Hartree Fock and its Implementation from Scratch

Malte F. Lange*

Department of Chemistry and James Franck Institute,
University of Chicago, Chicago, Illinois 60637 USA

I. MOTIVATION

Learning quantum mechanics is difficult enough, and adding onto that a separate framework for actual implementation into code makes it difficult to keep track of the moving parts, especially in the nascent stage. I have written the following in the hopes that it will help me, and others, make sense of the moving parts, in at least Hartree Fock theory, and use that foundation to then really understand what happens in computational chemistry software packages. The following is adapted and borrows heavily from a course taught by Prof. Stefan Stoll at the University of Washington (http://depts.washington.edu/chem/people/faculty/stoll.html). My code is written in Python and includes extra elements that should help ease the learning curve for using and implementing code into PySCF [1].

The general format will be basic theory for Hartree Fock and building the Self Consistent Field (SCF) cycle. Then each section thereafter will be dedicated to building all of the matrices which make up Hartree Fock theory and will be used in the SCF cycle. At the end, I will also include appendices for more general coverage of material such as recursion and sections which cover some of the Python modules/language used in the code.

In addition to this write-up, I will make my code accessible, so that interested parties may focus on modules that they're interested in and import others. One of the appendices will also cover the general format of the code and how to format your own code to integrate seamlessly into the existing structure. If you wish to implement your own code completely from scratch, I will provide everything necessary to do so as well.

I wish you the best in your efforts in electronic structure and I hope the following will be useful to you as well.

II. COMPUTATIONAL CHEMISTRY

Just a quick guiding word on computational chemistry. There are many approximations that are made in theoretical and computational chemistry and it's important to distinguish approximations made for either or the other. For example, Hartree Fock is a theoretical approximation, which uses only a single Slater determinant rather than an infinite sum. Using a finite basis set is a computational approximation, which is used to keep computing time finite. The distinction becomes important, especially when one is saying something about post-Hartree Fock methods such as Coupled Cluster or Configuration Interaction, because they are formally exact for Hydrogen for

* langemf@uchicago.edu

CCSD or CISD, and thus suffer only from computational error.

Overall computational chemistry and theoretically chemistry are tied closely together, but the intention of each is slightly different; whereas theoretical chemistry attempts to find new approximations to exact (or other approximate methods), computational chemistry deals with the actual code and algorithms that reduce the computational complexity. In addition to the above reason for distinguishing the two, it also helps immensely when thinking about why certain approximations are made and keeping some of the moving parts in order. The main approximations to distinguish for what we are building is that Hartree Fock is an approximation to the exact solution of the Schrödinger equation, and that we are making the computational approximation of using a finite basis set. That said, using a finite basis set itself does not make Hartree Fock any less formally exact, it is just a numerical approximation.

III. HARTREE FOCK THEORY

Hartree Fock theory is an attempt to solve the Schrödinger equation with a single Slater determinant. Hartree Fock is theoretical method, and it wasn't until Roothaan provided his idea of basis sets, that Hartree Fock theory could be written in equations that could be implemented on a computer. I will not derive the Hartree Fock equations, I will simply use the result and show how one can determine the Roothaan equations. These equations are called the Roothaan-Hall equations and may look familiar in matrix form,

$$\boldsymbol{FC} = \boldsymbol{SC}\boldsymbol{\epsilon}. \tag{1}$$

The central players are the Fock matrix, $\boldsymbol{F}$, the coefficient matrix, $\boldsymbol{C}$, the overlap matrix, $\boldsymbol{S}$, and the eigenvalue or orbital energy matrix (vector), $\boldsymbol{\epsilon}$. As one can see, this equation is an eigenvalue equation that can't be readily solved for $\boldsymbol{C}$, instead requiring an iterative procedure to converge to a correct $\boldsymbol{C}$. This method is usually referred to as a Self-Consistent Field (SCF), which is covered in the next section.

Hartree Fock is, in my opinion, a satisfying theory in that it is intuitively and physically grounded and relatively simple to implement as a starting point in computational chemistry. Once you have a good grasp of Hartree Fock theory, it'll be much easier to transfer the same ideas to post Hartree Fock methods, such as Configuration Interaction (CI), Multi-Body Perturbation Theory (MBPT) or Coupled Cluster (CC). While going through the following, I would recommend also reading Modern Quantum Chemistry by Szabo and Ostlund[2] to make sure you also follow the theory, since this is mainly focused on the computational implementation.

