Handout 3:
Essential Coding Skills for Implementing Molecular Integrals

A programming language of choice for both entry-level and many expert programmers is Python. This language is not only powerful and flexible, it is also relatively easy to learn. For these reasons we will use it for the programming projects, including the Guided Exercises at the end of the present document.

If you do not already have Python3 installed go to `www.python.org`; from the Downloads page, select `Python3` for your operating system. Python3 is the version of Python employed in the programming projects. The installation procedures differ depending on operating system, as do specifics of writing and running the Python `.py` code files. Generally, in any operating system, the files can either be written with a simple text editor, or by using a software interface called an Integrated Development Environment (IDE), which allows one to easily write, debug, and execute code all in one program; enter an Internet such query such as "download and install IDE for linux." You will also need to install the NumPy and SymPy libraries; enter an Internet search query such as "download and install numpy for windows."

Alternatively, there are pre-compiled packages that include the most common additional libraries of Python (such as NumPy and SymPy) along with an IDE. Winpython is an easy-to-use distribution for Windows. Anaconda is a similar, though more complex, distribution that has versions for Windows, MacOS, and Linux. Both Winpython and Anaconda include the IDE Spyder.

As a first program, create a new file called `myfirstprogram.py`, and then in either your text editor or IDE type in the following line of code:

```
print('Hello, world!')
```

Run `myfirstprogram.py` by clicking the "Run" button in your IDE. Alternatively, double click on the saved file from a folder. If you work in a terminal, then enter the following at the `$` prompt.

```
$ python myfirstprogram.py
```

You should see

```
Hello, world!
```

printed to your screen. Congratulations, you have coded a complete, functioning program!

Now things get a little more complicated with it comes to implementing molecular integrals. However, we will present only the minimum, *essential* features of Python that you need to complete the programming projects. There are many techniques and functionalities of Python that would allow one to complete the project in a more sophisticated manner (tuples, object-oriented programming, etc.); the reader familiar with such techniques is encouraged to employ them.

3.1 Introducing the Python Types, Nested Arrays, and Printing

For the molecular integrals and Hartree–Fock programming projects, you will need to store different kinds of data. Python provides several types of data. The data types that we will be concerned with, and some examples of each, are:

- **Integer**: Numbers with no decimal point.

  `3, -2, 118`

- **Float**: Numbers with a decimal point.

  `-1.00, 3.14159, 6.022e23, 6.626e-34`

- **String**: Non-numerical characters. (Notice that `'1234'` is a string and not an integer, because it is in quotes.)

  ```
  'H', 'Li, '1234'
  ```

- **Dictionary**: In curly brackets, a dictionary is made up of **keys** and **values** separated by a colon.

  ```
  {'H': 1.0094, 'He':  1.9998, 'Li':  3.001}
  ```

- **Array**: A collection marked by square brackets, made up of **elements** separated by commas.

  ```
  [0.0004, 1.0, 0.34523425]
  ```

The fact that arrays themselves are types means that arrays can be **nested** within arrays, like so:

```
[[0.00000, 0.00000, 0.45403],[0.0000,2.70672,-1.81610],[0.00000,-2.70672,-1.81610]]
```

This **nested array** is the set of x, y, z coordinates of water, as shown in the input `.xyz` file in Fig. 1. The first element of the outer array is itself an "inner" array made up of individual elements giving the x, y, and z positions (in units Bohr), respectively, of the oxygen atom; the second element is an array made up of the x, y, and z positions of the first hydrogen atom; the third element contains an array for the x, y, z coordinates of the second hydrogen atom.

```
3

O      0.00000     0.00000     0.22700
H      0.00000     1.35300    -0.90800
H      0.00000    -1.35300    -0.90800
```

Figure 1: The first line of an `.xyz` file is the number of atoms present in the input molecule. The second line is either empty or holds text, such as `these are the coordinates of water in units Bohrs`. The remaining lines feature a different atom on each line, where the first column contains the atom's atomic symbol, followed by the x, y, and z coordinates, respectively, of that atom's position.

Each of these described data types can be assigned to **variables**. For example, to **declare** an array-assigned variable `d`, that holds nested arrays of contraction coefficients that form part of the STO-3G basis set, the code would be:

```
# STO-3G contraction coefficients pulled from EMSL; applies to all atoms
d = [[0.44463454,0.53532814,0.15432897], [0.70011547,0.39951283,-0.09996723], [0.39195739,0.60768372,0.15591627]]
```

We have added a **comment** to a line before the array. Any code or text that appears after a pound sign is inactive. This is very useful for writing notes to remind yourself what certain code means. (It may be obvious what the code means to you now, but come back to it in two weeks and you may be scratching your head.) Here we remind ourselves that `d` is the nested array holding contraction coefficients, which apply to all atoms. We can also reformat the code like so:

```
# STO-3G contraction coefficients pulled from EMSL; applies to all atoms
d = [[0.44463454, 0.53532814, 0.15432897],   # contraction coefficients for 1s orbitals
     [0.70011547, 0.39951283,-0.09996723],   # contraction coefficients for 2s orbitals
     [0.39195739, 0.60768372, 0.15591627]]   # contraction coefficients for 2p orbitals (x, y, or z)
```

Parentheses or brackets permit breaking up one continuous line of code over several lines, which aids visualization. Here we have also reminded ourselves, with comments, that the first element of the outer array is the three contraction coefficients for the 1s orbitals, that the second element of the outer array contains the contraction coefficients for the $2s$ orbitals, and so on.

You can **print** a string of the second element of the `alpha` array by including in your code the following line:

```
print(d[1])
```

Running this Python program will then display

```
[0.70011547, 0.39951283,-0.09996723]
```

Note that Python uses **zero-based indexing**: It counts beginning at the number 0, rather than the number 1. Hence, `d[1]` is actually the second element of `d`. You can also write

```
print(d[1,2])
```

to print just the third contraction coefficient of the 2*s* orbitals.

We can generate a more detailed, user-friendly output by adding together strings in the `print` command argument (where the argument is content within parentheses).

```
print("The third contraction coefficient for a 2s orbital is: " + str(d[1,2]) )
```

Because we coupled a string (`'The third contraction coefficient ...'`) with data that is not of string-type (the array `d`), we needed to add the `str` keyword around `d[1,2]`. Running this program will then display

```
The third contraction coefficient for a 2s orbital is: -0.09996723.
```

When building the atomic orbital basis, regular use will made be of *dictionaries*, one of the Python types introduced earlier. We will create a dictionary that converts atomic symbols (which appear at the beginning of all but the first two lines of `.xyz` input files; see Fig. 1) to atomic numbers:

```
Z = {'H':1,'He':2,'Li':3,'Be':4,'B':5,'C':6,'N':7,'O':8,'F':9,'Ne':10}
```

Here, the key `'Li'` has a value of 3. If one wished to create an integer variable called `atnumBoron` holding the atomic number of boron, we would write the following code:

```
atnumBoron = Z['B']
```

Try printing `atnumBoron`, and confirm it gives the atomic number of boron.

Python is not generally sensitive to spacing. For example,

```
d = [[0.44463454, 0.53532814, 0.15432897],   # contraction coefficients for 1s orbitals
     [0.70011547, 0.39951283,-0.09996723],   # contraction coeffcieints for 2s orbitals
     [0.39195739, 0.60768372, 0.15591627]]   # contraction coefficients for 2p orbitals (x, y, or z)
```

functions the same as

```
d   =  [
          [0.44463454,0.53532814,0.15432897],  [0.70011547,0.39951283,-0.09996723],
       [0.39195739,0.60768372,0.15591627]
  ]
```

One important exception to Python's non-restrictive treatment of spacing is found in **functions**, which will be discussed later.

3.2 Handling Matrices with NumPy

While it is possible to use a Python two-dimensional array (or nested array) as a matrix, matrices and matrix operations are most easily and powerfully handled with the NumPy library. Computational quantum chemistry is replete with matrices: the overlap and one-electron integral matrices are two-dimensional matrices; the electron-electron repulsion integral matrices are of four dimensions. To use NumPy (which needs to have been installed on your computer), the following line is added to the top of a code file:

```
import numpy as np # import the NumPy library
```

By importing NumPy, we can also calculate exponential functions and use a precise value of π. For example, to calculate the value e^{π} would be `np.exp(np.pi)`.

