}{2}\nabla_1^2 - \frac{2}{\|\vec{r}_{1A}\|} + \int_{-\infty}^{\infty}|\psi_1(\vec{r}_2)|^2\frac{1}{\|\vec{r}_{12}\|}d\vec{r}_2$$

and for the Hydrogen molecule, the Fock operator takes the following form:

$$\hat{F}(\vec{r}) = -\frac{1}{2}\nabla_1^2 - \frac{1}{\|\vec{r}_{1A}\|} - \frac{1}{\|\vec{r}_{1B}\|} + \int_{-\infty}^{\infty}|\psi_1(\vec{r}_2)|^2\frac{1}{\|\vec{r}_{12}\|}d\vec{r}_2$$

For the ground state, we only consider one orbital to fill, which simplifies the exchange operator's index-switch operator in both Fock operators. This also has the effect of making the Coulomb and exchange integrals identical, greatly simplifying the equation. Note that: $\psi_n^*(\vec{r}_2)\psi_n(\vec{r}_2) = |\psi_1(\vec{r}_2)|^2$. For the Helium atom, our value for Z_A is 2 as there are two protons in the nucleus and for the Hydrogen molecule, we

consider both Hydrogen nuclei containing one proton each, so $Z_A = Z_B = 1$. The nuclei are separated by 0.74 Å. Since we are solving this problem using Hartree atomic units, we must convert this value accordingly when considering the coordinates of the hydrogen nuclei. In Hartree atomic units, the atomic unit of length converts the Bohr radius a_0 to 1. Therefore, 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.

In order to discretize these equations, the second order differentiation and the finite integration must be numerically performed, in the three spatial dimensions x, y, and z. For the second order differentiation, it will discretize in the following manner:

$$\begin{aligned}\nabla^2 f(x,y,z) \approx & \frac{f(x+\Delta x,y,z) - 2f(x,y,z) + f(x-\Delta x,y,z)}{\Delta x^2} \\ & + \frac{f(x,y+\Delta y,z) - 2f(x,y,z) + f(x,y-\Delta y,z)}{\Delta y^2} \\ & + \frac{f(x,y,z+\Delta z) - 2f(x,y,z) + f(x,y,z-\Delta z)}{\Delta z^2}\end{aligned}$$

If we choose the same discretization size, such that $\Delta x = \Delta y = \Delta z = h$, the discretization takes the following simplification:

$$\nabla^2 f(x,y,z) \approx \frac{f(x+h,y,z) + f(x,y+h,z) + f(x,y,z+h) - 6f(x,y,z) + f(x-h,y,z) + f(x,y-h,z) + f(x,y,z-h)}{h^2}$$

As seen in the previous assignment, by forming a column vector in the following form:

$$|\psi\rangle = [\psi(0,0,0),\ldots,\psi(N,0,0),\psi(0,1,0),\ldots,\psi(N,1,0),\ldots,\psi(0,N,0),\ldots,\psi(N,N,N)]^T$$

The indices found in this vector will depend on the limits chosen for the solution in all directions, as well as the number of partitions N. With three dimensions and N discretization points, the total number of equations to solve will be N^3. The sparse matrix $\boldsymbol{S}_{\nabla^2}$ for the Laplacian differential operator on three variables for $N = 3$ and $h = 1$ is shown below. Three rows have been highlighted to demonstrate how they correspond to an equation in the total solution.

```
    N = 3
    S =
    +-------+-------+-------+-------+-------+-------+-------+-------+-------+
    |-6 1   | 1     |       | 1     |       |       |       |       |       | f(0,0,0)
    | 1-6 1 |   1   |       |   1   |       |       |       |       |       | f(1,0,0)
    |   1-6 |     1 |       |     1 |       |       |       |       |       | f(2,0,0)
    +-------+-------+-------+-------+-------+-------+-------+-------+-------+
    | 1     |-6 1   | 1     |       | 1     |       |       |       |       | f(0,1,0)
    |   1   | 1-6 1 |   1   |       |   1   |       |       |       |       | f(1,1,0)
    |     1 |   1-6 |     1 |       |     1 |       |       |       |       | f(2,1,0)
    +-------+-------+-------+-------+-------+-------+-------+-------+-------+
    |       | 1     |-6 1   |       |       | 1     |       |       |       | f(0,2,0)
    |       |   1   | 1-6 1 |       |       |   1   |       |       |       | f(1,2,0)
    |       |     1 |   1-6 |       |       |     1 |       |       |       | f(2,2,0)
    +-------+-------+-------+-------+-------+-------+-------+-------+-------+
    | 1     |       |       |-6 1   | 1     |       | 1     |       |       | f(0,0,1)
    |   1   |       |       | 1-6 1 |   1   |       |   1   |       |       | f(1,0,1)
    |     1 |       |       |   1-6 |     1 |       |     1 |       |       | f(2,0,1)
    +-------+-------+-------+-------+-------+-------+-------+-------+-------+
    |       | 1     |       | 1     |-6 1   | 1     |       | 1     |       | f(0,1,1) (+)
    |       |   1   |       |   1   | 1-6 1 |   1   |       |   1   |       | f(1,1,1) (*)
    |       |     1 |       |     1 |   1-6 |     1 |       |     1 |       | f(2,1,1) (&)
    +-------+-------+-------+-------+-------+-------+-------+-------+-------+
    |       |       | 1     |       | 1     |-6 1   |       |       | 1     | f(0,2,1)
    |       |       |   1   |       |   1   | 1-6 1 |       |       |   1   | f(1,2,1)
    |       |       |     1 |       |     1 |   1-6 |       |       |     1 | f(2,2,1)
    +-------+-------+-------+-------+-------+-------+-------+-------+-------+
    |       |       |       | 1     |       |       |-6 1   | 1     |       | f(0,0,2)
    |       |       |       |   1   |       |       | 1-6 1 |   1   |       | f(1,0,2)
    |       |       |       |     1 |       |       |   1-6 |     1 |       | f(2,0,2)
    +-------+-------+-------+-------+-------+-------+-------+-------+-------+
    |       |       |       |       | 1     |       | 1     |-6 1   | 1     | f(0,1,2)
    |       |       |       |       |   1   |       |   1   | 1-6 1 |   1   | f(1,1,2)
    |       |       |       |       |     1 |       |     1 |   1-6 |     1 | f(2,1,2)
    +-------+-------+-------+-------+-------+-------+-------+-------+-------+
    |       |       |       |       | 1     |       | 1     |-6 1   |       | f(0,2,2)
    |       |       |       |       |   1   |       |   1   | 1-6 1 |       | f(1,2,2)
    |       |       |       |       |     1 |       |     1 |   1-6 |       | f(2,2,2)
    +-------+-------+-------+-------+-------+-------+-------+-------+-------+

(+) f(1,1,1) + f(0,2,1) + f(0,1,2) -6f(0,1,1) +           + f(0,0,1) + f(0,1,0)
(*) f(2,1,1) + f(1,2,1) + f(1,1,2) -6f(1,1,1) + f(0,1,1) + f(1,0,1) + f(1,1,0)
(&)          + f(2,2,1) + f(2,1,2) -6f(2,1,1) + f(1,1,1) + f(2,0,1) + f(2,1,0)
```

The next operator to be considered for discretization is the integration that is performed on the Coulomb and exchange terms. For this integration, we can extend the one dimensional numerical integration intro three dimensions. First we consider the expanded form of the spatial integral:

$$\int_{-\infty}^{\infty} f(\vec{r})d\vec{r} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y,z)\,dx\,dy\,dz$$

We then split the three spatial dimensions into N partitions in all cardinal directions to quantize the problem into discretized volumes of the solution, Δx, Δy, and Δz as we did in the differential discretization. Once again, if we choose the same discretization size, such that $\Delta x = \Delta y = \Delta z = h$, we evenly divide the solution space into partitions. By evaluating the function at each coordinate, multiplying the result by our discretization size h, and summing over the solution space, we obtain a Riemann sum, which is a trivial implementation of numerical integration. The sums approximate the integral as follows:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y,z)\,dx\,dy\,dz \approx \sum_{k_z=0}^{N}\sum_{k_y=0}^{N}\sum_{k_x=0}^{N}\left[w^{[k_x,k_y,k_z]}f(k_x,k_y,k_z)\right]$$
$$w_{k_x} = w_{k_y} = w_{k_z} = \{0,h,h,...,h\}$$

One limitation of this approximated integration is that the infinite limit definite integral required by the original equation is now confined within the limits we choose for the problem and the N partitions we make out of the space in all directions. Choosing the limits and N must be done carefully as the matrices generated for the solution will easily balloon into very difficult diagonalization problems.

Both the Helium atom and the Hydrogen molecule share the following integral term:

$$\int_{-\infty}^{\infty} |\psi_1(\vec{r}_2)|^2 \frac{1}{\|\vec{r}_{12}\|} d\vec{r}_2$$

For every iteration of the Hartree-Fock algorithm, a solution set will be generated for $\psi_1(\vec{r}_1)$ when the Fock matrix $\mathbf{F}$ is diagonalized. This solution is then re-used in the upcoming iteration as the solution for $\psi_1(\vec{r}_2)$ for use in the integrals. For the first iteration, a version of $\psi_1(\vec{r}_2)$ will be use where every entry is 0. If that fails to converge, then a noisy solution will be attempted. Another thing to take into account is the fact that there will be multiple solutions in the eigenvectors. For every energy to be plotted, that energy's resulting wavefunction will need to be used in each iteration step. Using the sums above, $\psi_1(x_2,y_2,z_2)$ will be evaluated and a value will be obtained. Depending on which energy is being calculated, that energy's eigenvector will be chosen to calculate the integral. This integral also possesses a distance calculation between the current and other electron. This calculation relies on the position of the current step as seen below:

$$\frac{1}{\|\vec{r}_{12}\|} = \frac{1}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}}$$

This integral will be evaluated for every partition in the problem; therefore, the values for x_1, y_1, and z_1 will correspond to those found in the column vector $|\psi\rangle$ as defined above. This integration operator can be summarized as follows:

$$I(x_1,y_1,z_1) = \int_{-\infty}^{\infty} |\psi_1(\vec{r}_2)|^2 \frac{1}{\|\vec{r}_{12}\|} d\vec{r}_2$$

This equation will form a diagonal in a matrix J of size $N \times N \times N$ as follows:

$$\boldsymbol{J} = \begin{bmatrix} I(x_1,y_1,z_1) & 0 & \dots & 0 \\ 0 & I(x_1,y_1,z_1) & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & I(x_1,y_1,z_1) \end{bmatrix}$$

The final term of the Fock equation that needs to be considered is the nuclear attraction operator. For the Helium atom case, we have a single term, and for the Hydrogen molecule we have two terms. For the Helium atom, the expanded nuclear attraction term is as follows:

$$A_{He}(x_1, y_1, z_1) = -\sum_A \frac{Z_A}{\|\vec{r}_{1A}\|} = -\frac{2}{\sqrt{(x_1^2 + y_1^2 + z_1^2)}}$$

With the Helium nucleus placed at the origin of the problem, the distance between the electron and the nucleus becomes the distance of the electron from the origin. Also, the atomic number for Helium is 2, which puts that number in the denominator.