IV. SELF-CONSISTENT FIELD

Hartree Fock is eventually solved using a Self-Consistent Field algorithm. The idea is that Hartree-Fock is a mean-field theory (electrons see all other electrons as an average cloud), and so if we find the correct field (density matrix), then we can find the correct coefficient matrix to solve the Roothaan-Hall equations. Doing this isn't too difficult, but has two main parts, building up the matrices that make up the Fock matrix and the overlap matrix, and then performing an iterative procedure to converge to the correct density matrix. How to build the correct matrices is covered in detail below, here I would like to summarize the important points for an SCF algorithm (see

also Szabo pg. 146) and then give a guide for implementing your own.

1. Specify molecule with nuclear coordinates, $\{\boldsymbol{R_C}\}$, atomic numbers, Z_C, number of electrons, N, and a basis set, $\{\phi_\mu\}$.
2. Calculate all required matrices, $\boldsymbol{S}$, $\boldsymbol{T}$, $\boldsymbol{H_{core}}$, $\boldsymbol{V_{ne}}$, $\boldsymbol{ERIS}$ (see below).
3. Obtain a guess at the density matrix, $\boldsymbol{P}$
4. Calculate $\boldsymbol{V_{ee}}$ and add it to $\boldsymbol{H_{core}}$ to make $\boldsymbol{F}$.
5. Obtain the coefficient matrix, $\boldsymbol{C}$ by solving Eq. (1).
6. Generate a new $\boldsymbol{P}$ from the new $\boldsymbol{C}$
7. Determine convergence, if not converged, repeat starting at 4.
8. Once converged, one can use the different matrices, now including $\boldsymbol{P}$ and $\boldsymbol{C}$ to calculate desired properties, such as energy.

I changed the algorithm slightly from the one presented in Szabo and Ostlund, because there are simpler ways to diagonalizing the Fock matrix than the way presented there. That said, it is important to remember that we are mainly interested in an equation that looks something like,

$$\boldsymbol{F'C'} = \boldsymbol{C'\epsilon}, \tag{2}$$

where the primes indicate that these matrices have been rotated by a unitary matrix derived from the overlap matrix. For details on how to perform the algorithm as written in Szabo (pg. 144-146). This will simply be hidden in the code by using the scipy module in Python.

V. GENERAL QUANTITIES

Here are some quantities that will be used throughout the implementation. Many of them have to do with converting a two point problem into a center of mass problem and combining Gaussians.The first is the normalization constant for a given orbital,

$$N_k = \left(\frac{2}{\pi}\right)^{3/4} \frac{2^{a_x+a_y+a_z}\alpha_k^{(2(a_x+a_y+a_z)+3)/4}}{\sqrt{(2a_x-1)!!(2a_y-1)!!(2a_x-1)!!}} \tag{3}$$

where a_w is the angular momentum in dimension w, α_k is the exponent of the kth PGO, and $(x)!!$ is a double factorial, given by,

$$n!! = \begin{cases} n\cdot(n-2)\cdots 1 & \text{if } n>0 \text{ odd} \\ n\cdot(n-2)\cdots 2 & \text{if } n>0 \text{ even} \\ 1 & \text{if } n=-1,0 \end{cases}. \tag{4}$$

And finally, the product of two Gaussian (which will happen every time we find any matrix elements) is given by,

$$e^{-\alpha|\boldsymbol{r}-\boldsymbol{A}|^2}e^{-\beta|\boldsymbol{r}-\boldsymbol{B}|^2} = K_{AB}e^{-\alpha|\boldsymbol{r}-\boldsymbol{P}|^2} \tag{5}$$

where,

$$K_{AB} = e^{-\alpha\beta/(\alpha+\beta)|\boldsymbol{A}-\boldsymbol{B}|^2}, \tag{6}$$

$$p = \alpha + \beta, \tag{7}$$

and

$$\boldsymbol{P} = \frac{\alpha\boldsymbol{A} + \beta\boldsymbol{B}}{\alpha + \beta}. \tag{8}$$

$\boldsymbol{P}$ is the weighted midpoint of $\boldsymbol{A}$ and $\boldsymbol{B}$. The ease of multiplying two Gaussians and also integrating them is why we choose to use Guassian-type orbitals, rather than Slater orbitals, or similarly difficult to integrate functions.