Here is how to create a two-dimensional matrix. The following simple example features the number 1 along the diagonal, with all non-diagonal elements being equal to 0.

```
# construct a matrix with diagonal elements = 1
mymatrix = np.matrix( [[1,0,0],[0,1,0],[0,0,1]] )

print(mymatrix)
```

Running this code will display

```
[[1 0 1]
 [0 1 0]
 [0 0 1]]
```

Typically, we will create an empty matrix with dimensions that are determined by the number K of atomic orbitals present in the basis. For water, the minimal basis configuration describes

$$K = \phi_{\mathrm{H1s}} + \phi_{\mathrm{H1s}} + \phi_{\mathrm{O1s}} + \phi_{\mathrm{O2s}} + \phi_{\mathrm{O2p}_x} + \phi_{\mathrm{O2p}_y} + \phi_{\mathrm{O2p}_z} = 7$$

atomic orbitals in the basis. Thus, the overlap integral matrix $\mathbf{S}$ is $K \times K = 7 \times 7$ in dimension. The corresponding code for generating an empty `S` matrix of $K \times K$ dimension is

```
K = 7 # the number of atomic orbitals in the basis
S = np.zeroes((K,K)) # create an empty 7x7 matrix to be filled in later
```

The elements of `S` can be accessed by loops and modified by functions, which we shall do later in this document. The two-electron integrals make up a rank-4 tensor of $K \times K \times K \times K$ dimensions:

```
# K was already set to 7, so the following line will create a 7x7x7x7 tensor filled with zeros
G = np.zeros((K,K,K,K))
```

For an even simpler molecular system like H_2, for which 2 atomic orbitals ($K = 2$) form the basis when using a minimal basis set like STO-3G, then

```
import numpy as np # import the NumPy library

K = 2 # the number of atomic orbitals in the basis
S = np.zeroes((K,K)) # create an empty 2x2 matrix to be filled in later
G = np.zeros((K,K,K,K)) # create a rank-4 tensor, 2x2x2x2

print(S)
```

gives

```
[[0 0]
 [0 0]]
```

whereas `print(G)` gives

```
[[[[ 0.  0.]
   [ 0.  0.]]

  [[ 0.  0.]
   [ 0.  0.]]]


 [[[ 0.  0.]
   [ 0.  0.]]

  [[ 0.  0.]
   [ 0.  0.]]]]
```

We will now discuss how to access individual matrix elements, so that we can calculate their values using the proper formulas.

3.3 Functions and Loops

Loops allow us to perform repetitive tasks without having to repeat code, which can be error-prone. A simple example of using a loop is having our quantum chemistry program calculate the number of electrons present in a neutrally-charged molecule. (Remember that the orbitals of our minimal basis provide the *space* for electrons to occupy, and orbitals are used in the molecular integral calculations. It is in the Hartree–Fock method that electrons are effectively "inserted" into these atomic orbitals, which are then mixed to form molecular orbitals. The Hartree–Fock procedure depends on the number of electrons to form the molecular orbitals from the atomic orbitals.) We can calculate the number of electrons assuming a neutrally charged molecule, such that the number of electrons is simply the sum of atomic numbers.

We earlier created a dictionary `Z` containing *keys* that are atomic symbols (strings), and *values* that are atomic numbers (integers). Assume that we have an array called `atoms`, the array's elements of which are the atomic symbol strings for each atom present in the input molecule. For example, having read the input file for water, shown in Fig. 1, the `atoms` array would be

```
atoms = ['O','H','H']
```

This array would be automatically constructed while the input file is being read in by the code. (Reading input files will be addressed in the Guided Exercises.)

We can take advantage of the fact that each element of `atoms` is a string that corresponds to some key within `Z`. We can then use the value corresponding to that key, which is the atomic number integer, to sum the total number of electrons. This is shown in the following code.

```
atoms = ['O','H','H']

n = 0 # this integer variable will hold the sum of the atomic numbers
for A in atoms:
  n = n + Z[A] # add to whatever n already equals the atomic number of index A
print(n)
```

While `n = n + Z[A]` is of course not algebraically correct, in computer code it serves to *update* the variable `n`.

Notice that the code within the for loop is indented; this is necessary to designate those lines which are part of the loop, and separate it from those that are not. The first line following the for line that is not indented (here, the `print` statement) then lies *outside* of whatever task the loop is executing. (See for yourself what happens when the print statement is included inside the loop versus when it is outside the loop.)

For the current step in the loop through each element of `atoms`, `A` is a variable *equal* to that element. So in the first loop through atoms, `A = 'O'`. Thus, `Z[A]` is equivalent to saying `Z['O']`, and this returns the integer 8. The integer 8 is added to `n` (which initially equals 0), and then the next cycle is begun. `A` now equals `'H'`, and so `Z['H']` returns 1, and this is added to `n`, for a total of 9. The final step in the cycle adds the remaining 1 to `n`. With `atoms` now exhausted, the for loop ends, having determined for us that the number of protons (and hence the number of electrons) in our neutrally charged molecule is 10. This is what would be `printed` to the computer screen.

Here is another, slightly different way of using a loop. We will often need to compute the distance between two atoms. Let `R` be a nested array holding the coordinates of the H_2 molecule: Element one for the first hydrogen atom, and element two for the other hydrogen. (Normally, `R` will automatically be created while the `.xyz` input file is being read in, as would `atoms`. For now we assign the coordinates directly.)

```
# define coordinates of the H2 molecule
R = [[0.0, 0.0,-0.8],
     [0.0, 0.0, 0.8]]

RA = R[0] # coordinates of the first hydrogen
RB = R[1] # coordinates of the second hydrogen
```

In this code we also **instantiated** (or "created") the variables `RA` and `RB` to separately hold the coordinates of the individual atoms, which will make this demonstration a little clearer. As we saw in Handout 1, the equation for the squared distance between orbitals ϕ_A (centered on atom A at position $\vec{A}$) and ϕ_B (centered at $\vec{B}$), is

$$|\vec{AB}|^2 = |\vec{A} - \vec{B}|^2 = (\vec{A}_x - \vec{B}_x)^2 + (\vec{A}_y - \vec{B}_y)^2 + (\vec{A}_z - \vec{B}_z)^2 \,. \tag{1}$$

One straightforward way of computing this would be

```
ABsquared = (RA[0]-RB[0])**2 + (RA[1]-RB[1])**2 + (RA[2]-RB[2])**2
```

Recall that `RA[0]` would be the x-coordinate of atom A, `RA[1]` the y-coordinate, and `RA[2]` the z-coordinate; likewise for `RB`. `ABsquared` is a variable that holds a float, a number that is the result of using Eq. 1 as implemented in the code. The distance between $\vec{A}$ and $\vec{B}$ for each coordinate is in parentheses, and the squaring of each coordinate difference is achieved with `**2`. (Cubing can be achieved with `**3`, finding the square root with `**(1/2)`, and so on.)

Alternatively, we could employ a for loop.

```
ABsquared = 0.0 # instantiate the variable ABsquared, its quantity to be calculated in a for loop
for i in range(0, 3):
  ABsquared = ABsquared + (RA[i]-RB[i])**2
```

What distinguishes this for loop from our previous example is that the index `i` is simply a counting, integer variable, rather than being equal to an *element* of an array. Thus, in the range from 0 up to (but not including!) 3, `i` will be equal to: `0` in the first pass, and then `1` in the second pass, and end with `2`. If we didn't know that the "length" (the number of elements) of the array was 3, we could have also written

```
ABsquared = 0.0 # instantiate the variable ABsquared, its quantity to be calculated in a for loop
for i in range(0, len(RA)):
  ABsquared = ABsquared + (RA[i]-RB[i])**2
```

We're assuming here that `len(RA)` – that is, the *length* of `RA` – is equal to `RB`.

