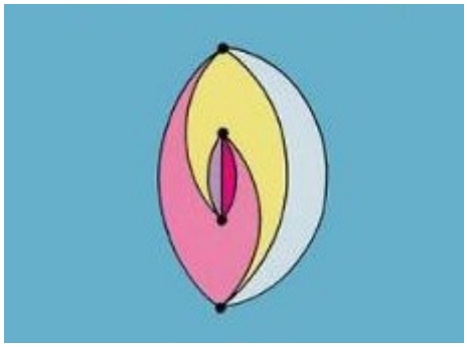
Practical Introduction to Hartree-Fock Algorithm using Python

Laksh
May 7, 2019 · 10 min read ★



We will write a Hartree-Fock algorithm completely from scratch in Python and use it to find the (almost) exact energy of simple diatomic molecules like H_2

Prerequisites

I will assume you have read the first three chapters of "Modern Quantum Chemistry" by Szabo and Ostlund, or any other similar book, or have taken an introductory course into computational quantum chemistry. I will be referring to said book throughout the post. The book is very cheap (£10 on Amazon) and is a good investment.

I'll also assume you have had a bit of practice coding in python, and know the basics, like how for loops work, etc.

I will go through the important maths again here and there. This is to function as a reminder, and it will not make sense if you are reading about this for the first time. If this is the case, simply refer to the book.

In many lines of the code, I will put a page reference to Szabo & Ostlund (I will just refer to the page number mostly).

Advice for following this tutorial

I suggest you have a jupyter notebook open, and simply code along. If you don't understand a line of code, test it in another line to pick apart its function.

If you get stuck/some code doesn't work

In this case, I recommend going to my GitHub and downloading the Jupyter Notebook for this. Then, replace the coding giving an error with the code from the notebook.

https://github.com/aced125/Hartree_Fock_jupyter_notebook

Summary of the practical steps

On pp146 of Szabo & Ostlund (I will stop referring to the name of the book from here on), we will find a summary of the key steps in Hartree-Fock, and go through these.

Imports

We will need some imports for this. Go ahead and install these if you haven't already.

```
C:\Users\laksh>pip install numpy scipy
```

Installing packages in the command line

```
In [1]: # Imports

        import numpy as np
        from numpy import *
        import scipy
        from scipy.special import erf
```

Imports required

1a) Specify a molecule (a set of nuclear coordinates, atomic numbers, and a number of electrons).

.XYZ files are a common file type to store data about chemical structure. As an example, take the .XYZ file of pyridine.

Example [edit]

The pyridine molecule can be described in the XYZ format by the following:

```
11

C       -0.180226841      0.360945118     -1.120304970
C       -0.180226841      1.559292118     -0.407860970
C       -0.180226841      1.503191118      0.986935030
N       -0.180226841      0.360945118      1.685965030
C       -0.180226841     -0.781300882      0.986935030
C       -0.180226841     -0.837401882     -0.407860970
H       -0.180226841      0.360945118     -2.206546970
H       -0.180226841      2.517950118     -0.917077970
H       -0.180226841      2.421289118      1.572099030
H       -0.180226841     -1.699398882      1.572099030
H       -0.180226841     -1.796059882     -0.917077970
```

.xyz file of pyridine

Let's write a function to read in this type of file.

In [14]:
```
def xyz_reader(file_name):
    # Reads an xyz file (https://en.wikipedia.org/wiki/XYZ_file_format) and returns the number of atoms,
    # atom types and atom coordinates.

    file = open(file_name,'r')

    number_of_atoms = 0
    atom_type = []
    atom_coordinates = []

    for idx,line in enumerate(file):
        # Get number of atoms
        if idx == 0:
            try:
                number_of_atoms = line.split()[0]
            except:
                print("xyz file not in correct format. Make sure the format follows: https://en.wikipedia.org/wiki/XYZ_file_format

        # Skip the comment/blank line
        if idx == 1:
            continue

        # Get atom types and positions
        if idx != 0:
            split = line.split()
            atom = split[0]
            coordinates = [float(split[1]),
                           float(split[2]),
                           float(split[3])]

            atom_type.append(atom)
            atom_coordinates.append(coordinates)

    file.close()

    return number_of_atoms, atom_type, atom_coordinates
```

We will now read in the XYZ file for HeH:

```
2
```

```
He      0       0       0
H       0       0       1.4632
```

XYZ file for HeH

You can simply make this file in any text editor (remember to name the file HeH.xyz). Note, we are using the experimentally determined bond length, in atomic units.

```
In [15]: file_name = "D:\\blog_post_images\\hartree-fock_tutorial\\HeH.xyz"

In [16]: number_of_atoms, atom_type, atom_coordinates = xyz_reader(file_name)

In [17]: atom_type
Out[17]: ['He', 'H']

In [18]: atom_coordinates
Out[18]: [[0.0, 0.0, 0.0], [0.0, 0.0, 1.4632]]
```

We will choose the number of electrons in the system later on. We can set this to whatever we like.

