Hartree-Fock Program Project

C. David Sherrill
School of Chemistry and Biochemistry
Georgia Institute of Technology

Updated 14 April 2017
Based on notes by Dr. Yukio Yamaguchi (University of Georgia)
C++ notes provided by Prof. Edward Hohenstein (CCNY)

1 General Information and Outline of the Hartree–Fock Procedure

Goals: Write a computer program to perform a closed-shell restricted Hartree-Fock computation, given nuclear repulsion energy and one- and two-electron integrals. These notes describe three strategies for accomplishing this goal, of which you should choose one: (1) reading the required data from a plain text file, for the specific case of STO-3G H_2O at a specific geometry; (2) obtaining the required data directly from the Psi4 program package using Psi4's Python front-end and the Psi4's available Python function library; or (3) obtaining the required data from Psi4 using C++ and Psi4's C++ libraries.

An advantage of strategy (1), the plain text file for STO-3G H_2O, is that you are free to use any computer programming language that you wish (or Matlab, Mathematica, etc.). The disadvantages are that your program will be limited to working only for this one test case (unless you can independently generate the required integrals), and that you will have to create/find all required subroutines (e.g., matrix diagonalization, matrix multiplication, etc.)

An advantage of strategies (2) or (3) is that they will work on general molecules, and you can leverage existing subroutines provided by Psi4. However, the current documentation for Psi4 C++ libraries is not highly organized, and moreover C++ is generally harder to learn and/or requires more lines of code to accomplish tasks than Python. Documentation of all functions required to accomplish the project using Python (strategy (2)) is provided below. Hence, for an initial project strategy (2) is probably preferred over strategy (3).

Additional Information: See chapter 3 (sections 3.4.4 - 3.4.6) of Modern Quantum Chemistry, 1st Ed., Revised, A. Szabo and N. S. Ostlund (McGraw-Hill, New York, 1989).

Procedure: Here we will briefly outline the primary computational steps. In subsequent sections, we will discuss strategies for implementing these steps.

1. Get the nuclear repulsion energy (Enuc).
2. Use Psi to compute one-electron integrals
   (a) Compute the overlap integrals (S).
   (b) Compute the kinetic energy integrals (T).
   (c) Compute the potential energy integrals (V).
   (d) Form the core Hamiltonian (H), via $H_{\mu\nu} = T_{\mu\nu} + V_{\mu\nu}$.
3. Construct the orthogonalizing matrix $S^{-1/2}$
   (a) Diagonalize the S matrix,
   $$U^{\dagger} S U = \Lambda. \tag{1}$$
   (b) Form the $S^{-1/2}$ matrix,
   $$S^{-1/2} = U \Lambda^{-1/2} U^{\dagger}. \tag{2}$$
4. Construct an initial (guess) density matrix
   (a) Form the "core" Fock matrix in the orthogonalized basis via
   $$F'_0 = (S^{-1/2})^{\dagger} H S^{-1/2}. \tag{3}$$
   (b) Diagonalize the initial Fock matrix using a standard eigenvalue routine such as the DSYEV routine in the LAPACK library.
   $$C_0'^{\dagger} F'_0 C'_0 = \epsilon. \tag{4}$$
   Note: Two steps below this, you will be forming the density matrix using the columns of C that correspond to occupied orbitals. You need to know which columns of C correspond to occupied orbitals and which ones correspond to unoccupied orbitals. Many standard diagonalizers will sort the eigenvalues and eigenvectors for you, and if they sort them in increasing order of the eigenvalues, then you can just take the first $N/2$ columns of C when forming the D matrix, where N is the number of electrons (and hence $N/2$ is the number of doubly-occupied orbitals). But if your matrix diagonalizer does not sort the eigenvalues/eigenvectors, you will have to sort them yourself or else somehow make sure you are using the correct columns of C when forming D below.
   (c) Form the initial SCF eigenvector matrix in the original basis
   $$C_0 = S^{-1/2} C'_0. \tag{5}$$

(d) Form the initial density matrix, D

$$D_{\mu\nu} = \sum_{i}^{N/2} C_{\mu i} C_{\nu i}, \tag{6}$$

where N is the number of electrons (and hence $N/2$ is the number of doubly-occupied orbitals).

