Programming the Self-Consistent Field Method Using Python

Alexander Yu. Sokolov

Department of Chemistry and Biochemistry
The Ohio State University
Columbus, OH 43210

August 23, 2017

1 The self-consistent field algorithm

The following algorithm can be used to implement the self-consistent field method (SCF) for the restricted Hartree-Fock theory (RHF) in the atomic-orbital basis:

1. **Specify molecular geometry, basis set.**
2. **Compute initial data.**
   (a) **Compute nuclear repulsion energy (E_{nuc}).**
   (b) **Compute overlap integrals (S).**
   (c) **Compute one-electron integrals ($H_{\mu\nu} = \langle\mu|\hat{h}|\nu\rangle = T_{\mu\nu} + V_{\mu\nu}$).**
   (d) **Compute two-electron integrals** $[(\mu\nu|\rho\sigma)]$.
   Note that most of the quantum chemistry codes compute two-electron integrals sorted in the Chemists' notation.
3. **Construct the $\mathbf{S^{-1/2}}$ matrix.**
   (a) **Diagonalize the S matrix.**

$$\sum_{\mu\nu} U_{\mu\mu'} S_{\mu\nu} U_{\nu\nu'} = s_{\mu'} \delta_{\mu'\nu'} \tag{1}$$

   (b) **Compute the $\mathbf{S^{-1/2}}$ matrix.**

$$(S^{-1/2})_{\mu\nu} = \sum_{\mu'} U_{\mu\mu'} \frac{1}{\sqrt{s_{\mu'}}} U_{\nu\mu'} \tag{2}$$

4. **Construct initial density matrix.**
   (a) **Form initial Fock matrix neglecting the two-electron term.**

$$\mathbf{F} = \mathbf{H} \tag{3}$$

   (b) **Compute transformed Fock matrix.**

$$\tilde{\mathbf{F}} = \mathbf{S^{-1/2}}\,\mathbf{F}\,\mathbf{S^{-1/2}} \tag{4}$$

(c) **Diagonalize the $\tilde{\mathbf{F}}$ matrix.**

$$\tilde{\mathbf{F}}\,\tilde{\mathbf{C}} = \tilde{\mathbf{C}}\,\boldsymbol{\epsilon} \tag{5}$$

(d) **Compute SCF eigenvector matrix.**

$$\mathbf{C} = \mathbf{S}^{-1/2}\,\tilde{\mathbf{C}} \tag{6}$$

(e) **Compute initial density matrix.**

$$D_{\mu\nu} = \sum_{i}^{n/2} C_{\mu}^{i*} C_{\nu}^{i} \ . \tag{7}$$

5. The SCF iteration.

(a) **Compute the new Fock matrix including the two-electron contribution.**

$$F_{\mu\nu} = \langle \mu | \hat{h} | \nu \rangle + \sum_{\rho\sigma}^{N} D_{\rho\sigma} (2 \langle \mu\rho | \nu\sigma \rangle - \langle \mu\rho | \sigma\nu \rangle) \tag{8}$$

(b) **Compute the electronic and total energies.**

$$E_{elec} = \sum_{\mu\nu}^{N} D_{\mu\nu} (H_{\mu\nu} + F_{\mu\nu}) \tag{9}$$

$$E_{total} = E_{nuc} + E_{elec} \tag{10}$$

(c) **Compute transformed Fock matrix.**

$$\tilde{\mathbf{F}} = \mathbf{S}^{-1/2}\,\mathbf{F}\,\mathbf{S}^{-1/2} \tag{11}$$

(d) **Diagonalize the $\tilde{\mathbf{F}}$ matrix.**

$$\tilde{\mathbf{F}}\,\tilde{\mathbf{C}} = \tilde{\mathbf{C}}\,\boldsymbol{\epsilon} \tag{12}$$

(e) **Compute SCF eigenvector matrix.**

$$\mathbf{C} = \mathbf{S}^{-1/2}\,\tilde{\mathbf{C}} \tag{13}$$

(f) Compute the new density matrix.

$$D_{\mu\nu} = \sum_{i}^{n/2} C_{\mu}^{i*} C_{\nu}^{i} \ . \tag{14}$$

(g) Test convergence of the energy and the density matrix.

i. Check RMS of the density matrix elements:

$$\Delta_{rms} = \sqrt{\sum_{\mu\nu} (D_{\mu\nu}^{new} - D_{\mu\nu}^{old})^2} < \Delta_D \tag{15}$$

ii. Check energy difference:

$$\Delta E = E_{elec}^{(n)} - E_{elec}^{(n-1)} < \Delta_E \tag{16}$$

(h) If SCF is converged, stop the iterations. If not, continue.