1b) Specify a basis set. Also, set the number of electrons for the system (pp152)

We want to represent our Slater-like orbitals as linear combinations of Gaussian orbitals so that the integrals can be performed easily. For a discussion turn to pp152 of Szabo and Ostlund. In brief, a **Gaussian can be specified by two parameters: its center, and its exponent**. Furthermore, since we are representing slater orbitals as a **sum** of Gaussian orbitals, we need **contraction coefficients**. The exponents and contraction coefficients are optimized by a least-squares fitting procedure. More information here: Hehre, Stewart, Pople, 1969.

The zeta coefficients are the exponents of the Slater orbitals, and they have been optimized by the variational principle. They are in essence an effective nuclear charge of an atom. They have been historically estimated using Slater's rules, which you might come across in an undergraduate Chemistry course.

```
In [41]: # Basis set variables

         # STO-nG (number of gaussians used to form a contracted gaussian orbital - pp153)
         STOnG = 3

         # Dictionary of zeta values (pp159-160, 170)
         zeta_dict = {'H': [1.24], 'He':[2.0925], 'Li':[2.69,0.80],'Be':[3.68,1.15],
                      'B':[4.68,1.50],'C':[5.67,1.72]}   #Put zeta number in list to accomodate for possibly more basis sets (eg 2s orbital
```

```
# Dictionary containing the max quantum number of each atom,
# for a minimal basis STO-nG calculation
max_quantum_number = {'H':1,'He':1,'Li':2,'Be':2,'C':2}


# Gaussian contraction coefficients (pp157)
# Going up to 2s orbital (W. J. Hehre, R. F. Stewart, and J. A. Pople. J. Chem. Phys. 51, 2657 (1969))
# Row represents 1s, 2s etc...
D = np.array([[0.444635, 0.535328, 0.154329],
              [0.700115,0.399513,-0.0999672]])

# Gaussian orbital exponents (pp153)
# Going up to 2s orbital (W. J. Hehre, R. F. Stewart, and J. A. Pople. J. Chem. Phys. 51, 2657 (1969))
alpha = np.array([[0.109818, 0.405771, 2.22766],
                  [0.0751386,0.231031,0.994203]])

# Basis set size
B = 0
for atom in atoms:
    B += max_quantum_number[atom]
```

Variables needed to define the basis set

Some more book-keeping is below. Most importantly, here is where we store the number of electrons. The storage of atom charges is required for calculation of the potential energy (although this is not that important per se since the potential energy just raises the overall energy by a constant value).

In [42]:
```
# Other book-keeping

# Number of electrons (Important!!)
N = 2

# Keep a dictionary of charges
charge_dict = {'H': 1, 'He': 2, 'Li':3, 'Be':4,'B':5,'C':6,'N':7,'O':8,'F':9,'Ne':10}
```

Some more book-keeping

2. Computing all the required integrals in the Gaussian basis

2.1) Writing definitions for integrals between the Gaussian functions

We want to form the Fock matrix in the basis of our atomic orbitals. But our atomic orbitals are a linear sum of Gaussian orbitals. The integrals between individual Gaussian orbitals can be calculated easily and their derivations are given in the back of the book (pp410).

2.2) The product of two Gaussians is a Gaussian (pp410)

This lovely property allows easy calculation of integrals. Let's write a function that takes in two Gaussians and spits out a new Gaussian.

