Handout 4: Implementation of the Molecular Integrals

4.1 Prepare your integrals code

It is at the reader or instructor's discretion whether to include all molecular integrals and Hartree–Fock code in one file, or to break the code up over several files. In Handout 3 (introduction to programming), the latter was recommended because it makes managing the code easier. The suggested file structure was as follows: `basis.py` (to hold STO-3G basis set information and construct the atomic orbital basis), `oei.py` (to calculate one-electron integrals $\mathbf{S}$, $\mathbf{T}$, $\mathbf{V}$), `eri.py` (to calculate two-electron integrals $\mathbf{G}$), and `main.py` (to "drive" the first three files and pass results into the Hartree–Fock code `hf.py` described in the next Handout). It was discussed in Handout 3 how, from within one `.py` file, one can call a function from another `.py` file.

In most of these code files you will need to import a few modules from the SciPy library to perform some specialized mathematical operations (e.g., double factorial). Additionally, you should import the NumPy library, because both simple (e.g., matrix construction, matrix multiplication) and sophisticated (e.g., diagonalization) linear algebra operations are easily handled with NumPy, and will prove especially useful when coding the Hartree–Fock method. In the current project, we will use NumPy to build empty two-dimensional matrices (each of which will hold the overlap, kinetic energy, and electron-nuclear attraction integrals), as well as a rank-4 tensor (essentially just a four-dimensional matrix) to hold the electron-electron repulsion integrals; the elements will then be filled in using the appropriate matrix element equations.

4.2 Read the input `.xyz` file

In the `main.py` file ...

1. Read in the `.xyz` file.
   - Form a one-dimensional list itemizing `atoms` (e.g., `['O', 'H', 'H']`).
   - Form a two-dimensional nested list of the atom's x, y, z coordinates (`[[0.000,0.000,0.000],...]`). If the coordinates aren't already in units Bohr, make sure to convert them.

4.3 Build the STO-3G basis

In `basis.py` ...

1. Construct two arrays that hold the basis set information (we will use the minimal basis set STO-3G). One array should be allocated for α_p exponential factors, and the other for d_p contraction coefficients (here, p is a general index, which is a for ϕ_A orbitals, b for ϕ_B orbitals, and so on). The α_p containing array will need to be nested two times in order to hold values for distinct atoms, distinct subshells within atoms, and the three individual values. The d_a array will only need to be nested once in order to hold values for different shells, because the same contraction coefficients apply to all atoms. It is suggested that you include basis set information for at least the first ten atoms of the periodic table, which would mean returning the length of α_p yields the value `10`.

   The required STO-3G basis set values for α_p and d_p can be found at `https://bse.pnl.gov/bse/portal`. (See Fig. 6 of Handout 1 for downloading and intepreting basis set values.)

2. Create a dictionary `orbitalconfiguration` that returns the minimal basis atomic orbital configuration of any atom through the second row of the periodic table (the first ten atoms).
   - For example, passing in `'O'` or `'C'` returns `['1s','2s','2px','2py','2pz']`

3. Looping through the `atoms`, and then through the `orbitalconfiguration` of each atom ...
   - Append to a `sto3Gbasis` array, such that
     - each element is a dictionary, and
     - the keys of each dictionary host ...
       (a) List of α_p exponential factors (depends on orbital type)

(b) List of d_p contraction coefficients (depends on orbital type)
(c) List of atom coordinates (x, y, z)
(d) l (= 0 if $1s$, $2s$, $2p_y$, $2p_z$; = 1 if $2p_x$)
(e) m (= 0 if $1s$, $2s$, $2p_x$, $2p_z$; = 1 if $2p_y$)
(f) n (= 0 if $1s$, $2s$, $2p_x$, $2p_y$; = 1 if $2p_z$)

- By way of example, calling information on just the $2p_x$ orbital of the oxygen atom should return something close to

```
{'a': [0.380389, 1.1695961, 5.0331513], 'd': [0.39195739, 0.60768372, \
  0.15591627], 'R': [0.0, 0.0, 0.454025755], 'l': 1, 'm': 0, 'n': 0}
```

- *Tip:* When looping through the atoms, you will need not only the atom identity (e.g., hydrogen), but also its index (e.g., row 2 of the x, y, z coordinates in the input file). Consider using

```
for i, atom in enumerate(atoms):
```

  to return both, where `i` is the numerical index.