2 Using Python to implement SCF

Python is a very powerful high-level programming language that can be used to implement complicated algorithms in just a few lines of code. Using its extensive mathematics library and the third-party libraries, such as Numpy and Scipy, many methods in quantum chemistry can be programmed very efficiently. In addition, Python is compatible with other popular programming languages (such as C++ and Fortran), which can be used to enhance Python's functionality and computational efficiency. This short overview is intended to illustrate some of the Python's capabilities. For more details, please refer to the information available online.

2.1 Matrix and tensor operations using Numpy

A powerful way to perform matrix and tensor operations in Python is to use Numpy. To check if Numpy is available on your system, open the Python interpreter (run the Python executable) and in the command line type `import numpy`, e.g.:

```
[Silverblade:~] alex% python
Python 2.7.13 (default, Mar 23 2017, 10:12:46)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>>
```

If running the `import` command does not cause any errors, your Python should be configured to work with Numpy. Since Numpy is an external library, it is not loaded by default. Thus, every Python script that uses Numpy should contain the `import numpy` line. As any Python library, Numpy can be loaded with an alias:

```
>>> import numpy as np
```

The above line allows to shorten the Numpy function calls, e.g. `np.sqrt(4.0)` instead of `numpy.sqrt(4.0)`.

Numpy allows to efficiently store and manipulate data using the so-called *numpy arrays*. For example, a matrix can be represented using a two-dimensional numpy array as follows:

```
>>> A = np.random.random((4,4))
>>> print A
[[ 0.41876825  0.46548118  0.72390022  0.82810599]
 [ 0.91278577  0.6193322   0.89722354  0.92845717]
 [ 0.63918077  0.56033718  0.24796446  0.74611299]
 [ 0.07634179  0.81549647  0.95896538  0.01995572]]
```

In the above example, we used the Numpy function `random` of the `np.random` module to generate a 4×4 matrix with random floating-point numbers. A powerful feature of the numpy arrays is that they can be used to represent tensors with more than two dimensions (up to 32). As an example, a $2 \times 2 \times 2 \times 2$ four-dimensional array can be created as:

```
>>> X = np.empty((2,2,2,2))
>>> X
array([[[[ 0.,  0.],
         [ 0.,  0.]],

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

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

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

        [[ 0.,  0.],
         [ 0.,  0.]]]])
```

In this example, we created an empty numpy array that is filled with floating-point zeros. Note that in both examples above the dimensions of the numpy arrays are represented as Python *tuples*, e.g. `(4,4)` or `(2,2,2,2)`. Thus, the number of elements in a tuple defines the number of dimensions in a numpy array. There are many other ways to create numpy arrays. For example, a numpy array can be obtained by "stacking" two existing numpy arrays (using functions `np.vstack()` or `np.hstack()`) or by creating an empty array (using `np.array([])`) and appending data (using `np.append()`). Other Python data structures, such as tuples or lists, can be converted to numpy arrays as well.

Now let's go back to our first example and define a new matrix with dimensions 4×2:

```
>>> B = np.random.random((4,2))
>>> B
array([[ 0.19960546,  0.55020058],
       [ 0.97162975,  0.53432274],
       [ 0.17318349,  0.87559476],
       [ 0.94732401,  0.11567287]])
```

We can multiply the matrices $\mathbf{A}$ and $\mathbf{B}$ to compute a new matrix $\mathbf{C}$:

```
>>> C = np.dot(A,B)
>>> C
array([[ 1.44571604,  1.20875634],
       [ 1.81889269,  1.72614007],
       [ 1.42177835,  0.95449993],
       [ 0.99258038,  1.319715  ]])
```

Note that the operation `A * B` is not a matrix multiplication, it is an element-wise multiplication instead. Since $\mathbf{A}$ and $\mathbf{B}$ are matrices of different dimensions, the operation `A * B` will result in an error. Multiplying `A * A` will result in a matrix $\mathbf{A}$ with all elements squared:

```
>>> A * A
array([[  1.75366845e-01,   2.16672731e-01,   5.24031531e-01,
          6.85759525e-01],
       [  8.33177858e-01,   3.83572377e-01,   8.05010082e-01,
          8.62032724e-01],
       [  4.08552054e-01,   3.13977755e-01,   6.14863745e-02,
          5.56684597e-01],
       [  5.82806936e-03,   6.65034499e-01,   9.19614603e-01,
          3.98230596e-04]])
```

Multiplication of several matrices can be performed in one line, e.g. for $\mathbf{D} = \mathbf{B^T A B}$ we can write:

```
>>> D = np.dot(B.T, np.dot(A, B))
>>> D
array([[ 3.24238683,  3.33394475],
       [ 3.12702581,  2.57578468]])
```

A more elegant way to perform such a matrix multiplication is to use the `reduce` function:

```
>>> D = reduce(np.dot, (B.T, A, B))
>>> D
array([[ 3.24238683,  3.33394475],
       [ 3.12702581,  2.57578468]])
```

where we used the `.T` attribute of the numpy array `B` to generate its transpose. An alternative way to compute the transpose of a matrix is using the `np.transpose()` function.