The unnormalized 1s primitive Gaussian at $\mathbf{R}_A$ is

$$\tilde{g}_{1s}(\mathbf{r} - \mathbf{R}_A) = e^{-\alpha|\mathbf{r}-\mathbf{R}_A|^2} \qquad \text{(A.1)}$$

We will use α, β, γ, and δ for orbital exponents of functions centered at $\mathbf{R}_A$

$\mathbf{R}_B$, $\mathbf{R}_C$, $\mathbf{R}_D$, respectively. The reason why Gaussians simplify multicenter integrations is that the product of two 1s Gaussians, each on different centers, is proportional to a 1s Gaussian on a third center. Thus

$$\tilde{g}_{1s}(\mathbf{r} - \mathbf{R}_A)\tilde{g}_{1s}(\mathbf{r} - \mathbf{R}_B) = \tilde{K}\tilde{g}_{1s}(\mathbf{r} - \mathbf{R}_P) \tag{A.2}$$

where the proportionality constant $\tilde{K}$ is

$$\tilde{K} = \exp[-\alpha\beta/(\alpha + \beta)|\mathbf{R}_A - \mathbf{R}_B|^2] \tag{A.3}$$

The third center P is on a line joining the centers A and B,

$$\mathbf{R}_P = (\alpha\mathbf{R}_A + \beta\mathbf{R}_B)/(\alpha + \beta) \tag{A.4}$$

The exponent of the new Gaussian centered at $\mathbf{R}_P$ is

$$p = \alpha + \beta \tag{A.5}$$

The product of two Gaussians is Gaussian. The proof is straightforward and is left to the reader as an exercise

In [55]:
```
# Integrals between Gaussian orbitals (pp410)

def gauss_product(gauss_A, gauss_B):
    # The product of two Gaussians gives another Gaussian (pp411)
    # Pass in the exponent and centre as a tuple

    a, Ra = gauss_A
    b, Rb = gauss_B
    p = a + b
    diff = np.linalg.norm(Ra-Rb)**2            # squared difference of the two centres
    N = (4*a*b/(pi**2))**0.75                  # Normalisation
    K = N*exp(-a*b/p*diff)                     # New prefactor
    Rp = (a*Ra + b*Rb)/p                       # New centre

    return p, diff, K, Rp
```

Code implementation

Note that in our code, we have absorbed the normalizing factors into K, and thus do not need to worry about normalisation.

2.3) The Overlap and Kinetic integrals between two Gaussians (pp411)

```
# Overlap integral (pp411)
def overlap(A, B):
    p, diff, K, Rp = gauss_product(A, B)
    prefactor = (pi/p)**1.5
    return prefactor*K

# Kinetic integral (pp412)
def kinetic(A,B):
    p, diff, K, Rp = gauss_product(A, B)
    prefactor = (pi/p)**1.5
```

```
    a, Ra = A
    b, Rb = B
    reduced_exponent = a*b/p
    return reduced_exponent*(3-2*reduced_exponent*diff)*prefactor*K
```

Overlap and kinetic energy integrals

2.4) The Potential integral, the Multi-electron Tensor and Boys Integral (pp412)

To get the potential integral and multi-electron tensor, we need to define a variant of the **Boys** function, which in turn (for this case) is related to the error function.

We now introduce the F_0 function, which is defined by

$$F_0(t) = t^{-1/2} \int_0^{t^{1/2}} dy\, e^{-y^2} \qquad \text{(A.31)}$$

It is related to the error function by

$$F_0(t) = \tfrac{1}{2}(\pi/t)^{1/2} erf(t^{1/2}) \qquad \text{(A.32)}$$

```
# Fo function for calculating potential and e-e repulsion integrals.
# Just a variant of the error function
# pp414
def Fo(t):
    if t == 0:
        return 1
    else:
        return (0.5*(pi/t)**0.5)*erf(t**0.5)
```

Fo function

For higher orbitals (2p, 3d, etc) we can't express the Boys function in terms of the error function and different methods are required. This has been subject to great academic study. Carrying on, we can now give the potential and multi-electron integrals:

```
# Nuclear-electron integral (pp412)
def potential(A,B,atom_idx):
    p,diff,K,Rp = gauss_product(A,B)
    Rc = atom_coordinates[atom_idx] # Position of atom C
    Zc = charge_dict[atoms[atom_idx]] # Charge of atom C


    return (-2*pi*Zc/p)*K*Fo(p*np.linalg.norm(Rp-Rc)**2)

# (ab|cd) integral (pp413)
def multi(A,B,C,D):
    p, diff_ab, K_ab, Rp = gauss_product(A,B)
    q, diff_cd, K_cd, Rq = gauss_product(C,D)
    multi_prefactor = 2*pi**2.5*(p*q*(p+q)**0.5)**-1
```

```
    return multi_prefactor*K_ab*K_cd*Fo(p*q/(p+q)*np.linalg.norm(Rp-Rq)**2)
```