5. Perform the SCF iterations

   (a) Form the new Fock matrix, F, from the density matrix and the two-electron integrals

   $$F_{\mu\nu} = H_{\mu\nu} + \sum_{\rho\sigma}^{AO} D_{\rho\sigma} \left\{2[\mu\nu|\rho\sigma] - [\mu\rho|\nu\sigma]\right\}. \tag{7}$$

   (b) Calculate the electronic energy

   $$E = \sum_{\mu\nu}^{AO} D_{\mu\nu} \left(H_{\mu\nu} + F_{\mu\nu}\right) + E_{nuc}. \tag{8}$$

   (c) Transform the Fock matrix to the orthonormal basis

   $$F' = (S^{-1/2})^{\dagger} F S^{-1/2}. \tag{9}$$

   (d) Diagonalize the Fock matrix

   $$C'^{\dagger} F' C' = \epsilon. \tag{10}$$

   As discussed above, you probably want to ensure the eigenvectors (columns of C) are sorted so that the corresponding eigenvalues are in increasing order.

   (e) Construct the new SCF eigenvector matrix

   $$C = S^{-1/2} C' \tag{11}$$

   (f) Form the new density matrix

   $$D_{\mu\nu} = \sum_{i}^{N/2} C_{\mu i} C_{\nu i}, \tag{12}$$

   where $N/2$ is the number of doubly-occupied orbitals.

   (g) Test for convergence of the energy.

   $$\Delta E = E^{n} - E^{n-1} < \delta_E \tag{13}$$

(h) Optionally, also test convergence of the density and/or Fock matrix. One simple way to do this is just to compute the RMS change in the density matrix:

$$D_{rms} = \left[\sum_{\mu\nu}^{AO} \left(D^n_{\mu\nu} - D^{n-1}_{\mu\nu} \right)^2 \right]^{1/2} < \delta_D \tag{14}$$

This criterion is fine for small systems. For very large systems, it will become harder for this criterion to be met for a fixed RMS cutoff value. An alternative criterion that is sometimes used is the commutator of the density matrix and the Fock matrix, which will go to zero at convergence.

$$[D, F]_{\mu\nu} = \sum_{\rho\sigma}^{AO} S_{\mu\rho} D_{\rho\sigma} F_{\sigma\nu} - F_{\mu\rho} D_{\rho\sigma} S_{\sigma\nu} \tag{15}$$

$$\| [D, F] \|_F < \delta_D \tag{16}$$

(i) If not converged, do another iteration.

2 Project writeup

Before discussing three different technical strategies for performing the project, if you are doing this project as a class project, below are some suggestions for what to include in your project write-up (if you are just doing this as part of a research experience there is probably no reason to do a formal writeup).

1. Include a *short* introduction giving a brief re-cap of what Hartree-Fock is and how it works, in your own words
2. Give an introduction to how you coded the program: what language (or program like Mathematica) did you use, why did you pick that language, what features of the language were helpful, what things did the language not provide automatically that you had to code up yourself, did you have to use any special tricks, etc.
3. Provide the actual source code of your Hartree-Fock program and any helper functions you had to write
4. Provide at least one sample output from your program (including the corresponding input file, if you are not using the H_2O integrals file from Coding Strategy #1 below).
5. If you did not obtain the correct Hartree-Fock energy, explain what you think might have gone wrong with your program.

3 Coding strategy #1: reading data from a text file for a specific test case

The required data for STO-3G H_2O at its equilibrium geometry is available on the Sherrill group website at http://vergil.chemistry.gatech.edu/h2oints.txt. The file should be self-explanatory. There are 7 orbitals for this system in the STO-3G basis. Note that the numbering of orbitals in the two-electron integrals starts at 0. Feel free to reformat the file to suit your purposes.