The Hydrogen molecule on the other hand takes a slightly different form. A hydrogen molecule has two nuclei in its system and they are separated by 1.39839733222307 atomic units of length as calculated above. For this problem, the Hydrogen nuclei will be placed $\frac{1.39839733222307}{2}$ in the positive and negative x directions from the origin. With an atomic number of 1 for the Hydrogen element, the nuclear attraction terms will take the following form:

$$A_{H_2}(x_1, y_1, z_1) = -\sum_A \frac{Z_A}{\|\vec{r}_{1A}\|} = -\frac{1}{\sqrt{(\frac{-1.39839733222307}{2} - x_1)^2 + y_1^2 + z_1^2}} - \frac{1}{\sqrt{(\frac{1.39839733222307}{2} - x_1)^2 + y_1^2 + z_1^2}}$$

The nuclear attraction operator will form a diagonal in a matrix A of size $N \times N \times N$ as follows:

$$\boldsymbol{A} = \begin{bmatrix} A(x_1, y_1, z_1) & 0 & \dots & 0 \\ 0 & A(x_1, y_1, z_1) & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & A(x_1, y_1, z_1) \end{bmatrix}$$

Now we have all of the components necessary to implement a solution. To summarize, for the Helium atom, the following matrix system will be solved:

$$\boldsymbol{F}_{He} |\psi\rangle = \boldsymbol{\epsilon} |\psi\rangle$$
$$\boldsymbol{F}_{He} = -\frac{1}{2h^2}\boldsymbol{S}_{\nabla^2} + \boldsymbol{A}_{He} + \boldsymbol{J}$$

and for the Hydrogen molecule, the following matrix system will be solved:

$$\boldsymbol{F}_{H_2} |\psi\rangle = \boldsymbol{\epsilon} |\psi\rangle$$
$$\boldsymbol{F}_{H_2} = -\frac{1}{2h^2}\boldsymbol{S}_{\nabla^2} + \boldsymbol{A}_{H_2} + \boldsymbol{J}$$

2.2 Hartree-Fock Energy Calculation Derivation

The orbital energies do not sum the Hartree-Fock energy. Koopman's theorem states that in restricted Hartree-Fock, the energies associated with occupied orbitals approximate the ionization energy of that orbital, and the energies associated with unoccupied orbitals approximate the energy required to place an electron into that orbital. The total energy of the system must be calculated using the following equation which accounts for the double contribution of electrons in the double electron integrals (hence the $\frac{1}{2}$):

$$E_o = \sum_i^N \langle i| \hat{h} |i\rangle + \frac{1}{2}\sum_{i>j}^N [ij\|ij]$$

This implies that for the total energy, we can take the expectation value of the resulting Fock operators for both the Helium atom and the Hydrogen molecule, while making sure to halve the contribution of the Coulomb and exchange terms and also using the calculated ground state orbital which both electrons occupy. Also, since both electrons occupy the same spin orbital, terms will end up combining (e.g. $\langle 1| \hat{h} |1\rangle + \langle 1| \hat{h} |1\rangle = 2\langle 1| \hat{h} |1\rangle$)

$$\begin{aligned} E_{o(He)} = E_{o(H_2)} &= \sum_i^2 \langle i| \hat{h} |i\rangle + \frac{1}{2}\sum_{i>j}^N [ij|ij] \\ &= \langle 1| \hat{h} |1\rangle + \langle 2| \hat{h} |2\rangle + \frac{1}{2}([11|22] + [12|12]) \\ &= 2\langle 1| \hat{h} |1\rangle + \frac{1}{2}[11|22] + \frac{1}{2}[12|12] \end{aligned}$$

The total energy for the Helium molecule and Hydrogen atom in their ground states should then be equal to:

$$\begin{aligned} E_{o(He)} = 2\int_{-\infty}^{\infty} \psi_1^*(\vec{r}_1)\left[-\frac{1}{2}\nabla_1^2 - \frac{2}{\|\vec{r}_{1A}\|}\right]\psi_1(\vec{r}_1)d\vec{r}_1 + \frac{1}{2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \psi_1^*(\vec{r}_1)\psi_1(\vec{r}_1)\frac{1}{\|\vec{r}_{12}\|}\psi_2^*(\vec{r}_2)\psi_2(\vec{r}_2)d\vec{r}_2 d\vec{r}_1 \\ + \frac{1}{2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \psi_1^*(\vec{r}_1)\psi_2(\vec{r}_1)\frac{1}{\|\vec{r}_{12}\|}\psi_2^*(\vec{r}_2)\psi_1(\vec{r}_2)d\vec{r}_2 d\vec{r}_1 \end{aligned}$$

$$\begin{aligned} E_{o(H_2)} = 2\int_{-\infty}^{\infty} \psi_1^*(\vec{r}_1)\left[-\frac{1}{2}\nabla_1^2 - \frac{1}{\|\vec{r}_{1A}\|} - \frac{1}{\|\vec{r}_{1B}\|}\right]\psi_1(\vec{r}_1)d\vec{r}_1 + \frac{1}{2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \psi_1^*(\vec{r}_1)\psi_1(\vec{r}_1)\frac{1}{\|\vec{r}_{12}\|}\psi_2^*(\vec{r}_2)\psi_2(\vec{r}_2)d\vec{r}_2 d\vec{r}_1 \\ + \frac{1}{2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \psi_1^*(\vec{r}_1)\psi_2(\vec{r}_1)\frac{1}{\|\vec{r}_{12}\|}\psi_2^*(\vec{r}_2)\psi_1(\vec{r}_2)d\vec{r}_2 d\vec{r}_1 \end{aligned}$$

Since we only calculate a single, real-valued, orbital using our program, it can then be assumed that $\psi_1^*(\vec{r}) = \psi_1(\vec{r}) = \psi_2^*(\vec{r}) = \psi_2(\vec{r})$. This simplifies the total energy equations as follows:

$$E_{o(He)} = 2\int_{-\infty}^{\infty} |\psi_1(\vec{r}_1)|^2\left[-\frac{1}{2}\nabla_1^2 - \frac{2}{\|\vec{r}_{1A}\|}\right]d\vec{r}_1 + \int_{-\infty}^{\infty}|\psi_1(\vec{r}_1)|^2\int_{-\infty}^{\infty}\frac{1}{\|\vec{r}_{12}\|}|\psi_1(\vec{r}_2)|^2 d\vec{r}_2 d\vec{r}_1$$

$$E_{o(H_2)} = 2\int_{-\infty}^{\infty} |\psi_1(\vec{r}_1)|^2 \left[-\frac{1}{2}\nabla_1^2 - \frac{1}{\|\vec{r}_{1A}\|} - \frac{1}{\|\vec{r}_{1B}\|}\right] d\vec{r}_1 + \int_{-\infty}^{\infty} |\psi_1(\vec{r}_1)|^2 \int_{-\infty}^{\infty} \frac{1}{\|\vec{r}_{12}\|} |\psi_1(\vec{r}_2)|^2 \, d\vec{r}_2 d\vec{r}_1$$

The $\frac{1}{2}$ Coulomb and exchange terms become identical, which merges them into a single term.

3 Program Implementation

The program for this assignment was written in Python and the source can be viewed in Section 6.1. It relies heavily on the SciPy package to perform the matrix algebra required to obtain the results. The program constructs the sparse matrices outlined in the previous section, obtains solutions based on convergence criteria, and outputs the plots and data for further analysis.

The program features command line arguments for the user to set various runtime variables and features the ability to load and save the data in a specially formatted file. These files will also contain the configuration values for the simulation run as well as metrics to gauge how quickly each solution was able to converge to a final solution. The user can specify the number of partitions N to use for the problem, resulting in an $N \times N \times N$ sized sparse matrix to calculate eigenvectors and eigenvalues from. Increasing the partition size has a severe impact on performance, so the user must be careful to pick a reasonable size. The largest impact to performance was found to be the calculation of the various integrals and filling the respective matrices with the integration results.

In order to optimize the calculations, sparse matrices provided by the SciPy package are used to reduce the memory footprint as well as enable more effective linear algebra operations. All of the sparse matrices are combined together into the Hermitian representing the Fock matrix, and the SciPy `scipy.sparse.linalg.eigsh` function is used to calculate the associated eigenvalues and eigenvectors. The solver was given a looser tolerance as well as an increased maximum limit of iterations to help converge on solutions as using the default settings would prevent the solver from converging at times.

Only the lowest six eigenvalues and eigenvectors are calculated for from the Fock matrix. The target energy level (the ground state which should be the lowest value) is used as the convergence criteria. If the percent change between two iterations for the selected orbital is less than what was specified, the simulation finishes.

The simulation will split up the task of filling up certain matrices among the available CPUs on the host computer. After profiling the simulator program, it was discovered that the majority of the downtime was spent calculating integration matrices and filling them up. By distributing the workload among all of the available CPU resourses, a significant performance gain was observed. The two areas where multiprocessing is implemented is in the integration for the total system energy as well as filling up the integration matrix for the iterative solution.

The matplotlib Python package is used to generate 4D plots of the resulting orbitals. A 3D scatter plot is used with a heatmap and a mask. The mask allows the plot to selectively ignore points which fall out of a certain range, which allows the plots to effectively illustrate the calculated orbitals. The orbitals are plotted from their lowest energy to the highest energy. Since the resulting waveform will have both negative and positive values, the eigenvector is squared so that the resulting waveform only has positive values. This corresponds to the probability density of the waveform.

