Lab 2: QMC Basics

1. Overview

This lab is generally focused on the basics of performing quality quantum Monte Carlo (QMC) calculations. Practical topics covered in this lab include wavefunction optimization with variational Monte Carlo (VMC), diffusion Monte Carlo (DMC) timestep extrapolation, DMC population control bias, and automation of DMC workflows in the context of pseudopotential testing. Similar tests are an essential part of most QMC studies and the other QMC topics covered here are completely transferrable to larger, production calculations of more complicated material systems. In this lab, participants will test the quality of the Burkatzki-Filippi-Dolg oxygen pseudopotential by calculating the ionization potential of atomic oxygen and the binding properties of the oxygen dimer with DMC.

1.1 Getting the most out of this lab

The outline below shows the overall structure of the lab. Those who are new to QMC or QMCPACK should probably work through the contents in order. If you have a specific application/target system you would like to explore with QMCPACK, be sure to leave enough time for the optional material in section 4. Feel free to discuss your answers/results to the questions/exercises at the end of each section with a lab instructor.

1. **Overview**
   Overview of lab content.
   - **1.1 Getting the most out of this lab**
     This section.
   - **1.2 Lab directories and files**
     Description of directories and files used in the lab.
   - **1.3 The QMCPACK input file and XML**
     XML as used by QMCPACK. Reduced example of input file structure.
2. **Testing PP atomic properties: optimization, diffusion Monte Carlo**
   Calculate ionization potential of oxygen using pre-generated QMCPACK input files.
   - **2.1 Getting and converting a pseudopotential**
     Download PP from BFD database. Convert to QMCPACK format with `ppconvert`.
   - **2.2 Optimization walkthrough: neutral O atom**
     Theoretical background on trial wavefunction & optimization. QMCPACK optimization walkthrough. Fully annotated input file (`O.q0.opt.in.xml`) and explanation of Jastrow & optimization inputs.
   - **2.3 DMC timestep extrapolation I: neutral O atom**
     Theoretical background on timestep & population control biases. DMC

timestep extrapolation walkthrough w/ QMCPACK and explanation of DMC inputs.

2.4 DMC timestep extrapolation II: IP of oxygen
Optimization and timestep extrapolation of charged oxygen atom. Timestep extrapolation of DMC ionization potential & comparison w/ experimental data.

3. Testing PP dimer properties: DMC workflow automation
Calculate oxygen dimer binding curve w/ the Project Suite workflow automation system.

3.1 Example Project Suite input
Explanation of Project Suite inputs for simple VMC workflow (Python).

3.2 Automated binding curve of the oxygen dimer
Explanation of optimization & DMC inputs. Workflow w/ single optimization at eqm. bond length and several DMC runs for stretched/compressed dimer. Comparison of fitted eqm. bond length and dissociation energy w/ experimental data.

4. (Optional) Running your system with QMCPACK
Generate input files for (and optionally run) PWSCF and QMCPACK for your own physical system with the Project Suite. The 8-atom cubic unit cell of diamond is provided as a runnable example.

A. Basic Python constructs
Appendix with brief overview of Python syntax: intrinsics, container types, conditional statements, iteration, functions w/ keyword arguments. Possibly useful for those new to Python in working with the Project Suite (consult as needed).

1.2 Lab directories and files

```
Lab_2_QMC_Basics/

 docs                     - documentation
    Lab_2_QMC_Basics.pdf    - this document
    Lab_2_Slides.pdf        - slides presented during the lab
    Project_Suite.pdf       - slides on QMCPACK automation (supplementary)

 oxygen_atom              - oxygen atom calculations
    ip_conv.py              - tool to fit oxygen IP vs timestep
    O.q0.dmc.in.xml         - neutral O DMC input file
    O.q0.dmc.qsub.in        -    "    "  "  submission file
    O.q0.opt.in.xml         -    "    " optimization input file
    O.q0.opt.qsub.in        -    "    "  "  submission file
    O.q0.pwscf.h5           -    "    "  orbitals file
```

```
    O.q1.dmc.in.xml         - charged O DMC input file
    O.q1.dmc.qsub.i         -    "    "  "  submission file    n
    O.q1.opt.in.xml         -    "    " optimization input file
    O.q1.opt.qsub.i         -    "    "  "  submission file    n
    O.q1.pwscf.h5           -    "    "  orbitals file
    reference               - directory w/ completed runs
    submit_O_q0_dmc         - executable to submit neutral DMC
    submit_O_q0_opt         -    "       "    "       "    optimization
    submit_O_q1_dmc         -    "       "    "    charged DMC
    submit_O_q1_opt         -    "       "    "       "    optimization

 oxygen_dimer              - oxygen dimer calculations
    dimer_fit.py             - tool to fit dimer binding curve
    O_dimer.py               - automation script for dimer calculations
    pseudopotentials         - directory for pseudopotentials
    reference                - directory w/ completed runs

 your_system               - calculations with your own physical system
     example.py              - generates input files for your system
     pseudopotentials        - directory for pseudopotentials
     reference               - directory w/ completed runs
```

1.3 The QMCPACK input file and XML

This section introduces XML as it is used in QMCPACK's input file. The focus is on the XML file format itself and the general structure of the input file rather than an exhaustive discussion of all keywords and structure elements. Specific keywords and the relevant XML elements are discussed in context as they are encountered in the lab. Participants will work with a complete, annotated input file for the oxygen atom during the wavefunction optimization walkthrough in section 2.2.

QMCPACK uses XML to represent structured data in its input file. Instead of text blocks like

```
begin project
  id     = vmc
  series = 0
end project

begin vmc
  move     = pbyp
  blocks   = 200
  steps    =  10
```

```
  timestep = 0.4
end vmc
```

QMCPACK input looks like

```
   <project id="vmc" series="0">
   </project>

   <qmc method="vmc" move="pbyp">
      <parameter name="blocks"  >  200 </parameter>
      <parameter name="steps"   >   10 </parameter>
      <parameter name="timestep">  0.4 </parameter>
   </qmc>
```

XML elements start with `<element_name>`, end with `</element_name>`, and can be nested within each other to denote substructure (the trial wavefunction is composed of a Slater determinant and a Jastrow factor, which are each further composed of ...). `id` and `series` are attributes of the `<project/>` element. XML attributes are generally used to represent simple values, like names, integers, or real values. Similar functionality is also commonly provided by `<parameter/>` elements like those shown above.

The overall structure of the input file reflects different aspects of the QMC simulation: the simulation cell, particles, trial wavefunction, Hamiltonian, and QMC run parameters. A condensed version of the actual input file is shown below:

```
<?xml version="1.0"?>
<simulation>

  <project id="vmc" series="0">
    ...
  </project>

  <qmcsystem>

    <simulationcell>
      ...
    </simulationcell>

    <particleset name="e">
      ...
```

```
    </particleset>

    <particleset name="ion0">
      ...
    </particleset>

    <wavefunction name="psi0" ... >
      ...
      <determinantset>
        <slaterdeterminant>
          ..
        </slaterdeterminant>
      </determinantset>
      <jastrow type="One-Body" ... >
         ...
      </jastrow>
      <jastrow type="Two-Body" ... >
        ...
      </jastrow>
    </wavefunction>

    <hamiltonian name="h0" ... >
      <pairpot type="coulomb" name="ElecElec" ... />
      <pairpot type="coulomb" name="IonIon"   ... />
      <pairpot type="pseudo" name="PseudoPot" ... >
        ...
      </pairpot>
    </hamiltonian>

   </qmcsystem>

   <qmc method="vmc" move="pbyp">
     <parameter name="warmupSteps">   20 </parameter>
     <parameter name="blocks"     >  200 </parameter>
     <parameter name="steps"      >   10 </parameter>
     <parameter name="timestep"   >  0.4 </parameter>
   </qmc>

</simulation>
```

The omitted portions (...) are more fine-grained inputs such as the axes of the simulation cell, the number of up and down electrons, positions of atomic species, external orbital files, starting Jastrow parameters, and external pseudopotential files. Relevant portions will be explained in more detail throughout the lab.

2. Testing PP atomic properties: optimization, diffusion Monte Carlo

2.1 Getting and converting a pseudopotential

The Burkatzki-Filippi-Dolg (BFD) pseudopotential (PP) database is a respected source of pre-tested PP's for use in QMC calculations. The pseudopotentials are represented in gaussian basis sets and are naturally suited for the study of molecular systems (*e.g.* using GAMESS to obtain orbitals). The PP's can also be used in solid state calculations (*e.g.* using Quantum Espresso to obtain orbitals), but the planewave cutoff required for converged results may become prohibitive for heavier elements. In this case, DFT-based pseudopotentials may be generated with the OPIUM package (beyond the scope of this lab). In either case, testing pseudopotentials in relevant environments should be performed. To illustrate a subset of possible tests while gaining familiarity with QMCPACK we will work with an oxygen pseudopotential from the BFD database.