4. The basis is now built. When you call `sto3Gbasis`, built for the input `h2o.xyz`, seven items should be returned. Why is this?

4.4 Overlap integrals

In `oei.py` ...

1. Build an empty $K \times K$ matrix called `S`, where K is the number of atomic orbitals included in the basis. Use `np.zeros` to generate an empty 2D NumPy matrix:

```
S = np.zeros((K, K))
```

This will be your overlap integral matrix $\mathbf{S}$, or $S_{A,B} \in \mathbb{R}^{K\times K}$. The equation for any individual matrix element $(A|B)$ of $\mathbf{S}$ is

$$\begin{aligned}(A|B) &= \int\int\int \phi_A \phi_B \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \\ &= \sum_{a=1}^{K_a}\sum_{b=1}^{K_b} \mathrm{d}_a \mathrm{d}_b N_a N_b \int\int\int \chi_a(\vec{A}, \alpha_a, l_A, m_A, n_A)\chi_b(\vec{B}, \alpha_b, l_B, m_B, n_B)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z\end{aligned} \tag{1}$$

where χ_a is the Gaussian primitive for orbital A. The linear combination of these primitives form our atomic orbital ϕ_A, therefore we must sum up to K_a, which is the number of primitives per orbital (3 for `sto3Gbasis`, because we take a linear combination of *three* Gaussian functions to emulate one Slater-type orbital). Recall that α_a is the exponential factor for the current primitive (of 3 total) of orbital A, d_a is the contraction coefficient, $\vec{A}$ are the atomic x, y, z coordinates of orbital ϕ_A (because the orbitals are *centered* on the atoms), and l_A, m_A, and n_A are the angular momenta corresponding to the x, y, and z directions, respectively.

This should provide a fairly concrete sense of how a matrix element is evaluated, however we still require analytical expressions for the multiple integral component in order to code it up. We will come to that shortly.

2. Loop over each element $(A|B)$ of `S` in order to access and modify that element.
   - Do so with two nested loops, with each loop corresponding to orbital index A or B.
   - While looping over an orbital index, you'll need to pull its corresponding orbital information from the `sto3Gbasis` array. (*Tip:* Use the `enumerate` functionality.)

3. Now that we've "accessed" a matrix element with our two nested loops, we need to access elements within the basis that we've pulled for that index. For example, for the STO-3G basis set, there are three α_a exponent factors and three d_a contraction coefficients that we will be passing on to another function.

   - Once again, we need to obtain the basis components for each index we're looping over. In other words, we need yet two more nested loops within the nested loops we've already formed.
   - *Tip:* We can simultaneously retrieve the pair of α_a and d_a values necessary for the current primitive by using another useful Python function called `zip`. For example, when looping within index `A`, the basis for which we might name `bA`, we can call the α_a and d_a elements as follows:

     ```
         for a, da in zip(bA['a'],bA['d']):
     ```

4. Now that we've accessed specific α_p and d_p values to calculate the integral corresponding to one Gaussian primitive, we will need these values, the position coordinates (x, y, z) of each atom that the orbitals we're evaluating correspond to, and each orbital's angular momentum values (l, m, n). We then plug these into our analytical expression for the overlap integral:

$$\int\int\int \chi_a(\vec{A},\alpha_a,l_A,m_A,n_A)\chi_b(\vec{B},\alpha_b,l_B,m_B,n_B)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = \exp\left(-\frac{\alpha_a\alpha_b}{\gamma}|\vec{AB}|^2\right)\times S_x\times S_y\times S_z \tag{2}$$

where

$$S_x = \sqrt{\frac{\pi}{\gamma}}\sum_{k=0}^{\lfloor(l_A+l_B)/2\rfloor} c_{2k}(l_A,l_B,\vec{PA}_x,\vec{PB}_x)\frac{(2k-1)!!}{(2\gamma)^k}\,, \tag{3}$$

where $\lfloor x\rfloor$ is the floor function (conveniently, the `int(x)` function of Python returns the largest integer value less than or equal to the argument, which is the definition of the floor function), l_A and l_B are the angular momenta of orbital indices A and B, respectively, and where