So far, we have performed operations that involve all elements of a numpy array (e.g., matrix multiplication multiplies all elements of two matrices). To access a single element of an array, we can simply specify the indices of the element in square brackets (e.g., `A[3,2]` or `X[1,0,1,0]`, note that the 0-counting convention is used). Another very powerful way to access subsets of elements of a numpy array is to use *slicing*. As an example, we can use slicing to print the third row of the matrix $\mathbf{A}$:

```
>>> A[2,:]
array([ 0.63918077,  0.56033718,  0.24796446,  0.74611299])
```

Here, the first index 2 is used to specify the row, while the entry : specifies that all of the columns need to be accessed. Instead of accessing all columns, we can request to access only a subset. For example, we can restrict the range of columns to the second and third columns only:

```
>>> A[2,1:3]
array([ 0.56033718,  0.24796446])
```

Slicing can be used to set elements:

```
>>> A[2,:] = 1.0
>>> A
array([[ 0.41876825,  0.46548118,  0.72390022,  0.82810599],
       [ 0.91278577,  0.6193322 ,  0.89722354,  0.92845717],
       [ 1.        ,  1.        ,  1.        ,  1.        ],
       [ 0.07634179,  0.81549647,  0.95896538,  0.01995572]])
```

Alternatively, it can be used to access certain patterns of elements, such as elements with all even indices

```
>>> A[::2,::2]
array([[ 0.41876825,  0.72390022],
       [ 0.63918077,  0.24796446]])
```

or all odd indices

```
>>> A[1::2,1::2]
array([[ 0.6193322 ,  0.92845717],
       [ 0.81549647,  0.01995572]])
```

Slicing is not only very powerful, but can also be very computationally efficient. For other ways to access elements of the numpy arrays, please refer to the Numpy manual.

Finally, we consider how we can manipulate data stored in multi-dimensional numpy arrays. Much of the Numpy functionality available for the two-dimensional arrays can be used for the arrays with more dimensions. One difference is that, since multi-dimensional tensors have more than two indices, the multiplication of two tensors is no longer uniquely defined. For this reason, we will not use the `np.dot()` function to compute the product of two multi-dimensional arrays, but instead consider a more general function called `np.einsum()`. [Note that in certain cases the `np.dot()` function

can be used to compute the product of two multi-dimensional arrays, see the Numpy manual]. To illustrate how `np.einsum()` works, let's consider the following tensor operation (so-called *tensor contraction*) as an example:

$$G_{xyzw} = \sum_{pq} E_{xpzq} F_{pyqw} \ , \tag{17}$$

where we multiply two four-dimensional tensors by summing over the two common indices to obtain another four-dimensional tensor. Once the tensors **E** and **F** are computed and are stored as four-dimensional numpy arrays, the tensor contraction in Eq. (17) can be implemented as:

```
>>> G = np.einsum('xpzq,pyqw->xyzw', E, F)
```

The first argument of the `np.einsum()` function specifies how the tensor operation should be performed: the four indices of tensors `E` and `F` are separated by a comma, while the indices of the third tensor `G` are separated by the `->` symbol, and the summation is performed over the repeated indices (so-called *Einstein summation*). The `np.einsum()` function is very flexible, it can perform operations with tensors of arbitrary dimensions (including the one- and two-dimensional arrays). For example, the norm of the tensor **F** ($N = \sqrt{\sum_{pqrs} F_{pqrs}^2}$) can be computed as:

```
>>> N = np.sqrt(np.einsum('pqrs,pqrs', F, F))
```

While `np.einsum()` can be used to perform complicated tensor operations in a compact form, it is less computationally efficient than `np.dot()`. For this reason, using `np.einsum()` is recommended for the initial (pilot) implementations only, while for the efficient implementations it should be avoided. In the latter case, it is often possible to reduce the dimensionality of the multi-dimensional tensors by transposing the indices (using `np.transpose()`) and reshaping them down to the two-dimensional arrays (`np.reshape()`), which can be multiplied as "super-matrices" using the `np.dot()` function.