To obtain the pseudopotential, go to http://www.burkatzki.com/pseudos/index.2.html and click on the "Select Pseudopotential" button. Next click on oxygen in the periodic table. Click on the empty circle next to "V5Z" (a large gaussian basis set) and click on "Next". Select the Gamess format and click on "Retrive Potential". Helpful information about the pseudopotential will be displayed. The desired portion is at the bottom (the last 7 lines). Copy this text into the editor of your choice and save it as `O.BFD.gamess` (be sure to include a newline at the end of the file). To transform the pseudopotential into the fsatom xml format used by QMCPACK, use the `ppconvert` tool:

```
ppconvert --gamess_pot O.BFD.gamess --s_ref "1s(2)2p(4)" \
 --p_ref "1s(2)2p(4)" --d_ref "1s(2)2p(4)" --xml O.BFD.xml
```

Observe the notation used to describe the reference valence configuration for this helium-core PP: `1s(2)2p(4)`. The `ppconvert` tool uses the following convention for the valence states: the first s state is labeled `1s` (`1s`, `2s`, `3s`, ...), the first p state is labeled `2p` (`2p`, `3p`, ...), the first d state is labeled `3d` (`3d`, `4d`, ...). Copy the resulting xml file into the `oxygen_atom` directory.

Note: the command to convert the PP into QM Espresso's UPF format is similar:

```
ppconvert --gamess_pot O.BFD.gamess --s_ref "1s(2)2p(4)" \
 --p_ref "1s(2)2p(4)" --d_ref "1s(2)2p(4)" --log_grid --upf O.BFD.upf
```

For reference, the text of `O.BFD.gamess` should be:

```
O-QMC GEN 2 1
3
6.00000000 1 9.29793903
55.78763416 3 8.86492204
-38.81978498 2 8.62925665
1
38.41914135 2 8.71924452
```

The full QMCPACK pseudopotential is also included in `oxygen_atom/reference/O.BFD.xml`.

2.2 Optimization walkthrough: neutral O atom

The aim of this section is to obtain a trial wavefunction of reasonable quality for the neutral oxygen atom. The first subsection provides background regarding the wavefunction for this system, including the specific form of the Jastrow factors used in QMCPACK. A brief discussion of wavefunction optimization is also given. The second subsection contains the actual walkthrough to follow for the lab.

Background on trial wavefunction and optimization

The trial wavefunction used to describe the neutral oxygen atom is of the standard Slater-Jastrow form:

$$\Psi_T = e^{-(J_1+J_2)} D^{\uparrow}(\{\phi_u^{\uparrow}\}_{u=1}^{N^{\uparrow}}) D^{\downarrow}(\{\phi_d^{\downarrow}\}_{d=1}^{N^{\uparrow}}) \tag{2.2.1}$$

The orbitals forming the spin-restricted Slater determinants ($D^{\uparrow}/D^{\downarrow}$) are obtained from DFT or Hartree-Fock (*e.g.* via Quantum Espresso) and are fixed. The ground state of the (pseudo) oxygen atom is spin polarized with $N^{\uparrow} = 4$ and $N^{\downarrow} = 2$.

The part of the wavefunction we will be optimizing is the Jastrow factor ($e^{-(J_1+J_2)}$), which in this case includes one- (electron-ion) and two- (electron-electron) body correlation functions. The Jastrow factor is symmetric under same-spin electron exchange and does not affect the DMC fixed node approximation. Optimization of the Jastrow factor does, however, improve the efficiency of the DMC calculation and reduces additional approximations due to non-local pseudopotentials (locality approximation, T-moves).

The explicit form of the one-body Jastrow factor we will be using is

$$J_1 = \sum_{e=1}^{N^{\uparrow}+N^{\downarrow}} U_1^{\uparrow/\downarrow}(|r_e - r_O|) \tag{2.2.2}$$

where r_e refers to the electron positions and r_O is the position of the oxygen ion. The $U_1^{\uparrow/\downarrow}$ term is a one-dimensional radial function represented with piecewise continuous cubic

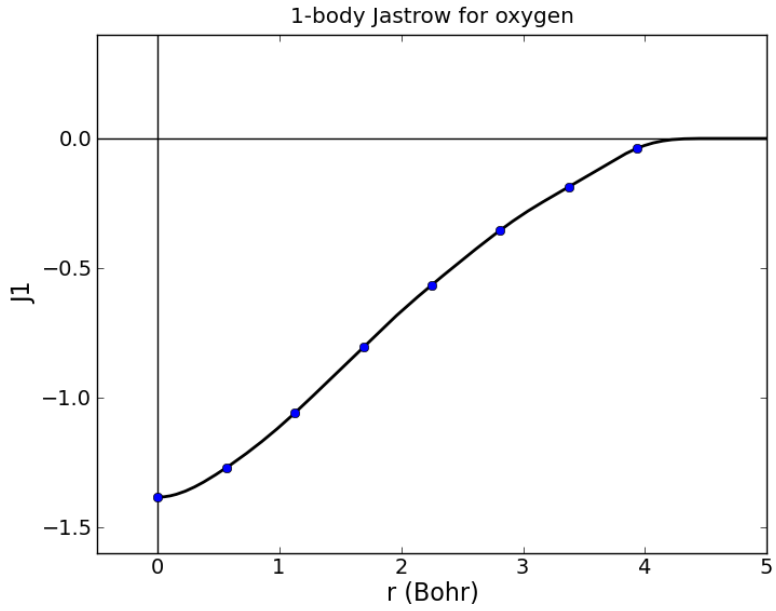

Figure 2.1: Optimized U_1 function for 1-body Jastrow factor of an oxygen atom.

polynomials (B-splines). The adjustable parameters to be optimized are the "knots" of the B-splines which are simply the values of the U_1 function at uniformly spaced grid points (See fig. 2.1 for an example of a U_1 spline function with 8 knots).

The two-body Jastrow factor is spin resolved ($r^{\uparrow}/r^{\downarrow}$ are up/down electron positions):

$$J_2 = \sum_{u<u'} U_2^{\uparrow\uparrow/\downarrow\downarrow}(|r_u^{\uparrow} - r_{u'}^{\uparrow}|) + \sum_{d<d'} U_2^{\uparrow\uparrow/\downarrow\downarrow}(|r_d^{\downarrow} - r_{d'}^{\downarrow}|) + \sum_{u,d} U_2^{\uparrow\downarrow}(|r_u^{\uparrow} - r_d^{\downarrow}|) \tag{2.2.3}$$

For an atom, Padé functions are appropriate for $U_2^{\uparrow\uparrow/\downarrow\downarrow}$ and $U_2^{\uparrow\downarrow}$:

$$U_2(r) = \frac{Ar}{1 + Br} \tag{2.2.4}$$

Only $B^{\uparrow\uparrow/\downarrow\downarrow}$ and $B^{\uparrow\downarrow}$ are adjustable since the A parameters are fixed by the electron-electron cusp conditions.

Wavefunction optimization essentially relies on two inequalities regarding energy and variance:

$$E_T(P) = \frac{\langle \Psi_T(P)|\hat{H}|\Psi_T(P)\rangle}{\langle \Psi_T(P)|\Psi_T(P)\rangle} \geq E_0 \tag{2.2.5}$$

$$V_T(P) = \frac{\langle \Psi_T(P)|\hat{H}^2|\Psi_T(P)\rangle}{\langle \Psi_T(P)|\Psi_T(P)\rangle} - \left(\frac{\langle \Psi_T(P)|\hat{H}|\Psi_T(P)\rangle}{\langle \Psi_T(P)|\Psi_T(P)\rangle}\right)^2 \geq 0 \tag{2.2.6}$$

Here E_0 is the ground state energy, $E_T(P)$ is the trial energy, $V_T(P)$ is the trial variance, and P denotes the set of adjustable parameters in the trial wavefunction. Equality is reached

only for the true ground state wavefunction and so the trial wavefunction can be improved by attempting to minimize a chosen cost function:

$$C(P) = \alpha E_T(P) + (1-\alpha)V_T(P). \tag{2.2.7}$$

Iterative varational Monte Carlo methods have been developed to handle the non-linear optimization problem $\min_P C(P)$. We will be using the linearized optimization method of Umrigar, *et al.* (PRL **98** 110201 (2007)). Let us try this now with QMCPACK.

Optimization walkthrough with QMCPACK

Enter the `oxygen_atom` directory and copy over the oxygen pseudopotential (`O.BFD.xml`) you downloaded and converted (section 2.1). Alternatively, the already converted pseudopotential is located in the `oxygen_atom/reference` directory. All files prefixed with "`O.q0`" relate to the neutral oxygen atom.