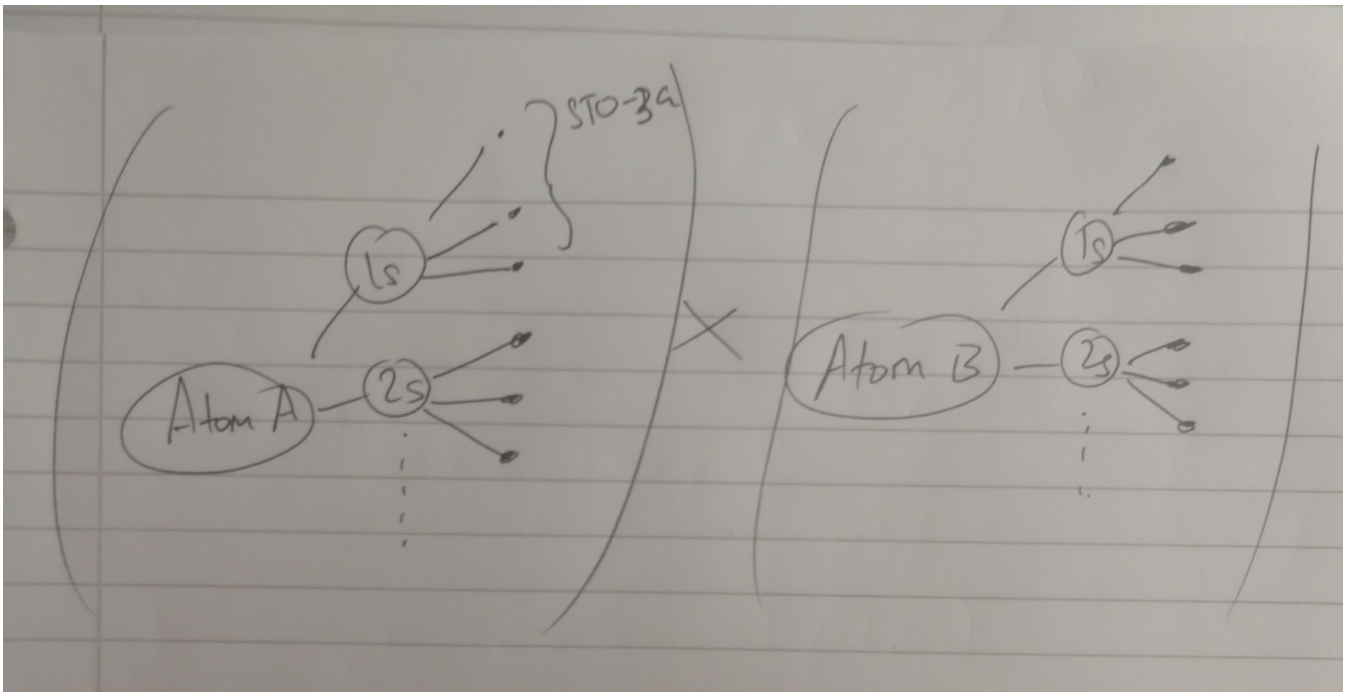
potential and multi-electron integrals

3. Computing integrals in the atomic orbital basis

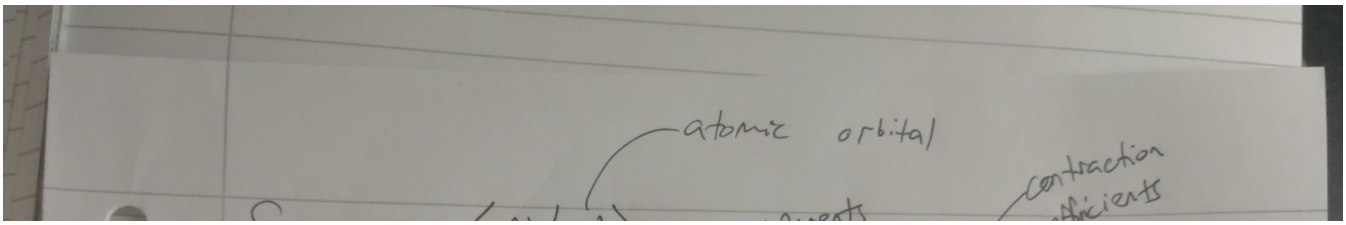
This is probably the trickiest part of this tutorial, and care needs to be taken thinking about the for loops. The idea is that at each stage of the for loop, we store information so that we don't have to keep calling the same things over and over again. It doesn't matter here, because we are doing a very simple calculation. But it would matter for more expensive calculations.