A Windows batch file was created to more easily construct simulation runs for execution and data retrieval. The batch file executes concurrent scripts which run in the background.

4 Results

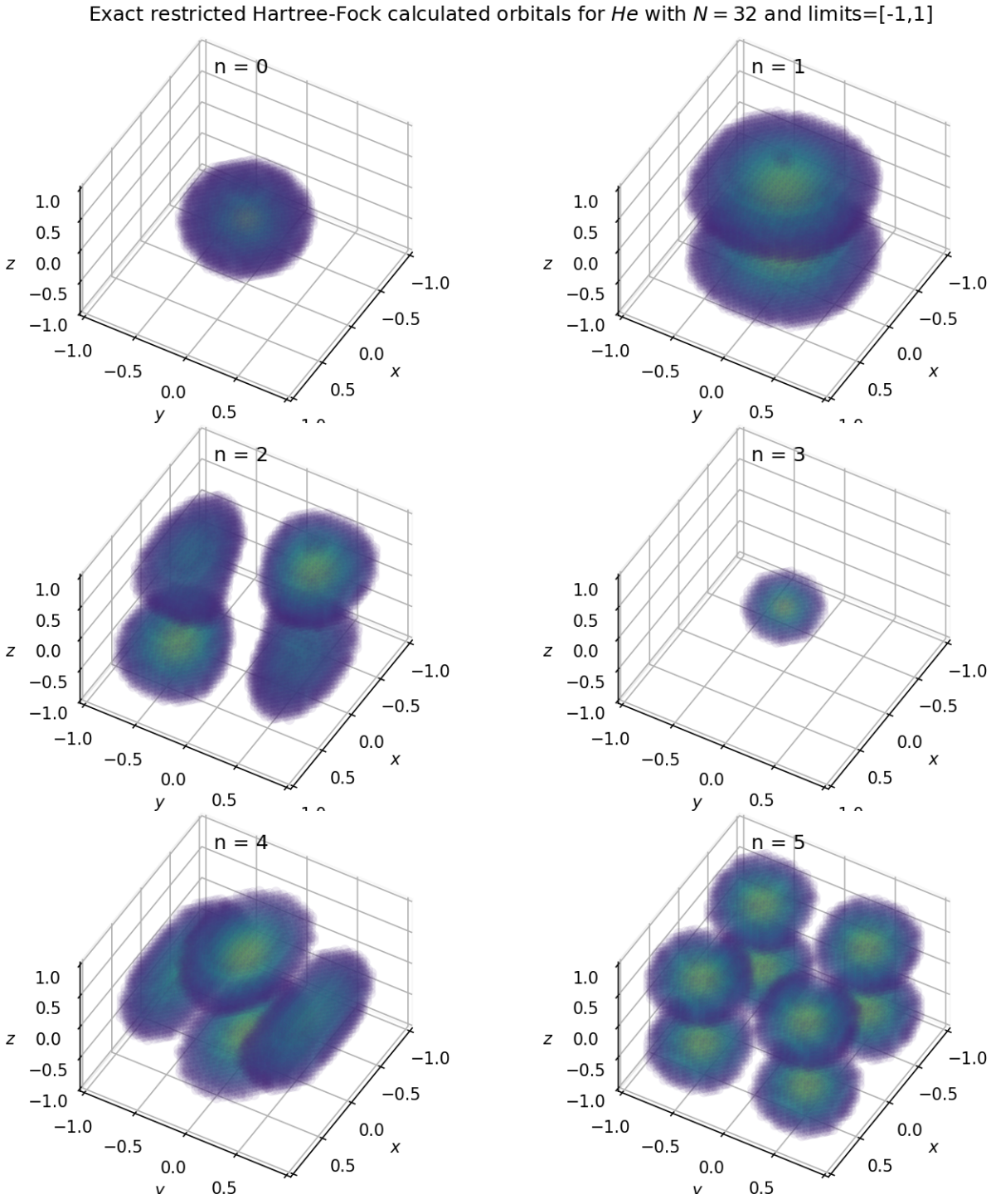

Figure 1: Calculated orbitals for He atom using limits of [-1,1]

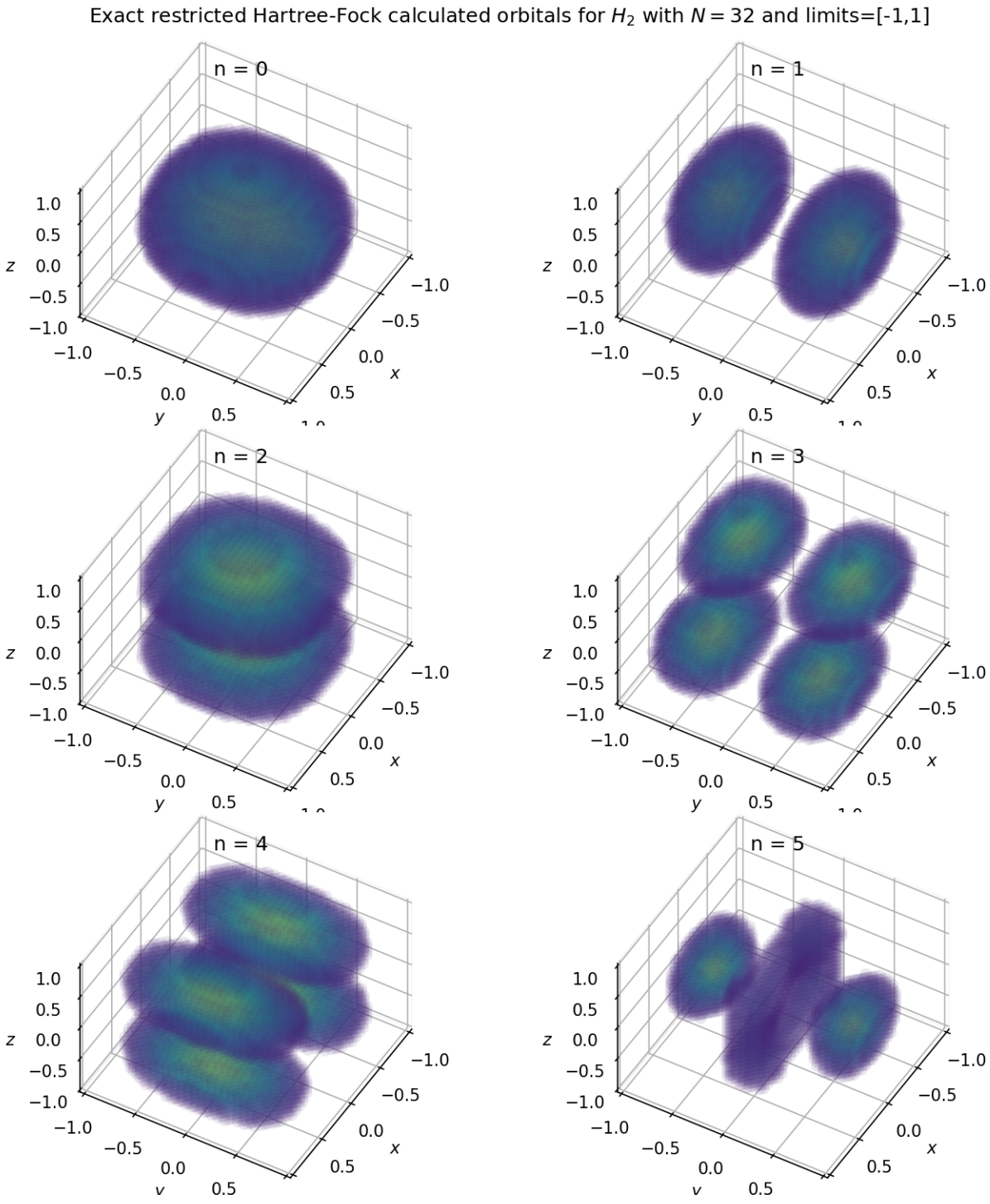

Figure 2: Calculated orbitals for H_2 molecule using limits of [-1,1]

