Instructor: Prof. Lee M. Thompson E-mail: *lee.thompson.1@louisville.edu* Office: CB 251

1. Derive the time dependence of the wavefunction

$$\Psi(x,t) = \exp\left(\frac{-iEt}{\hbar}\right)\Psi(x,0) \tag{1}$$

from the time-dependent component of the Schrödinger equation

$$i\hbar\frac{\partial\Psi}{\partial t} = \hat{H}\Psi \tag{2}$$
$$\Psi(\mathbf{r},t) = \Phi(\mathbf{r})\Theta(t) \tag{3}$$
$$i\hbar\Phi(\mathbf{r})\frac{\partial\Theta(t)}{\partial t} = -\frac{\hbar^2}{2m}\Theta(t)\frac{d^2\Phi(\mathbf{r})}{d\mathbf{r}^2} + V(\mathbf{r})\Phi(\mathbf{r})\Theta(t) \tag{4}$$
$$i\hbar\frac{1}{\Theta(t)}\frac{d\Theta(t)}{dt} = E \tag{5}$$
$$\int\frac{1}{\Theta(t)}d\Theta = \frac{-iE}{\hbar}\int dt \tag{6}$$
$$\ln(\Theta(t)) = \frac{-iEt}{\hbar} \tag{7}$$
$$\Theta(t) = \exp\left(\frac{-iEt}{\hbar}\right) \tag{8}$$

2. Write the following expressions in bra-ket notation

a)

$$\int f^*(x)\psi(x)dx \tag{9}$$

b)

$$(\hat{H} - 5)\psi(x) = E'\psi(x) \tag{10}$$

c)

$$\frac{\int\psi^*(x)\hat{H}\psi_n(x)dx}{E_k - E_n} \tag{11}$$

d)

$$\psi(x)\int\psi^*(x')\psi(x')dx' \tag{12}$$

a)

$$\langle f(x)|\psi(x)\rangle \tag{13}$$

b)

$$(\hat{H} - 5)|\psi(x)\rangle = E'|\psi(x)\rangle \tag{14}$$

c)

$$\frac{\langle\psi(x)|\hat{H}|\psi_n(x)\rangle}{E_k - E_n} \tag{15}$$

d)

$$|\psi(x)\rangle\langle\psi(x')|\psi(x')\rangle \tag{16}$$

.

3. This week we are going to:

   - Set up a code repository with GitHub
   - Run a Gaussian calculation to output a "matrix file" which contains intermediates from electronic structure calculations
   - Use MQC to read in intermediates and be able to start manipulating them

Setting up your Git repository

We are going to set up a repository for revision control of your code using Git. Using revision control enables you to have a record of code as it develops, return to previous versions if desired, collaborate with multiple developers, fork and branch code to preserve stable and development versions, and merge together code developed in different branches. As you will be working on your own on the code, the main benefit is having a record of code so that if anything goes wrong you can return to the previous working version. Git repositories can be hosted on GitHub which is an online service that behaves like a server, enabling you to have a single version that can be accessed by others to fork, clone, use and develop. You can use GitHub to develop code on more than one computer by pushing and pulling updates to GitHub.

First log into the CRC Type "vi .bashrc" and then add lines "export GIT_EDITOR=vim", "export GIT_AUTHOR_EMAIL"**your louisville email**"" and "export GIT_AUTHOR_NAME="**your name**"". Type "mkdir CHEM_555". Last week you made a file named test.f03. Type "mv test.f03 CHEM_555" and then "cd CHEM_555". Type "git init" to initialize your git repository. Now type "git status" and you will see test.f03 and makefile has been identified as new, but not tracked in the repository. Type "git add test.f03" to track the file (you can check the output of "git status" to see that test.f03 is now being tracked but has been modified) and "git add makefile", and now commit (save) the changes, which effectively makes a checkpoint of your added code, by typing "git commit". Add a descriptive message to display so you know at what point this commit corresponds to (press i to enter insert mode) and then save and quit (:wq after exiting insert mode using Esc). If you decide against committing your code, you can cancel the commit by exiting (:wq) without inserting any text. At this point you could push your commit to GitHub with "git push" but you have not set up the remote server, so you must do this first.