Finally, for loops also allow us to systematically access the elements of a matrix (as of **S**, **T**, **V**, and **G**). To demonstrate this, we will prepare a simple program that computes the overlap integrals for H_2 using the STO-1G basis set. Before showing how loops allow us to access matrix elements, we first consider some details of what the program will entail, and discuss **functions**.

As Fig. 2 below shows, the overlap matrix will have four matrix elements: the overlap of orbital ϕ_A (on one hydrogen) with itself; the overlap of orbital ϕ_A with orbital ϕ_B (which is on the other hydrogen); the overlap of orbital ϕ_B with orbital ϕ_A (this will give the same value as the overlap of orbital ϕ_A with orbital ϕ_B); and the overlap of orbital ϕ_B with itself. These integrals correspond to specific values of the **S** (or coded `S`) matrix.

Because orbitals are normalized, we should expect that the overlap integrals of orbitals with themselves (which make up the diagonal elements of the matrix) equal 1; the rest will be some value between, or equal

	ϕ_A	ϕ_B
ϕ_A	$\int \phi_A \phi_A \, \mathrm{d}\tau$	$\int \phi_A \phi_B \, \mathrm{d}\tau$
ϕ_B	$\int \phi_B \phi_A \, \mathrm{d}\tau$	$\int \phi_B \phi_B \, \mathrm{d}\tau$

$\equiv$

	ϕ_A	ϕ_B
ϕ_A	`S[0,0]`	`S[0,1]`
ϕ_B	`S[1,0]`	`S[1,1]`

Figure 2: The 2×2 overlap integral matrix $\mathbf{S}$ for the 1s orbital ϕ_A on hydrogen atom A, and the $1s$ orbital ϕ_B on hydrogen atom B.

to, 0 and 1. Knowing these facts will form a good basic check test of our code. In the STO-1G basis set for hydrogen, there is only one contraction coefficient $\mathrm{d}_p = 1.0$, and only one $\alpha_p = 0.28294$.

The equation for an overlap integral matrix element of two identical, $1s$ orbitals can easily be derived from Eqs. 28 and 29 of Handout 1. It is

$$S_{AB} = N^2 \times \exp\left[-\left(\frac{\alpha^2}{2\alpha}\right)|\vec{AB}|^2\right] \times \left(\frac{\pi}{2\alpha}\right)^{3/2}, \tag{2}$$

where the contraction coefficient $\mathrm{d}_p = 1.0$ is implied. The normalization constant N is

$$N = \left(\frac{2\alpha}{\pi}\right)^{3/4}. \tag{3}$$

We discussed earlier the mathematical form of $|\vec{AB}|^2$, and how to code it.

Let's define a **function** that allows us to call the calculation of `ABsquared` at any time, so that if we need to we can use it more than once. We will then use this function to find the distance between the two atoms specified in `R`. Finally, we will print the result.

```
def compute_ABsq(RA, RB):
  """
  This function computes the square of the distance between two atoms, or two atom-centered orbitals.
  It follows Eq. 1 of Handout 2.
  RA and RB are arrays of the coordinates of atoms (or orbitals) A and B, 3 elements each.
  """
  ABsquared = 0.0
  for i in range(len(RA)): # equivalent to range(0, len(RA))
    ABsquared = ABsquared + (RA[i]-RB[i])**2
  return ABsquared

H2distance = compute_ABsq(R[0], R[1]) # equivalent to passing RA and RB to the function

print("Distance between the two atoms of dihydrogen = " + str(H2distance) )
```

We created the function with the "define function" keyword `def`. The keyword is followed by what we want to call the function, `compute_ABsq`. We then say the function has two **arguments**, `RA` and `RB`, which are passed *in* to the function for the function to use.

We also have an expanded comment type, created by blocking text between three quotation marks. Here the comment indicates that `RA` and `RB` are both expected to be arrays; if they are not, the function will return an error when running the program. For example, if we accidentally pass in a float instead of an array, we would get the error

```
TypeError: 'float' object is not subscriptable
```

That is, it has encountered a Python data type error. If, on the other hand, we do pass in an array but it doesn't have enough elements for what the loop "expects," we will get the following error:

```
Index Error: list index out of range
```

A comment after the first line of the for loop indicates that the `0` at the beginning of `range` isactually zero by default, and hence not necessary to explicitly include. Of course, any lower bound of a range other than 0 must be explicitly declared.

Just as with a loop, every line that's indented after `def compute_ABsq(RA, RB):` is a part of the function. When the text returns to non-indented form, it is outside the function. The line `H2distance = compute_ABsq(R[0], R[1])` is outside the function, and is *calling* the function `compute_ABsq`. It is passing into `compute_ABsq` the positions of the first hydrogen atom and the second hydrogen we defined for our H_2 molecule in `R`, which you will recall is a nested array. What is **returned** by the function (`ABsquared`) is stored in the variable `H2distance`. The print statement results in the display

```
Distance between the two atoms of dihydrogen = 1.6
```

when the code is run.

Similarly, the functions for computing the normalization constant and the overlap integral would be

```
def compute_norm(alphaconstant):
  """
  Calculate and return the normalization constant N, according to Eq. 3 of Handout 2.
  """
  N = (2*alphaconstant / np.pi)**(3/4)
  return N

def compute_overlap(a,RA,RB):
  """
  Calculate a matrix element of the S matrix by calling on compute_norm and compute_ABsq.
  """
  Selement = (compute_norm(a))**2 * np.exp(-(a**2 / (2*a)) * compute_ABsq(RA,RB)) * (np.pi / (2*a))**(3/2)
  return Selement
```

This code reveals that one can call another function from *within* a function. In the function `compute_overlap` we call `compute_norm`, passing into it whatever value for `alpha` we have. We also call the distance function `compute_ABsq`, passing into it the x, y, z coordinates of atom A (in the form of the array `RA`) and the x, y, z coordinates of atom B (array `RB`). Note that the variable name for α is `alphaconstant` in the `compute_norm` function, but `a` in the `compute_overlap` function. This is perfectly acceptable, as long as the same variable name is used *within* a function. Make sure that you can see the direct connections between this code and Eqs. 2 and 3.

It only remains to fill the empty matrix `S` using these functions. We will do so with a loop. Let's first create a simple matrix other than `S` for demonstration, and call it `mymatrix`. Assuming we've imported NumPy, the code

```
mymatrix = np.matrix([[1,2],[3,4]])
print(mymatrix)
```

gives

```
[[ 1 2 ]
 [ 3 4 ]]
```

If we apply a single loop to `mymatrix`, and have an integer variable called `counter` to keep track of which loop cycle we are in, then

```
counter = 1 # track where we are in the loop
for A in range(len(mymatrix)): # lower bound of 0
  print("Loop " + str(counter) + ": mymatrix[" +str(i) + "] = " + str(mymatrix[i]))
  counter = counter + 1
```

Notice that we set the upper bound on our range to the length of the matrix, which would be just one dimension of the matrix (the number of columns). This is okay, and it will be for all of our molecular integral matrices, because they are square, $K \times K$ matrices (the number of rows is equal to the number of columns). The display of this computation would be

```
Loop 1: [[ 1.  2.]]
Loop 2: [[ 3.  4.]]
```

This output would be fine if we wanted entire rows of matrices. But usually we want individual matrix elements. To access elements, we must **nest** loops.

```
mymatrix = np.matrix([[1,2],[3,4]])

counter = 1 # track where we are in the nested loop
for A in range(len(mymatrix)):
  for B in range(len(mymatrix)):
    print("Loop" + str(counter) + ": mymatrix[" + str(A)+ "," + str(B) + "] = " + str(mymatrix[A,B]))
    counter = counter + 1
```

As you can see, deeper indenting is required when nesting loops: one indentation per loop. In this looping scheme, the row of `mymatrix` is accessed with index `A`, the first row being `A = 0`. Then *within* this row, each *element* of row `A` is accessed with index `B`. A matrix element corresponding to row `A` and column `B`, then, is specified by `mymatrix[A,B]`. Once every possible element (or "column") `B` is accessed within row `A`, then the next row (`A + 1`) is accessed. Thus, the output is

```
Loop 1: mymatrix[0,0] = 1.0
Loop 2: mymatrix[0,1] = 2.0
Loop 3: mymatrix[1,0] = 3.0
Loop 4: mymatrix[1,1] = 4.0
```