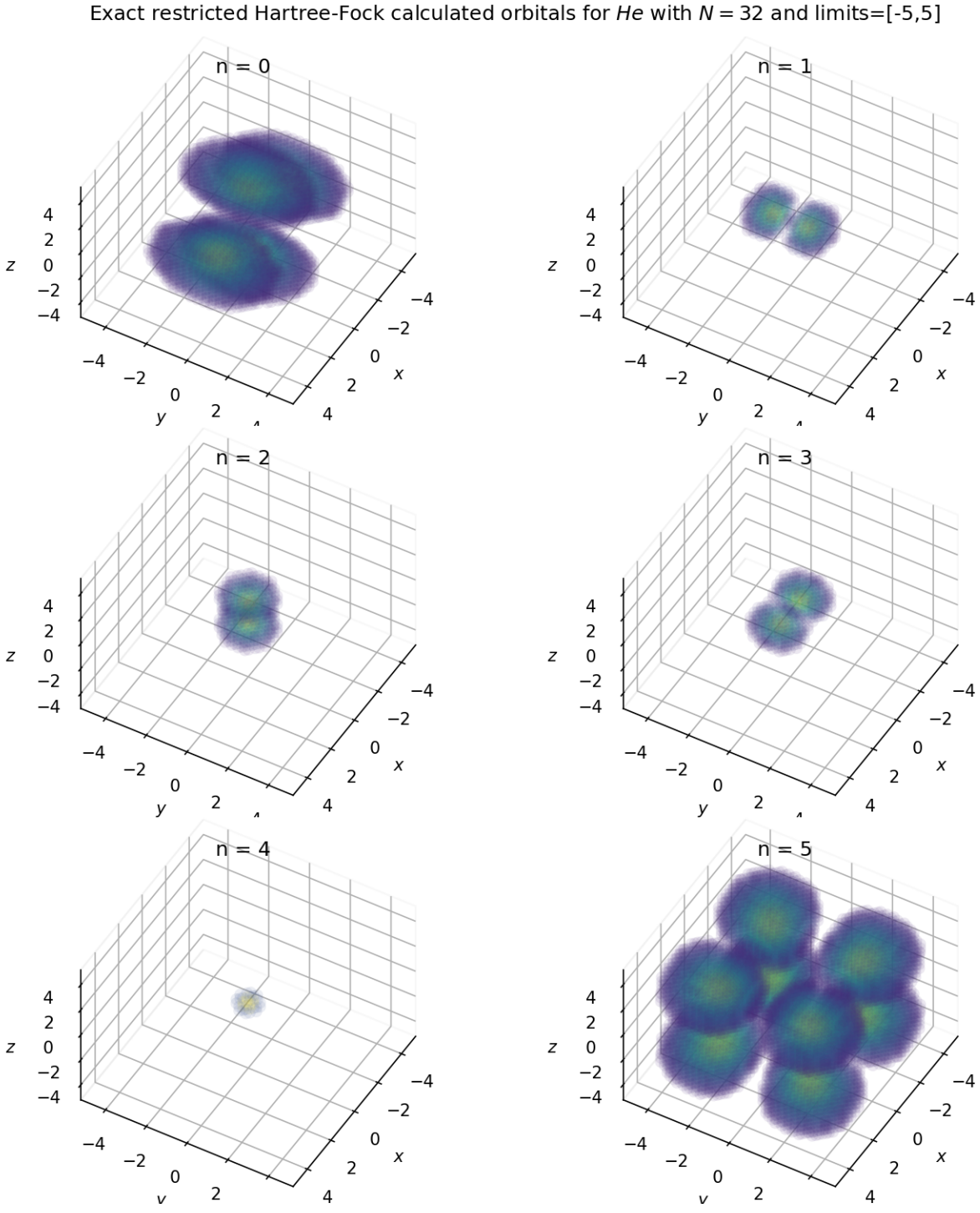

Figure 3: Calculated orbitals for He atom using limits of [-5,5]

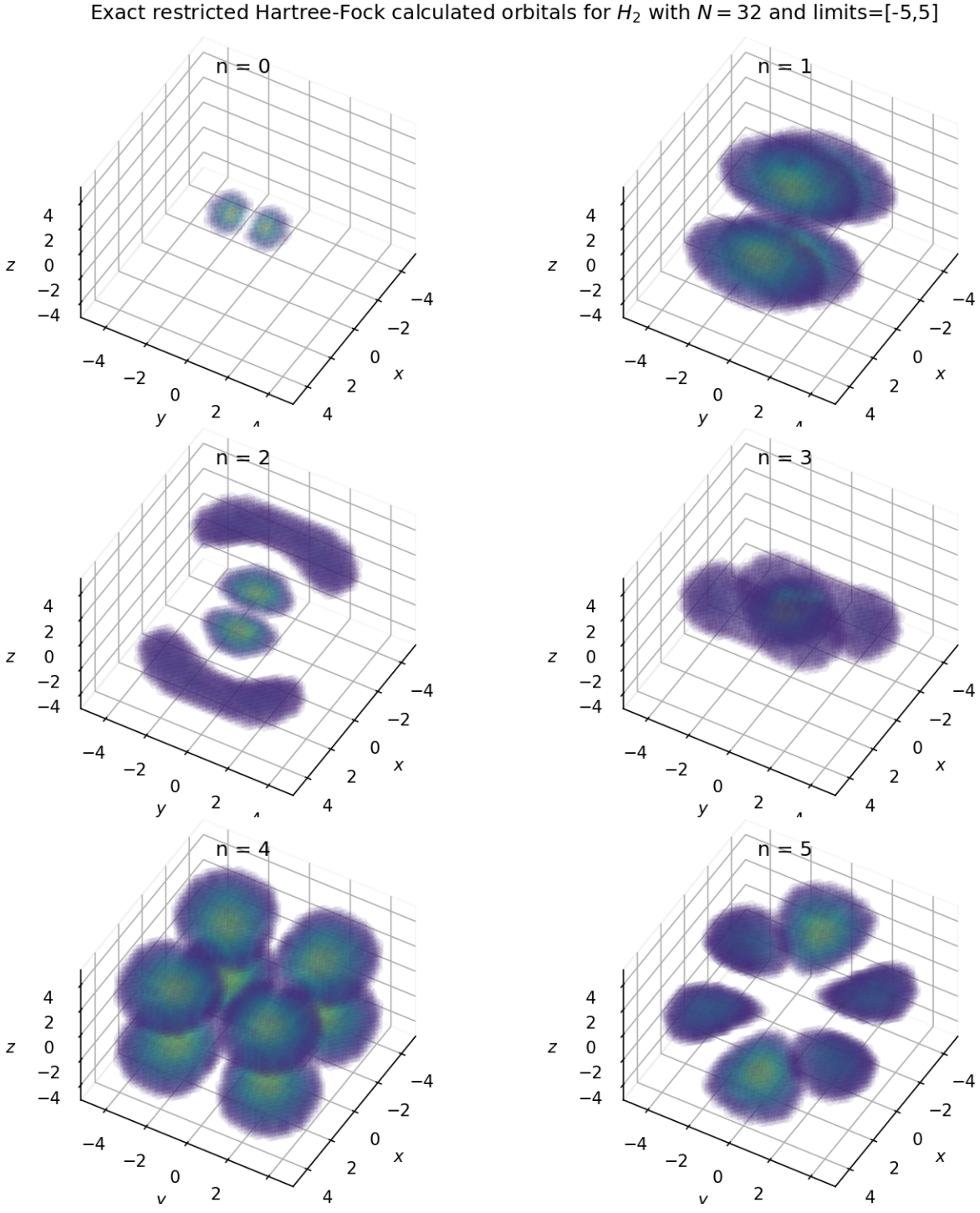

Figure 4: Calculated orbitals for H_2 molecule using limits of [-5,5]

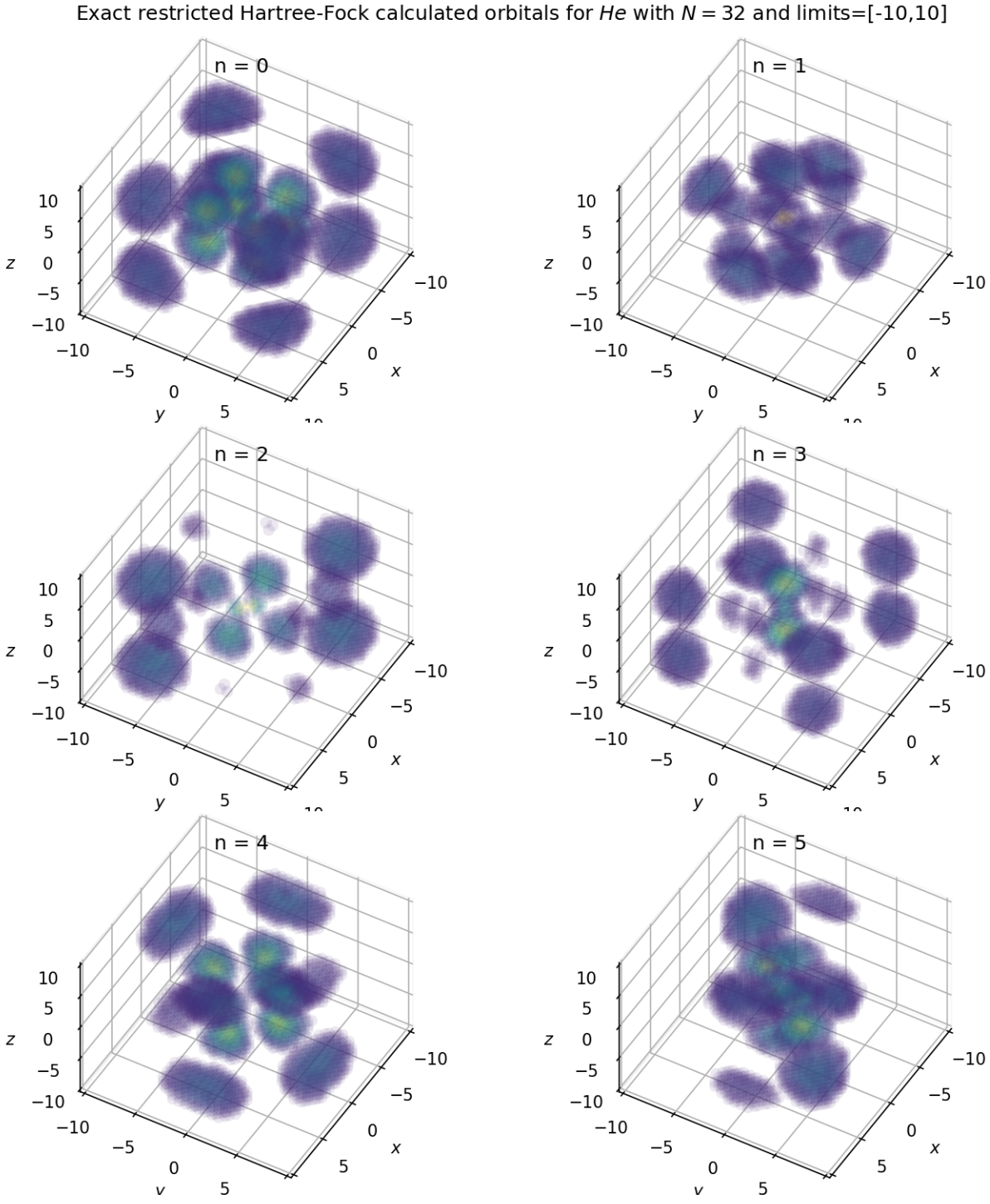

Figure 5: Calculated orbitals for He atom using limits of [-10,10]