$$\vec{P} = \frac{\alpha_a\vec{A}+\alpha_b\vec{B}}{\alpha_a+\alpha_b}\,, \tag{4}$$

$$\vec{PA} = \vec{P}-\vec{A}\,, \tag{5}$$

$$|\vec{AB}|^2 = (\vec{A}_x-\vec{B}_x)^2+(\vec{A}_y-\vec{B}_y)^2+(\vec{A}_z-\vec{B}_z)^2\,, \tag{6}$$

$$\gamma = \alpha_a+\alpha_b\,, \tag{7}$$

$$c_k(l,m,a,b) = \sum_{i=0}^{l}\sum_{j=0}^{m}\binom{l}{i}\binom{m}{j}a^{l-i}b^{m-j};\quad i+j\overset{!}{=}k\,. \tag{8}$$

Note that notation such as $\vec{P}_x$ refers specifically to the x component of the position vector $\vec{P}$. The function c_k is a generic expression that will be used by more than just equation 3 – hence, the generic form of its arguments. The exclamation point above the equals sign indicates that the function is evaluated *only* when the sum of indices i and j is equal to the argument k.

Additionally, SciPy includes a function to calculate a double factorial:

```
misc.factorial2(2*j-1, exact=True)
```

The double factorial function can be applied to non-integer arguments, and cost-saving approximations made with `exact=False`. But the function is fastest if it is applied to exact integers, which is all that we are dealing with – hence, `exact` is set to `True`.

Binomial coefficients can be computed as:

```
special.binom(m,j)
```

5. This Gaussian primitive is almost fully evaluated. However, we need to multiply it by its contraction coefficients d_a and d_b, as well as its normalization constants N_a and N_b, the latter of which are evaluated as follows:

$$N(\alpha, l, m, n) = \left[\left(\frac{2\alpha}{\pi} \right)^{3/2} \frac{(4\alpha)^{l+m+n}}{(2l-1)!!(2m-1)!!(2n-1)!!} \right]^{1/2} \tag{9}$$

6. Having looped through every orbital for both indices, and in turn every primitive for every orbital, the evaluation of the overlap integrals is complete!

4.5 Kinetic energy integrals

In `oei.py` ...

1. The evaluation of the kinetic energy integrals proceeds in much the same manner as it does for the overlap integrals. There are simply more terms that go into each matrix element, which has the general form $(A| - \frac{1}{2}\nabla^2|B)$; the operator features the Laplacian ∇^2, which is the sum of second partial derivatives with respect to x, y, and z, individually. Form another $K \times K$ matrix called $\mathbf{T}$.

2. As before, loop through every orbital pair to access a specific matrix element, and implement the following working equation:

$$\begin{aligned}(A| - \frac{1}{2}\nabla^2|B) = \sum_{a=1}^{K_a}\sum_{b=1}^{K_b} \mathrm{d}_a \mathrm{d}_b N_a N_b \int\int\int \chi_a(\vec{A}, \alpha_a, l_A, m_A, n_A) \Big\{ \alpha_b \left[2(l_B + m_B + n_B) + 3\right] \chi_b(\vec{B}, \alpha_b, l_B, m_B, n_B) - \\ 2\alpha_b^2 \Big[\chi_b(\vec{B}, \alpha_b, l_B + 2, m_B, n_B) + \chi_b(\vec{B}, \alpha_b, l_B, m_B + 2, n_B) + \chi_b(\vec{B}, \alpha_b, l_B, m_B, n_B + 2) \Big] - \\ \frac{1}{2} \Big[l_B(l_B - 1)\chi_b(\vec{B}, \alpha_b, l_B - 2, m_B, n_B) + m_B(m_B - 1)\chi_b(\vec{B}, \alpha_b, l_B, m_B - 2, n_B) + n_B(n_B - 1)\chi_b(\vec{B}, \alpha_b, l_B, m_B, n_B - 2) \Big] \Big\} \\ \mathrm{d}x\, \mathrm{d}y\, \mathrm{d}z \end{aligned} \tag{10}$$

While the presence of the integral signs might lead you to believe we haven't presented a code-able expression, you should be able to see that this equations consists of the sum of seven distinct overlap integrals, with some leading factors consisting of constant terms α_p, angular momentum components l, m, n, *etc.*. As such, this matrix element depends heavily on Eq. 2. The kinetic energy integrals are, mathematically, nothing more than a sum of overlap integrals with coefficients!