Important note: The printout of the two-electron integrals only lists the *permutationally unique* integrals. Recall that the two-electron integrals have 8-fold permutational symmetry when using real orbitals:

$$[pq|rs] = [qp|rs] = [pq|sr] = [qp|sr] = [rs|pq] = [sr|pq] = [rs|qp] = [sr|qp]. \tag{17}$$

The equations given above for the Fock matrix assume all possible permutations of integrals are available. That means that in forming the Fock matrix you may find yourself needing integral [21|34], and you need to realize that it doesn't appear that way in the list, it appears as [43|21]. The integrals listed follow the canonical order that $p \geq q$, $r \geq s$, and with the indices on the left being larger than those on the right (this last requirement can be stated more formally by saying Psi4 requires that the "super-indices" pq and rs satisfy the requirement that $pq \geq rs$, where $pq = (p(p+1)/2) + q$, and analogously for rs, and with orbital numbering beginning with 0).

4 Coding strategy #2: Writing a general program in Python using Psi4

Another option for getting the required integrals is to interface to a program like Psi4. This is relatively easy to do using Psi4's very user/programmer-friendly Python front-end, which allows user input files to include not only molecule/computation information, but also any valid Python code. This means that it is possible to write a Hartree–Fock program in Psi4 using only the Psi4 binary and a single user input file. Python is a high-level language (i.e., it is easy to accomplish a lot in relatively few lines of code), it is widely used, and it is fairly easy to learn.

This approach will require you to install (or have access to) a binary of the Psi4 program. If you do not already have login access to a machine with Psi4 on it, you can either download and install a Psi4 binary (easiest), or download the Psi4 source and compile and install it. To download and install a binary, see the

"Psi4 Downloads".

If you create your program this way, once the Psi4 binary is available, you just need to create your program in an input file according to the template provided in Figure 1 and execute it from Psi4 like this: `psi4 my-program.in`.

Figure 1: Overall skeleton of a Psi4 program in an input file.

```
# Your program goes first, at the top of the file
def simple_scf(molecule):

  # make sure the molecule object gets correctly populated with
  # the current geometry
  molecule.update_geometry()

  # your Python program goes here, takes a Psi4 molecule as input

# after your program, then specify a molecule using normal
# Psi4 input and run the SCF using your python SCF code and then also
# with the usual Psi4 SCF code, so that you can compare final energies.
# You should be able to pick any small molecule you like
# (although we will assume RHF so make it closed-shell).

molecule mol {
0 1
O
H 1 1.0
H 1 1.0 2 104.5
symmetry c1
}

set {
basis sto-3g
}

simple_scf(mol)
set scf_type direct
energy('scf')
```

Psi4 provides routines that make one- and two-electron integrals available in Python, and a Matrix class that allows various operations like matrix multiplication, matrix diagonalization, etc. It also has a Molecule class that allows one to grab useful information from a user-specified molecule, like its overall charge, etc. All the functions necessary to write a Hartree–Fock program should be given below. However, you can find a complete list of all Psi4 functions available from Python in the Linking C++ and Python section of the Psi4 Manual.

Figure 2 illustrates how to use Psi4's integral generation routines to form all the required one- and two-electron integrals and make them accessible through Psi Matrix objects. Note: these matrices are all stored in core RAM; because the two-electron integrals are an $\mathcal{O}(N^4)$ quantity, you should stick to small test cases with your program. The integrals will be generated for the current default Molecule object; the skeleton program above will correctly specify a `mol` as the default (and only) molecule object, so there is no ambiguity.

Figure 2: Using Psi4's MintsHelper to generate integrals

```
# Integral Generation
wfn = Wavefunction.build(mol, get_global_option("basis"))
mints = MintsHelper(wfn.basisset())
S = mints.ao_overlap()
T = mints.ao_potential()
V = mints.ao_kinetic()
I = mints.ao_eri()
```

`S`, `T`, and `V` are all square matrices of size `nbf` x `nbf`, where `nbf` is the number of AO basis functions, which can be obtained by querrying the matrices about their number of rows (or columns), like this: `nbf = S.rows(0)`. The two-electron integrals are formally represented by a 4-dimensional tensor, but the Psi4 `MintsHelper` object packs them into a (2-dimensional) Matrix for convenience. It accomplishes this by computing a "composite" row index $pq = p * \texttt{nbf} + q$, where p and q are individual orbital indices.