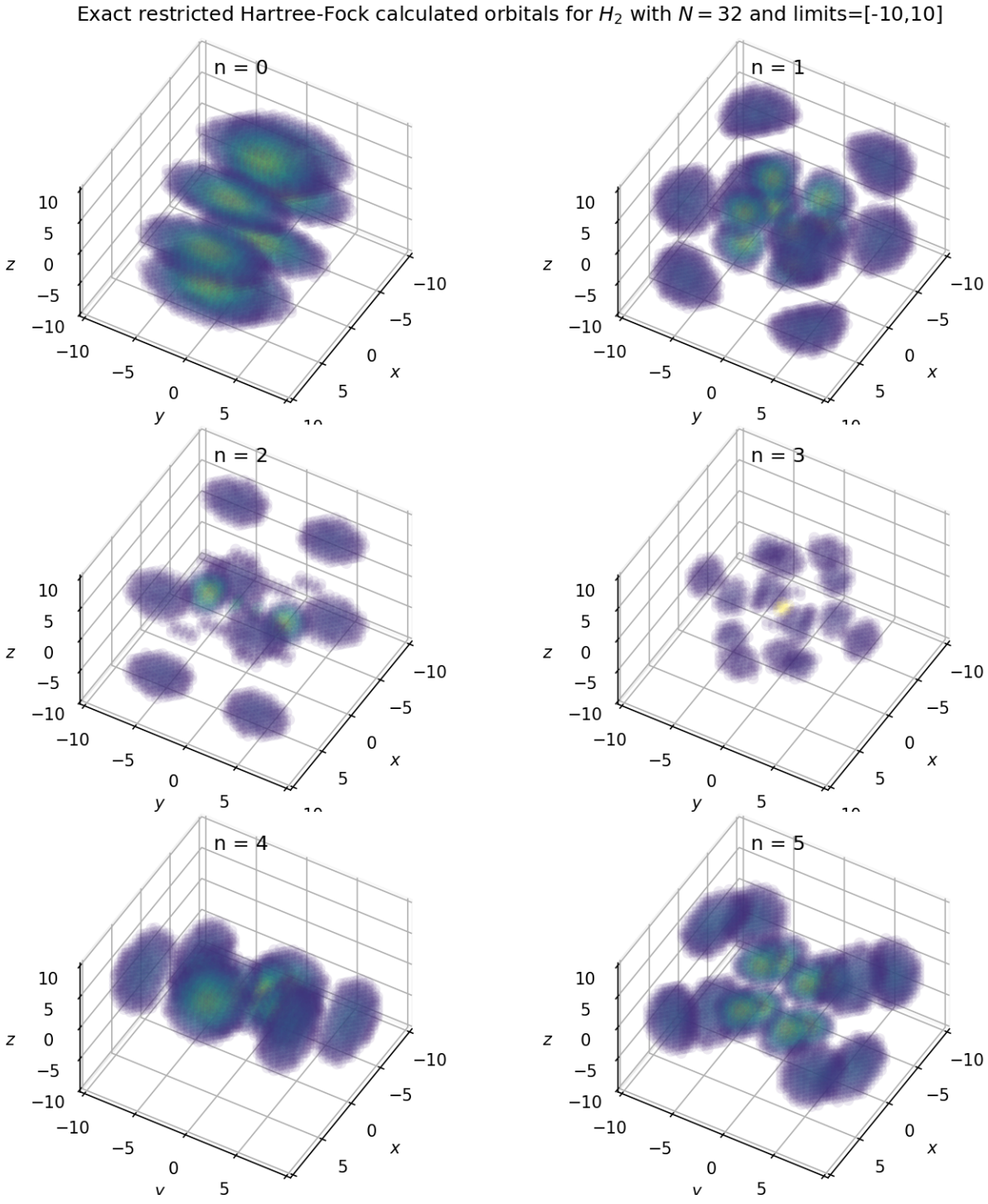

Figure 6: Calculated orbitals for H_2 molecule using limits of [-10,10]

n	1	2	3	4	5	6
E_n	−1.235253	3.503880	7.280588	7.298584	9.161709	10.836914
E_{total}			0.078488			

Table 1: The first six orbital energy levels and the total energy level obtained from applying HF to the Helium atom using limits of [-1,1]

n	1	2	3	4	5	6
E_n	0.537510	3.390950	4.290782	7.448408	7.795215	8.640438
E_{total}			−0.265565			

Table 2: The first six orbital energy levels obtained from applying HF to the Hydrogen molecule using limits of [-1,1]

n	1	2	3	4	5	6
E_n	−0.114483	−0.051618	−0.051522	−0.051513	0.034001	0.065295
E_{total}			−0.231081			

Table 3: The first six orbital energy levels obtained from applying HF to the Helium atom using limits of [-5,5]

n	1	2	3	4	5	6
E_n	−0.109023	−0.101815	−0.008226	−0.008226	0.065541	0.115528
E_{total}			−0.247352			

Table 4: The first six orbital energy levels obtained from applying HF to the Hydrogen molecule using limits of [-5,5]

n	1	2	3	4	5	6
E_n	−0.003174	0.000104	0.013754	0.013754	0.016740	0.016740
E_{total}			−0.235931			

Table 5: The first six orbital energy levels obtained from applying HF to the Helium atom using limits of [-10,10]

n	1	2	3	4	5	6
E_n	−0.017581	−0.003930	0.011870	0.012046	0.015739	0.016650
E_{total}			−0.272178			

Table 6: The first six orbital energy levels obtained from applying HF to the Hydrogen molecule using limits of [-10,10]

5 Discussion

- One of the major problems encountered with this implementation was the sensitivity to the small constant value used to prevent divide-by-zero conditions in the attraction and repulsion calculations. When the number was too small, the solver was often unable to find eigenvalues and eigenvectors for some iterations. A constant of 0.001 was experimentally found that enabled solution to converge properly. However, it is strongly suspected that this caused the system to become too "loose" to find the real solutions where the Coulomb attraction and repulsion wasn't being properly simulated.

- Obtaining these final results took several hours to converge on a result. With a partition size of $N = 32$, the sparse matrices were $N \times N \times N = 32{,}768$ in row and column length, for a total of 1,073,741,824 entries. At first, it was suspected that finding eigenvalues and eigenvectors was taking the longest. However, as mentioned in the program section, the longest time spent is actually calculating the integrals and filling up matrices with their respective results. Many more hours were spent in the development phase trying to tweak and debug the program and only after profiling the program was it discovered where the computation time was being spent. This was an incredible challenge. In future implementations, the integrations will likely be implemented by a library to iron out their quirks.

- The orbital energies and total energy for the system do not appear to be accurate when comparing them to literature. This suggests a major problem in this assignment's derivation of the equations and algorithm or perhaps an implementation problem in the program.

- Vastly different solutions were obtained when using smaller limits of $[-1, 1]$ and larger limits of $[-10, 10]$ for the problem, with the sweet spot appearing to be limits of $[-5, 5]$. This is likely tied to the problem being constrained into a virtual box of sorts where the electrons weren't free to orbit further. It is assumed that the more correct orbital results are those for when the limits are set to larger values. However, when the limits were set too large, very elaborate solutions started appearing where the electrons would fill intricate voids. The most reasonable results appears to have come from using limits of $[-5, 5]$.

- There were repeated eigenvalues (of approximately the same value) with very similar shapes. This is likely the solver finding identical or similar orbitals in different orientations. Perhaps this could be resolved using a finer partition size or somehow baking into the calculations that the space the entity occupies is actually infinite while only solving for a small window. It was reassuring to see some familiar shapes in the orbitals that were previously spotted as illustrations in textbooks.

- The current implementation does not treat the solution space as an infinite space and is more similar to a closed box. Either the limits need to be made large enough for the system or the implementation needs to simulate a larger space. The numerical integration, for example, may be able to extend its range, to allow the "other" electron to smear further.

- The solutions are extremely sensitive to partition size and the limits of the problem. Since the solver only reports the first six lowest eigenvalues, it's likely that similar solutions get moved in and out of these six slots as the parameters change.

6 Code Listings and Data

6.1 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 math
import functools
import time
import pickle
import datetime # for timestamping
import multiprocessing # for multiprocessing (MP) of matrix generation
import tqdm # progress bar for MP

numpy.set_printoptions(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
H2_BOND_LENGTH_ATOMIC_UNITS = 1.39839733222307
TINY_NUMBER = 0.001
IDX_X = 0
IDX_Y = 1
IDX_Z = 2
IDX_START = 0
```

```
IDX_END = 1
DATETIME_STR_FORMAT = '[%Y/%m/%d-%H:%M:%S]'
ENABLE_MP = True # multiprocessing

#
# This object encapsulates the results for pickling/unpickling
#
class Results:

    def __init__(self, eigenvectors=None, eigenvalues=None, args=None):

        self.eigenvectors = eigenvectors
        self.eigenvalues = eigenvalues
        self.args = args
        self.iteration_times = []
        self.total_time = None
        # keep track of total energies and eigenvalues
        self.historical_total_orbital_energies = []
        self.historical_eigenvalues = []

    def load(self, input_file):
        with open(input_file, 'rb') as file:
            tmp_dict = pickle.load(file)

        self.__dict__.update(tmp_dict)

    def save(self, output_file):
        with open(output_file, 'wb') as file:
            pickle.dump(self.__dict__, file, 2)

    def display_data(self):
        for key in self.__dict__:
            console_print(key)
            console_print(self.__dict__[key])

#
# This function generates the plots
#
def make_plots(results):

    # with help from this SO answer (4D scatter plots in matplotlib)
    # https://stackoverflow.com/a/66939879

    # number of partitions in the solution
    N = results.args.num_partitions
    # limits of sim
    limits = (-1*results.args.limit, results.args.limit)
    # generate coordinates
    coords = generate_coordinates(limits[IDX_START], limits[IDX_END], N)

    # eigenvectors
    eigenvectors = results.eigenvectors

    # total energy levels
    total_energy_levels = len(results.eigenvalues)

    # create figure object
    fig = plt.figure(figsize=(9,10), dpi=150, constrained_layout=True)

    # modify subject formatting
    if results.args.target_subject == 'he':
        title_subject = 'He'
    elif results.args.target_subject == 'h2':
        title_subject = 'H_2'

    fig.suptitle('Exact␣restricted␣Hartree-Fock␣calculated␣orbitals␣for␣$%s$␣with␣$N=%d$␣and
        ␣limits=[%d,%d]' % (title_subject, N, limits[IDX_START], limits[IDX_END]))
    axes = []
```

```
    # create results table
    for current_energy_level in range(total_energy_levels):

        # reshape eigenvectors for specified
        data = numpy.square(eigenvectors[:,current_energy_level].reshape((N,N,N)).transpose
            ())

        # three rows, two columns, index
        axes.append(fig.add_subplot(3, 2, current_energy_level + 1, projection='3d',
            proj_type='ortho'))