4.6 Electron-nuclear attraction integrals

In `oei.py` ...

1. The evaluation of the electron-nuclear attraction integrals is distinct from the construction of overlap and integral matrices in that we will now sum n two-dimensional matrices, each of which we call $\mathbf{V}_C$ corresponding to the index C for each atom, to form our matrix $\mathbf{V}$, as follows:

$$\mathbf{V} = \mathbf{V}_1 + \mathbf{V}_2 + ... + \mathbf{V}_n \tag{11}$$

where n is the number of atoms.

We will need to collect atomic number and coordinates for this new, third index C. However, we do *not* need to access, for matrix $\mathbf{V}_C$, the α_p exponential factors and contraction coefficients of index C.

2. The matrix element for any individual matrix element of $\mathbf{V}_C$ has the general form

$$(A|\frac{-Z_C}{\vec{r}_{iC}}|B) = \int\int\int \phi_A \left(\frac{-Z_C}{\vec{r}_{iC}}\right) \phi_B \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \tag{12}$$

where the operator features the atomic number Z of the nucleus C; the denominator indicates the distance between electron i and nucleus C.

The working equation for the matrix element is:

$$\begin{aligned}(A|\frac{-Z_C}{\vec{r}_{iC}}|B) = &\sum_{a=1}^{K_b}\sum_{b=1}^{K_b} \mathrm{d}_a \mathrm{d}_b N_a N_b \times \Big\{ - Z_C \times \frac{2\pi}{\gamma} \times \exp\left(\frac{-\alpha_a\alpha_b}{\gamma}|\vec{AB}|^2\right) \\ &\times \sum_{l=0}^{l_A+l_B}\sum_{r=0}^{\lfloor l/2\rfloor}\sum_{i=0}^{\lfloor (l-2r)/2\rfloor} v_{l,r,i}(l_A, l_B, \vec{A}_x, \vec{B}_x, \vec{C}_x, \gamma) \\ &\times \sum_{m=0}^{m_A+m_B}\sum_{s=0}^{\lfloor m/2\rfloor}\sum_{j=0}^{\lfloor (m-2s)/2\rfloor} v_{m,s,j}(m_A, m_B, \vec{A}_y, \vec{B}_y, \vec{C}_y, \gamma) \\ &\times \sum_{n=0}^{n_A+n_B}\sum_{t=0}^{\lfloor n/2\rfloor}\sum_{k=0}^{\lfloor (n-2t)/2\rfloor} v_{n,t,k}(n_A, n_B, \vec{A}_z, \vec{B}_z, \vec{C}_z, \gamma) \\ &\times F_{l+m+n-2(r+s+t)-(i+j+k)}(\gamma|\vec{PC}|^2)\Big\}\end{aligned} \tag{13}$$

where

$$v_{l,r,i}(l_A, l_B, \vec{A}_x, \vec{B}_x, \vec{C}_x, \gamma) = (-1)^a c_l(l_A, l_B, \vec{PA}_x\vec{PB}_x)\frac{(-1)^i l! \vec{PC}_x^{\,l-2r-2i}\epsilon^{r+i}}{r!i!(l-2r-2i)!} \tag{14}$$

$$\epsilon = \frac{1}{4\gamma} \tag{15}$$

Here, $\vec{PC}_x$ is the x component of a new vector $\vec{PC} = \vec{P} - \vec{C}$, the distance between the midpoint position $\vec{P}$ and the third index position $\vec{C}$. It's worth trying to see the pattern in the many indices present in the summations – for example, the floor functions constituting the upper range of certain sums involve indices being counted over from sums earlier in the same line of the equation. This interdependence of sums is largely due to the treatment of the product of polynomials that form the angular momentum terms of the contracted Gaussian type orbitals (see Handout 2). These summation indices, in turn, are arguments for the function v.