Open `O.q0.opt.in.xml` with your favorite text editor. This is a QMCPACK input file configured for wavefunction optimization with the linear method. Take a minute to familiarize yourself with the general format and contents of the input file. The major sections are the simulation cell, description of particle species (electrons & ions/atoms), the trial wavefunction (orbitals, Slater determinants, and Jastrow factors), the Hamiltonian, and finally inputs describing the quantum Monte Carlo process (linear optimization in this case). Portions marked with "`<!-- ... -->`" are comments describing these sections. XML is not the easiest to read, but this can be helped by using an editor with color highlighting such as `emacs` or `vi`.

The most important parts to focus on for the purposes of this exercise are the Jastrow factors and the inputs to the linear optimization method. Input specifying the one-body electron-ion Jastrow factor corresponding to eq. 2.2.2 is

```
<jastrow type="One-Body" name="J1" function="bspline" source="ion0" print="yes">
  <correlation elementType="O" size="8" rcut="4.5" cusp="0.0">
    <coefficients id="eO" type="Array">
      0 0 0 0 0 0 0 0
    </coefficients>
  </correlation>
</jastrow>
```

The XML describes $U_1^{\uparrow/\downarrow}(r)$ as a B-spline with 8 knots, no cusp at the origin (the oxygen pseudopotential is finite at $r = 0$), and vanishing beyond 4.5 Bohr. The initial guess of zero for each of the 8 knot parameters corresponds to $U_1^{\uparrow/\downarrow}(r) = 0$. The input for the two-body electron-electron Jastrow is similar:

```
<jastrow type="Two-Body" name="J2" function="pade" print="yes">
  <correlation speciesA="u" speciesB="u">
    <var id="uu_b" name="B">   0.6   </var>
  </correlation>
  <correlation speciesA="u" speciesB="d">
    <var id="ud_b" name="B">   1.0   </var>
  </correlation>
</jastrow>
```

The XML describes $U_2^{\uparrow\uparrow/\downarrow\downarrow}(r)$ and $U_2^{\uparrow\downarrow}(r)$ from eq. 2.2.3 as Padé functions with initial guesses of $B^{\uparrow\uparrow} = 0.6$ Bohr^{-1} and $B^{\uparrow\downarrow} = 1.0$ Bohr^{-1} for the adjustable parameters.

The relevant portion of the input describing the linear optimization process is

```
<loop max="MAX">
  <qmc method="linear" move="pbyp" checkpoint="-1">
    <cost name="energy"              >  ECOST   </cost>
    <cost name="unreweightedvariance">  UVCOST  </cost>
    <cost name="reweightedvariance"  >  RVCOST  </cost>
    <parameter name="timestep"       >  TS      </parameter>
    <parameter name="samples"        >  SAMPLES </parameter>
    <parameter name="warmupSteps"    >  300     </parameter>
    <parameter name="blocks"         >  800     </parameter>
    <parameter name="subSteps"       >  10      </parameter>
    <parameter name="nonlocalpp"     >  yes     </parameter>
    <parameter name="useBuffer"      >  yes     </parameter>
    ...
  </qmc>
</loop>
```

An explanation of each input variable can be found below. The remaining variables control specialized internal details of the linear optimization algorithm. The meaning of these inputs is beyond the scope of this lab and reasonable results are often obtained keeping these values fixed.

energy Fraction of trial energy in the cost function.

unreweightedvariance Fraction of unreweighted trial variance in the cost function. Neglecting the weights can be more robust.

reweightedvariance Fraction of trial variance (including the full weights) in the cost function.

timestep Timestep of the VMC random walk, determines spatial distance moved by each electron during MC steps. Should be chosen such that the acceptance ratio of MC moves is around 50% (30-70% is often acceptable). Reasonable values are often between 0.2 and 0.6 Ha^{-1}.

samples Total number of MC samples collected for optimization, determines statistical error bar of cost function. Often efficient to start with a small number of samples (5-20k) and then increase (20-100k). More samples may be required if the wavefunction contains a large number of variational parameters. MUST be be a multiple of the number of threads/cores (use multiples of 512 on Vesta).

warmupSteps Number of MC steps discarded as a warmup or equilibration period of the random walk. If this is too small, it will bias the optimization procedure.

blocks Number of average energy values written to output files. Should be greater than 200 for meaningful statistical analysis of output data (*e.g.* via `qmca`).

subSteps Number of MC steps in between energy evaluations. Each energy evaluation is expensive so taking a few steps to decorrelate between measurements can be more efficient. Will be less efficient with many substeps.

nonlocalpp,useBuffer If no, evaluate non-local pseudopotential derivatives approximately during optimization. This saves time and often does not affect optimization results unless the non-local contribution to the energy is large.

loop max Number of times to repeat the optimization. Using the resulting wavefunction from the previous optimization in the next one improves the results. Typical choices range between 4 and 20.

The three components of the cost function, energy, unreweighted variance, and reweighted variance should sum to one. Dedicating 100% of the cost function to unreweighted variance is often a good choice. Another common choice is to try 90/10 or 80/20 mixtures of reweighted variance and energy.

Replace `MAX`, `EVCOST`, `UVCOST`, `RVCOST`, `TS`, and `SAMPLES` in the two `loop`'s with appropriate starting values in the `O.q0.opt.in.xml` input file. Submit the optimization job to Vesta's queue by typing `./submit_O_q0_opt`. The job should only take a few minutes for reasonable values of loop `max` and `samples`.

Log file output will appear in `O_q0_opt.output`. The beginning of each linear optimization will be marked with text similar to

```
=========================================================
  Start QMCFixedSampleLinearOptimize
  File Root O_q0_opt.s011 append = no
=========================================================
```

At the end of each optimization section the change in cost function, new values for the Jastrow parameters, and elapsed wallclock time are reported:

```
OldCost: 7.4701713964e-01 NewCost: 7.4681622535e-01 Delta Cost:-2.0091428584e-04
...
  <optVariables href="O_q0_opt.s011.opt.xml">
eO_0 -9.5623201640e-01 1 1  ON 0
eO_1 -8.4728730387e-01 1 1  ON 1
eO_2 -6.8954452383e-01 1 1  ON 2
eO_3 -4.9327199567e-01 1 1  ON 3
eO_4 -3.2560096773e-01 1 1  ON 4
eO_5 -1.9567566480e-01 1 1  ON 5
eO_6 -1.2940405487e-01 1 1  ON 6
eO_7 -9.5221474839e-02 1 1  ON 7
uu_b 4.2002038228e-01 0 1  ON 8
ud_b 6.3472757070e-01 0 1  ON 9
  </optVariables>
...
  QMC Execution time = 7.0060820112e+00 secs
```

The cost function should decrease during each linear optimization (`Delta cost < 0`). Try "`grep OldCost *.output`". You should see something like this:

```
OldCost: 1.3644746067e+00 NewCost: 1.1049104640e+00 Delta Cost:-2.5956414268e-01
OldCost: 1.0690085060e+00 NewCost: 8.3206148222e-01 Delta Cost:-2.3694702381e-01
OldCost: 7.8558402137e-01 NewCost: 7.2478477600e-01 Delta Cost:-6.0799245374e-02
OldCost: 7.3070322298e-01 NewCost: 7.1655770805e-01 Delta Cost:-1.4145514926e-02
OldCost: 1.2184771084e+00 NewCost: 1.1923197177e+00 Delta Cost:-2.6157390699e-02
OldCost: 6.8740347812e-01 NewCost: 6.8733036689e-01 Delta Cost:-7.3111228164e-05
OldCost: 6.9683928634e-01 NewCost: 6.9681780340e-01 Delta Cost:-2.1482934426e-05
OldCost: 6.7982953532e-01 NewCost: 6.7982948866e-01 Delta Cost:-4.6667065545e-08
OldCost: 6.8674328187e-01 NewCost: 6.8674327833e-01 Delta Cost:-3.5391565234e-09
OldCost: 7.5998537866e-01 NewCost: 7.5965629336e-01 Delta Cost:-3.2908530361e-04
OldCost: 7.0771416413e-01 NewCost: 7.0765392787e-01 Delta Cost:-6.0236255172e-05
OldCost: 7.4701713964e-01 NewCost: 7.4681622535e-01 Delta Cost:-2.0091428584e-04
```

Blocked averages of energy data, including the kinetic energy and components of the potential energy, are written to `scalar.dat` files. The first is named "`O_q0_opt.s000.scalar.dat`", with a series number of zero (`s000`). In the end there will be `MAX1+MAX2` of them, one for each series.