        # change viewing angles
        axes[-1].view_init(45, 30)

        axes[-1].xaxis.pane.fill = False
        axes[-1].yaxis.pane.fill = False
        axes[-1].zaxis.pane.fill = False

        # set limits
        axes[-1].set_xlim3d(limits[IDX_START], limits[IDX_END])
        axes[-1].set_ylim3d(limits[IDX_START], limits[IDX_END])
        axes[-1].set_zlim3d(limits[IDX_START], limits[IDX_END])

        axes[-1].set_xlabel('$x$')
        axes[-1].set_ylabel('$y$')
        axes[-1].set_zlabel('$z$')

        # set title
        axes[-1].set_title('n = %d' % current_energy_level, y=0.95)

        # create a mask for the data to make the visualization clearer
        mask = data > (data.max() * 0.1)
        idx = numpy.arange(int(numpy.prod(data.shape)))
        x, y, z = numpy.unravel_index(idx, data.shape)
        x, y, z = numpy.meshgrid(coords[IDX_X], coords[IDX_Y], coords[IDX_Z])
        axes[-1].scatter(x, y, z, c=data.flatten(), s=50.0 * mask, edgecolor='face', alpha
            =0.1, marker='o', cmap='viridis', linewidth=0)

    plot_file_name = results.args.output_file.replace('.xyzp','') + ('.png')
    fig.savefig(plot_file_name)
    # plt.show()

#
# Console formatter
#
def console_print(string=''):

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

    datetime_now = datetime.datetime.now()
    print(datetime_now.strftime(DATETIME_STR_FORMAT) + ' ' + string)

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

    console_print(' ** Exact Hartree-Fock simulator **')

    # results object
    results = Results()

    ## extract arguments

    console_print(' ** Arguments:')
```

```
    # input file to process
    input_file = cmd_args.input_file

    # if we have an input file, it will have data and args, load them
    if input_file:
        console_print('Input %s supplied, reading its data and arguments...' % input_file)
        results.load(input_file)
        args = results.args
        args.input_file = input_file
        eigenvectors = results.eigenvectors
        eigenvalues = results.eigenvalues
    else:
        args = cmd_args

    # print args
    for arg in vars(args):
        console_print('\t' + arg + ' ' + str(getattr(args, arg)))

    # output file to save results to
    output_file = args.output_file

    # energy level to converge on
    # always use value provided by command line
    energy_level = cmd_args.energy_level

    # target subject to run sim on
    target_subject = args.target_subject

    # convergence condition percentage
    convergence_percentage = args.convergence_percentage

    # number of eigenvalues to calculate
    total_energy_levels = 6

    # damping factor
    damping_factor = args.damping_factor

    # number of partitions in the solution
    N = args.num_partitions

    # limits of sim
    limits = (-1*args.limit, args.limit)

    # generate coordinates
    coords = generate_coordinates(limits[IDX_START], limits[IDX_END], N)

    # calculate partition size
    h = coords[IDX_X][1]-coords[IDX_X][0]

    ## start program

    console_print(' ** Program start!')

    console_print('\tPartition size: %f' % h)

    # check if we're simulating something new (no input file specified)
    if not input_file:

        # generate attraction matrix for hydrogen molecule
        attraction_matrix_hydrogen = attraction_matrix_gen(attraction_func_hydrogen, N,
            coords)

        # generate attraction matrix for helium molecule
        attraction_matrix_helium = attraction_matrix_gen(attraction_func_helium, N, coords)

        # generate laplacian matrix
        laplacian_matrix = second_order_laplacian_3d_sparse_matrix_gen(N)
```

```
        # generate kinetic matrix
        kinetic_energy_matrix = (-1.0/(2.0*h**2))*laplacian_matrix

        # create base solution
        eigenvectors = numpy.ndarray(((N**3), total_energy_levels))

        # last eigenvalues
        last_total_energy = 0

        # first iteration
        first_iteration = True

        # iteration counter
        iteration_count = 1

        # total time
        total_time_start = time.time()

        console_print('␣**␣Simulation␣start!␣')

        # main loop
        while True:

            # iteration time
            iteration_time_start = time.time()

            if first_iteration:
                console_print('␣**␣First␣iteration,␣zeros␣used␣as␣first␣guess')
                first_iteration = False
            else:
                console_print('␣**␣Iteration:␣%d' % iteration_count)

            # Modify eigenvectors to help with convergence
            console_print('␣**␣Modifying␣eigenvector␣values␣with␣damping␣factor␣of␣%f' %
                damping_factor)
            eigenvector = eigenvectors[:,energy_level] * damping_factor

            # create integration matrix
            console_print('␣**␣Generating␣integration␣matrix')
            # turn our eigenvector into a square matrix and square all of the terms
            orbital_values_squared = numpy.square(eigenvector).reshape((N,N,N)).transpose()

            integration_matrix = integration_matrix_gen(orbital_values_squared, coords)

            # create Fock matrix
            if target_subject == 'h2':
                fock_matrix = kinetic_energy_matrix + attraction_matrix_hydrogen +
                    integration_matrix
            elif target_subject == 'he':
                fock_matrix = kinetic_energy_matrix + attraction_matrix_helium +
                    integration_matrix
            else:
                console_print('Fatal␣error,␣exiting.')
                quit()

            # get (total_energy_levels) eigenvectors and eigenvalues and order them from
                smallest to largest
            console_print('␣**␣Obtaining␣eigenvalues␣and␣eigenvectors...')
            try:
                eigenvalues, eigenvectors = scipy.sparse.linalg.eigsh(fock_matrix, k=
                    total_energy_levels, which='SM', maxiter=10000, tol=1e-4)
            except ArpackNoConvergence:
                console_print('␣**␣Fatal␣error!␣Couldn\'t␣converge!')
                console_print('␣**␣Will␣apply␣damping␣factor␣again...')
                continue

            # check percentage difference between previous and current eigenvalues
```

```
            # total_energy = numpy.sum(eigenvalues)
            # only compare the orbital we're trying to converge on
            total_energy = eigenvalues[energy_level]

            # record history
            results.historical_eigenvalues.append(eigenvalues)
            results.historical_total_orbital_energies.append(total_energy)

            # calculate total energy
            total_energy_percent_diff = abs(total_energy - last_total_energy)/((total_energy
                 + last_total_energy) / 2)

            console_print(' ** Orbital %d energy %% diff: %.3f%%' % (energy_level,
                total_energy_percent_diff * 100.0))

            # update last value
            last_total_energy = total_energy

            # update iteration count
            iteration_count = iteration_count + 1

            # append iteration time
            iteration_time = time.time() - iteration_time_start
            results.iteration_times.append(iteration_time)

            console_print(' ** Iteration end! Iteration time: %.3f seconds**' %
                iteration_time)
            console_print(' ** Eigenvalues:')
            console_print(eigenvalues)

            # check if we meet convergence condition
            if abs(total_energy_percent_diff) < (convergence_percentage/100.0):
                break

        # append total time
        total_time_end = time.time()
        total_time = total_time_end - total_time_start
        results.total_time = total_time
        console_print(' ** Simulation end! ')
        console_print(' ** Total time: %.3f seconds **' % results.total_time)

        # construct file name
        if not output_file:
            output_file = target_subject + '_N%d_l%d.xyzp' % (N, limits[IDX_END])
            args.output_file = output_file

        console_print(' ** Saving results to ' + output_file)
        results.eigenvectors = eigenvectors
        results.eigenvalues = eigenvalues
        results.args = args
        results.save(output_file)

    console_print(' ** Solution summary: **')

    console_print(' ** Solution took %d iterations to converge with a convergence criteria
        of %.1f%% in %.3f seconds' % (len(results.iteration_times), results.args.
        convergence_percentage, results.total_time))
    for iteration, iteration_time in enumerate(results.iteration_times):
        console_print('\tIteration %i took %.3f seconds' % (iteration, iteration_time))

    # solution results display

    # console_print(' ** Coordinates:')
    # console_print(coords)

    console_print(' ** Eigenvalues:')
    console_print(eigenvalues)
```

```
    console_print('␣**␣Orbital␣energies:')
    for n in range(len(eigenvalues)):
        console_print('\tn=%d␣orbital␣energy:␣%f' % (n, eigenvalues[n]))
        eigenvector = eigenvectors[:,n]
        squared_eigenvector_3d = lambda x, y, z : numpy.square(eigenvector).reshape((N,N,N))
            .transpose()[x, y, z]
        expectation = integrate(squared_eigenvector_3d, coords)
        console_print('\t\tPsi_%d␣expectation␣value:␣%f' % (n, expectation))

    console_print('␣**␣Total␣Energy:')
    eigenvector = eigenvectors[:,n]
    console_print('\tn=%d␣total␣energy:␣%f' % (energy_level, calculate_total_energy(
        target_subject, eigenvector, coords)))

    # plot data
    console_print('␣**␣Plotting␣data...')
    make_plots(results)

#
# This function calculates the total energy of the system
#
def calculate_total_energy(target_subject, orbital_values, coords):

    # extract N
    N = int(numpy.cbrt(len(orbital_values)))

    # extract h
    h = coords[IDX_X][1] - coords[IDX_X][0]

    # turn orbital values into 3d array
    orbital_values_squared = numpy.square(orbital_values.reshape((N,N,N)).transpose())

    # calculate kinetic energy

    # helper lambdas
    # interpret a matrix as a function f(x,y,z)
    matrix_to_func = lambda matrix : lambda x, y, z : matrix[x, y, z]

    # calculate kinetic energy
    diff_result = diff_2(matrix_to_func(orbital_values_squared), coords)
    kinetic_energy = -1.0*(integrate(matrix_to_func(diff_result), coords))

    # calculate nuclear attraction
    if target_subject == 'he':
        attraction_vals = attraction_val_matrix_gen(attraction_func_helium, coords)
    elif target_subject == 'h2':
        attraction_vals = attraction_val_matrix_gen(attraction_func_hydrogen, coords)
    else:
        console_print('Fatal␣error,␣exiting.')
        quit()

    nuclear_attraction = 2*integrate(matrix_to_func(orbital_values_squared*attraction_vals),
         coords)