The Psi4 Matrix objects have various built-in capabilities. Some of them are as follows:

1. Create new Matrix X with *nrows* rows and *ncols* cols: `X = Matrix(nrows, ncols)`

2. Get the p, q element of X: `val = X.get(p,q)`

3. Set the p, q element of X to *val*: `X.set(p,q,val)`

4. Add J to F: `F.add(J)`

5. Subtract K from F: `F.subtract(K)`

6. Multiply X by 2: `X.scale(2.0)`
7. Copy X to Y: `Y = X.clone()`
8. Matrix multiplication $C = \beta C + \alpha AB$ (set $\beta = 0$ if the result of αAB is to overwrite C instead of add to it): `X.gemm(transa, transb, alpha, A, B, beta)`, where `transa = True` if we need to transpose matrix A (otherwise `transa = False`), and `transb = True` if we need to transpose matrix B (otherwise `transb = False`).
9. Diagonalization of a symmetric Matrix:
   `X.diagonalize(evecs, evals, DiagonalizeOrder.Ascending)` will diagonalize symmetric matrix X, placing the eigenvectors in matrix `evecs` (one eigenvector per column) and the corresponding eigenvalues in `evals`, which is a Vector. Both `evecs` and `evals` should have already been created by the user (e.g., `evecs = Matrix(n,n)` and `evals = Vector(n)`). Note: this routine does not necessarily actually change the original matrix X to diagonal form.
10. Raise a matrix to the p-th power: `X.power(p, 0.0)`
11. Compute an element-wise dot product between two matrices (i.e., $\sum_{pq} A_{pq} \cdot B_{pq}$): `r = A.vector_dot(B)`

For this project you can specify molecular information like the charge, multiplicity, and number of electrons directly in your code, but it is more elegant to parse these out of the Molecule object itself. This is easily done as in Figure 3. Since your program assumes RHF, if you parse the information this way, your program should check to make sure `mult = 1` before it continues.

Figure 3: Using Psi4's Molecule object to get information

```
charge = molecule.molecular_charge()
mult   = molecule.multiplicity()
Z = 0
for A in range(molecule.natom()):
    Z += molecule.Z(A)
ndocc = int(Z / 2) - (charge / 2) # number of doubly-occupied orbitals
Enuc = molecule.nuclear_repulsion_energy()
```

5 Coding strategy #3: Writing a general program in C++ using Psi4

[Note: Psi4 has changed over the last couple of years since we wrote these notes. The plugin information is likely obsolete. I recommend using one of the two options above until we can update this part of the notes. –CDS]

This section describes how to use integrals and C++ libraries from Psi4 to write your program as a Psi4 "plugin." A plugin is compiled as a shared library and loaded by Psi4 at runtime. This has the advantage of making your program completely separate from the Psi4 source, while still allowing you access to all of Psi4's libraries. You will need to download the source code to the Psi4 package from www.psicode.org and get it compiled successfully.

1. **Creating a new plugin**
   Rather conveniently, you can create a new plugin by running PSI4 with the `--new-plugin` command line option.

   ```
       psi4 --new-plugin [plugin name]
   ```

   This will create a new directory with a Makefile, minimal source file, and sample input.

2. **Compiling your plugin**
   If you created your plugin with `--new-plugin` and your environment was configured properly, everything should just work. Typing `Make` in your plugin directory should produce an appropriately named `.so` file.

3. **Running your plugin**
   In the PSI4 input file, there needs to be a call to `plugin_load("Path to your plugin")` for initialization and a call to `plugin("Path to your plugin")` to run it. The sample input file created by `--new-plugin` will do this properly.

General features of PSI4
The following is an overview of the important classes in PSI4 that will be needed to complete this project. Examples of how to construct these classes can be found in the accompanying example plugin.

1. `boost::shared_ptr`
   These have permeated the entirety of the code. Essentially, these are just pointers with reference counting; moreover, they can be treated as though they were allocated with `new`, but do not need to be deleted.