Single factorials can be calculated similarly to double factorials:

```
misc.factorial(l-2*r-2*i, exact=True)
```

The Boys function for x greater than zero

$$F_\nu(x) = \int_0^1 t^{2\nu}\exp\left(-xt^2\right)\,dt = \frac{1}{2x^{(\nu+\frac{1}{2})}} \times \gamma(\nu+\frac{1}{2}, x) \times \Gamma(\nu+\frac{1}{2}) \tag{16}$$

includes both a lower incomplete gamma function γ (not to be confused with our *variable* $\gamma = \alpha_a + \alpha_b$), and a normal Gamma function Γ, the latter of which was not discussed in the derivation, but *is* necessary because Python normalizes the lower incomplete gamma function γ by dividing it by Γ.

Evaluation of this in Python relies on two special SciPy functions:

```
F = 0.5 * x**(-(nu + 0.5)) * special.gammainc(nu + 0.5, x) * \
      special.gamma(nu + 0.5)
```

For very small x (say, $x < 1 \times 10^{-6}$), the equation breaks down. This is because, as x approaches zero in the denominator of the first quotient of the expression, the equation "blows up." However, we can use a Taylor expansion of the function and truncate at second-order:

$$F_\nu(x) \approx \frac{1}{2\nu+1} - \frac{x}{2\nu+3} \tag{17}$$

where x has been removed from the denominator. An `if` statement should be employed to allow both argument types.

3. Notice that there are three summation symbols per x, y, and z direction of the function v. Thus, implementation of the equation for evaluating integration over *one* Gaussian primitive requires *nine* loops. Thus, to fully evaluate a matrix element as a contraction of the three primitives (in STO-3G), these nine loops are *within* the two blocks of nested loops needed to access the matrix elements and the α_p and d_p values.

4.7 Electron-electron repulsion integrals

In `eri.py` ...

1. Having conquered the electron-nuclear attraction integrals, the two-electron repulsion integrals are effectively just an extension of these – i.e., more loops! There is one key difference: We must form a rank-4 tensor instead of just a two-dimensional matrix. So instead of looping over two indices to access a two-dimensional matrix element, we must loop over four in order to access any one element of this $K \times K \times K \times K$ "matrix". Nevertheless, the looping technique is just the same as before.

   Use NumPy to form $\mathbf{G}$.

2. The two-electron operator is $(1/\vec{r}_{ij})$, and describes the inverse distance between electrons i and j. This operator acts on the product of orbitals ϕ_C and ϕ_D; after we've multiplied this result by the product of orbitals A and B, we integrate over all space to get the matrix element $(AB|CD)$.

   The working equation for the matrix element $(AB|CD)$ is

$$\begin{aligned}
G =& \int\int\int \phi_A \phi_B \left(\frac{1}{\vec{r}_{ij}}\right) \phi_C \phi_D \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z \\
=& \sum_{a=1}^{K_a} \sum_{b=1}^{K_b} \sum_{c=1}^{K_c} \sum_{d=1}^{K_d} \mathrm{d}_a \mathrm{d}_b \mathrm{d}_c \mathrm{d}_d N_a N_b N_c N_d \\
&\times \left\{ \frac{2\pi^2}{\gamma_P \gamma_Q} \left(\frac{\pi}{\gamma_P + \gamma_Q}\right)^{1/2} \times \exp\left(\frac{-\alpha_a \alpha_b}{\gamma_P} |\vec{AB}|^2\right) \times \exp\left(\frac{-\alpha_c \alpha_d}{\gamma_Q} |\vec{CD}|^2\right) \right. \\
&\times \sum_{l_P=0}^{l_A+l_B} \sum_{r_P=0}^{\lfloor l_P/2 \rfloor} \sum_{l_Q=0}^{l_C+l_D} \sum_{r_Q=0}^{\lfloor l_Q/2 \rfloor} \sum_{i=0}^{\lfloor (l_P+l_Q-2r_P-2r_Q)/2 \rfloor} g_x \\
&\times \sum_{m_P=0}^{m_A+m_B} \sum_{s_P=0}^{\lfloor m_P/2 \rfloor} \sum_{m_Q=0}^{m_C+m_D} \sum_{s_Q=0}^{\lfloor m_Q/2 \rfloor} \sum_{j=0}^{\lfloor (m_P+m_Q-2s_P-2r_Q)/2 \rfloor} g_y \\
&\times \sum_{n_P=0}^{n_A+n_B} \sum_{t_P=0}^{\lfloor n_P/2 \rfloor} \sum_{n_Q=0}^{n_C+n_D} \sum_{t_Q=0}^{\lfloor n_Q/2 \rfloor} \sum_{k=0}^{\lfloor (n_P+n_Q-2t_Q-2t_Q)/2 \rfloor} g_z \\
&\left. \times F_\nu(|\vec{PQ}|^2/4\delta) \right\}
\end{aligned} \tag{18}$$