    # calculate Coulomb/exchange contribution
    two_electron_integration = two_electron_integration_calc(orbital_values_squared, coords)

    return kinetic_energy + nuclear_attraction + two_electron_integration

#
# This functions returns whether or not the matrix is symmetric
#
def is_symmetric(A, tol=1e-8):
    return scipy.sparse.linalg.norm(A-A.T, scipy.Inf) < tol;

#
# This functions returns whether or not the matrix is symmetric
#
```

```
def is_hermitian(A, tol=1e-8):
    return scipy.sparse.linalg.norm(A-A.H, scipy.Inf) < tol;

#
# This function takes in a matrix index (row or column) and returns the
# associated coordinate indices as a tuple.
#
@functools.lru_cache
def matrix_index_to_coordinate_indices(matrix_index, N):

    # Z is kind of like the MSB, as it changes less often, so we'll treat this
    # as base conversion of sorts where to new base is N

    base_10_num = matrix_index
    digits = []

    # while we have a base 10 number to work with
    for i in range(3):
        # do moduluses to extract the digits of the new number
        digits = digits + [base_10_num % N]
        # go to the next digit
        base_10_num = base_10_num // N

    return tuple(digits)

#
# This function coverts coordinate indices to a matrix index
#
def coordinate_indices_to_matrix_index(coord_indices, N):

    return coord_indices[IDX_X] + coord_indices[IDX_Y]*N + coord_indices[IDX_Z]*N*N
#
# This function takes in coordinate indices and returns the coordinates
# associated with them.
#
@functools.lru_cache
def coordinate_index_to_coordinates(coord_indices, coords):

    return (coords[IDX_X][coord_indices[IDX_X]], coords[IDX_Y][coord_indices[IDX_Y]], coords
        [IDX_Z][coord_indices[IDX_Z]])

#
# Generate the coordinates for the solution space, with the assumption that it
# is cubic.
#
def generate_coordinates(minimum, maximum, N):

    x = tuple(numpy.linspace(minimum, maximum, N))
    y = tuple(numpy.linspace(minimum, maximum, N))
    z = tuple(numpy.linspace(minimum, maximum, N))

    return (x, y, z)

#
# This function returns the nuclear attraction for an electron in the Helium
# element simulation. A small number tiny_number is provided to prevent divide
# by zero scenarios.
#
def attraction_func_helium(coords, h):

    x = coords[IDX_X]
    y = coords[IDX_Y]
    z = coords[IDX_Z]

    tiny_number = TINY_NUMBER

    denominator = math.sqrt(x**2 + y**2 + z**2)
```

```
    return -((2.0/(tiny_number + denominator)))

#
# This function returns the nuclear attraction for an electron in the Hydrogen
# molecule simulation. A small number tiny_number is provided to prevent divide
# by zero scenarios.
#
def attraction_func_hydrogen(coords, h):

    x = coords[IDX_X]
    y = coords[IDX_Y]
    z = coords[IDX_Z]

    tiny_number = TINY_NUMBER

    denominator_1 = math.sqrt(((H2_BOND_LENGTH_ATOMIC_UNITS/2) - x)**2 + y**2 + z**2)
    denominator_2 = math.sqrt(((-H2_BOND_LENGTH_ATOMIC_UNITS/2) - x)**2 + y**2 + z**2)

    return -((1.0/(tiny_number + denominator_1)) + (1.0/(tiny_number + denominator_2)))

#
# This functions calculates the repulsion between two electrons. A small number
# TINY_NUMBER is provided to prevent divide by zero scenarios.
#
@functools.lru_cache(maxsize=8192)
def repulsion_func(coords_1, coords_2, h):

    x1 = coords_1[IDX_X]
    y1 = coords_1[IDX_Y]
    z1 = coords_1[IDX_Z]

    x2 = coords_2[IDX_X]
    y2 = coords_2[IDX_Y]
    z2 = coords_2[IDX_Z]

    tiny_number = TINY_NUMBER

    denominator = math.sqrt((x2 - x1)**2 + (y2 - y1)**2 + (z2 - z1)**2)

    return ((1.0/(tiny_number + denominator)))

# This function generates an N*N*N matrix A with the specified attraction
# function. This matrix contains values that evaluates the attraction
# function at the specified coordinates. i.e, A[x,y,z] = attraction_func
# (x,y,z) Note that this is different from the matrix generated by
# attraction_matrix_gen, which is used to solve a linear system of equations.
def attraction_val_matrix_gen(attraction_func, coords):

    # get partition size
    h = coords[IDX_X][1] - coords[IDX_X][0]
    # get number of partitions
    N = len(coords[IDX_X])

    # generate a solution space
    solution_space = numpy.empty((N,N,N))

    for xi in range(N):
        for yi in range(N):
            for zi in range(N):

                x = coords[IDX_X][xi]
                y = coords[IDX_Y][yi]
                z = coords[IDX_Z][zi]

                solution_space[xi,yi,zi] = attraction_func((x, y, z), h)

    return solution_space
```

```
#
# This function generates an N*N*N matrix A with the inner result of the
# integration term (combined Coulomb/exchange)
#
def inner_integration_val_matrix_gen(orbital_values_squared, coords):

    # get partition size
    h = coords[IDX_X][1] - coords[IDX_X][0]
    # get number of partitions
    N = len(coords[IDX_X])

    # generate coordinates
    coordinates = []
    for xi in range(N):
        for yi in range(N):
            for zi in range(N):
                coordinates.append((coords[IDX_X][xi],coords[IDX_Y][yi],coords[IDX_Z][zi]))

    # this takes an extremely long time!!!
    if ENABLE_MP:
        # multiprocessing filling of the diagonal
        with multiprocessing.Pool(processes = multiprocessing.cpu_count()-1,
            maxtasksperchild=1000) as pool:
            func = functools.partial(integration_term_func, orbital_values_squared, coords)
            # diagonal = pool.map(func, coordinates)
            solution_space = list(tqdm.tqdm(pool.imap(func, coordinates), total=(N**3),
                ascii=True, smoothing=0.01))

    # reshape solution space
    solution_space = numpy.array(solution_space).reshape((N,N,N)).transpose()

    return solution_space

#
# This function calculates the Coulomb/exchange two electron integral
# expectation value used in the total energy calculation.
#
def two_electron_integration_calc(orbital_values_squared, coords):

    # get partition size
    h = coords[IDX_X][1] - coords[IDX_X][0]
    # get number of partitions
    N = len(coords[IDX_X])

    # calculate inner integral matrix
    console_print('** Calculating inner two electron integral for all solution space...')
    inner_integration_val_matrix = inner_integration_val_matrix_gen(orbital_values_squared,
        coords)

    # running sum
    sum = 0

    # calculate the integration over the solution space of the specified psi squared
    for xi in range(N):
        for yi in range(N):
            for zi in range(N):
                # first integration weight is 0
                if xi == 0 or yi == 0 or zi == 0:
                    w = 0
                else:
                    w = h
                sum = sum + w*(orbital_values_squared[xi, yi, zi]*
                    inner_integration_val_matrix[xi, yi, zi])

    return sum

#
# This function generates an attraction matrix with the specified attraction
```

```python
# function and coordinates
#
def attraction_matrix_gen(attraction_func, N, coords):

    # extract h
    h = coords[IDX_X][1] - coords[IDX_X][0]

    # use scipy sparse matrix generation
    # create the diagonal
    diagonal = [attraction_func(coordinate_index_to_coordinates(
        matrix_index_to_coordinate_indices(i, N), coords), h) for i in range(N**3)]

    # now generate the matrix with the desired diagonal
    matrix = scipy.sparse.spdiags(data=diagonal, diags=0, m=N**3, n=N**3)

    return matrix.tocoo()

#
# This function generates the second order laplacian matrix for 3D space for N
# partitions.
#
#   e.g. N = 3
#
#     +-------+-------+-------+-------+-------+-------+-------+-------+-------+
#     |-6 1   | 1     |       | 1     |       |       |       |       |       |
#     | 1-6 1 |   1   |       |   1   |       |       |       |       |       |
#     |   1-6 |     1 |       |     1 |       |       |       |       |       |
#     +-------+-------+-------+-------+-------+-------+-------+-------+-------+
#     | 1     |-6 1   | 1     |       | 1     |       |       |       |       |
#     |   1   | 1-6 1 |   1   |       |   1   |       |       |       |       |
#     |     1 |   1-6 |     1 |       |     1 |       |       |       |       |
#     +-------+-------+-------+-------+-------+-------+-------+-------+-------+
#     |       | 1     |-6 1   |       |       | 1     |       |       |       |
#     |       |   1   | 1-6 1 |       |       |   1   |       |       |       |
#     |       |     1 |   1-6 |       |       |     1 |       |       |       |
#     +-------+-------+-------+-------+-------+-------+-------+-------+-------+
#     | 1     |       |       |-6 1   | 1     |       | 1     |       |       |
#     |   1   |       |       | 1-6 1 |   1   |       |   1   |       |       |
#     |     1 |       |       |   1-6 |     1 |       |     1 |       |       |
#     +-------+-------+-------+-------+-------+-------+-------+-------+-------+
#     |       | 1     |       | 1     |-6 1   | 1     |       | 1     |       |
#     |       |   1   |       |   1   | 1-6 1 |   1   |       |   1   |       |
#     |       |     1 |       |     1 |   1-6 |     1 |       |     1 |       |
#     +-------+-------+-------+-------+-------+-------+-------+-------+-------+
#     |       |       | 1     |       | 1     |-6 1   |       |       | 1     |
#     |       |       |   1   |       |   1   | 1-6 1 |       |       |   1   |
#     |       |       |     1 |       |     1 |   1-6 |       |       |     1 |
#     +-------+-------+-------+-------+-------+-------+-------+-------+-------+
#     |       |       |       | 1     |       |       |-6 1   | 1     |       |
#     |       |       |       |   1   |       |       | 1-6 1 |   1   |       |
#     |       |       |       |     1 |       |       |   1-6 |     1 |       |
#     +-------+-------+-------+-------+-------+-------+-------+-------+-------+
#     |       |       |       |       | 1     |       | 1     |-6 1   | 1     |
#     |       |       |       |       |   1   |       |   1   | 1-6 1 |   1   |
#     |       |       |       |       |     1 |       |     1 |   1-6 |     1 |
#     +-------+-------+-------+-------+-------+-------+-------+-------+-------+
#     |       |       |       |       |       | 1     |       | 1     |-6 1   |
#     |       |       |       |       |       |   1   |       |   1   | 1-6 1 |
#     |       |       |       |       |       |     1 |       |     1 |   1-6 |
#     +-------+-------+-------+-------+-------+-------+-------+-------+-------+
def second_order_laplacian_3d_sparse_matrix_gen(N):