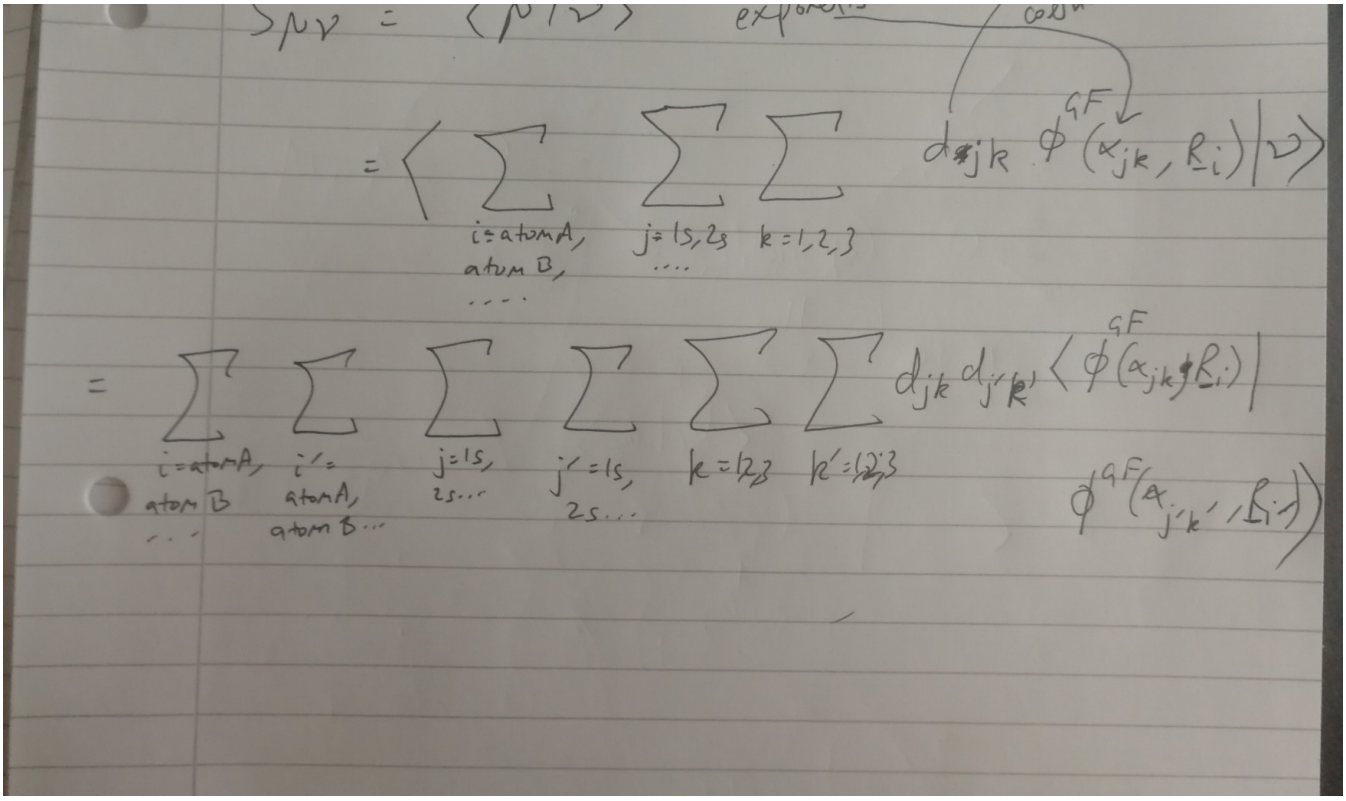
We will iterate first through the atoms. On each atom, we iterate through its orbitals. Finally, for each orbital, we iterate through its three Gaussians. We perform this triple iteration over each atom.

In this simple case, we could have just summed over the three Gaussians on each atom directly (because each atom has only 1 atomic orbital). But by doing it this way, we can easily extend our program to solve more complicated molecules, which we will do in a future tutorial.



Visual depiction of sum





The different summations we are carrying out. Note, for the multi-electron integral we would need to carry out double the amount of sums, that is, 12 sums.