2. `SharedMatrix` ≡ `boost::shared_ptr<Matrix>`
   This is PSI's `Matrix` class. Many matrix operations are defined for this class, the full definition can be found in

   ```
   $PSI/src/lib/libmints/matrix.h
   ```

   To avoid using the `Matrix` class, the underlying array is stored two dimensionally as a `double**` pointing to a contiguous chuck of memory. This can be obtained for some `SharedMatrix`, A, as

   ```
   double **Aptr = A->pointer();
   ```

3. `Process::environment`
   The pertinence of this object is that it allows communication between the PSI4 program and your plugin. References to various objects contained in PSI can be obtained from this object.

4. `Molecule`
   PSI's molecule class contains information about the molecular geometry and is needed to construct many other classes in PSI4. `Molecule` is defined in

   ```
   $PSI/src/lib/libmints/molecule.h
   ```

5. `BasisSet`
   This class contains the basis set definition for the molecule. It contains information such as the number of basis functions and the number of shells. `BasisSet` is defined in

   ```
   $PSI/src/lib/libmints/basisset.h
   ```

6. `Options`
   This class allows you to interact with the PSI4 input file by defining options that are unique to your plugin.

   ```
   $PSI/src/lib/liboptions/liboptions.h
   ```

   Functions that begin with `add_` can be used to introduce new options of a certain type in the `read_options` function in the plugin. The corresponding calls to `get_` will retrive the value of the option set in the input or the default value.

Selected useful functions

This is a list of functions that may prove useful in writing your program. Some of these functions are defined in:

```
$PSI/src/lib/libciomr/libciomr.h
```

1. **Allocating 1D arrays**
   Obviously, `malloc` and `new` are available. In PSI, the allocation of 1D arrays of doubles is wrapped in `double *init_array(unsigned long int)`; the allocation of a 1D array of integers is possible via `int *init_int_array(unsigned long int)`. These functions will also initialize the arrays to zero. Arrays allocated in this manner should be freed with the usual `free`.

2. **Allocating 2D arrays**
   To allocate a contiguous 2D array of doubles, PSI provides `double **block_matrix(unsigned long int, unsigned long int)`, which also zeros the array. These can be freed with `void free_block(double **)`.

3. **Obtaining useful quantities**

   (a) **Number of basis functions:** `int BasisSet::nbf()`

   (b) **Number of shells:** `int BasisSet::nshell()`

   (c) **Nuclear repulsion energy:** `double Molecule::nuclear_repulsion_energy()`

   (d) **Number of electrons:** This is actually difficult to extract from PSI in this context. It could be made an input parameter to your plugin. The alternative is to get the net charge from `Molecule` and add up the atomic charges to determine the number of electrons.

Useful linear algebra routines

Here are a list of the PSI wrappers to several useful BLAS and LAPACK functions. Note that the difference in C-style versus FORTRAN-style indexing can lead to some confusion when calling these functions. In some cases, notably DGEMV and DGEMM, PSI will take arguments in C-style and perform the necessary transpositions to FORTRAN-style indexing when it calls the underlying BLAS routine. The BLAS and LAPACK wrappers are found in the following files along with extensive comments:

`$PSI/src/lib/libqt/blas_intfc.cc`
`$PSI/src/lib/libqt/blas_intfc23`
`$PSI/src/lib/libqt/lapack_intfc.cc` Don't forget that BLAS and LAPACK will assume arrays are stored in linear memory. If you use PSI's functions for allocating memory, this won't become a problem.

1. **Scaling a vector**

```
void C_DSCAL(unsigned long int length, double alpha, double *vec, int inc)
```

2. **Copying a vector**

   ```
   void C_DCOPY(unsigned long int length, double *x, int inc_x,
   double *y, int inc_y)
   ```

3. **Adding two vectors**

   ```
   void C_DAXPY(unsigned long int length, double a, double *x, int inc_x,
   double *y, int inc_y)
   ```

4. **Dot product**

   ```
   double C_DDOT(unsigned long int length, double *x, int inc_x,
   double *y, int inc_y)
   ```

5. **Matrix-vector multiplication**

   ```
   void C_DGEMV(char trans, int m, int n, double alpha, double* a, int lda,
   double* x, int incx, double beta, double* y, int incy)
   ```

6. **Matrix-matrix multiplication**

   ```
   void C_DGEMM(char transa, char transb, int m, int n, int k, double alpha,
   double* a, int lda, double* b, int ldb, double beta, double* c, int ldc)
   ```

7. **Matrix diagonalization**

   ```
   int C_DSYEV(char jobz, char uplo, int n, double* a, int lda, double* w,
   double* work, int lwork)
   ```

Computing integrals with PSI4

1. **Set up**
   In order to create an object than generates integrals, an `IntegralFactory` must be constructed in order to build one-body or two-body integral objects. The factory will do some generic initialization and allows basis sets to be specified.