In this section I would also like to include the equations for the density matrix, $\boldsymbol{P}$ and the electron energy that will be used in the SCF procedure above. For the density matrix, I would use the following form,

$$P_{\mu\nu} = 2\sum_{a}^{N/2} C_{\mu a} C_{\nu a}^{*}, \tag{9}$$

where the sum is over the electron count divided by two (Restricted Hartree Fock) and the C's refer to the coefficient matrix. For the first iteration, I would recommend using a density matrix derived from the non-interaction Fock matrix, $\boldsymbol{F}_{non-int} = \boldsymbol{H_{core}}$. It would also be good to define the core Hamiltonian by,

$$\boldsymbol{H_{core}} = \boldsymbol{T} + \boldsymbol{V_{ne}}. \tag{10}$$

In order to then determine the interacting Fock matrix, we need to include $\boldsymbol{V_{ee}}$, which is built using the ERIS,

$$V_{ee,\mu\nu} = \sum_{\kappa\lambda} P_{\kappa\lambda}\Big[(\mu\nu|\lambda\kappa) - \frac{1}{2}(\mu\kappa|\lambda\nu)\Big], \tag{11}$$

where we are summing over atomic orbitals. With this matrix, we can now calculate the interacting Fock matrix (Step 4. above). And finally, all of these matrices will eventually allow us to calculate the electronic energy of our molecule via,

$$E_0 = \frac{1}{2}\sum_{\mu\nu} P_{\mu\nu}(2H_{core,\mu\nu} + F_{\mu\nu}). \tag{12}$$

The total energy is then determined by adding the electronic energy to the nuclear-nuclear energy,

$$V_{nn} = \sum_{A}\sum_{B>A} \frac{Z_A Z_B}{\boldsymbol{r}_A - \boldsymbol{r}_B}, \tag{13}$$

with a total energy,

$$E_{tot} = E_0 + V_{nn}. \tag{14}$$

So to start with, complete the next few sections to build all of the necessary matrices from scratch, then use these to seed the SCF procedure above to determine $\boldsymbol{P}$ and $\boldsymbol{F}$, which will finally allow you to calculate the electronic and total energies of the molecule. This might seem like a lot, but taking this one section at a time will hopefully make it easier to tackle and also build up the picture as you go.

VI. OVERLAP MATRIX

The overlap matrix contains matrix elements which describe the overlap of the basis functions. It is important to note that this is not the same as the density matrix, the overlap matrix will remain constant throughout the SCF cycle. The overlap matrix elements $S_{\mu\nu}$ are integrals written as a linear combination of integrals over primitives (PGOs),

$$S_{\mu\nu} = \langle\mu|\nu\rangle = \sum_{k\,on\,\mu}\sum_{l\,on\,\mu} d_{\mu k} d_{\nu l} N_k N_l [k\boldsymbol{a}|l,\boldsymbol{b}], \tag{15}$$

where μ, ν are basis functions, k, l are primitives, $d_{\mu k}$ are primitive coefficients, N_k are normalization constants, and $[k\boldsymbol{a}|l,\boldsymbol{b}]$ and the vectors $\boldsymbol{a}$ are the angular momentum vectors for the given primitive. The 3D primitive integrals given by,

$$[k\boldsymbol{a}|l,\boldsymbol{b}] = \left(\frac{\pi}{p}\right)^{3/2} K_{AB} I_x^S(a_x,b_x) I_y^S(a_y,b_y) I_z^S(a_z,b_z), \tag{16}$$

where K_{AB} is a pre-factor based on the primitives, p is the sum of primitive exponents, and $I_w^S(a_w,b_w)$ are 1D primitive integrals. The 1D integrals can be obtained analytically using,

$$I_w^S(a_w,b_w) = \sum_{i=0}^{(a_w+b_w)/2} f_{2i}(a_w,b_w,P_w,A_w,B_w)\frac{(2i-1)!!}{(2p)^i}, \qquad w = (x,y,z) \tag{17}$$

where the w subscript refers to one of the axes being used, $\boldsymbol{A}$ and $\boldsymbol{B}$ are primitive coordinates, $\boldsymbol{P}$ is the weighted midpoint, $(x)!!$ is a double factorial, and f_κ is given by,

$$f_\kappa(a_w,b_w,P_w,A_w,B_w) = \sum_{j=max(0,\kappa-a_w)}^{min(\kappa,b_w)} \binom{a_w}{\kappa-j}\binom{b_w}{j}(P_w-A_w)^{a_w-\kappa+j}(P_w-B_w)^{b_w-j}. \tag{18}$$

The double factorial is defined as above in Eq. (4).

The overlap matrix is symmetric, $S_{\mu\nu} = S_{\nu\mu}$, and all diagonal elements are equal to 1 with off-diagonal elements, $|S_{\mu\nu}| < 1$. To improve the speed of your code, you only need to calculate the lower or upper triangle, add the transpose and identity matrix.