On GitHub, log into your account, and in the top-right corner, click on the "+" symbol and then click "Create Repository" from the drop down menu. Write "CHEM_555" in the box titled

Repository name and then click Create repository. On the next page, copy the SSH repository address in the Quick setup box to the clipboard. Go back to the command line (on the CRC) and type "git remote add origin **url_that_you_copied**" and then "git remote -v" to check that the remote has been added correctly. Now you have set up the remote you can type "git push origin master" to push your commit to GitHub. You only need to set up the remote on GitHub (this paragraph) the first time you set up a repository.

As you develop code, when you want to create a checkpoint, type "git status" to see which files have changed and which files are not tracked. Add files using "git add **filename**" as necessary. You can also type "git diff **filename**" if you want to see what has changed since the last commit and "git checkout **filename**" to reset the file to the version at the previous checkpoint (be careful because you will lose all changes to the file since the last commit). Once you have added all the files, type "git commit" and save the commit message. You can view the log of commits by typing "git log –graph" and push to the GitHub remote using "git push". If you have multiple users pushing to the same GitHub remote, you will need to use "git pull" to copy changes from the GitHub remote to your local repository.

Obtaining a matrix file from Gaussian

We will now run a Gaussian calculation in order to extract a matrix file containing electronic structure intermediate matrices that you can read into your code with MQC. First we will need to install a few scripts to make running and monitoring Gaussian calculations easier. Type "cd" and then type "git clone git@github.com:thompsonresearchgroup/crc_tools.git bin". Type "cd bin" then "ln -s gsub_crc gsub" and then "cd". Your settings on the CRC already allow you to access the scripts inside the folder named bin on your home directory so there should be no further set-up required. Now we will create a Gaussian input file that can output a matrix file, using H_2 as our example molecule. Type "vi workshop_4.com" and add the following text to the file:

```
#P output=matrix nosymm scf=conven int=noraff

test

0 1
H
H 1 0.7

workshop_4.mat
```

Exit out of the file with :wq. Type "module load g16reva03_lcpu" and then type "gsub workshop_4.com" which will submit the job to the queuing system on the CRC. If you get an error saying "command not found", type "vi /.bashrc" and add the line "export PATH="/home/**ULink_ID**/bin:$PATH"" and then back on the command line type "source /.bashrc". You can check the status of your calculation by typing "qstat" and view the output file by typing "less workshop_4.log" (press h to see a list of options for navigating files with less). Upon completion the calculation should have produced a file named "workshop_4.mat" which you can check using the "ls" command.

Reading a matrix file using MQC

Now that we have produced a matrix file, we will write a code using Fortran/MQC that can read in the intermediates and print them out. Type "cd /CHEM_555" and then type "cp test.f03 workshop_4.f03" and "vi workshop_4.f03". Will will modify the code as follows:

1. Change the name of the program on the first and last lines to workshop_4.
2. Delete the lines where we declare the type of variables A, B and C, as well as the lines where we assign them, multiply and print them (i.e. you should just have a bare-bones piece of code).
3. We will need a variable that holds the file name of the matrix file so the code knows where to access the data. First we must declare the variable as a character string. Write "character(len=:),allocatable::fileName" on the line after implicit none. Then, leave a blank line followed by "call mqc_get_command_argument(1,fileName)" tells the code to read the first command line argument into the variable *fileName*.
4. Now that we know the name of the file where the data is stored, we need to load the data into a variable within the code, which we will call *matFile*. First, in the type declaration section, add a new line "type(mqc_gaussian_unformatted_matrix_file)::matFile". Then in the main code block, on the line after the command argument call, add the line "call matFile%load(fileName)".
5. The variable *matFile* loads the data from the matrix file so that we can now load up specific variables containing various electronic structure intermediates. We are going to create three variables – a wavefunction object that contains variables relating to electronic structure, a molecule object that contains data regarding nuclear structure, and a two-electron integral object which contains the two-electron integrals. In the type declaration section, add the lines