When the job has finished, use the `qmca` tool to assess the effectiveness of the optimization process. To look at just the total energy and the variance, type "`qmca -q ev O_q0_opt*scalar*`". This will print the energy, variance, and the variance/energy ratio in Hartree units:

```
                          LocalEnergy              Variance           ratio
O_q0_opt  series 0  -15.568764 +/- 0.003421   1.382681 +/- 0.056604   0.0888
O_q0_opt  series 1  -15.638500 +/- 0.005014   1.067662 +/- 0.019865   0.0683
O_q0_opt  series 2  -15.802163 +/- 0.002680   0.834521 +/- 0.007037   0.0528
O_q0_opt  series 3  -15.840982 +/- 0.001791   0.752242 +/- 0.009477   0.0475
O_q0_opt  series 4  -15.841584 +/- 0.003301   1.097355 +/- 0.252991   0.0693
O_q0_opt  series 5  -15.848602 +/- 0.003280   0.728377 +/- 0.019288   0.0460
O_q0_opt  series 6  -15.850839 +/- 0.001870   0.723159 +/- 0.008173   0.0456
O_q0_opt  series 7  -15.848411 +/- 0.002449   0.708589 +/- 0.007225   0.0447
...
```

Plots of the data can also be obtained with the "`-p`" option ("`qmca -p -q ev O_q0_opt*scalar*`").

Identify which optimization series is the "best" according to your cost function. It is likely that multiple series are similar in quality. Note the `opt.xml` file corresponding to this series. This file contains the final value of the optimized Jastrow parameters to be used in the DMC calculations of the next section of the lab.

Questions and Exercises

1. What is the acceptance ratio of your optimization runs? (use "`qmca --help`" if necessary) Do you expect the Monte Carlo sampling to be efficient?
2. How do you know when the optimization process has converged?
3. Why is the mean and the error of the variance sometimes large? Consider using "`qmca -t ...`" to investigate.
4. Optimization is sometimes sensitive to initial guesses of the parameters. If you have time, try varying the initial parameters, including the cutoff radius (`rcut`) of the one-body Jastrow factor (remember to change `id` in the `<project/>` element). Do you arrive at a similar set of final Jastrow parameters? What is the lowest variance you are able to achieve?

2.3 DMC timestep extrapolation I: neutral O atom

The diffusion Monte Carlo (DMC) algorithm contains two biases in addition to the fixed node and pseudopotential approximations that are important to control: timestep and population control bias. The following subsection briefly discusses the origin of timestep and population control biases in DMC and how they can be minimized or extrapolated away. As before, the second subsection contains the lab walkthrough with QMCPACK. By the end of the section, we will have a solid DMC estimate of the ground state energy of oxygen.

Background on timestep and population control bias

DMC improves over the VMC algorithm by projecting toward the true many-body electronic ground state of the system. The projection operator is the (importance sampled) imaginary time propagator, which is also known as the thermodynamic density matrix:

$$\hat{\rho} = e^{-t\hat{H}} \tag{2.3.1}$$

The direct action of the projection operator on a trial wavefunction in position space

$$\langle R|e^{-t\hat{H}}|\Psi_T\rangle = \int dR' \rho(R, R'; t)\Psi_T(R') \tag{2.3.2}$$

cannot be calculated in a straightforward fashion since the analytic form of $\rho(R, R'; t) = \langle R|\hat{\rho}|R'\rangle$ is unknown. In order to make the algorithm computationally tractable, the finite time projection operator is expanded as a product of short-time projection operators

$$\langle R|e^{-tH}|\Psi_T\rangle = \langle R|e^{-\tau\hat{H}}e^{-\tau\hat{H}}\cdots e^{-\tau\hat{H}}|\Psi_T\rangle \tag{2.3.3}$$

$$= \int dR_1 dR_2 \cdots dR_M \rho(R, R_1; \tau)\rho(R_1, R_2; \tau)\cdots\rho(R_{M-1}, R_M; \tau)\Psi_T(R_M) \tag{2.3.4}$$

The advantage here is that reasonable approximations of the short time propagators are known. Common approximations have the form

$$\rho(R, R'; \tau) = e^{D(R,R';\tau)}e^{B(R,R';\tau)} + \mathcal{O}(\tau^2) \tag{2.3.5}$$

where $D(R, R'; \tau)$ and $B(R, R'; \tau)$ represent drift and branching terms, respectively. DMC results are biased for any finite timestep (τ). The bias can be eliminated by extrapolating to zero timestep. In practice this is done by performing a series of runs with decreasing timesteps and then fitting the results.

The drift term can be sampled with standard Monte Carlo methods, while the branching term is incorporated as a weight assigned to each random walker. Instead of accumulating the weight, it is more efficient to "branch" each walker according to the weight, resulting in some walkers being deleted and others copied multiple times. If left uncontrolled, the walker population (P) may vanish or diverge. A stable algorithm is obtained by adjusting the branching weight to preserve the overall number of walkers on average. Population control also biases the results, but usually to a lesser extent than timestep error (the bias is proportional to $1/P$). A common rule of thumb is to use at least a couple thousand walkers. This bias should be checked occasionally by performing runs with varying numbers of walkers.

Timestep extrapolation with QMCPACK

In the same directory you used to perform wavefunction optimization (`oxygen_atom`) you will find a sample DMC input file for the neutral oxygen atom named `O.q0.dmc.in.xml`. Open this file in a text editor and note the differences from the optimization case. The XML describing the wavefunction is no longer present. In its place is the line

```
<include href="OPT_XML"/>
```

Replace "OPT_XML" with the `opt.xml` file corresponding to the best Jastrow parameters you found in the last section. The `include` element essentially amounts to an in-place copy and paste of the contents of the `opt.xml` file.

The QMC calculation section at the bottom is also different. The linear optimization blocks have been replaced with XML describing a VMC run followed by DMC. The input keywords are described below.

timestep Timestep of the VMC/DMC random walk. In VMC choose a timestep corresponding to an acceptance ratio of about 50%. In DMC the acceptance ratio is often above 99%.

warmupSteps Number of MC steps discarded as a warmup or equilibration period of the random walk.

steps Number of MC steps per block. Physical quantities, such as the total energy, are averaged over walkers and steps.

blocks Number of blocks. This is also the number of average energy values written to output files. Should be greater than 200 for meaningful statistical analysis of output data (*e.g.* via `qmca`). The total number of MC steps each walker takes is `blocks`×`steps`.

samples VMC only. This is the number of walkers used in subsequent DMC runs. Each DMC walker is initialized with electron positions sampled from the VMC random walk.

nonlocalmoves DMC only. If yes/no, use the locality approximation/T-moves for nonlocal pseudopotentials. T-moves generally improve the stability of the algorithm and restore the variational principle for small systems (T-moves version 1).

The purpose of the VMC run is to provide initial electron positions for each DMC walker. Setting `walkers` = 1 in the VMC block ensures there will be only one VMC walker per execution thread. There will be a total of 512 VMC walkers in this case (see `O.q0.dmc.qsub.in`). We want the electron positions used to initialize the DMC walkers to be decorrelated from one another. A VMC walker will often decorrelate from its current position after propagating for a few Ha^{-1} in imaginary time (in general this is system dependent). This leads to a rough rule of thumb for choosing `blocks` and `steps` for the VMC run (`VWALKERS` = 512 here):

$$\texttt{VBLOCKS} \times \texttt{VSTEPS} \geq \frac{\texttt{DWALKERS}}{\texttt{VWALKERS}} \frac{5\ \mathrm{Ha}^{-1}}{\texttt{VTIMESTEP}} \tag{2.3.6}$$

Fill in the VMC XML block with appropriate values for these parameters. There should be more than one DMC walker per thread and enough walkers in total to avoid population control bias (see previous subsection).

To study timestep bias, we will perform a sequence of DMC runs over a range of timesteps (0.1 Ha^{-1} is too large and timesteps below 0.002 Ha^{-1} are probably too small). A common approach is to select a fairly large timestep to begin with and then decrease the timestep by a factor of two in each subsequent DMC run. The total amount of imaginary time the walker population propagates should be the same for each run. A simple way to accomplish this is to choose input parameters in the following way

$$\begin{aligned} \texttt{timestep}_n &= \texttt{timestep}_{n-1}/2 \\ \texttt{warmupSteps}_n &= \texttt{warmupSteps}_{n-1} \times 2 \\ \texttt{blocks}_n &= \texttt{blocks}_{n-1} \\ \texttt{steps}_n &= \texttt{steps}_{n-1} \times 2 \end{aligned} \tag{2.3.7}$$

Each DMC run will require about twice as much computer time as the one preceeding it. Note that the number of blocks is kept fixed for uniform statistical analysis. `blocks` $\times$ `steps` $\times$ `timestep` ~ 60 Ha^{-1} is sufficient for this system.

Choose an initial DMC timestep and create a sequence of N timesteps according to 2.3.7. Make N copies of the DMC XML block in the input file

```
<qmc method="dmc" move="pbyp">
   <parameter name="warmupSteps"         >    DWARMUP         </parameter>
   <parameter name="blocks"              >    DBLOCKS         </parameter>
   <parameter name="steps"               >    DSTEPS          </parameter>
   <parameter name="timestep"            >    DTIMESTEP       </parameter>
   <parameter name="nonlocalmoves"       >    yes             </parameter>
</qmc>
```

Fill in `DWARMUP`, `DBLOCKS`, `DSTEPS`, and `DTIMESTEP` for each DMC run according to 2.3.7. Submit the DMC timestep extrapolation run to the queue with `submit_O_q0_dmc`. The run should take only a few minutes to complete.