VII. KINETIC ENERGY MATRIX

The kinetic matrix is one of the terms contributing to the non-interacting portion of the Hamiltonian, H_{core}. The kinetic energy matrix elements are also written as a linear combination of primitives, although the integrals in this case get more complex, since we have the $-\frac{1}{2}\bigtriangledown^2$ operator. The elements can be written as,

$$T_{\mu\nu} = \langle\mu| -\frac{1}{2}\bigtriangledown^2 |\nu\rangle = \sum_{k\,on\,\mu}\sum_{l\,on\,\nu} d_{\mu k} d_{\nu l} N_k N_l [k,\boldsymbol{a}| -\frac{1}{2}\bigtriangledown^2 |l,\boldsymbol{b}] \tag{19}$$

where most of the terms are the same as in the overlap integral, except the 3D kinetic energy integral is given by,

$$[k,\boldsymbol{a}| -\frac{1}{2}\bigtriangledown^2 |l,\boldsymbol{b}] = I_x^T + I_y^T + I_z^T. \tag{20}$$

The 1D integrals can be written as a sum over 3D overlap integrals with changed angular momenta,

$$I_w^T = \beta(2b_w + 1)[k, \boldsymbol{a}|l, \boldsymbol{b}] - 2\beta^2[k, \boldsymbol{a}|l, \boldsymbol{b} - 2_w] - \frac{1}{2}b_w(b_w - 1)[k, \boldsymbol{a}|l, \boldsymbol{b} - 2_w]. \tag{21}$$

The above equation calls the 3D overlap integrals in Eq. (16). It is also important to point out that $\boldsymbol{b} - 2_w$ is taking the angular momentum vector $\boldsymbol{b}$ and subtracting unit 2 from the wth dimension, so $\boldsymbol{b} = [0, 0, 2] - 2_z = [0, 0, 0]$. The kinetic energy matrix is also symmetric, $T_{\mu\nu} = T_{\nu\mu}$, so to reduce computing time, you only need to compute the upper diagonal and diagonal.

VIII. ELECTRON-NUCLEAR ATTRACTION MATRIX ELEMENTS

The Electron-Nuclear Attraction matrix is the second half of the non-interacting portion of the Hamiltonian, H_{core}. This part of the Hamiltonian takes into account the attraction that (negative) electrons experience due to the presence of the (positive) nuclei. It is important to remember that electrons in molecules feel this force from all nuclei, thus we sum over all nuclei. The highest level expression for the matrix elements is simple,

$$V_{ne,\mu\nu} = -\sum_C Z_C \left\langle \mu | r_C^{-1} | \nu \right\rangle, \tag{22}$$

where the sum enumerates of atoms, C, and Z_C gives the atomic charge. The integral equation looks simple as well, summing over primitives,

$$\langle \mu | r_C^{-1} | \nu \rangle = \sum_{k\,on\,\mu} \sum_{l\,on\,\nu} d_{\mu k} d_{\nu l} N_k N_l [k, \boldsymbol{a} | r_C^{-1} | l, \boldsymbol{b}] \tag{23}$$

where the integral is now over primitives. It is common to write the integral, $[k, \boldsymbol{a}|r_C^{-1}|l, \boldsymbol{b}]$, as $[\boldsymbol{a}|r_C^{-1}|\boldsymbol{b}]$, for notational brevity. However, these integrals are more complex than the previous ones, requiring recursive relations in order to express integrals of non-zero angular momentum in terms of integrals over zero angular momentum terms. We also need to introduce auxiliary integrals as presented in the paper by Obara and Saika[3]. The vertical recursion relations (VRR) displace magnitude in the angular momentum to magnitude in the auxiliary variable, m. Combining the recursion relations and auxiliary integrals, we will finally be left with an integral over only s-type orbitals (zero angular momentum),

$$[\boldsymbol{0}|r_C^{-1}|\boldsymbol{0}]^{(m)} = \frac{2\pi}{p} K_{AB} F_m(T), \tag{24}$$

where K_{AB} and p are the same as previously, T is given by,

$$T = p|\boldsymbol{P} - \boldsymbol{C}|^2, \tag{25}$$

and $F_m(T)$ is the Boys function,

$$F_m(T) = \int_0^1 t^{2m} e^{-Tt^2} dt. \tag{26}$$

The integral in the Boys function can be solved using Gamma functions,

$$F_m(T) = \frac{\gamma(m + \frac{1}{2}, T)}{2T^{m+1/2}} = \frac{\Gamma(m + \frac{1}{2}) - \Gamma(m + \frac{1}{2}, T)}{2T^{m+1/2}}, \tag{27}$$

where the three different gamma functions are given by,

$$\Gamma(a) = \int_0^\infty t^{a-1}e^{-t}dt \qquad \gamma(a,z) = \int_0^z t^{a-1}e^{-t}dt \qquad \Gamma(a,z) = \int_z^\infty t^{a-1}e^{-t}dt. \tag{28}$$