   ```
   type(mqc_wavefunction)::wavefunction
   type(mqc_molecule_data)::molecule
   type(mqc_twoERIs)::twoeris
   ```

   Then in the main code block add

   ```
   call matFile%getESTObj('wavefunction',wavefunction)
   call matFile%getMolData(molecule)
   call matFile%get2ERIs('regular',twoeris)
   ```

   to fill the variables with data.
6. Now we are in a position where we can start developing code. First I will explain a little about the variables we have just constructed, taking the *wavefunction* variable as a representative example. MQC uses an object-orientated programming paradigm. The wavefunction is an *object* that contains *data* and *procedures*. The data is a list of different variables – in our case e.g. molecular orbital coefficients, number of electrons, overlap matrix etc. (we will cover the data in more detail in another workshop) – while procedures are subroutines or functions that can act on data within the object. Complicating matters somewhat is the fact that data in a procedure can also be an object with its own (different) data and procedures. In addition, you can have a *class* which is an object that extends other objects (e.g. the

shape class contains the object rectangle which contains the object square) so that code can be written to work with any object in the class without having to write different code for different objects.

7. It is very common that objects have a print procedure that allows you to output the values of data stored in the object. We are going to call procedures to output data stored in the variables we declared. First let's print some information about the molecule object. In the typing block, add the line "type(mqc_scalar)::Vnn" and in the main code block, add the line "Vnn = molecule%getNucRep()" to output the nuclear repulsion into the variable Vnn. On the next line write "call Vnn%print(6,'Nuclear Repulsion Energy (au)')" to print the nuclear repulsion energy. Now in the main code block add the line "call molecule%print(6)" to output the nuclear geometry.

8. We will now print data from the wavefunction object. Add the line to the main code block "call wavefunction%print(6,'all')". Note that you could print a specific variable by changing the string option (the various options are overlap, core hamiltonian, orbital energies, mo coefficients,density, scf density, fock, nbasis, nalpha, nbeta, nelectrons, charge, multiplicity, type, complex, all) e.g. call wavefunction%print(6,'type') to display if the wavefunction is from a restricted, unrestricted, or general Hartree-Fock calculation.

9. We can also print the two-electron integrals, using the line "call twoeris%print(6,'AO 2ERIs')", where the string is just a heading that is printed along with the data.

Once you have modified your file as described above, you should have the following code:

```
program workshop_4

use mqc_gaussian

implicit none
character(len=:),allocatable::fileName
type(mqc_gaussian_unformatted_matrix_file)::matFile
type(mqc_wavefunction)::wavefunction
type(mqc_molecule_data)::molecule
type(mqc_twoERIs)::twoeris
type(mqc_scalar)::Vnn

call mqc_get_command_argument(1,fileName)
call matFile%load(fileName)
call matFile%getESTObj('wavefunction',wavefunction)
call matFile%getMolData(molecule)
call matFile%get2ERIs('regular',twoeris)
Vnn = molecule%getNucRep()
call Vnn%print(6,'Nuclear Repulsion Energy (au)')
call molecule%print(6)
call wavefunction%print(6,'all')
call twoeris%print(6,'AO 2ERIs')

end program workshop_4
```

Compile the code using the instructions from the last workshop. Once your code has compiled you can run it on the command line using "./workshop_4.exe ../workshop_4.mat". In addition if you wish to save the output, your can redirect it to a file using "./workshop_4.exe ../workshop_4.mat > workshop_4.out".

If you type "git status" you should now see that there are several untracked files in your git repository. Type "git add workshop_4.f03" and then "git commit" with a message such as 'Wrote code as described in tutorial 4". Now enter "git push" to update your repository on git hub. You can check your new commit by typing "git log".