Additionally, we can perform tasks at different levels of the nested loop. So, if we wanted to print that the end of a row of `mymatrix` was reached, we would do so by writing code at the appropriate level of indentation, that of the "outer" loop.

```
counter = 1 # track where we are in the nested loop
for A in range(len(mymatrix)):
  for B in range(len(mymatrix)):
    print("Loop " + str(counter) + ": mymatrix[" + str(A)+ "," + str(B) + "] = " + str(mymatrix[A,B]))
    counter = counter + 1
  print("The end of row " + str(A+1) + " has been reached.") # A+1 to account for zero-based indexing
```

The output for this code is

```
Loop 1: mymatrix[0,0] = 1.0
Loop 2: mymatrix[0,1] = 2.0
The end of row 1 has been reached.
Loop 3: mymatrix[1,0] = 3.0
Loop 4: mymatrix[1,1] = 4.0
The end of row 2 has been reached.
```

To access elements of an N-dimensional matrix, we require the nesting of N loops: two loops for a two-dimensional matrix, one loop for a one-dimensional matrix (or vector), four loops for a four-dimensional matrix (or rank-4 tensor), and so on.

Now that we've demonstrated how to acess matrix elements, we will apply this technique to filling in our empty `S` overlap integral matrix. Recalling that hydrogen has one $\alpha = 0.28294$ in the STO-1G basis set, and that we already constructed a function for computing a matrix element, then

```
H1salpha = 0.28294 # exponential factor for hydrogen 1s orbital in a STO-1G basis
for A in range(len(S)):
  for B in range(len(S)):
    S[A,B] = compute_overlap(H1salpha,R[A],R[B])
```

First, the overlap when `A = B = 0` is calculated: This is the overlap of ϕ_A with itself. Then, with `A` unchanged, `B` is *incremented* to 1, and the overlap of ϕ_A with ϕ_B is calculated. `B`'s length does not go beyond 1, so then `A` is incremented by 1 (to `A = 1`) and `B` is again at 0: The overlap of ϕ_B with ϕ_A is calculated. Finally, with `A` still at `A = 1`, `B` is incremented to `1`, such that `A = B = 1` and the overlap of ϕ_B with itself is computed.

Let's put this discussion in context by presenting all of the code constituting a complete program that computes the overlap integral matrix of dihydrogen with the STO-1G basis set. Dropping some of the intermediate print statements we included for demonstration earlier, this complete program would be

```
import numpy as np

R = [[0,0, 0.0,-0.8],
     [0.0, 0.0, 0.8]]

def compute_ABsq(RA, RB):
  """
  This function computes the square of the distance between two atoms, or two atom-centered orbitals.
  It follows Eq. 1 of Handout 2.
  RA and RB are arrays of the coordinates of atoms (or orbitals) A and B, 3 elements each.
  """
  ABsquared = 0.0 # instantiate the variable ABsquared, its quantity to be calculated in a for loop
  for i in range(len(RA)):
    ABsquared = ABsquared + (RA[i] - RB[i])**2
  return ABsquared

def compute_norm(alphaconstant):
  """
  Calculate and return the normalization constant N, according to Eq. 3 of Handout 2.
  """
  N = (2*alphaconstant / np.pi)**(3/4)
  return N

def compute_overlap(a,RA,RB):
  """
  Calculate a matrix element of the S matrix by calling on compute_norm and compute_ABsq.
  Follows Eq. 2 of Handout 2.
  """
  S = (compute_norm(a))**2 * np.exp(-(a**2/(2*a)) * compute_ABsq(RA,RB)) * (np.pi/(2*a))**(3/2)
  return S

H1salpha = 0.28294

S = np.zeroes((2,2)) # create an empty 2x2 S matrix

# access and calculate each matrix element
for A in range(len(S)):
  for B in range(len(S)):
    S[A,B] = compute_overlap(H1salpha,R[A],R[B])

print("Overlap integral matrix (S) =\n" + str(S))
```

The `\n` in the print statement pushes whatever comes next to the following line, so that the printout of this code looks like

```
Overlap integral matrix (S) =
[[ 1.          0.91343706]
 [ 0.91343706  1.        ]]
```

These are the overlap integrals for H_2 in the STO-1G basis, and can be represented as shown in Fig. 3.

As expected, the diagonal elements representing self-overlap are equal to exactly 1. All values of this matrix can be considered as percentages of overlap, so that while an orbital obviously overlaps 100% with

$$\begin{array}{|c|c|c|}\hline & \phi_A & \phi_B \\ \hline \phi_A & \int \phi_A \phi_A \, \mathrm{d}\tau & \int \phi_A \phi_B \, \mathrm{d}\tau \\ \hline \phi_B & \int \phi_B \phi_A \, \mathrm{d}\tau & \int \phi_B \phi_B \, \mathrm{d}\tau \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline & \phi_A & \phi_B \\ \hline \phi_A & 1.000 & 0.913 \\ \hline \phi_B & 0.913 & 1.000 \\ \hline \end{array}$$

Figure 3: The 2×2 overlap integral matrix $\mathbf{S}$ for the dihydrogen molecule, with a STO-1G basis set, and interatomic distance of 1.6 a_0.

itself, one hydrogen $1s$ orbital overlaps with 91.3% of the other hydrogen's $1s$ orbital. This high degree of overlap is due to the identical orbital symmetry, as well as the relative proximity of the two atoms. (Atomic orbitals that are farther apart and have different orbital angular momentum will overlap less.))

3.4 The Final Skill: Conditional Logic (If Statements)

Most of the molecular integrals programming project is dedicated to accessing matrix elements with loops (as shown earlier) and calling on functions (also discussed earlier) to execute some task that is frequently used, like calculating a matrix element. There is one final tool we need in our programming toolkit to implement all of the molecular integrals.

If statements are a powerful means of giving the programmer greater control over their code. It allows them to execute line(s) of code only under the *condition* that some criterion is met: *If* something is true, *then* perform this action. (By implication, if that something is *not* true, then don't perform the action.) If statements are one kind of **conditional logic** that programmers frequently use.

There are only a few cases where we will use if statements in our programming projects. Here is one example. Recall from the discussion of the Hartree–Fock method in Handout 1 that it is iterative: The molecular orbital coefficients $\mathbf{C}$ are tweaked step-by-step (in a loop) until the *change* in total energy of the molecular system drops below some threshold. The energy changes usually get smaller with every step. We call this threshold, where the change in energy is sufficiently small, `convergence`. Usually it is set to a very small number like 1.0×10^{-7} Hartrees.

```
def compute_HF(electroncount, basissize, S, T, V, G):
  """
  Calculate electronic energy iteratively by the Hartree-Fock method.
  Depends on number of electrons, the number of orbitals in the basis (basissize), and the molecular integrals.
  """

  # code to set up HF iteration as a for loop

  convergence = 1.0e-7
  for iteration in range(0, 20): # stop looping after 20 iterations if HF doesn't converge

    # code to variationally optimize the molecular orbitals, and calculate electronic energy Eel

    if abs(Eel - old_Eel) < convergence:
      break
```

You will learn in the Hartree–Fock programming project (Handout 4) all of the details that go into this code, including what goes into `# code to ...` We are presently concerned with just two lines:

```
    if abs(Etot - old_Etot) < convergence:
      break
```

These two lines of code command the program to "break" out of the Hartree–Fock for loop *if* the absolute difference in energy between the present iteration energy (`Eel`) and the previous iteration energy (`old_Eel`) is less than 1.0×10^{-7} Hartrees. By implication, if the change in energy is greater than or equal to 1.0×10^{-7} Hartrees, continue looping.

Here is another important example. At the beginning of the programming project, our (minimal) atomic orbital basis will be built by passing through the list of atoms (which are represented by strings, *e.g.* `'C'`) specified in the `atoms` array, drawn from the input `.xyz` file. For each atom, we **append**, or add, to an array called `sto3Gbasis`, where each element is an "atomic orbital" in the form of a dictionary. The dictionary will contain the orbital's atom-centered coordinates, the atomic number of the atom it's centered on, the orbital angular momentum (p_y implies $l = 0, m = 1, n = 0$), and the orbital's basis set values for α_p and d_p.