It is important to separately also implement the following limit,

$$\lim_{T\to 0} F_m(T) = \frac{1}{2m+1}, \tag{29}$$

by making sure you have an if statement in the code to catch anything below a certain cutoff magnitude (ie. 10^{-12}) to make sure you have no errors. If the orbitals have non-zero angular momentum, we must use the following two vertical recursion relations (VRR) to reduce the integral down to the above zero angular momentum form. For the two primitives involved, we have,

$$\begin{aligned}[\boldsymbol{a}|r_C^{-1}|\boldsymbol{b}]^{(m)} = (\boldsymbol{P}-\boldsymbol{A})_w[\boldsymbol{a}-1_w|r_C^{-1}|\boldsymbol{b}]^{(m)} + (\boldsymbol{C}-\boldsymbol{P})_w[\boldsymbol{a}-1_w|r_C^{-1}|\boldsymbol{b}]^{(m+1)} \\ +\frac{a_w-1}{2p}\Big([\boldsymbol{a}-2_w|r_C^{-1}|\boldsymbol{b}]^{(m)} - [\boldsymbol{a}-2_w|r_C^{-1}|\boldsymbol{b}]^{(m+1)}\Big) \\ +\frac{b_w}{2p}\Big([\boldsymbol{a}-1_w|r_C^{-1}|\boldsymbol{b}-1_w]^{(m)} - [\boldsymbol{a}-1_w|r_C^{-1}|\boldsymbol{b}-1_w]^{(m+1)}\Big)\end{aligned} \tag{30}$$

and,

$$\begin{aligned}[\boldsymbol{a}|r_C^{-1}|\boldsymbol{b}]^{(m)} = (\boldsymbol{P}-\boldsymbol{B})_w[\boldsymbol{a}|r_C^{-1}|\boldsymbol{b}-1_w]^{(m)} + (\boldsymbol{C}-\boldsymbol{P})_w[\boldsymbol{a}|r_C^{-1}|\boldsymbol{b}-1_w]^{(m+1)} \\ +\frac{b_w-1}{2p}\Big([\boldsymbol{a}|r_C^{-1}|\boldsymbol{b}-2_w]^{(m)} - [\boldsymbol{a}|r_C^{-1}|\boldsymbol{b}-2_w]^{(m+1)}\Big) \\ +\frac{a_w}{2p}\Big([\boldsymbol{a}-1_w|r_C^{-1}|\boldsymbol{b}-1_w]^{(m)} - [\boldsymbol{a}-1_w|r_C^{-1}|\boldsymbol{b}-1_w]^{(m+1)}\Big).\end{aligned} \tag{31}$$

The first relation reduces the angular momentum on $\boldsymbol{a}$ and the second on $\boldsymbol{b}$. $\boldsymbol{P}$ is again the midpoint of $\boldsymbol{A}$ and $\boldsymbol{B}$, while $\boldsymbol{C}$ is the position of the nucleus and w again refers to a specific coordinate, x, y, or z.

When implementing the recursion relations into code, I would recommend having an if statement for the zero angular momentum case, and elif statements for the other two relations for each Cartesian dimension. The electron nuclear matrix is also symmetric ($V_{\mu\nu} = V_{\nu\mu}$), so use the same trick as for the kinetic energy matrix.

IX. ELECTRON-ELECTRON REPULSION INTEGRALS

The electron-electron repulsion integrals (ERIS) are similar to the electron-nuclear attraction integrals, except they are a four-point integral now and require not only vertical recursion relations (VRRs), but also horizontal recursion relations (HRRs). They are usually written as, $(\mu\nu|\kappa\lambda)$ and will be used to build the electron electron potential $\boldsymbol{V_{ee}}$, which is used to build the Fock matrix, $\boldsymbol{F}$ Expressing the 4-point integrals into two separate recursion relations with three points was developed by Head-Gordon and Pople[4] and was based on the recursion relations above by Obara and Saika. If you'd like to learn more about the details of how a four-point integral can be condensed using auxiliary functions, please read the original papers, they are very interesting.