QMCPACK will create files prefixed with `O_q0_dmc`. The log file is `O_q0_dmc.output`. As before, block averaged data is written to `scalar.dat` files. In addition, DMC runs produce `dmc.dat` files which contain energy data averaged only over the walker population (one line per DMC step). The `dmc.dat` files also provide a record of the walker population at each step.

Use the `PlotTstepConv.pl` to obtain a linear fit to the timestep data (type "`PlotTstepConv.pl O.q0.dmc.in.xml 40`"). You should see a plot similar to fig. 2.2. The tail end of the text output displays the parameters for the linear fit. The "`a`" parameter is the total energy extrapolated to zero timestep in Hartree units.

```
...
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
```

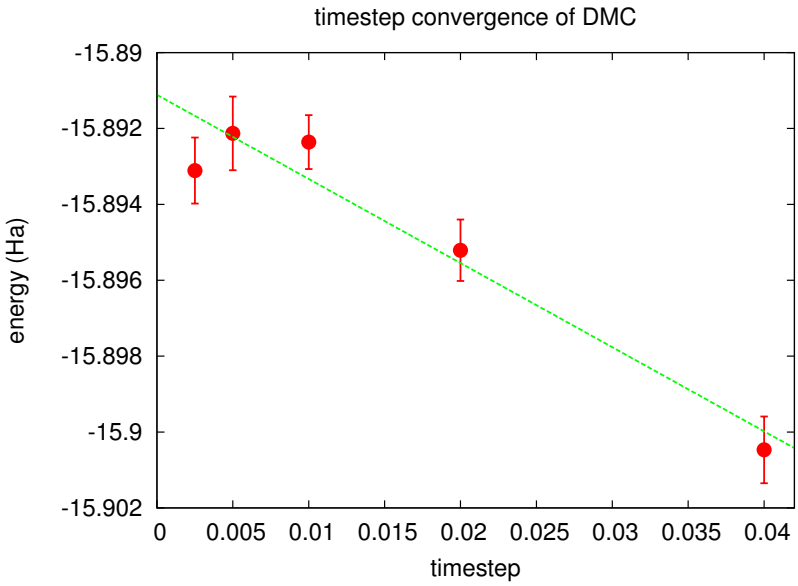

Figure 2.2: Linear fit to DMC timestep data from `PlotTstepConv.pl`.

```
a                       = -15.8911          +/- 0.000756      (0.004757%)
b                       = -0.221687         +/- 0.03757       (16.95%)
...
```

Questions and Exercises

1. What is the $\tau \to 0$ extrapolated value for the total energy?
2. What is the maximum timestep you should use if you want to calculate the total energy to an accuracy of 0.05 eV? For convenience, 1 Ha = 27.2113846 eV.
3. What is the acceptance ratio for this (bias$<$ 0.05 eV) run? Does it follow the rule of thumb for sensible DMC (acceptance ratio $>$ 99%) ?
4. Check the fluctuations in the walker population (`qmca -t -q nw O_q0_dmc*dmc.dat --noac`). Does the population seem to be stable?
5. (Optional) Study population control bias for the oxygen atom. Select a few population sizes (use multiples of 512 to fit cleanly on a single Vesta partition). Copy `O.q0.dmc.in.xml` to a new file and remove all but one DMC run (select a single timestep). Make one copy of the new file for each population, set "`samples`", and

choose a unique `id` in `<project/>`. Make submission files similar to `submit_O_q0_dmc` and `O.q0.dmc.qsub.in` and run one job at a time to avoid crowding the lab allocation. Use `qmca` to study the dependence of the DMC total energy on the walker population. How large is the bias compared to timestep error? What bias is incurred by following the "rule of thumb" of a couple thousand walkers? Will population control bias generally be an issue for production runs on modern parallel machines?

2.4 DMC timestep extrapolation II: IP of oxygen

In this section, we will repeat the calculations of the prior two sections (optimization, timestep extrapolation) for the +1 charge state of the oxygen atom. Comparing the resulting 1st ionization potential (IP) with experimental data will complete our first test of the BFD oxygen pseudopotential. In actual practice, higher IP's could also be tested prior to performing production runs.

Obtaining the timestep extrapolated DMC total energy for ionized oxygen should take much less (human) time than for the neutral case. For convenience, the necessary steps are briefly summarized below.

1. Copy the linear optimization blocks you used in `O.q0.opt.in.xml` to `O.q0.opt.in.xml`.
2. Submit the optimization job to Vesta's queue with `submit_O_q1_opt`.
3. Identify the optimal set of parameters with `qmca`.
4. Replace `OPT_XML` in `submit_O_q1_dmc` with the `opt.xml` file containing the optimal parameters.
5. Copy the VMC and DMC blocks you used in `O.q0.dmc.in.xml` to `O.q1.dmc.in.xml`.
6. Submit the DMC timestep job to Vesta's queue with `submit_O_q1_dmc`.
7. Obtain the DMC total energy extrapolated to zero timestep with `PlotTstepConv.pl`.

The process listed above, which excludes additional steps for orbital generation and conversion, can become tedious to perform by hand in production settings where many calculations are often required. For this reason automation tools are introduced for calculations involving the oxygen dimer in section 3 of the lab.

Questions and Exercises

1. What is the $\tau \to 0$ extrapolated DMC value for the 1st ionization potential of oxygen?
2. How does the extrapolated value compare to the experimental IP? Go to http://physics.nist.gov/PhysRefData/ASD/ionEnergy.html and enter "`O I`" in the box labeled "`Spectra`" and click on the "`Retrieve Data`" button. For comparison the LDA value is 12.25 eV.

3. What can we conclude about the accuracy of the pseudopotential? What factors complicate this assessment?

4. Explore the sensitivity of the IP to the choice of timestep. Type "`ip_conv.py`" to view three timestep extrapolation plots: two for the $q = 0, 1$ total energies and one for the IP. Is the IP more, less, or similarly sensitive to timestep than the total energy?

5. What is the maximum timestep you should use if you want to calculate the ionization potential to an accuracy of 0.05 eV? What factor of cpu time is saved by assessing timestep convergence on the IP (a total energy difference) vs. a single total energy?

6. Are the acceptance ratio and population fluctuations reasonable for the $q = 1$ calculations?

3. Testing PP dimer properties: DMC workflow automation

In this section we will use automation tools to calculate the DMC total energy of the oxygen dimer over a series of bond lengths. The equilibrium bond length and binding energy of the dimer will be determined by performing a polynomial fit to the data (Morse potential fits should be preferred in production tests). Comparing these values with correponding experimental data provides a second test of the BFD pseudopotential for oxygen.

Production QMC projects are often composed of many similar workflows. The simplest of these is a single DMC calculation involving four different compute jobs:

1. Orbital generation via Quantum Espresso or GAMESS.
2. Conversion of orbital data via `pw2qmcpack.x` or `convert4qmc`.
3. Optimization of Jastrow factors via QMCPACK.
4. DMC calculation via QMCPACK.

Simulation workflows quickly become more complex with increasing costs in terms of human time for the researcher. Automation tools can decrease both human time and error if used well.

The set of automation tools we will be using is known as the Project Suite (PS), which is distributed with QMCPACK. The PS is capable of generating input files, submitting and monitoring compute jobs, passing data between simulations (such as relaxed structures, orbital files, optimized Jastrow parameters, etc.), and data analysis. The user interface to the PS is through a set of functions defined in the Python programming language. User scripts which execute simple workflows resemble input files and do not require programming experience. More complex workflows require only basic programming constructs (*e.g.* for loops and if statements). PS input files/scripts should be easier to navigate than QMCPACK input files and more efficient than submitting all the jobs by hand.