You might need to spend some time convincing yourself of this, or (even better) try to code it out yourself.

```
# Initialise matrices
S = np.zeros((B,B))
T = np.zeros((B,B))
V = np.zeros((B,B))
multi_electron_tensor = np.zeros((B,B,B,B))


# Iterate through atoms
for idx_a, val_a in enumerate(atoms):

    # For each atom, get the charge and centre
    Za = charge_dict[val_a]
    Ra = atom_coordinates[idx_a]

    # Iterate through quantum numbers (1s, 2s etc)
    for m in range(max_quantum_number[val_a]):

        # For each quantum number, get the contraction
        # coefficients, then get zeta,
        # then scale the exponents accordingly (pp158)
        d_vec_m = D[m]
        zeta = zeta_dict[val_a][m]
        alpha_vec_m = alpha[m]*zeta**2

        # Iterate over the contraction coefficients
        for p in range(STOnG):


            # Iterate through atoms once again (more info in blog post)
            for idx_b, val_b in enumerate(atoms):
                Zb = charge_dict[val_b]
                Rb = atom_coordinates[idx_b]
```

```
            Rb = atom_coordinates[idx_b]
            for n in range(max_quantum_number[val_b]):
                d_vec_n = D[n]
                zeta = zeta_dict[val_b][n]
                alpha_vec_n = alpha[n]*zeta**2
                for q in range(STOnG):
```

```
        # Iterate through atoms once again (more info in blog post)
        for idx_b, val_b in enumerate(atoms):
            Zb = charge_dict[val_b]
            Rb = atom_coordinates[idx_b]
            for n in range(max_quantum_number[val_b]):
                d_vec_n = D[n]
                zeta = zeta_dict[val_b][n]
                alpha_vec_n = alpha[n]*zeta**2
                for q in range(STOnG):


                    # This indexing is explained in the blog post.
                    # In short, it is due to Python indexing
                    # starting at 0.

                    a = (idx_a+1)*(m+1)-1
                    b = (idx_b+1)*(n+1)-1

                    # Generate the overlap, kinetic and potential matrices

                    S[a,b] += d_vec_m[p]*d_vec_n[q]*overlap((alpha_vec_m[p],Ra),(alpha_vec_n[q],Rb))
                    T[a,b] += d_vec_m[p]*d_vec_n[q]*kinetic((alpha_vec_m[p],Ra),(alpha_vec_n[q],Rb))

                    for i in range(N_atoms):
                        V[a,b] += d_vec_m[p]*d_vec_n[q]*potential((alpha_vec_m[p],Ra),(alpha_vec_n[q],Rb),i)
```

Note the extra sum over atoms to get the entire potential energy matrix

```
for i in range(N_atoms):
    V[a,b] += d_vec_m[p]*d_vec_n[q]*potential((alpha_vec_m[p],Ra),(alpha_vec_n[q],Rb),i)


# 2 more iterations to get the multi-electron-tensor
for idx_c, val_c in enumerate(atoms):
    Zc = charge_dict[val_c]
    Rc = atom_coordinates[idx_c]
    for k in range(max_quantum_number[val_c]):
        d_vec_k = D[k]
        zeta = zeta_dict[val_c][k]
        alpha_vec_k = alpha[k]*zeta**2
        for r in range(STOnG):
            for idx_d, val_d in enumerate(atoms):
                Zd = charge_dict[val_d]
                Rd = atom_coordinates[idx_d]
                for l in range(max_quantum_number[val_d]):
                    d_vec_l = D[l]
                    zeta = zeta_dict[val_d][l]
                    alpha_vec_l = alpha[l]*zeta**2
                    for s in range(STOnG):
                        c = (idx_c+1)*(k+1)-1
                        d = (idx_d+1)*(l+1)-1
                        multi_electron_tensor[a,b,c,d] += d_vec_m[p]*d_vec_n[q]*d_vec_k[r]*d_vec_l[s]*(
                        multi((alpha_vec_m[p],Ra),
                              (alpha_vec_n[q],Rb),
                              (alpha_vec_k[r],Rc),
                              (alpha_vec_l[s],Rd))
                        )
```

We carry out the iteration 2 more times to get the multi-electron tensor

If you got through that, great! The rest of the algorithm is relatively straightforward.