The end point for the ERIS is when all orbitals have been reduced to s-type orbitals (zero angular momentum), which can be written in a way similar to above, but slightly more complicated,

$$[\boldsymbol{00}|\boldsymbol{00}]^{(m)} = \frac{2\pi^{5/2}}{pq\sqrt{p+q}}K_{AB}K_{CD}F_m(T), \tag{32}$$

where

$$T = \frac{pq}{p+q}|\boldsymbol{P}-\boldsymbol{Q}|^2, \tag{33}$$

and K_{AB}, K_{CD} are given in the General Quantities section (V) with A,B and C,D centers, and $F_m(T)$ is again the Boys function. To get to all s-type orbitals, we'll need to use VRRs to get from $[\boldsymbol{a0}|\boldsymbol{c0}]^{(m)}$ down to $[\boldsymbol{00}|\boldsymbol{00}]^{(m)}$. There are two VRRs that you'll use,

$$\begin{aligned}[\boldsymbol{a0}|\boldsymbol{c0}]^{(m)} = (\boldsymbol{P}-\boldsymbol{A})_w[(\boldsymbol{a}-1_w)\boldsymbol{0}|\boldsymbol{c0}]^{(m)} + (\boldsymbol{W}-\boldsymbol{P})_w[(\boldsymbol{a}-1_w)\boldsymbol{0}|\boldsymbol{c0}]^{(m+1)} \\ +\frac{a_w-1}{2p}\Big([(\boldsymbol{a}-2_w)\boldsymbol{0}|\boldsymbol{c0}]^{(m)} - \frac{q}{p+q}[(\boldsymbol{a}-2_w)\boldsymbol{0}|\boldsymbol{c0}]^{(m+1)}\Big) \\ +\frac{c_w}{2(p+q)}[(\boldsymbol{a}-1_w)\boldsymbol{0}|(\boldsymbol{c}-1_w)\boldsymbol{0}]^{(m+1)},\end{aligned} \tag{34}$$

and

$$\begin{aligned}[\boldsymbol{a0}|\boldsymbol{c0}]^{(m)} = (\boldsymbol{Q}-\boldsymbol{C})_w[\boldsymbol{a0}|(\boldsymbol{c}-1_w)\boldsymbol{0}]^{(m)} + (\boldsymbol{W}-\boldsymbol{Q})_w[\boldsymbol{a0}|(\boldsymbol{c}-1_w)\boldsymbol{0}]^{(m+1)} \\ +\frac{c_w-1}{2q}\Big([\boldsymbol{a0}|(\boldsymbol{c}-2_w)\boldsymbol{0}]^{(m)} - \frac{p}{p+q}[\boldsymbol{a0}|(\boldsymbol{c}-2_w)\boldsymbol{0}]^{(m+1)}\Big) \\ +\frac{a_w}{2(p+q)}[(\boldsymbol{a}-1_w)\boldsymbol{0}|(\boldsymbol{c}-1_w)\boldsymbol{0}]^{(m+1)}.\end{aligned} \tag{35}$$

And to get from $[\boldsymbol{ab}|\boldsymbol{cd}]^{(m)}$ to $[\boldsymbol{a0}|\boldsymbol{c0}]^{(m)}$ you'll need two HRRs, which move angular momentum from $\boldsymbol{b}$ and $\boldsymbol{d}$ to $\boldsymbol{a}$ and $\boldsymbol{c}$. The forms are,

$$[\boldsymbol{ab}|\boldsymbol{cd}]^{(m)} = [(\boldsymbol{a}+1_w)(\boldsymbol{b}-1_w)|\boldsymbol{cd}]^{(m)} + (\boldsymbol{A}-\boldsymbol{B})_w[\boldsymbol{a}(\boldsymbol{b}-1_w)|\boldsymbol{cd}]^{(m)}, \tag{36}$$

and

$$[\boldsymbol{ab}|\boldsymbol{cd}]^{(m)} = [\boldsymbol{ab}|(\boldsymbol{c}+1_w)(\boldsymbol{d}-1_w)]^{(m)} + (\boldsymbol{C}-\boldsymbol{D})_w[\boldsymbol{ab}|\boldsymbol{c}(\boldsymbol{d}-1_w)]^{(m)}. \tag{37}$$

Again, I would recommend implementing the recursion relations in the order presented, so an if statement for the endpoint, then the VRRs, and finally the HRRs. This might seem backwards, but recursion relations unravel in reverse order, so we want to write our statements in opposite order. See (X) below for more on recursion.
Once you have the overlap matrix (S), kinetic energy matrix (T), electron-nuclear attraction matrix (Vne) and the ERIS correct, you'll have all of the pieces for writing the SCF procedure. If you're choosing to skip making these matrices, you can just use the `integrals` module that I wrote.