3.1 Example Project Suite input

The Project Suite (PS) is driven by simple user-defined scripts that resemble keyword-driven input files. An example PS input file that performs a single VMC calculation is shown below. Take a moment to read it over and especially note the comments (prefixed with "`#`") explaining most of the contents. If the input syntax is unclear you may want to consult portions of appendix A, which gives a condensed summary of Python constructs. For more information about the functionality and effective use of the Project Suite, consult `docs/Project_Suite.pdf` first. More information can be found in the user guide distributed with QMCPACK, although examples in this lab series and `Project_Suite.pdf` are more

up to date (if `qmcpack` is the location of your QMCPACK distribution, the user guide can be found at `qmcpack/project_suite/documentation/project_suite_user_guide.pdf`).

```
#! /usr/bin/env python

# import project suite functions
from project import settings,Job,get_machine,run_project
from project import generate_physical_system
from project import generate_qmcpack,vmc

settings(                               # project suite settings
    pseudo_dir    = './pseudopotentials', # location of PP files
    runs          = '',                   # root directory for simulations
    results       = '',                   # root directory for simulation results
    status_only   = 0,                    # show simulation status, then exit
    generate_only = 0,                    # generate input files, then exit
    sleep         = 3,                    # seconds between checks on sim. progress
    machine       = 'vesta',              # name of local machine
    account       = 'QMC_2014_training'   # charge account for cpu time
    )

vesta = get_machine('vesta')            # allow max of one job at a time (lab only)
vesta.queue_size = 1

qmcjob = Job(                           # specify job parameters
    nodes   = 32,                           # use 32 Vesta nodes
    threads = 16,                           # 16 OpenMP threads per node (32 MPI tasks)
    hours   = 1,                            # wallclock limit of 1 hour
                                            # use QMCPACK executable
    app     = '/soft/applications/qmcpack/build_XL_real/bin/qmcapp'
    )

qmc_calcs = [                           # list QMC calculation methods
    vmc(                                    #   VMC
        walkers     =   1,                  #     1 walker
        warmupsteps =  50,                  #    50 MC steps for warmup
        blocks      = 200,                  #   200 blocks
        steps       =  10,                  #    10 steps per block
        timestep    =  .4                   #   0.4 1/Ha timestep
        )]

dimer = generate_physical_system(       # make a dimer system
    type       = 'dimer',                   # system type is dimer
    dimer      = ('O','O'),                 # dimer is two oxygen atoms
```

```
    separation = 1.2074,                # separated by 1.2074 Angstrom
    Lbox       = 15.0,                  # simulation box is 15 Angstrom
    units      = 'A',                   # Angstrom is dist. unit
    net_spin   = 2,                     # nup-ndown is 2
    O          = 6                      # pseudo-oxygen has 6 valence el.
    )

qmc = generate_qmcpack(                 # make a qmcpack simulation
    identifier   = 'example',           # prefix files with 'example'
    path         = 'scale_1.0',         # run in ./scale_1.0 directory
    system       = dimer,               # run the dimer system
    job          = qmcjob,              # set job parameters
    input_type   = 'basic',             # basic qmcpack inputs given below
    pseudos      = ['O.BFD.xml'],       # list of PP's to use
    orbitals_h5  = 'O2.pwscf.h5',       # file with orbitals from DFT
    bconds       = 'nnn',               # open boundary conditions
    jastrows     = [],                  # no jastrow factors
    calculations = qmc_calcs            # QMC calculations to perform
    )

run_project(qmc)                        # write input file and submit job
```

3.2 Automated binding curve of the oxygen dimer

Enter the `oxygen_dimer` directory. Copy your BFD pseudopotential from the atom runs into `oxygen_dimer/pseudopotentials`. Open `O_dimer.py` with a text editor. The overall format is similar to the example file shown in the last section. The header material, including PS imports, settings, and the job parameters for QMC are identical. The main difference is that optimization and DMC runs are being performed rather than a single VMC run.

Following the job parameters, inputs for the optimization method are given. The keywords should all be familiar from the QMCPACK XML input files you used previously:

```
linopt1 = linear(
    energy               = 0.0,
    unreweightedvariance = 1.0,
    reweightedvariance   = 0.0,
    timestep             = 0.4,
    samples              = 5000,
    warmupsteps          = 50,
    blocks               = 200,
    substeps             = 1,
    nonlocalpp           = True,
```

```
    usebuffer            = True,
    walkers              = 1,
    minwalkers           = 0.5,
    maxweight            = 1e9,
    usedrift             = True,
    minmethod            = 'quartic',
    beta                 = 0.025,
    exp0                 = -16,
    bigchange            = 15.0,
    alloweddifference    = 1e-4,
    stepsize             = 0.2,
    stabilizerscale      = 1.0,
    nstabilizers         = 3
    )
```

Requesting multiple loop's with different numbers of samples is more compact than in XML:

```
linopt1 = ...

linopt2 = linopt1.copy()
linopt2.samples = 20000 # opt w/ 20000 samples

linopt3 = linopt1.copy()
linopt3.samples = 40000 # opt w/ 40000 samples

opt_calcs = [loop(max=8,qmc=linopt1), # loops over opt's
             loop(max=6,qmc=linopt2),
             loop(max=4,qmc=linopt3)]
```

The VMC/DMC method inputs should also look familiar:

```
qmc_calcs = [
    vmc(
        walkers     =   1,
        warmupsteps =  30,
        blocks      =  20,
        steps       =  10,
        substeps    =   2,
        timestep    =  .4,
```

```
        samples     = 2048
        ),
    dmc(
        warmupsteps   = 100,
        blocks        = 400,
        steps         =  32,
        timestep      = 0.01,
        nonlocalmoves = True
        )
    ]
```

As in the example in the last section, the oxygen dimer is generated with the `generate_physical_system` function:

```
dimer = generate_physical_system(
    type       = 'dimer',
    dimer      = ('O','O'),
    separation = 1.2074*scale,
    Lbox       = 15.0,
    units      = 'A',
    net_spin   = 2,
    O          = 6
    )
```

Similar syntax can be used to generate crystal structures or to specify systems with arbitrary atomic configurations and simulation cells. Notice that a "`scale`" variable has been introduced to stretch or compress the dimer.

Next, objects representing QMCPACK simulations are constructed with the `generate_qmcpack` function:

```
opt = generate_qmcpack(
    identifier   = 'opt',
    ...
    jastrows     = [('J1','bspline',8,4.5),
                    ('J2','pade',0.5,0.5)],
    calculations = opt_calcs
    )
sims.append(opt)
```

```
qmc = generate_qmcpack(
    identifier   = 'qmc',
    ...
    jastrows     = [],
    calculations = qmc_calcs,
    dependencies = (opt,'jastrow')
    )
sims.append(qmc)
```

Shared details such as the run directory, job, pseudopotentials, and orbital file have been omitted (...). The "opt" run will optimize a 1-body B-spline Jastrow with 8 knots having a cutoff of 4.5 Bohr and a 2-body Padé Jastrow with up-up and up-down "B" parameters set to 0.5 1/Bohr. The Jastrow list for the DMC run is empty and a new keyword is present: dependencies. The usage of dependencies above indicates that the DMC run depends on the optimization run for the Jastrow factor. The PS will submit the "opt" run first and upon completion it will scan the output, select the optimal set of parameters, pass the Jastrow information to the "qmc" run and then submit the DMC job. Independent job workflows are submitted in parallel when permitted (we have explicitly prevented this for this lab by setting queue_size=1 for Vesta). No input files are written or job submissions made until the "run_project" function is reached.

As written, 0_dimer.py will only perform calculations at the equilibrium separation distance of 1.2074 Angstrom. Modify the file now to perform DMC calculations across a range of separation distances with each DMC run using the Jastrow factor optimized at the equilibrium separation distance. The necessary Python for loop syntax should look something like this:

```
sims = []
for scale in [1.00,0.90,0.95,1.05,1.10]:
    ...
    dimer = ...
    if scale==1.00:
        opt = ...
        ...
    #end if
    qmc = ...
    ...
#end for
run_project(sims)
```

Note that the text inside the for loop and the if block must be indented by precisely four spaces. If you use Emacs, changes in indentation can be performed easily with Cntrl-C >

and Cntrl-C < after highlighting a block of text (other editors should have similar functionality). If you see something like "SyntaxError: invalid syntax" print to the screen when you run O_dimer.py later on, consult the completed file in oxygen_dimer/reference.

The values of "scale" in the loop must be a subset of [0.90,0.925,0.95,0.975,1.00,1.025,1.05,1.075,1.10] since orbital files have been pre-generated with PWSCF for only these values. If other values are selected, the job will be submitted but QMCPACK will fail when it attempts to read the non-existent O2.pwscf.h5 file (in later labs we will run PWSCF to generate the orbital files directly with the PS). Begin with the reduced set of scale values shown above.

Change the "status_only" parameter in the "settings" function to 1 and type "./O_dimer.py" at the command line. This will print the status of all simulations:

```
Project starting
  checking for file collisions
  loading cascade images
    cascade 0 checking in
  checking cascade dependencies
    all simulation dependencies satisfied
  cascade status
    setup, sent_files, submitted, finished, got_output, analyzed
    000000  opt  ./scale_1.0
    000000  qmc  ./scale_1.0
    000000  qmc  ./scale_0.9
    000000  qmc  ./scale_0.95
    000000  qmc  ./scale_1.05
    000000  qmc  ./scale_1.1
    setup, sent_files, submitted, finished, got_output, analyzed
```

In this case, a single independent simulation "cascade" (workflow) has been identified, containing one "opt" and five dependent "qmc" runs. The six status flags (setup, sent_files, submitted, finished, got_output, analyzed) each show 0, indicating that no work has been done yet.