2.2 Defining new functions in Python

Programming computer algorithms often involves performing the same task multiple times. Such tasks can be conveniently defined as functions, which can be called in several places of the program. Python makes defining new

functions very easy. We will illustrate this on a simple example. Let us define a function that takes a numpy array as an input and returns a list of its three largest elements. First, we define the function as:

```
>>> import numpy as np
>>> def max_elements(A):
...     # Turn a numpy array into a one-dimensional array
...     A = A.ravel()
...     # Sort the array in the descending order
...     A[::-1].sort()
...     # Return first three elements as a Python list
...     return A[:3].tolist()
...
```

We can now use this function to print the three largest elements of a random matrix:

```
>>> X = np.random.random((4,4))
>>> print max_elements(X)
[0.9840955434199267, 0.952108251869013, 0.9236377591745225]
>>> X = np.random.random((4,4))
>>> print max_elements(X)
[0.893858566081185, 0.7728361321722338, 0.746132464101978]
>>> X = np.random.random((4,4))
>>> print max_elements(X)
[0.9711829325900079, 0.9198351757368676, 0.8354676131068733]
```

The function we defined above can also be used for arrays with more than two dimensions, e.g.:

```
>>> Y = np.random.random((4,4,4,4))
>>> print max_elements(Y)
[0.990771191307948, 0.98952424412145, 0.982675647370982]
```

Although this is a very simple example, Python can be used to define more complicated objects, such as a function with a varying number of arguments or a recursive function that calls itself multiple times.

2.3 Obtaining one- and two-electron integrals from Pyscf

Computing the one- and two-electron integrals is always the first step in quantum chemical simulations. Most of the quantum chemistry programs already provide the ability to efficiently compute integrals for a specified basis set and store them in a file or in memory. In this project, we will use the Pyscf program to generate the one- and two-electron integrals. Pyscf has many useful features for developing methods in quantum chemistry and is mostly written using Python, which allows us to use many of its functions (such as obtaining the integrals or the molecular geometry) directly in a Python script.

In order to use Pyscf, you will need to download the source code and compile it. Once the compilation is successful, the path to the Pyscf directory needs to be appended to the `PYTHONPATH` environment variable. This is important to be able to use Pyscf as a Python library. For example, if Pyscf is installed in `/home/ays3/codes/pyscf`, the `/home/ays3/codes/` path needs to be appended to the `PYTHONPATH` environment variable. [It is recommended to update the `.bashrc` shell script to make sure that `PYTHONPATH` always contains the path to Pyscf]. To check if your Python is configured to work with Pyscf, open the Python interpreter and type `import pyscf`. If importing the pyscf module does not cause any errors, your Python should be configured to work with Pyscf.

Now let's consider an example of how we can use Pyscf to run a SCF computation for the N_2 molecule using the sto-3g basis set. This can be done using the following input Python script:

```
# Import modules
import numpy as np
import pyscf.gto
import pyscf.scf

# Specify molecular geometry and basis set
mol = pyscf.gto.M(
    verbose = 5,
    atom = [
        ['N', (0.0,  0.0,  0.0)],
        ['N', (1.1,  0.0,  0.0)],
        ],
    basis = 'sto-3g',
```

```
    symmetry = True)

# Create an instance of the Restricted Hartree-Fock Python class
mf = pyscf.scf.RHF(mol)
# Specify the level of details that need to be printed
mf.verbose = 4
# Specify the convergence parameter
mf.conv_tol = 1e-10
# Run the SCF computation and obtain the energy
ehf = mf.scf()
# Analyze molecular orbitals
mf.analyze()
```

Many other examples can be found in the `examples` directory of the Pyscf source code.

Pyscf can be used to compute integrals directly without executing the SCF computation. To do that, we can use the `pyscf.gto` module as follows:

```
# Import modules
import numpy as np
import pyscf.gto

# Set options to make Numpy printing more clear
np.set_printoptions(linewidth=150, edgeitems=10, suppress=True)

# Specify molecular geometry and basis set
mol = pyscf.gto.M(
    verbose = 5,
    atom = [
        ['N', (0.0,  0.0,  0.0)],
        ['N', (1.1,  0.0,  0.0)],
        ],
    basis = 'sto-3g',
    symmetry = True)

# Obtain the number of atomic orbitals in the basis set
nao = mol.nao_nr()
# Obtain the number of electrons
nelec = mol.nelectron
```

```
# Compute nuclear repulsion energy
enuc = mol.energy_nuc()
# Compute overlap integrals
ovlp = mol.intor('cint1e_ovlp_sph')
# Compute one-electron kinetic integrals
T = mol.intor('cint1e_kin_sph')
# Compute one-electron potential integrals
V = mol.intor('cint1e_nuc_sph')
# Compute two-electron repulsion integrals (Chemists' notation)
v2e = mol.intor('cint2e_sph').reshape((nao,)*4)
```

The computed integrals are stored in the numpy array format. These integrals can be used to implement the algorithm outlined in Section 1.