X. APPENDIX: RECURSION

A. Fibonacci

The concept of recursion is relatively simple, it is a function which calls upon itself over and over again until it hits some desired endpoint. Implementing recursion relations can be difficult, so below I try to walk through a simple example of the Fibonacci sequence and then also demonstrate some pseudo-code for the integral case.

The Fibonacci sequence is readily recognizable when written as: 0,1,1,2,3,5,8,13,21,...
This can be re-written in the form of a recursion relation,

$$F(n) = F(n-1) + F(n-2), \tag{38}$$

which states that any Fibonacci number is a sum of the previous two. However, if we simply left the definition as is, then there would be no termination point and F(n-2) may call F(-1), which we know shouldn't happen. To make sure the Fibonacci sequence is well-defined, we define two endpoints (or starting points numerically),

$$F(0) = 0 \qquad , \qquad F(1) = 1 \tag{39}$$

Now when at some point we make the call, F(2) = F(1) + F(0), we can actually calculate the end values.
How would this look in terms of code?

```
def fibonacci(n):
   if n == 0:
      return 0
    elif n == 1:
      return 1
    else:
      return fibonacci(n-1) + fibonacci(n-2)
```

Here we can see that to determine the nth number, we are making calls to the function 'fibonacci' within itself with lower values of n, until we hit 0 or 1, in which case we return 0 or 1 and then unravel the recursion upwards. This effectively creates a tree structure in which the final value is the first node and everything is calculated from the youngest generation to the starting node.

B. Integrals

The integral relations above are also recursive, although much more complicated than the simple Fibonacci code above. However, if we break down what is going on, it might be easier to see the recursion and write something which successfully computes what we'd like. Please only read the following when you are truly stuck on this part, as part of this project is for you to figure some of the coding things out yourself.
The general schematic for the integral recursion could be written as,

```
def integral(a, b):
   if ENDPOINT:
      return ENDVALUE
    elif not ENDPOINT:
      return integral(a, b-1) + integral(a-1, b)
```

Hopefully the general structure is clear now, the important pieces are finding the ENDPOINT conditions, the ENDVALUE, and making sure that you are unraveling the recursion in the correct order.

The following are a few tips that you should only look at when you're truly stuck or done and you want to see how I thought about it:

Here is probably the best way to call the ENDPOINT and ENDVALUE

```
def eris():
    if a = [0,0,0] and b = [0,0,0] and c = [0,0,0] and d = [0,0,0]:
      T = p*q/(p+q)*P.distance(Q)**2
      return 2*np.pi**(5/2)/(p*q*np.sqrt(p+q))*KAB*KCD*Fm(T)
    # Make sure that you have no negative angular momentum
   elif any(n < 0 for n in a) or any(n < 0 for n in b) or any(n < 0 for n in c) or any(n
       < 0 for n in d):
      return 0
```

I ended up splitting the eris into five functions. Then I have one more parent function which calls prim_eris().

```
ENDPOINT = a = [0,0,0] and b = [0,0,0] and c = [0,0,0] and d = [0,0,0]
ZEROS = any(n < 0 for n in a) or any(n < 0 for n in b) or any(n < 0 for n in c) or any(n
    < 0 for n in d)

def prim_eris():
   if ENDPOINT:
      return ENDVALUE
   elif ZEROS:
      return 0
      else:
      return prim_eris_hrr()

def prim_eris_hrr():
   if ENDPOINT:
      return ENDVALUE
    elif b = [0,0,0] and d = [0,0,0]:
      return prim_eris_vrr()
    elif ZEROS:
      return 0
    elif b[0] != 0:
      return eris_hrr_single()
    elif b[1] != 0:
      return eris_hrr_single()
    ...
    elif d[2] != 0:
      return eris_hrr_single()

def eris_hrr_single():
   return prim_eris_hrr()

def prim_eris_vrr():
   if ENDPOINT:
      return ENDVALUE
    elif ZEROS:
      return 0
    elif a[0] != 0:
      return eris_vrr_single()
    ...
    elif c[2] != 0:
      return eris_vrr_single()
```

```
def eris_vrr_single():
   return prim_eris_vrr()
```