Now change "status_only" back to 0, set "generate_only" to 1, and run O_dimer.py again. This will perform a dry-run of all simulations. The dry-run should finish in about 20 seconds:

```
Project starting
  checking for file collisions
  loading cascade images
    cascade 0 checking in
  checking cascade dependencies
```

```
    all simulation dependencies satisfied

  starting runs:
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  poll 0  memory 88.54 MB
    Entering ./scale_1.0 0
      writing input files  0 opt
    Entering ./scale_1.0 0
      sending required files  0 opt
      submitting job  0 opt
    Entering ./scale_1.0 1
      Would have executed:  qsub --mode script --env BG_SHAREDMEMSIZE=32 opt.qsub.in

  poll 1  memory 88.54 MB
    Entering ./scale_1.0 0
      copying results  0 opt
    Entering ./scale_1.0 0
      analyzing  0 opt

  poll 2  memory 88.87 MB
    Entering ./scale_1.0 1
      writing input files  1 qmc
    Entering ./scale_1.0 1
      sending required files  1 qmc
      submitting job  1 qmc
    ...
    Entering ./scale_1.0 2
      Would have executed:  qsub --mode script --env BG_SHAREDMEMSIZE=32 qmc.qsub.in
    ...

Project finished
```

The PS polls the simulation status every 3 seconds and sleeps in between. The "scale_*" directories should now contain several files:

```
scale_1.0
 O2.pwscf.h5
 O.BFD.xml
 opt.in.xml
 opt.qsub.in
 qmc.in.xml
 qmc.qsub.in
```

```
 sim_opt
    analyzer.p
    input.p
    sim.p
 sim_qmc
     analyzer.p
     input.p
     sim.p
```

Take a minute to inspect the generated input (`opt.in.xml`, `qmc.in.xml`) and submission (`opt.qsub.in`, `qmc.qsub.in`) files. The pseudopotential file `O.BFD.xml` has been copied into each local directory. Two additional directories have been created: `sim_opt` and `sim_qmc`. The `sim.p` files in each directory contain the current status of each simulation. If you run `O_dimer.py` again, it should not attempt to rerun any of the simulations:

```
Project starting
  checking for file collisions
  loading cascade images
    cascade 0 checking in
    cascade 8 checking in
    cascade 2 checking in
    cascade 4 checking in
    cascade 6 checking in
  checking cascade dependencies
    all simulation dependencies satisfied

  starting runs:
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  poll 0  memory 60.10 MB
Project finished
```

This way one can continue to add to the `O_dimer.py` file (*e.g.* adding more separation distances) without worrying about duplicate job submissions.

Let's actually submit the optimization and DMC jobs now. Reset the state of the simulations by removing the `sim.p` files ("`rm ./scale*/sim*/sim.p`"), set "`generate_only`" to `0`, and rerun `O_dimer.py`. It should take about 15 minutes for all the jobs to complete. You may wish to open another terminal to monitor the progress of the individual jobs while the current terminal runs `O_dimer.py` in the foreground. You can begin the first exercise below once the optimization job completes.

Questions and Exercises

1. Evaluate the quality of the optimization at `scale=1.0` using the `qmca` tool. Did the optimization succeed? How does the variance compare with the neutral oxygen atom? Is the wavefunction of similar quality to the atomic case?

2. Evaluate the traces of the local energy and the DMC walker population for each separation distance with the `qmca` tool. Are there any anomalies in the runs? Is the acceptance ratio reasonable? Is the wavefunction of similar quality across all separation distances?

3. Use the `dimer_fit.py` tool located in `oxygen_dimer` to fit the oxygen dimer binding curve. To get the binding energy of the dimer, we will need the DMC energy of the atom. Before performing the fit, answer: What DMC timestep should be used for the oxygen atom results? The tool accepts three arguments ("`O_dimer.py P N E Eerr`"), `P` is the prefix of the DMC input files (should be "`qmc`" at this point), `N` is the order of the fit (use 2 to start), `E` and `Eerr` are your DMC total energy and error bar, respectively for the oxygen atom (in eV). A plot of the dimer data will be displayed and text output will show the DMC equilibrium bond length and binding energy as well as experimental values. How accurately does your fit to the DMC data reproduce the experimental values? What factors affect the accuracy of your results?

4. Refit your data with a fourth-order polynomial. How do your predictions change with a fourth-order fit? Is a fourth-order fit appropriate for the available data?

5. Add the four remaining "`scale`" values to the list in `O_dimer.py` that interpolate between the original set. Perform the DMC calculations and redo the fits. How accurately does your fit to the DMC data reproduce the experimental values? Should this pseudopotential be used in production calculations?

6. (Optional) Perform optimization runs at the extremal separation distances corresponding to `scale=[0.90,1.10]`. Are the individually optimized wavefunctions of significantly better quality than the one imported from `scale=1.00`? Why? What form of Jastrow factor might give an even better improvement?

4. (Optional) Running your system with QMCPACK

This section covers a fairly simple route to get started on QMC calculations of an arbitrary system of interest using the Project Suite (PS) automation system to setup input files and optionally perform the runs. The example provided in this section uses QM Espresso (PWSCF) to generate the orbitals forming the Slater determinant part of the trial wavefunction. PWSCF is a natural choice for solid state systems and it can be used for surface/slab and molecular systems as well, albeit at the price of describing additional vacuum space with plane waves.

To start out with, you will need pseudopotentials (PP's) for each element in your system in both the UPF (PWSCF) and FSATOM/XML (QMCPACK) formats. A good place to start is the Burkatzki-Filippi-Dolg (BFD) pseudopotential database (http://www.burkatzki.com/pseudos/index.2.html), which we have already used in our study of the oxygen atom. The database does not contain PP's for the 4th and 5th row transition metals or any of the lanthanides or actinides. If you need a PP that is not in the BFD database, you may need to generate and test one manually (*e.g.* with OPIUM, http://opium.sourceforge.net/). Otherwise, use `ppconvert` as outlined in section 2.1 to obtain PP's in the formats used by PWSCF and QMCPACK. Enter the `your_system` lab directory and place the converted PP's in `your_system/pseudopotentials`.

Before performing production calculations (more than just the initial setup in this section) be sure to converge the plane wave energy cutoff in PWSCF as these PP's can be rather hard, sometimes requiring cutoffs in excess of 300 Ry. Depending on the system under study, the amount of memory required to represent the orbitals (QMCPACK uses 3D B-splines) becomes prohibitive and one may be forced to search for softer PP's.

Beyond pseudopotentials, all that is required to get started are the atomic positions and the dimensions/shape of the simulation cell. The PS file `example.py` illustrates how to setup PWSCF and QMCPACK input files by providing minimal information regarding the physical system (an 8-atom cubic cell of diamond in the example). Most of the contents should be familiar from your experience with the automated calculations of the oxygen dimer binding curve in section 3 (if you've skipped ahead you may want to skim that section for relevant information). The most important change is the expanded description of the physical system:

```
# details of your physical system (diamond conventional cell below)
my_project_name = 'diamond_vmc'    # directory to perform runs
my_dft_pps      = ['C.BFD.upf']    # pwscf pseudopotentials
my_qmc_pps      = ['C.BFD.xml']    # qmcpack pseudopotentials
```

```
#  generate your system
#    units      :  'A'/'B' for Angstrom/Bohr
#    axes       :  simulation cell axes in cartesian coordinates (a1,a2,a3)
#    elem       :  list of atoms in the system
#    pos        :  corresponding atomic positions in cartesian coordinates
#    kgrid      :  Monkhorst-Pack grid
#    kshift     :  Monkhorst-Pack shift (between 0 and 0.5)
#    net_charge :  system charge in units of e
#    net_spin   :  # of up spins - # of down spins
#    C = 4      :  (pseudo) carbon has 4 valence electrons
my_system = generate_physical_system(
    units      = 'A',
    axes       = [[ 3.57000000e+00, 0.00000000e+00, 0.00000000e+00],
                  [ 0.00000000e+00, 3.57000000e+00, 0.00000000e+00],
                  [ 0.00000000e+00, 0.00000000e+00, 3.57000000e+00]],
    elem       = ['C','C','C','C','C','C','C','C'],
    pos        = [[ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
                  [ 8.92500000e-01, 8.92500000e-01, 8.92500000e-01],
                  [ 0.00000000e+00, 1.78500000e+00, 1.78500000e+00],
                  [ 8.92500000e-01, 2.67750000e+00, 2.67750000e+00],
                  [ 1.78500000e+00, 0.00000000e+00, 1.78500000e+00],
                  [ 2.67750000e+00, 8.92500000e-01, 2.67750000e+00],
                  [ 1.78500000e+00, 1.78500000e+00, 0.00000000e+00],
                  [ 2.67750000e+00, 2.67750000e+00, 8.92500000e-01]],
    kgrid      = (1,1,1),
    kshift     = (0,0,0),
    net_charge = 0,
    net_spin   = 0,
    C          = 4       # one line like this for each atomic species
    )

my_bconds       = 'ppp'  #  ppp/nnn for periodic/open BC's in QMC
                         #  if nnn, center atoms about (a1+a2+a3)/2
```

If you have a system you would like to try with QMC, make a copy of `example.py` and fill in the relevant information about the pseudopotentials, simulation cell axes, and atomic species/positions. Otherwise, you can proceed with `example.py` as it is.