2. **Computing one-electron integrals with PSI4**
   One electron integrals can be computed in a single function call with the result stored in a `SharedMatrix`.

   (a) **Overlap integrals**
   A call to `IntegralFactory::ao_overlap` will provide a one-body integral object than can be used to construct the overlap integrals.

(b) **Potential integrals**
A call to `IntegralFactory::ao_potential` will provide a one-body integral object than can be used to construct the nuclear attraction integrals.

(c) **Kinetic integrals**
A call to `IntegralFactory::ao_kinetic` will provide a one-body integral object than can be used to construct the kinetic energy integrals.

3. **Computing two-electron integrals with PSI4**
The evaluation of the two-electron integrals is the most computationally intensive operation in a Hartree-Fock program. An object capable of generating the two-electron integrals can be obtained from the `IntegralFactory`.

(a) **Two-body integral object**
By calling `IntegralFactory::eri()`, a two-body integral object can be obtained that will generate electron repulsion integrals. The integrals generated by this object are indexed in Chemist's notation.

(b) **Shell quartets**
Two-electron integrals are not evaluated individually, but, rather a shell quartet at a time (e.g. *ssss*, *sssp*, etc.). The individual integrals are stored in an array in the two-body object. A pointer to this array is available for some `shared_ptr<TwoBodyAOInt> eri`:

```
const double *buffer = eri->buffer();
```

Inside this buffer, the integrals are indexed with a compound index. That is for a shell quartet, $(PQ|RS)$, with N_P functions in shell P indexed with p, etc. The compound index for integral $(pq|rs)$ is given as:
$pqrs = p * N_Q * N_R * N_S + q * N_R * N_S + r * N_S + s.$

(c) **Loop structure**
While it is possible to interact with the two-body integral object in other ways, the code snippet in Figure 4 defines a transparent loop structure using `boost::shared_ptr<BasisSet> aoBasis`.

Figure 4: A naive loop structure to evaluate *every* two-electron integral.

```
// Loop over all shells
for(int MU=0; MU<aoBasis->nshell(); MU++) {
  // Get the number of functions in the shell
  int nummu = aoBasis->shell(MU).nfunction();
  for(int NU=0; NU<aoBasis->nshell(); NU++) {
    int numnu = aoBasis->shell(NU).nfunction();
    for(int RHO=0; RHO<aoBasis->nshell(); RHO++) {
      int numrho = aoBasis->shell(RHO).nfunction();
      for(int SIG=0; SIG<aoBasis->nshell(); SIG++) {
        int numsig = aoBasis->shell(SIG).nfunction();

        // Compute the shell quartet
        eri->compute_shell(MU,NU,RHO,SIG);

        // Loop over all functions in the shell
        for(int mu=0, munu=0, index=0; mu<nummu; mu++) {
          // Extract the absolute index of the first function
          int omu = aoBasis->shell(MU).function_index() + mu;
          for(int nu=0; nu<numnu; nu++, munu++) {
            int onu = aoBasis->shell(NU).function_index() + nu;
            for(int rho=0; rho<numrho; rho++) {
              int orho = aoBasis->shell(RHO).function_index() + rho;
              for(int sig=0; sig<numsig; sig++, index++) {
                int osig = aoBasis->shell(SIG).function_index() + sig;

                // buffer[index] contains (omu onu| orho osig)
                double tei = buffer[index];

              } // End sig function index
            } // End rho function index
          } // End nu function index
        } // End mu function index
      } // End SIG Shell index
    } // End RHO Shell index
  } // End NU Shell index
} // End MU Shell index
```