where we have a new third center $\vec{Q}$ located between orbitals ϕ_C and ϕ_D (see Eq. 4); additionally,

$$\gamma_P = \alpha_a + \alpha_b, \qquad \gamma_Q = \alpha_c + \alpha_d\,, \tag{19}$$

$$\delta = \frac{1}{4\gamma_P} + \frac{1}{4\gamma_Q}\,, \tag{20}$$

$$\nu = l_P + l_Q + m_P + m_Q + n_P + n_Q - 2(r_P + r_Q + s_P + s_Q + t_P + t_Q) - (i + j + k)\,, \tag{21}$$

where

$$\begin{aligned} g_x &\equiv g_{l_P,l_Q,r_P,r_Q,i}(l_A, l_B, \vec{A}_x, \vec{B}_x, \vec{P}_x, \gamma_P; l_C, l_D, \vec{C}_x, \vec{D}_x, \vec{Q}_x, \gamma_P) \\ &= (-1)^{l_P} \cdot \theta(l_P, l_A, l_B, \vec{PA}_x, \vec{PB}_x, r_P, \gamma_P) \cdot \theta(l_Q, l_C, l_D, \vec{QC}_x, \vec{QD}_x, r_Q, \gamma_Q) \\ &\times \frac{(-1)^i (2\delta)^{2(r_P+r_Q)} (l_P + l_Q - 2r_P - 2r_Q)!\delta^i \vec{PQ}_x^{\,l_P+l_Q-2(r_P+r_Q+i)}}{(4\delta)^{l_P+l_Q} i! [l_P + l_Q - 2(r_P + r_Q + i)]!}\,, \end{aligned} \tag{22}$$

$$\theta(l, l_A, l_B, a, b, \gamma) = c_l(l_A, l_B, a, b) \frac{l!\gamma^{r-l}}{r!(l-2r)!}\,. \tag{23}$$

3. For anything but simple, light molecules such as H_2, NumPy will produce some hard-to-read output when you print the massive electron-electron repulsion integral four-dimensional matrix. Here are a few things you can add to the beginning of your code to improve the appearance of printed matrices:

```
np.set_printoptions(threshold=np.inf) # suppresses "...", printing all elements
np.set_printoptions(precision=5) # reduces number of decimal places per element to 5
np.set_printoptions(linewidth=200) # lengthens printable line-width from 75 characters
np.set_printoptions(suppress=True) # suppresses hard-to-read exponential notation
```

4.8 Storing integrals

The process of calculating molecular integrals should be initiated from `main.py` by calling the appropriate functions for building the atomic orbital basis from `basis.py`. The basis can then be passed into `oei.py` to calculate **S**, **T**, and **V**, and `eri.py` to calculate **G**.

It can take several minutes to calculate these molecular integrals. The electron-electron repulsion integrals are particularly time-consuming. It is therefore worth saving the generated matrices to files that can later be opened, such as for when you're coding and debugging the Hartree–Fock procedure. (You don't want to have to re-compute the integrals for water every time you test your Hartree–Fock code.)

Let's say you had allocated a variable `fname` at the beginning of your code that holds the characters of the input file name preceding the `.xyz` extension – e.g., `h2o` from `h2o.xyz`. You can use `fname` to save the `S` NumPy matrix for the overlap integrals of water:

```
S_file = open('S.' + fname + '.npy','w')
np.save(S_file,S)
```

The `.npy` is important for NumPy's recognition of the file. In the future, then, you can easily load this saved NumPy matrix with

```
S = np.load('S.h2o.npy')
```