Lastly, since the kinetic and potential energy integrals aren't affected by the iterative process, we can just assign a variable to the sum of them, **Hcore**.

```
# Form Hcore
Hcore = T + V
```

Hcore

4. Symmetric Orthogonalisation of the Basis (pp144)

If we remember the Hartree-Fock equations in a basis (the Roothan equations), we cannot solve it like a normal eigenvalue equation due to the overlap matrix.

$$\mathbf{FC} = \mathbf{SC}\boldsymbol{\varepsilon}$$

The Roothan Equations

We can, however, transform into an orthogonal basis. There are several ways to find a matrix that will orthogonalize the basis set but we will use *symmetric orthogonalization*. We note that since **S** is Hermitian (symmetric in the case of real orbitals), **S** can always be diagonalized, the proof of which is in any linear algebra text. We can write:

$$\mathbf{U}^{\dagger}\mathbf{S}\mathbf{U} = \mathbf{s}$$

Diagonalisation of the overlap matrix

where **s** is a diagonal matrix. Then we can define:

$$\mathbf{X} \equiv \mathbf{S}^{-1/2} = \mathbf{U}\mathbf{s}^{-1/2}\mathbf{U}^{\dagger}$$

It is easy to show that:

$$\mathbf{S}^{-1/2}\mathbf{S}\mathbf{S}^{-1/2} = \mathbf{S}^{-1/2}\mathbf{S}^{1/2} = \mathbf{S}^{0} = \mathbf{1}$$

If we then rotate our orbital matrix with **X** we obtain:

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

Substituting the above into the Roothan equations yields:

$$\mathbf{F}\mathbf{X}\mathbf{C}' = \mathbf{S}\mathbf{X}\mathbf{C}'\boldsymbol{\varepsilon}$$

Left multiplying with the Hermitian-transpose of **X** we obtain:

$$\mathbf{F}'\mathbf{C}' = \mathbf{C}'\boldsymbol{\varepsilon}$$

where

$$\mathbf{F}' = \mathbf{X}^{\dagger}\mathbf{F}\mathbf{X}$$

Now we can easily solve the Roothan equations by diagonalizing **F'**. Below is a code implementation to obtain **X.**

```
# Symmetric Orthogonalisation of basis (p144)

evalS, U = np.linalg.eig(S)
diagS = dot(U.T,dot(S,U))
diagS_minushalf = diag(diagonal(diagS)**-0.5)
X = dot(U,dot(diagS_minushalf,U.T))
```

Symmetric orthogonalization

5. The Hartree-Fock Algorithm

We are finally in a position to write the iterative algorithm

The reason why Hartree-Fock is iterative is that the Fock matrix depends on the molecular orbitals. That is to say, you can't get the answer…without the answer. Of course, you can take a guess at the answer, and solve the Roothan equations. The solution you get will be better than your previous guess. \

In order to quantify when we stop making more guesses, we can see how the *orbital matrix* has changed compared to the last guess. This is completely valid, but it turns out that, probably due to convention, we use the **density matrix** instead (pp138).

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

The density matrix

One really important thing about the density matrix is to remember the sum is only over the occupied orbitals (in the closed shell case). It can, thus, be interpreted as a **bond order matrix** as well. Let's write a function to check the difference between the two most recent guesses of the density matrix.

```
def SD_successive_density_matrix_elements(Ptilde,P):
    x = 0
    for i in range(B):
        for j in range(B):
            x += B**-2*(Ptilde[i,j]-P[i,j])**2

    return x**0.5
```

Function to check convergence

We can now initiate a while loop that keeps repeating until convergence.

5.1 Take a guess at P

```
# Algorithm

# Initial guess at P
P = np.zeros((B,B))
P_previous = np.zeros((B,B))
P_list = []
```

We'll use the identity as a guess

Remember that we've defined B as our basis-set size, which is 2 in this case. We will also store the subsequent guesses of P to see how fast we converge.

5.2 Initiate the while loop

```
# Iterative process
threshold = 100
```

```
while threshold > 10**-4:
```

Loading...