Because each dictionary contains all of the orbital information we need for calculating molecular integrals, each orbital is effectively represented by this dictionary. Each atom is assigned orbitals by its atomic orbital “basis configuration,” which we discussed in Handout 1 and will now briefly re-define.

Any atom in the first row of the periodic table has the (minimal) atomic orbital basis configuration of, simply, $1s$ – just one orbital is available for the 1 or 2 electrons of hydrogen (electron configuration $1s^1$) or helium (electron configuration $1s^2$) to occupy. Any atom in the second row of the periodic table is endowed with orbitals according to the the minimal basis configuration $1s\,2s\,2p_x\,2p_y\,2p_z$, whether it be lithium (which has the ground state electron configuration $1s^2 2s^1$), or carbon (which has the ground state electron configuration $1s^2 2s^2 2p^2$), or neon ($1s^2 2s^2 2p^6$).

Basis configurations can be associated with particular atoms using another dictionary we will call `orbitalconfiguration`.

```
# minimal orbital basis configurations through the second row
orbitalconfiguration = {
  'H':['1s'],
 'He':['1s'],
 'Li':['1s','2s','2px','2py','2pz'],
 'Be':['1s','2s','2px','2py','2pz'],
  'B':['1s','2s','2px','2py','2pz'],
  'C':['1s','2s','2px','2py','2pz'],
  'N':['1s','2s','2px','2py','2pz'],
  'O':['1s','2s','2px','2py','2pz'],
  'F':['1s','2s','2px','2py','2pz'],
 'Ne':['1s','2s','2px','2py','2pz'],
 }
```

By using the dictionary `orbitalconfiguration`, we can append orbitals to an array representing our basis (`sto3Gbasis`) as each atom of the array `atoms` is accessed during the loop. As described earlier, the orbitals themselves are constructed as dictionaries containing all of the relevant orbital information, such as orbital angular momentum and basis set values.

```
# the basis will be constructed as an array;
# it is initially empty, but built up as orbitals (dictionaries) are appended to it in a for loop
sto3Gbasis = []

# looping through the atoms array, append orbitals (dictionaries) to the sto3Gbasis array,
#   where each element of the atoms array is termed atom;
# each atom has an orbital configuration associated with it, so loop through these too
for atom in atoms:
  for orbital in orbitalconfiguration(atom):
    if orbital == '1s':
      sto3Gbasis.append( {1s orbital dictionary} )  # add the atom's 1s orbital to basis
    elif orbital == '2s':
      sto3Gbasis.append( {2s orbital dictionary} )  # add the atom's 2s orbital to basis
    elif orbital == '2px':
      sto3Gbasis.append( {2px orbital dictionary} ) # add the atom's 2px orbital to basis
    elif orbital == '2py':
      sto3Gbasis.append( {2py orbital dictionary} ) # add the atom's 2py orbital to basis
    elif orbital == '2pz':
      sto3Gbasis.append( {2pz orbital dictionary} ) # add the atom's 2pz orbital to basis
```

The nature of orbitals as dictionaries will be fleshed out in the Guided Exercises. For now, the thing to pay attention to are the **if statements**. The code inside the `if` block will only be executed *if* the condition of `orbital == '1s'` is met. (Note that there are two equals signs, not just one. With two equals signs we're not *setting* `orbital` equal to the string value `'1s'`, rather we are seeing if it already *is* equal to the string value `1s`.) Sometimes one if statement is adequate, as we saw with the Hartree–Fock convergence condition. But in this case it is not, because we are interested in more than the condition that the orbital is $1s$. We are also interested in the possible condition that the orbital is $2s$, $2p_x$, and so on. For this, we follow `if` statements with `elif`, or “else-if”, statements, as shown.

3.5 Guided Exercises

In the following exercises you will be carefully guided through the beginning of the molecular integrals programming project, in which you will develop large parts of three of the five interacting `.py` files that will form the foundation of your quantum chemistry code. (It is certainly possible to have all of your code in one file, but management is considerably easier if it is broken up.) The five files we will prepare are:

1. `basis.py`: Basis set information and construction of the STO-3G basis (with the function `build_sto3Gbasis`) will be located here.You will develop and complete this code during these Guided Exercises.
2. `oei.py`: This will contain the code for calculating matrix elements of the one-electron integral matrices **S**, **T**, and **V**. You will be guided through coding the calculation of matrix elements of **S** during these Guided Exercises.
3. `eri.py`: The code for calculating matrix elements of the two-electron integrals that make up **G** will be in this file. This code will not be discussed here.
4. `hf.py`: This will contain the code for calculating the electronic energy, which depends on **S**, **T**, **V**, and **G**. Other than the Hartree–Fock example provided earlier in this document, we will not be discussing `hf.py` in these Guided Exercises.
5. `main.py`: This central code file will integrate, or "drive", the four other code files, beginning with reading in an input `.xyz` file. Then the basis will be built by accessing functions within `basis.py`. This is followed by constructing empty NumPy matrices, and calling on relevant functions in `oei.py` and `eri.py` to calculate the matrix elements. Finally, the matrices are passed into `hf.py`, where the total electronic energy of the molecule is calculated by the Hartree–Fock method. We will construct part of `main.py` in this set of exercises so we can build **S** and print the overlap integral matrix. (`h2o.xyz` as presented in Fig. 1 in this document will be our test case.)

Completing these exercises will provide practice in all of the coding techniques you will employ for the remainder of the project (building **T**, **V**, **G**, and computing the Hartee–Fock energy).

Throughout the following exercises, be sure to continually describe your code with detailed comments. Here are a couple *debugging* tips to keep in mind as you get started:

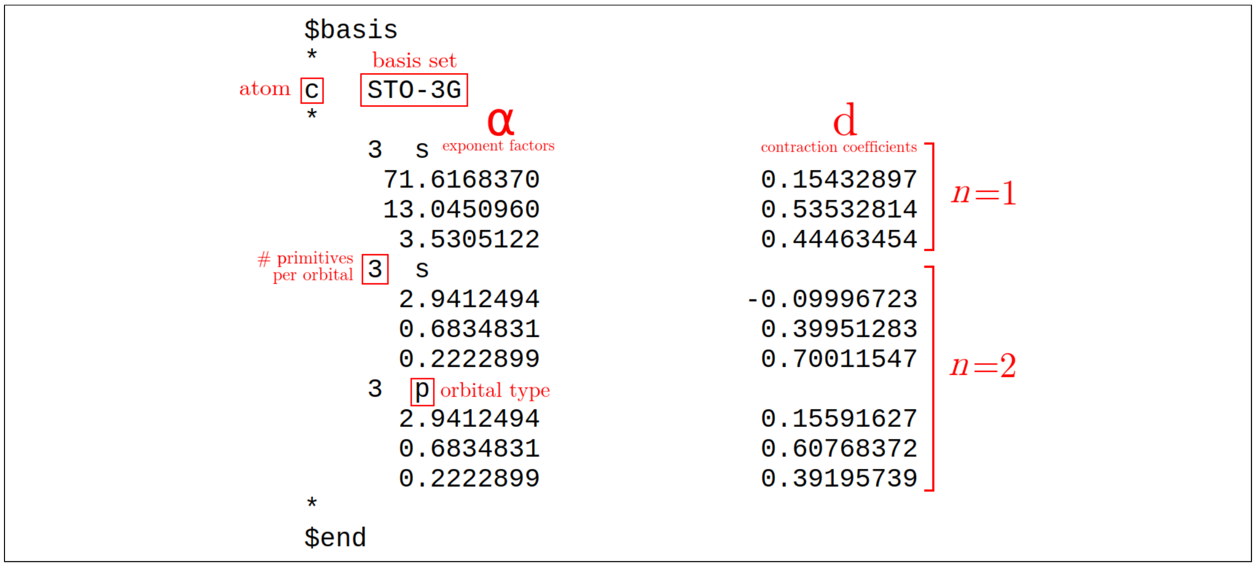
1. Errors you encounter can often be fixed by addressing a flagged line of bad code that Python reports when you try to run your code.
2. Be sure to regularly `print` any variables (numbers, matrices, *etc.*) that are formed during the development of your code, as this can help you infer where the offending code exists.