XI. APPENDIX: BASIS SETS

One of the most important considerations in computational chemistry is the basis set, which is the finite set that you will be creating your atomic and molecular orbitals from. A quick aside, basis sets are best downloaded from the EMSL database (https://bse.pnl.gov/bse/portal). In this appendix, I'll present an overview of the structure of basis set files and how I implemented my parser and `Orbital` object:
If we take a look at STO-3G, the first few lines (excluding comments) will be,

```
#BASIS SET: (3s) -> [1s]
H    S
      3.42525091           0.15432897
      0.62391373           0.53532814
      0.16885540           0.44463454
#BASIS SET: (3s) -> [1s]
He    S
      6.36242139           0.15432897
      1.15892300           0.53532814
      0.31364979           0.44463454
#BASIS SET: (6s,3p) -> [2s,1p]
Li    S
     16.1195750            0.15432897
      2.9362007            0.53532814
      0.7946505            0.44463454
Li    SP
      0.6362897           -0.09996723          0.15591627
      0.1478601            0.39951283          0.60768372
      0.0480887            0.70011547          0.39195739
```

Dissecting this, the commented line, #BASIS SET... gives the number of primitives that make up a given number of atomic orbitals, so for Li we have a total of 6 s-type primitives and 3 p-type primitives which make up 2 s-type orbitals and 1 p-type orbital. The second set of lines containing alpha strings give the element and the resulting atomic orbital, so for example, Li and the SP orbitals in the last case. The actual numbers are the coefficients ($d_{\mu k}$ and α in the equations above) and exponents for the orbital Gaussians, the exponents being in the first column and the coefficients in the second (and third) columns. Let's look at both the case of a hydrogen atom s-type orbital and a more complicated lithium p-type orbital.
General Primitive Gaussian form:

$$\psi_{k,\boldsymbol{A},\boldsymbol{a}}(\boldsymbol{r}) = (x - A_x)^{a_x}(y - A_y)^{a_y}(z - A_z)^{a_z} e^{-\alpha_k |\boldsymbol{r}-\boldsymbol{A}|^2} \tag{40}$$

Where k denotes the kth primitive, $\boldsymbol{A}$ are its coordinates, $\boldsymbol{a}$ is its angular momentum vector, and α_k is the exponent. An atomic orbital is simply a linear combination of these primitive Gaussians, giving rise to the form,

$$\chi_{\mu,\boldsymbol{A},\boldsymbol{a}}(\boldsymbol{r}) = \sum_k d_{\mu k} N_k \psi_{k,\boldsymbol{A},\boldsymbol{a}}(\boldsymbol{r}) \tag{41}$$

where $d_{\mu k}$ are the coefficients and the sum is over primitives.
Hydrogen, s-type orbital:
Coefficients: [0.15432897, 0.53532814, 0.44463454]
Exponents: [3.42525091, 0.62391373, 0.16885540]
Angular momentum: [0,0,0]
Lithium, sp-orbital, s-type:
Coefficients: [-0.09996723, 0.39951283, 0.70011547]
Exponents: [0.6362897, 0.1478601, 0.0480887]
Angular Momentum: [0,0,0]
Lithium, sp-orbital, p-type:
Coefficients: [0.15591627, 0.60768372, 0.39195739]
Exponents: [0.6362897, 0.1478601, 0.0480887]
Angular Momentum: [1,0,0], [0,1,0], [0,0,1]
Where for the sp-orbital, p-type we have three different angular momenta for the three possible dimensions (s-type are 0 in all dimensions).
The Object hierarchy that I implemented is the following:
`Molecule` → `Atom` → `Orbital` → `PGO`,
where `PGO` contains the coefficients and exponents that are found in the basis set file. Please see the code for all of the class variables and functions.

---

[1] Qiming Sun, Timothy C. Berkelbach, Nick S. Blunt, George H. Booth, Sheng Guo, Zhendong Li, Junzi Liu, James D. McClain, Elvira R. Sayfutyarova, Sandeep Sharma, Sebastian Wouters, and Garnet Kin-Lic Chan, "PySCF: the Python-based simulations of chemistry framework," Wiley Interdisciplinary Reviews: Computational Molecular Science , e1340 (2017).

[2] Attila Szabo and Neil Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*, 2nd ed. (Dover, 1996).

[3] Shigeru Obara and A. Saika, "Efficient recursive computation of molecular integrals over Cartesian Gaussian functions," The Journal of Chemical Physics **84**, 3963 (1986).

[4] Martin Head-Gordon and John A. Pople, "A method for two-electron Gaussian integral and integral derivative evaluation using recurrence relations," The Journal of Chemical Physics **89**, 5777–5786 (1988).