    # create block matrices
    # sub main block diagonals (containing -6 surrounded by 1s) found within N*N blocks
    # e.g N=3
    # +-------+
    # |-6 1   |
    # | 1-6 1 |
```

```
    # |   1-6 |
    # +-------+
    sub_main_block_diag = [-6.0 for i in range(N)]
    sub_main_block_outer_diags = [1.0 for i in range(N)]
    sub_main_block = scipy.sparse.spdiags(data=[sub_main_block_outer_diags,
        sub_main_block_diag, sub_main_block_outer_diags], diags=[-1,0,1], m=N, n=N)

    # create mini identity blocks found within N*N blocks
    # e.g N=3
    # +-------+
    # | 1     |
    # |   1   |
    # |     1 |
    # +-------+
    mini_ident_block = scipy.sparse.identity(N)

    # create a list of blocks to be used in the sparse.bmat function to generate
    # main block along the diagonal
    # e.g. N=3
    # +-------+-------+-------+
    # |-6 1   | 1     |       |
    # | 1-6 1 |   1   |       |
    # |   1-6 |     1 |       |
    # +-------+-------+-------+
    # | 1     |-6 1   | 1     |
    # |   1   | 1-6 1 |   1   |
    # |     1 |   1-6 |     1 |
    # +-------+-------+-------+
    # |       | 1     |-6 1   |
    # |       |   1   | 1-6 1 |
    # |       |     1 |   1-6 |
    # +-------+-------+-------+
    block_lists = [[None for i in range(N)] for j in range(N)]
    for i in range(N):
        for j in range(N):
            # add main blocks
            if j == i:
                block_lists[i][j] = sub_main_block
            # add diagonal blocks
            if (j == (i + 1)) and ((i + 1) != N):
                block_lists[i][j] = mini_ident_block
            if (j == (i - 1)) and ((i - 1) != -1):
                block_lists[i][j] = mini_ident_block
    main_block = scipy.sparse.bmat(block_lists)

    # create large identity blocks of size N*N
    # e.g. N=3
    # +-------+-------+-------+
    # | 1     |       |       |
    # |   1   |       |       |
    # |     1 |       |       |
    # +-------+-------+-------+
    # |       | 1     |       |
    # |       |   1   |       |
    # |       |     1 |       |
    # +-------+-------+-------+
    # |       |       | 1     |
    # |       |       |   1   |
    # |       |       |     1 |
    # +-------+-------+-------+
    large_ident_block = scipy.sparse.identity(N*N)

    # create a list of blocks to be used in the sparse.bmat function to generate
    # the final laplacian_matrix
    block_lists = [[None for i in range(N)] for j in range(N)]
    for i in range(N):
        for j in range(N):
            # add main blocks
```

```
            if j == i:
                block_lists[i][j] = main_block
            # add diagonal blocks
            if (j == (i + 1)) and ((i + 1) != N):
                block_lists[i][j] = large_ident_block
            if (j == (i - 1)) and ((i - 1) != -1):
                block_lists[i][j] = large_ident_block

    return scipy.sparse.bmat(block_lists)

#
# This is a generic numerical second-order central 3D differentiation (the same
# used in the matrix solution)
#
def diff_2(function, coords):

    # get partition size
    h = coords[IDX_X][1] - coords[IDX_X][0]
    # get number of partitions
    N = len(coords[IDX_X])

    # generate a solution space
    solution_space = numpy.empty((N,N,N))

    for xi in range(N):
        for yi in range(N):
            for zi in range(N):

                # central requires values from the past, if required, use 0
                if xi == 0:
                    x_past = 0
                else:
                    x_past = function(xi-1,yi,zi)

                if yi == 0:
                    y_past = 0
                else:
                    y_past = function(xi,yi-1,zi)

                if zi == 0:
                    z_past = 0
                else:
                    z_past = function(xi,yi,zi-1)

                # we'll always have present values
                present = function(xi,yi,zi)

                # central requires values from the future, if required, use 0
                if xi == (N-1):
                    x_future = 0
                else:
                    x_future = function(xi+1,yi,zi)

                if yi == (N-1):
                    y_future = 0
                else:
                    y_future = function(xi,yi+1,zi)

                if zi == (N-1):
                    z_future = 0
                else:
                    z_future = function(xi,yi,zi+1)

                # differentiate
                solution_space[xi,yi,zi] = ((x_future + y_future + z_future - (6*present) +
                    x_past + y_past + z_past)/(h**2))

    return solution_space
```

```
#
# This is a generic numerical integration over the cubic 3D space covered by the
# solution space.
#
def integrate(function, coords):

    # get partition size
    h = coords[IDX_X][1] - coords[IDX_X][0]
    # get number of partitions
    N = len(coords[IDX_X])

    # running sum
    sum = 0

    # calculate the integration over the solution space
    for xi in range(N):
        for yi in range(N):
            for zi in range(N):

                # first integration weight is 0
                if xi == 0 or yi == 0 or zi == 0:
                    w = 0
                else:
                    w = h
                row_index = xi + yi*N + zi*N*N
                sum = sum + w*function(xi, yi, zi)

    # return result
    return sum

#
# This function evaluates the integration function used by both the Helium
# element and the Hydrogen molecule.
#
def integration_term_func(orbital_values_squared, all_coords, coords_1):

    # get partition size
    h = all_coords[IDX_X][1] - all_coords[IDX_X][0]
    # get number of partitions
    N = len(all_coords[IDX_X])

    # running sum
    sum = 0

    # calculate the integration over the solution space of the specified psi squared
    for xi, x in enumerate(all_coords[IDX_X]):
        for yi, y in enumerate(all_coords[IDX_Y]):
            for zi, z in enumerate(all_coords[IDX_Z]):
                # first integration weight is 0
                if xi == 0 or yi == 0 or zi == 0:
                    w = 0
                else:
                    w = h
                matrix_index = xi + yi*N + zi*N*N
                coords_2 = (x, y, z)
                sum = sum + w*orbital_values_squared[xi, yi, zi]*repulsion_func(coords_1,
                    coords_2, h)

    return sum

# This function generates the the integration matrix
#
def integration_matrix_gen(orbital_values_squared, coords):

    # extract h from coordinates
    h = coords[IDX_X][1] - coords[IDX_X][0]
    # get number of partitions
```

```
    N = len(coords[IDX_X])

    # generate coordinates
    coordinates = [coordinate_index_to_coordinates(matrix_index_to_coordinate_indices(i, N),
         coords) for i in range(N**3)]

    # this takes an extremely long time!!!
    if ENABLE_MP:
        # multiprocessing filling of the diagonal
        with multiprocessing.Pool(processes = multiprocessing.cpu_count()-1,
            maxtasksperchild=1000) as pool:
            func = functools.partial(integration_term_func, orbital_values_squared, coords)
            # diagonal = pool.map(func, coordinates)
            diagonal = list(tqdm.tqdm(pool.imap(func, coordinates), total=(N**3), ascii=True
                , smoothing=0.01))
    else:
        # create the diagonal (no MP)
        diagonal = numpy.ndarray(N**3)
        for i in tqdm.tqdm(range(N**3), ascii=True):
            diagonal[i] = integration_term_func(orbital_values_squared, coords, coordinates[
                i])
        # add a new line after we're done progress
        console_print()

    # now generate the matrix with the desired diagonal
    matrix = scipy.sparse.spdiags(data=diagonal, diags=0, m=N**3, n=N**3)

    return matrix.tocoo()

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

    # arguments for the program
    parser.add_argument('-i', type=str, dest='input_file', action='store',
        help='input␣file␣(*.xyzp)␣to␣plot')

    parser.add_argument('-o', type=str, dest='output_file', action='store',
        help='path␣and␣name␣of␣the␣file␣containing␣the␣results␣of␣the␣current␣run')

    parser.add_argument('-e', type=int, default=0, dest='energy_level', action='store',
        choices=[0, 1, 2, 3, 4, 5],
        help='energy␣level␣to␣generate␣and/or␣plot')

    parser.add_argument('-t', type=str, default='h2', dest='target_subject', action='store',
         choices=['h2', 'he'],
        help='target␣subject␣to␣run␣exact␣HF␣sim␣on')

    parser.add_argument('-p', type=int, default=14, dest='num_partitions', action='store',
        help='number␣of␣partitions␣to␣discretize␣the␣simulation')

    parser.add_argument('-l', type=float, default=5, dest='limit', action='store',
        help='the␣x,y,z␣max␣limit,␣forming␣a␣cubic␣solution␣space')

    parser.add_argument('-c', type=float, default=1.0, dest='convergence_percentage', action
        ='store',
        help='percent␣change␣threshold␣for␣convergence')

    parser.add_argument('-d', type=float, default=0.1, dest='damping_factor', action='store'
        ,
        help='damping␣factor␣to␣apply␣to␣orbital␣results␣between␣iterations')

    args = parser.parse_args()

    main(args)
```

7 References

These aren't citing anything, but they were useful in helping me figure out this assignment.

- https://www.12000.org/my_notes/mma_matlab_control/KERNEL/KEse83.htm
- https://www.value-at-risk.net/numerical-integration-multiple-dimensions/
- https://chem.libretexts.org/Bookshelves/Physical_and_Theoretical_Chemistry_Textbook_Maps/Book%3A_Quantum_States_of_Atoms_and_Molecules_(Zielinksi_et_al)/09%3A_The_Electronic_States_of_the_Multielectron_Atoms/9.07%3A_The_Self-Consistent_Field_Approximation_(Hartree-Fock_Method)
- http://vergil.chemistry.gatech.edu/notes/hf-intro/hf-intro.html
- https://chem.libretexts.org/Bookshelves/Physical_and_Theoretical_Chemistry_Textbook_Maps/Physical_Chemistry_(LibreTexts)/08%3A_Multielectron_Atoms/8.07%3A_Hartree-Fock_Calculations_Give_Good_Agreement_with_Experimental_Data