3.5.1 Building `basis.py`

1. In a directory that will hold all of your quantum chemistry code, create a file called `basis.py`.
2. In `basis.py`, we will not need a preamble to import any libraries. Type in the code to create a dictionary called `Z` that holds the atomic symbols (keys) and atomic numbers (values) of the first ten elements of the periodic table, as demonstrated earlier.
3. After an empty line following the code you wrote for the previous step, type in the necessary code to create a nested array called `d`, which will hold the STO-3G d_p contraction coeffcients for $1s$, $2s$, and $2p$ orbitals. This also was demonstrated earlier in this Handout. (Don't forget comments!)
4. Create the dictionary `orbitalconfiguration`, as described earlier.
5. Create a nested array featuring the α_p factors. This will involve not just one nesting within the outer array, as with `d`, but two nestings, because *each atom* has its own, unique set of α_p factors. (The set of contraction coefficients in a STO-NG basis apply to all atoms). Like the contraction coefficients, values for $2p$ apply to $2p_x$, $2p_y$, and $2p_z$. Note also that $2s$ and $2p$ orbitals have the same α_p factors. Here's part of this code:

```
alpha = [
          [
           [    0.16885540,    0.62391373,    3.42525091],    # H  1s
           [    0.00000000,    0.00000000,    0.0000000 ]     # H  2s,2p
                                                       ],
          [
           [    0.31364979,    1.15892300,    6.36242139],    # He 1s
           [    0.00000000,    0.00000000,    0.0000000 ]     # He 2s,2p
                                                       ],
          [
           [    0.7946505,     2.9362007,    16.1195750 ],    # Li 1s
           [    0.0480887,     0.1478601,     0.6362897 ]     # Li 2s,2p
                                                       ],
           # alpha coefficients for atoms between Li and Ne
          [
           [   10.7946506,    37.7081510,   207.0156100 ],    # Ne 1s
           [    0.6232293,     1.9162662,     8.2463151 ]     # Ne 2s,2p
                                                       ]
        ]
```

Fill in the values corresponding to the elements between lithium and neon, as directed in dark blue text. Go to `https://bse.pnl.gov/bse/portal`, select the elements you need, and then select `STO-3G` from the scrolling menu on the left side of the page. From the "Format:" drop down menu below the periodic table, select Turbomole, because this is the format we presented in the main article. Type in the α values you see on the website in to your `alpha` array; the `alpha` values are in the left column, as was demonstrated in Fig. 6 of Handout 1, and duplicated here for convenience.



```
$basis
*
C    STO-3G
*
   3  s
    71.6168370                0.15432897
    13.0450960                0.53532814
     3.5305122                0.44463454
   3  s
     2.9412494               -0.09996723
     0.6834831                0.39951283
     0.2222899                0.70011547
   3  p
     2.9412494                0.15591627
     0.6834831                0.60768372
     0.2222899                0.39195739
*
$end
```

Figure 4: An example of the structure of the tabulated STO–3G basis set for the carbon atom. Many table formats are found; this is "Turbomole" selected from the drop-down menu (with "Optimized General Contractions") of the EMSL Basis Set Exchange, with "carbon" highlighted in the web site's periodic table. The red text was added in this figure to highlight salient features of the table. n is the principal quantum number.

The contraction coefficients are in the right column, and you should confirm this by noting that they are the same as those you already typed in for the `d` array.

As you enter in the α values, take care to heed the order of the basis set numbers! Compare the numbers shown in the starting examples with those printed on the website. The numbers going left to right inthe `alpha` array are the same as those going bottom to top on the EMSL website. Double (and triple!) check that you enter every digit correctly, and that you have every closing bracket for every opening bracket. If there is a typo, it could lead to numerical errors later on that would be very difficult to trace back to a simple transcription error from this admittedly tedious task.

This would be a good time to try running your program. At this point it won't print anything, but if running it produces an error, then there may be something as trivial as a missing closing bracket in your code. The printed error will often direct you to the line that can be found. It is good practice to frequently check that your code is running, even well before it is complete. The more code you write in, the more likely are errors to be produced, and the harder it becomes to track and fix them.

6. We now define a function `build_sto3Gbasis` that will house the code we introduced in our discussion of conditional logic. This will append to an array `sto3Gbasis` every atomic orbital in the molecule, based on the atomic orbital basis configuration. Type in the following code, and complete the unfinished code indicated in dark blue.

```
sto3Gbasis = [] # instantiate the array representing our basis

def build_sto3Gbasis(atoms, R):
  """
  This function depends on the atoms array, which lists strings of atomic symbols of the atoms of the input
  molecule, in the same order as the input .xyz file.
  This function additionally depends on the molecule's coordinates R (a nested array), where each element is
  an array holding the coordinates of each atom, also in the same order as atoms.
  """

  K = 0 # instantiates atomic orbital counter, which will be used for matrix dimensions

  # loop through the atoms array, and append orbitals (dictionaries) to the basis array
  # each atom (in atoms) has an orbital configuration associated with it, so loop through these too
  for i, atom in enumerate(atoms):
    for orbital in orbitalconfiguration[atom]:
      if orbital == '1s':
        sto3Gbasis.append(
                          {
                            'Z': Z[atom],                # atom name --> atomic number
                            'o': orbital,                # append the orbital-type string ('1s','2s',etc.)
                            'R': R[ i ],                 # get array [x,y,z] of i-th atom coordinates
                            'l': 0,                      # s orbital ==> 0 angular momentum
                            'm': 0,
                            'n': 0,
                            'a': a[ (Z[atom]-1) ][0],    # append list of 1s orbital exponential factors
                            'd': d[0]                    # append list of 1s orbital contraction coefficients
                          }
                        )
        K = K + 1 # increment orbital counter
      elif orbital == '2s':
        # append 2s orbital dictionary to sto3Gbasis
        # increment orbital counter K by one
      elif orbital == '2px':
        sto3Gbasis.append(
                          {
                            'Z': Z[atom],
                            'o': '2px',
                            'R': R[ i ],
                            'l': 1,                     # 2px orbital has angular momentum in x direction
                            'm': 0,
                            'n': 0,
                            'a': a[ (Z[atom]-1) ][1],   # 2p orbital exponent = 2s orbital exponent
                            'd': d[2]                   # append list of 2p orbital contraction coefficients
                          }
                        )
      # append 2py orbital under the condition that orbital == '2py'; increment K
      # append 2pz orbital under the condition that orbital == '2pz'; increment K

  # when this function is called, return both the basis and the orbital count
  return sto3Gbasis, K
```

While typing in this code, you will have noticed that the code for the function `build_sto3Gbasis` has changed just slightly from our earlier presentation of function `build_sto3Gbasis`. Namely, we have adopted the keyword `enumerate` in the for loop. We do so because this is a convenient way of *conjoining* the two kinds of loops we discussed earlier: One in which the loop index is simply a counting integer variable (`i = 0, 1, 2`, ..., upper bound of `range`), *and* one in which the index actually *equals* the element of whatever array is being looped through (`for atom in atoms:` ... `atom = 'O'`, `atom = 'H'`, ...).

This is useful because `R` is a nested array, and the i-th atom's coordinates are those coorresponding to whatever `atom` is being looped through. For example, if `atoms = ['H','F']`, which has the corresponding molecular coordinates `R = [[0.00, 0.00, 0.00],[0.00,0.00,1.79]]`, then when in the second loop that `atom = 'F'`, then `i = 1`, and `R[i] = R[1] = [0.00,0.00,1.79]`.

Pay careful attention to the indentations of for loops and if statements, because it is easy to throw an error if any of them are not aligned correctly. Continue to ensure you are closing every bracket or parentheses that you open. And include comments!

This completes `basis.py`. Confirm there are no obvious errors in your code by running it.

3.5.2 Building `oei.py`

7. Create a file in your programming project directory called `oei.py`. In this file we will write code to calculate the overlap integral (**S**) matrix elements.