5.3 Compute the Fock matrix with the initial guess of P (pp140–141)

```
# Iterative process
threshold = 100
while threshold > 10**-4:

    # Calculate Fock matrix with guess
    G = np.zeros((B,B))
    for i in range(B):
        for j in range(B):
            for x in range(B):
                for y in range(B):
                    G[i,j] += P[x,y]*(multi_electron_tensor[i,j,y,x]-0.5*multi_electron_tensor[i,x,y,j])
    Fock = Hcore + G
```

Calculate G, then calculate Fock

One qualm that the hawk-eyed might wonder is why only 1 instance of **P** comes out of the sum in **G**(and not twice, or even at all). This is because we want to find **F** in the basis of atomic orbitals. The ***coulomb*** and ***exchange*** operators are defined in the basis of ***molecular orbitals*** which we must expand in terms of our atomic basis. If the operators were defined in the atomic basis, we would not need any instance of **P**.

5.4 Symmetric Orthogonalisation, as discussed before

```
    # Calculate Fock matrix in orthogonalised base
    Fockprime = dot(X.T,dot(Fock, X))
    evalFockprime, Cprime = np.linalg.eig(Fockprime)

    #Correct ordering of eigenvalues and eigenvectors (starting from ground MO as first column of C, else we get the wrong P)
    idx = evalFockprime.argsort()
    evalFockprime = evalFockprime[idx]
    Cprime = Cprime[:,idx]

    C = dot(X,Cprime)
```

Note this is part of the while loop. Make sure your indentation is correct

The second block of code is to make sure the eigenvalues, and orbital matrix, is sorted in ascending order. This is not the case by default (for some reason) and so if this part is ignored, the density matrix will be computed wrongly in the next part.

5.5 Form the new Density matrix and check for convergence

```
    # Form new P (note, we only sum over electron pairs - we DON'T sum
    # over the entire basis set.

    for i in range(B):
        for j in range(B):
            for a in range(int(N/2)):
```

```
                P[i,j] = 2*C[i,a]*C[j,a]


    P_list.append(P)

    threshold = SD_successive_density_matrix_elements(P_previous,P)
    P_previous = P.copy()
```

Almost done!

Note the use of the convergence-checking function we wrote earlier. If we have converged, the while loop will break and we can simply read off our energies and orbital matrix.

5.6 Print results and we are done!

We can go ahead and enter this:

```
print('\n')
print('STO3G Restricted Closed Shell HF algorithm took {} iterations to converge'.format(len(P_list)))
print('\n')
print('The orbital energies are {} and {} Hartrees'.format(evalFockprime[0],evalFockprime[1]))
print('\n')
print(f'The orbital matrix is: \n\n{C}')
print('\n')
print(f'The density/bond order matrix is: \n\n{P}')
```

Note the two different types of string formatting: you can use any

The result (if everything has gone correctly) should be this:

```
STO3G Restricted Closed Shell HF algorithm took 6 iterations to converge


The orbital energies are -1.5974477442856965 and -0.061669520372867936 Hartrees


The orbital matrix is:

[[-0.80191685  0.78226417]
 [-0.33680061 -1.0684443 ]]


The density/bond order matrix is:

[[1.28614127 0.54017217]
 [0.54017217 0.22686931]]
```

Hurrah!

You might be wondering why the anti-bonding orbital is still negative? That's because we haven't added nuclear-nuclear repulsion. We can do that easily:

```
In [92]: # Nuclear repulsion
         def get_nuclear_repulsion():
             Nuc_repuls = 0
             for idx_a, A in enumerate(atoms):
                 for idx_b, B in enumerate(atoms):
                     if idx_a == idx_b:
                         continue
                     charge_A = charge_dict[A]
                     charge_B = charge_dict[B]
                     product = charge_A*charge_B
                     Ra = atom_coordinates[idx_a]
                     Rb = atom_coordinates[idx_b]
                     R = np.linalg.norm(Ra-Rb)
                     Nuc_repuls += product/R
             return Nuc_repuls*0.5

In [93]: get_nuclear_repulsion()

Out[93]: 1.366867140513942
```

Nuclear repulsion

We can see the anti-bonding orbital is raised in energy a lot more than the bonding orbital is lowered. This is especially so because the species is positively charged, but applies less strongly in the general case.

End Notes:

Thanks for following through. Leave a comment if you got stuck anywhere and I'll try to answer it. In the next tutorial, we will see what other properties we can obtain other than just the orbital energies (population analysis, dipole moment, etc.). In the tutorial after that, we will see what the main issue with Hartree-Fock is, and how we can improve on its predictions with second-order perturbation theory (**Moller-Plesset theory**). In the tutorial after that, we will see how to treat p and d orbitals. Combined with this, we can calculate the electronic structure accurately of much larger organic molecules.