The other new aspects are two additional compute jobs to generate the orbitals with PWSCF and convert them into the ESHDF format with `pw2qmcpack.x`:

```
# scf run to generate orbitals
scf = generate_pwscf(
    identifier   = 'scf',
```

```
    path         = my_project_name,
    job          = Job(nodes=32,hours=2,app=pwscf),
    input_type   = 'scf',
    system       = my_system,
    pseudos      = my_dft_pps,
    input_dft    = 'lda',
    ecut         = 200,   # PW energy cutoff in Ry
    conv_thr     = 1e-8,
    mixing_beta  = .7,
    nosym        = True,
    wf_collect   = True
    )

# conversion step to create h5 file with orbitals
p2q = generate_pw2qmcpack(
    identifier   = 'p2q',
    path         = my_project_name,
    job          = Job(cores=1,hours=2,app=pw2qmcpack),
    write_psir   = False,
    dependencies = (scf,'orbitals')
    )
```

Set "`generate_only`" to `1` and type "`./example.py`" or similar to generate the input files. All files will be written to "`./diamond_vmc`" ("`./[my_project_name]`" if you have changed "`my_project_name`" in the file). The input files for PWSCF, pw2qmcpack, and QMCPACK are `scf.in`, `pw2qmcpack.in`, and `vmc.in.xml`, repectively. Take some time to inspect the generated input files. If you have questions about the file contents, or run into issues with the generation process, feel free to consult with a lab instructor.

If desired, you can submit the runs directly with `example.py`. To do this, first reset the PS simulation record by typing "`rm ./diamond_vmc/sim*/sim.p`" or similar and set "`generate_only`" back to `0`. Next rerun `example.py` (you may want to redirect the text output).

Alternatively the runs can be submitted by hand:

```
qsub --mode script --env BG_SHAREDMEMSIZE=32 scf.qsub.in

(wait until JOB DONE appears in scf.output)

qsub --mode script --env BG_SHAREDMEMSIZE=32 p2q.qsub.in
```

Once the conversion process has finished the orbitals should be located in the file `diamond_vmc/pwscf_output/pwscf.pwscf.h5`. Open `diamond_vmc/vmc.in.xml` and re-

place "MISSING.h5" with "./pwscf_output/pwscf.pwscf.h5". Next submit the VMC run:

```
qsub --mode script --env BG_SHAREDMEMSIZE=32 vmc.qsub.in
```

Note: If your system is large, the above process may not complete within the time frame of this lab. Working with a stripped down (but relevant) example is a good idea for exploratory runs.

Once the runs have finished, you may want to begin exploring Jastrow optimization and DMC for your system. Example calculations are provided at the end of `example.py` in the commented out text).

A. Basic Python constructs

Basic Python data types (`int`, `float`, `str`, `tuple`, `list`, `array`, `dict`, `obj`) and programming constructs (`if` statements, `for` loops, functions w/ keyword arguments) are briefly overviewed below. All examples can be executed interactively in Python. To do this, type "`python`" at the command line and paste any of the shaded text below at the "`>>>`" prompt. For more information about effective use of Python, consult the detailed online documentation: https://docs.python.org/2/.

Intrinsic types: `int, float, str`

```
#this is a comment
i=5                     # integer
f=3.6                   # float
s='quantum/monte/carlo' # string
n=None                  # represents "nothing"

f+=1.4                  # add-assign (-,*,/ also): 5.0
2**3                    # raise to a power: 8
str(i)                  # int to string: '5'
s+'/simulations'        # joining strings: 'quantum/monte/carlo/simulations'
'i={0}'.format(i)       # format string: 'i=5'
```

Container types: `tuple, list, array, dict, obj`

```
from numpy import array  # get array from numpy module
from generic import obj  # get obj from generic module

t=('A',42,56,123.0)     # tuple

l=['B',3.14,196]        # list

a=array([1,2,3])        # array

d={'a':5,'b':6}         # dict
```

```
o=obj(a=5,b=6)          # obj

                        # printing
print t                 #  ('A', 42, 56, 123.0)
print l                 #  ['B', 3.1400000000000001, 196]
print a                 #  [1 2 3]
print d                 #  {'a': 5, 'b': 6}
print o                 #    a               = 5
                        #    b               = 6

len(t),len(l),len(a),len(d),len(o) #number of elements: (4, 3, 3, 2, 2)

t[0],l[0],a[0],d['a'],o.a  #element access: ('A', 'B', 1, 5, 5)

s = array([0,1,2,3,4])  # slices: works for tuple, list, array
s[:]                    #   array([0, 1, 2, 3, 4])
s[2:]                   #   array([2, 3, 4])
s[:2]                   #   array([0, 1])
s[1:4]                  #   array([1, 2, 3])
s[0:5:2]                #   array([0, 2, 4])

                        # list operations
l2 = list(l)            #   make independent copy
l.append(4)             #   add new element: ['B', 3.14, 196, 4]
l+[5,6,7]               #   addition: ['B', 3.14, 196, 4, 5, 6, 7]
3*[0,1]                 #   multiplication:  [0, 1, 0, 1, 0, 1]

b=array([5,6,7])        # array operations
a2 = a.copy()           #   make independent copy
a+b                     #   addition: array([ 6, 8, 10])
a+3                     #   addition: array([ 4, 5, 6])
a*b                     #   multiplication: array([ 5, 12, 21])
3*a                     #   multiplication: array([3, 6, 9])

                        # dict/obj operations
d2 = d.copy()           #   make independent copy
d['c'] = 7              #   add/assign element
d.keys()                #   get element names: ['a', 'c', 'b']
d.values()              #   get element values: [5, 7, 6]

                        # obj-specific operations
o.c = 7                 #   add/assign element
o.set(c=7,d=8)          #   add/assign multiple elements
```

An important feature of Python to be aware of is that assignment is most often by reference, *i.e.* new values are not always created. This point is illustrated below with an `obj` instance, but it also holds for `list`, `array`, `dict`, and others.

```
>>> o = obj(a=5,b=6)
>>>
>>> p=o
>>>
>>> p.a=7
>>>
>>> print o
  a               = 7
  b               = 6

>>> q=o.copy()
>>>
>>> q.a=9
>>>
>>> print o
  a               = 7
  b               = 6
```

Here `p` is just another name for `o`, while `q` is a fully independent copy of it.

Conditional Statements: `if/elif/else`

```
a = 5
if a is None:
    print 'a is None'
elif a==4:
    print 'a is 4'
elif a<=6 and a>2:
    print 'a is in the range (2,6]'
elif a<-1 or a>26:
    print 'a is not in the range [-1,26]'
elif a!=10:
    print 'a is not 10'
else:
    print 'a is 10'
```

```
#end if
```

The "`#end if`" is not part of Python syntax, but you will see text like this throughout the Project Suite for clear encapsulation.

Iteration: `for`

```
from generic import obj

l = [1,2,3]
m = [4,5,6]
s = 0
for i in range(len(l)):  # loop over list indices
    s += l[i] + m[i]
#end for

print s                  # s is 21

s = 0
for v in l:              # loop over list elements
    s += v
#end for

print s                  # s is 6

o = obj(a=5,b=6)
s = 0
for v in o:              # loop over obj elements
    s += v
#end for

print s                  # s is 11

d = {'a':5,'b':4}
for n,v in o.iteritems():# loop over name/value pairs in obj
    d[n] += v
#end for

print d                  # d is {'a': 10, 'b': 10}
```

Functions: `def`, argument syntax

```
def f(a,b,c=5):          # basic function, c has a default value
    print a,b,c
#end def f

f(1,b=2)                 # prints: 1 2 5


def f(*args,**kwargs):   # general function, returns nothing
    print args           #     args: tuple of positional arguments
    print kwargs         #   kwargs: dict of keyword arguments
#end def f

f('s',(1,2),a=3,b='t')   # 2 pos., 2 kw. args, prints:
                         #   ('s', (1, 2))
                         #   {'a': 3, 'b': 't'}

l = [0,1,2]
f(*l,a=6)                # pos. args from list, 1 kw. arg, prints:
                         #   (0, 1, 2)
                         #   {'a': 6}
o = obj(a=5,b=6)
f(*l,**o)                # pos./kw. args from list/obj, prints:
                         #   (0, 1, 2)
                         #   {'a': 5, 'b': 6}

f(                       # indented kw. args, prints
    blocks   = 200,      #   ()
    steps    = 10,       #   {'steps': 10, 'blocks': 200, 'timestep': 0.01}
    timestep = 0.01
    )

o = obj(                 # obj w/ indented kw. args
    blocks   = 100,
    steps    =  5,
    timestep = 0.02
    )

f(**o)                   # kw. args from obj, prints:
                         #   ()
                         #   {'timestep': 0.02, 'blocks': 100, 'steps': 5}
```