8. Import the NumPy library, and three modules of the SciPy library, by typing in the following code in the preamble (beginning lines) of `oei.py`.

```
import numpy as np  # to handle matrices, calculate pi and exponential functions, and more
from scipy import misc, special, linalg # to handle other specific math operations
```

9. Define a function for computing normalization constants, and call it `compute_norm`. This function will be employed during the calculation of matrix elements of **S**, **T**, **V** and **G**. The formula was derived in Handout 1, and is found there as Eq. 30. Type the following code in, and take care to notice the relationship between Handout 1 Eq. 30 and the code.

```
def compute_norm(alphaconstant,l,m,n):
  """
  Compute the normalization constant, using Eq. 30 from Handout 1.
  Applies to S, V, T, and G.
  """
  N = (2*alphaconstant / np.pi)**(3/2) * (?)**(?) \
      / (misc.factorial2(2*l-1) * misc.factorial2(?) * misc.factorial2(?)
  N = N**(1/2)

  return N
```

There are several things to notice here.

`N` is initially equated to just the bracketed part of the normalization constant N of Eq. 30. Then, in the next line of code, the square root of that result is taken. As before, the line `N = N**(1/2)` is not algebraically correct, but in code it serves to update the variable `N`, to reflect the full form of the normalization constant N as it appears in Handout 1 Eq. 30.

Another thing to notice is that because the equation for N is quite long, we break it up over several lines of code by adding a backslash at the end of the first line of the coded equation.

Additionally, double factorials are computed by including in the parentheses of `misc.factorial2()` whatever we want the double factorial of – that is, the *argument* of the function goes in the parentheses. Recall that we imported the `misc` module of the SciPy library in our preamble.

Complete the `compute_norm` function by substituting the dark blue placeholder text (question marks) with functioning code.

10. We will frequently need to compute third centers between some extant centers – e.g., the third center $\vec{P}$ between $\vec{A}$ and $\vec{B}$. Let's create a function based on Eq. 13 of Handout 1, and let α_a = `a`, α_b = `b`, $\vec{A}$ = `RA`, and $\vec{B}$ = `RB`

```
def compute_third_center(a, RA, b, RB):
  """
  Compute a Gaussian product third center (P betweeen A and B, or Q between C and D),
  according to Eq. 14 of Handout 1.
  """
  # code to calculate P in terms of the function's arguments
  return P
```

where of course `P` $= \vec{P}$. Implement Handout 1 Eq. 14 in the blue block of code.

11. Type in the function `compute_ABsq` for computing the distance between two orbitals (or atoms), as described earlier.

12. Now we write a function for Eq. 29 of Handout 1. This is among the most complex functions, but once you have a good grasp on it, then by understanding its application to the matrix **S** you will be able to apply it to the other matrices (**T**, **V**, **G**). Type in the following code.

```
def build_S(basis, S):
  """
  This function depends on the complete basis (an array of dictionaries, one dictionary
  per orbital), as well as basis size (number of atomic orbitals) K. It calls an
  intermediate function compute_Si (based on Handout 2 Eq. 29), as  well as the function
  for calculating the third center P.
  """

  # create an empty K x K NumPy matrix called S

  for A, bA in enumerate(basis): # access row A of S matrix; call basis info for orbital A
    for B, bB in enumerate(basis): # access column B of S matrix; call basis info for orbital B

      # code to manipulate matrix element S[A,B] (Handout 2, Eq. 29)

  return S
```

You know how to create empty NumPy matrices of any dimensions, so type in the code for instantiating `S` where you are directed to. We will fill in `# code ...` shortly, but first let's consider the nested for loop.

This nested loop is very similar to the example we provided earlier for computing the overlap integral matrix of dihydrogen with a STO-1G basis set. However, because we are now using a STO-3G basis, and want to generalize the code to atoms other than hydrogen, and to molecules of any size, we will loop through the array `basis` instead of the array `atoms`. This gives us access to the detailed basis set information for each orbital (which is represented by *three* Gaussian primitives now, not just one), which could be a $1s$ orbital of H, a $2p_y$ orbital of F, and one of many other possibilities. Once again we adopt `enumerate` loops. `A` is an integer counting variable, and `bA` is equal to an element of the array `basis`. `bA`, then, will be a dictionary because, as we've already discussed, each element of the array `basis` is an "orbital dictionary."

Initially, `A` and `B` are equal to 0, and `bA` and `bB` will both be the first dictionaries of the basis array – that is, both reference the same atom-centered orbital. Once code to manipulate matrix `S` on this upper-left-most matrix element (overlap of ϕ_A with ϕ_A) is complete, then, with `bA` still at 0 (which is the first row of matrix `S`), `bB` will be incremented by one, accessing the second element (or "column") of `S` in the first row. `bB` will increment as many times as there are elements of `basis`, before `bA` is incremented by one, and `bB` is again at 0 – the beginning of the second row of `S`.

This nested loop scheme gives us access to all elements of `S`.

However, we need to nest more loops *within* these two loops, precisely because each orbital is made up of *three* Gaussian primitives. Within each matrix element, then, a Gaussian overlap integral must be calculated three times, corresponding to each primitive of each orbital. The code is therefore expanded as follows:

```
def build_S(basis, K):
  """
  This function depends on the complete basis (an array of dictionaries, one dictionary
  per orbital), as well as basis size (number of atomic orbitals) K. It calls an
  intermediate function compute_Si (based on Handout 2 Eq. 29), as well as the function
  for calculating the third center P.
  """

  # create an empty K x K NumPy matrix called S

  for A, bA in enumerate(basis): # access row A of S matrix; call basis info for orbital A
    for B, bB in enumerate(basis): # access column B of S matrix; call basis info for orbital B

      # pull basis set information for orbital A, B
      for a, dA in zip(bA['a'],bA['d']): # a is alpha, dA is contraction coefficient
        # analogous for loop for basis set information of orbital B

          RA = bA['R'] # collect atom-centered coordinates of orbital A from basis
          # analogous coordinate information for orbital B

          lA,mA,nA = bA['l'],bA['m'],bA['n'] # collect orbital A angular momentum
          # analogous angular momentum information for orbital B

          RP = compute_third_center(a,b,RA,RB)

          S[A,B] = # Handout 2, Eq. 29

  return S
```

The new for loops serve to access individual elements of the basis set (α_p factors and d_p contraction coefficients). To do so, we have employed a new functionality of Python loops called `zip`. (Don't worry, this is the last variation on loops we need to introduce!) Looping with `zip` allows one to loop through *two* arrays *in tandem*: As the first element of the value of key `'a'` from dictionary `bA` is being accessed with `a`, so too is `'d'` with `dA`. The first elements of each array are accessed simultaneously, the second elements simultaneously, and the third elements simultaneously. This is essential, because the first α_p coefficient can *only* be paired with the first d_p contraction coeffcient, the second α_p with the second d, and the third α_p with the third d_p. Any another combination is meaningless.

Recall that the keys `'a'` and `'d'`, in each orbital dictionary of `basis`, have "values" that are arrays. In STO-3G, there are three elements in both `'a'` and `'d'`. If we had chosen STO-6G, there would be 6 elements in each of these arrays.

New variables, such as `RA`, `RB`, `lA`, `lB`, and so on, are instantiated to pull coordinate and angular momentum information from `bA` and `bB`.

Having accessed a matrix element, and having pulled relevant information about the individual orbitals from the basis, Eq. 29 can be evaluated. Using the fact that

```
S[A,B] = S[A,B] + ...
```

is equivalent to

```
S[A,B] += ...
```

let's flesh out this line of code.

```
S[A,B] += dA * ? * compute_norm(arguments) * ? * \
          np.exp (-( a*b/(a+b) ) * compute_ABsq(arguments)**2 ) * \
          compute_Si(lA,lB,RP[?]-RA[?],RP[?]-RB[?],a+b) * \   # Sx
          compute_Si(mA,mB,RP[?]-RA[?],RP[?]-RB[?],a+b) * \   # Sy
          compute_Si(nA,nB,RP[?]-RA[?],RP[?]-RB[?],a+b)       # Sz
```

Complete the code. Note that

$$\vec{PA} = \vec{P} - \vec{A}$$

such that

$$\vec{PA}_x = \vec{P}_x - \vec{A}_x, \quad \vec{PA}_y = \vec{P}_y - \vec{A}_y, \quad \vec{PA}_z = \vec{P}_z - \vec{A}_z$$

We have called a function `compute_Si`, which we have not yet coded. But it makes sense to code a generic S_i, because for each Gaussian primitive overlap integral we must compute something very similar three times (S_x, S_y, S_z).

13. Referring to Eq. 28 of Handout 2, enter and complete the following code:

```
def compute_Si(lA,lB,PA,PB,gamma):
  """
  Calculate the i-th coordinate contribution to the matrix element S[A,B].
  """
  for k in range( int((lA + lB)/2) + 1 ): # note range is up to and INCLUDING floor of (lA+lB)/2, hence +1
    Si += compute_ck(2*k,lA,lB,PA,PB) * np.sqrt( np.pi / gamma ) * \
          misc.factorial2(2*k-1) / (?)**k
```

The floor function, $\lfloor\rfloor$ depicted in Eq. 28 is equivalent to the integer `int()` function of Python. `Si` depends on one final function, c_k, which we implement next.

14. Define another function for computing the constant c_k. Referring to Eq. 21 of Handout 2, and calling the `binomial` function of the `special` module (part of the SciPy library), type in the following code:

```
def compute_ck(arguments):
  for i in range(l+1): # i from 0 up to and including l, hence +1
    #for loop over j
      if i + k == ?:
        ck += special.binom(l,k) * ? * a**(l-i) * ?
  return ?
```

Complete the code.

We are now finished with `oei.py` for the Guided Exercises, having implemented all of the code for computing the overlap integral matrix.

3.5.3 Building `main.py`

We need to tie together `basis.py` and `oei.py` with `main.py`.

15. Create a file in your programming project directory called `main.py`.

16. In the preamble of `main.py`, import the NumPy library, as you did for `oei.py`. Also import the library `argparse`. Finally, `import oei` and `import basis`, which will allow us to access all functions of these code files, which we prepared in the previous exercises.

17. After an empty line, type in the following code:

```
parser = argparse.ArgumentParser()
parser.add_argument('coords_file', metavar = 'coordinates.xyz', type=str)
args   = parser.parse_args()

inputfile = open(args.coords_file,'r') # open .xyz file without being able to modify it (read-only)
```

This code will allow the user of `main.py` to specify an input file when running the program, like so:

```
$ python main.py [.xyz file]
```

The variable `inputfile`, then, will hold the contents of the `.xyz` input file (e.g., `h2o.xyz`). (At some point the reader should explore features of `argparse` functionality, because with additional `add_arguments`, one can include greater input functionality. For example, you might wish to allow the user to control the molecule's charge or multiplicity if open-shell functionality is added to the closed-shell Hartree–Fock code that is presented in Handout 4.)

18. Now we will begin generating code to automatically read in the `.xyz` files and generate the `atoms` and coordinates array `R`. Construct a function called `read_inputfile` to read an `inputfile` line by line, at each line extracting the atom's atomic symbol and coordinates.

```
def read_inputfile(inputfile):
  """
  Builds two arrays from .xyz input file:
  1) atoms, containing elements of atomic symbols (strings) in order of input file
  2) coords, a nested array containing elements of coordinates (1 coordinates array per atom) also in order
  """
  atoms   = [] # this array will hold atomic symbols in order of input file
  coords  = [] # in same order as atoms, each element will be an array of i-th atom's coordinates

  # read through inputfile line by line
  for line in inputfile.readlines()[2:]: # [2:] means read only from the third line of the input file onwards
      atom, x, y, z = line.split()       # from each atom line collect atomic symbol, x, y , z
      atoms.append( atom )               # append to atoms the contents of the string variable atom
      coords.append( [float(x),float(y),float(z)] ) # append to coords current atom's coordinates (as array)

  return atoms, coords
```

19. Call the function `read_input_file` developed in the previous step, and pass into it `inputfile`, to return an array `atoms` and coordinates `R`.

```
# build array of atoms (atomic symbols) and coordinates, by calling function
?, ? = read_input_file(?)
```

20. Build the STO-3G basis by typing in and completing

```
# build the STO-3G basis using these inputs
orbitalbasis, basissize = basis.build_sto3Gbasis(?, ?) # basis size (number of AOs) ==> dimensions of matrix
```

If you are unsure what the arguments to the function call should be, return to the part of your code in `basis.py` where you defined the `build_sto3Gbasis` function, and use this information to determine what to pass into it. Also make sure you understand the relationship between `orbitalbasis` and `basissize`, and what is *returned* by the function `build_sto3Gbasis`.

21. Finally, pass in to the `build_S` function (which is in `oei.py`) the array `orbitalbasis` and the matrix dimension integer `basissize`, which we collected in the previous exercise.

```
# the result of calling on build_S is the overlap integral matrix
S = oei.build_S(?,?)
```

As the comment states, the result of calling `build_S`, in the form of `S` here, should be the complete overlap integral matrix **S**.

22. The exercises are complete! All that remains is to `print(S)`, and ensure that the result is identical to that shown in Fig. 5, for the input file also shown there.

A reminder: Errors you may encounter in your code are often fixed by addressing a flagged line of bad code that Python reports when you try to run your code. An additional debugging practice includes printing variables and results at several different stages throughout your code. For example, if everything prints as expected up until `Sx`, `Sy`, and/or `Sz` is being calculated, then there is very likely an error in your `compute_Si` function. Did you type in everything exactly as shown in this document? Are you sure you filled in the incomplete code correctly?

You have now practiced every coding skill you will require to complete the remainder of the coding projects in Handout 4 and Handout 5. Any other programming techniques you will need are described in those documents.

	$\vert \mathrm{O}1s)$	$\vert \mathrm{O}2s)$	$\vert \mathrm{O}2p_x)$	$\vert \mathrm{O}2p_y)$	$\vert \mathrm{O}2p_z)$	$\vert \mathrm{H}_a 1s)$	$\vert \mathrm{H}_b 1s)$
$(\mathrm{O}1s\vert$	1.00000	0.23670	0.00000	0.00000	0.00000	0.05695	0.05695
$(\mathrm{O}2s\vert$	0.23670	1.00000	0.00000	0.00000	0.00000	0.48979	0.48979
$(\mathrm{O}2p_x\vert$	0.00000	0.00000	1.00000	0.00000	1.00000	0.00000	0.00000
$(\mathrm{O}2p_y\vert$	0.00000	0.00000	0.00000	1.00000	0.00000	0.30738	-0.30738
$(\mathrm{O}2p_z\vert$	0.00000	0.00000	0.00000	0.00000	1.00000	-0.25785	-0.25785
$(\mathrm{H}_a 1s\vert$	0.05695	0.48979	0.00000	0.30738	-0.25785	1.00000	0.28279
$(\mathrm{H}_b 1s\vert$	0.05695	0.48979	0.00000	-0.30738	-0.25785	0.28279	1.00000

```
3

O     0.00000    0.00000    0.22700
H     0.00000    1.35300   -0.90800
H     0.00000   -1.35300   -0.90800
```

Figure 5: The outcome of evaluating every element of the overlap integral matrix **S** in the minimal STO-3G basis, with the `.xyz` coordinates (units Bohr) shown below it.

3.6 Other Resources

One's programming skills can always be improved through practice and exposure to new techniques. Further and broader introduction to programming in Python can be found in many excellent books and online introductory guides. At the time of this manuscript's writing, two excellent online introductory guides are CodeAcademy (`www.codeacademy.com`) and Learn Python the Hard Way (`www.learnpythonthehardway.org`).

The Internet is an incredible resource for answering your coding questions. If you're having a problem, or need to know if there is a way to perform some task in Python, a good search engine query can often yield results that provide exactly the information you are looking for (e.g., "how do I diagonalize a matrix in python with numpy"). Additionally, take advantage of colleagues that have coding experience. Ask them how they would go about coding something (but don't let them do the work for you!).