The Nexus User Guide

By Jaron T. Krogel

31 July 2015

Contents

1. Using this document

The Nexus User Guide provides an overview of Nexus (Sec. Sec. 2), instructions on how to install it (Sec. 3), an explanation of Nexus user scripts used to create simulation workflows (Sec. 4), and complete examples of electronic structure calculations using Nexus (Sec. 5). Reading all sections is recommended prior to beginning production work. The impatient may quickly visit "Nexus Installation" (Sec. 3) and see the examples section (Sec. 5) for template calculations to begin using Nexus immediately. If you cannot find what you need in this document, contact the main developer of Nexus (Jaron Krogel), at krogeljt@ornl.gov (but please make a thorough search first!).

Some basic understanding of Python is recommended for Nexus users. For a very quick introduction to the basics, read Appendix A. More information can be found under "Brushing up on Python" in the "Recommended Reading" Section (Sec. 6.3).

This document also provides a brief overview of Quantum Monte Carlo (QMC) from an applied perspective (Appendix B) and directions on where to go to learn more (Sec. 6). Consider reading "QMC Practice in a Nutshell" (Appendix B) and the review articles and online resources listed under "Quantum Monte Carlo: Theory and Practice" (Sec. 6.4) before proceeding to the overview (Sec. 2) and the examples (Sec. 5).

2. Overview of Nexus

2.1 What Nexus is

Nexus is a collection of tools, written in Python, to perform complex electronic structure calculations and analyze the results. The main focus is currently on performing arbitrary Quantum Monte Carlo (QMC) calculations with QMCPACK, however VASP, Quantum Espresso, and GAMESS are also supported. A single QMC calculation typically requires several previous calculations with other codes to produce a starting guess for the many-body wavefunction and convert it into a form that QMCPACK understands. Managing the resulting array of calculations, and the flow of information between them, quickly becomes unweildy to the researcher, demands a great deal of human time, and increases the potential for human error. Nexus reduces both the human time required and potential for error by automating the total simulation process.

2.2 What Nexus can do

The capabilities of Nexus currently include crystal structure generation, standalone Density Functional Theory (DFT) calculations with PWSCF (Quantum Espresso) or VASP, quantum chemical calculations with GAMESS, Hartree-Fock (HF) calculations of atoms with the SQD code (packaged with QMCPACK), complete QMC calculations with QMCPACK (including wavefunction optimization, Variational Monte Carlo (VMC), and Diffusion Monte Carlo (DMC) in periodic or open boundary conditions), automated job management on workstations (by acting as a virtual queue) and clusters/supercomputers including handling of dependencies between calculations and job bundling, and extraction of results from completed calculations for analysis. The integration of these capabilities permits the user to focus on the high-level tasks of problem formulation and interpretation of the results without (in principle) becoming too involved in the time-consuming, lower level details.

2.3 How Nexus is used

Use of Nexus currently involves writing a short Python script describing the calculations to be performed. This small script formed by the user closely resembles an input file for electronic structure codes. A key difference is that this "input file" represents executable code, and so variables are easily defined for use in expressions and more complicated simulation workflows (*e.g.* an equation of state) can be constructed with if/else logic and for loops. Knowledge of the Python programming language is helpful to perform complex calculations, but not essential for use of Nexus. Starting from working "input files" such as those covered in the "Complete Examples" section (5) is a good way to proceed.

3. Nexus Installation

Installation of Nexus can be accomplished by setting a single environment variable provided a working python environment exists.

3.1 Setting environment variables

To make your Python installation (must be Python 2.x as 3.x is not supported) aware of Nexus, simply set the PYTHONPATH environment variable. For example, in bash this would look like:

```
export PYTHONPATH=/your_download_path/nexus/library
```

Add this to *e.g.* your .bashrc file to make Nexus available in future sessions.

3.2 Installing Python dependencies

In addition to the standard Python installation, the `numpy` module must be installed for Nexus to function at a basic level. To realize the full range of functionality available, it is recommended that the `scipy`, `matplotlib`, and `h5py` modules be installed as well. Many of these packages are already available in various supercomputing environments. On a debian-based Linux system, such as Ubuntu, installation of these python modules is easily accomplished by invoking the following at the command line:

```
sudo apt-get install python-numpy
sudo apt-get install python-scipy python-matplotlib python-h5py
```

To install the Python modules on other platforms, please see "Helpful Links for Installing Python Modules" (section 6.1).

Of course, to run full calculations, the simulation codes and converters involved must be installed as well. These include a modified version of Quantum Espresso (`pw.x`, `pw2qmcpack.x`, optionally `pw2casino.x`), QMCPACK (`qmcapp`, `qmcapp_complex`, `convert4qmc`, `wfconvert`, `ppconvert`), SQD (`sqd`, packaged with QMCPACK), VASP, and/or GAMESS. Complete coverage of this task is beyond the scope of the current document, but please see "Helpful Links for Installing Electronic Structure Codes" (section 6.2).

4. Nexus User Scripts

Users interact with Nexus by writing a Python script that often resembles an input file. A Nexus user script typically consists of six main sections, as described below:

Nexus imports: functions unique to Nexus are drawn into the user environment.

Nexus settings: specify machine information and configure the runtime behavior of Nexus.

Physical system specification: create a data description of the physical system. Generate a crystal structure or import one from an external data file. Physical system details can be shared among many simulations.

Workflow specification: describe the simulations to be performed. Link simulations together by their data dependencies to form workflows.

Workflow execution: pass control to Nexus for active workflow management. Simulation input files are generated, jobs are submitted and monitored, output files are collected and preprocessed for later analysis.

Data analysis: control returns to the user to extract preprocessed simulation output data for further analysis.

Each of the six input sections is the subject of lengthier discussion: "Nexus imports" (Sec. 4.1), "Nexus settings" (4.2), "Physical system specification" (4.3), "Workflow specification" (4.4), "Workflow execution" (4.5), "Data analysis" (4.6). These sections are also illustrated in the abbreviated example script below. For more complete examples and further discussion, please refer to the user walkthroughs in Sec. 5.

4.1 Nexus imports

Each script begins with imports from the main Nexus module. Items imported include the interface to provide settings to Nexus, helper functions to make objects representing atomic structures or simulations of particular types (*e.g.* QMCPACK or VASP), and the interface to provide simulation workflows to Nexus for active management.

The import of all Nexus components is accomplished with the brief "`from nexus import *`". Each component can also be imported separately by name, as in the example below.

```
from nexus import settings                  # Nexus settings function
from nexus import generate_physical_system  # for creating atomic structures
from nexus import generate_pwscf            # for creating PWSCF sim. objects
from nexus import Job                       # for creating job objects
from nexus import run_project               # for active workflow management
```

This has the advantage of avoided unwanted namespace collisions with user defined variables. The major Nexus components available for import are listed in Table 4.1.

component	description
settings	Alter runtime behavior. Provide machine information.
generate_physical_system	Create atomic structure including electronic information.
generate_structure	Create atomic structure without electronic information.
generate_simulation	Create generic simulation object.
generate_pwscf	Create PWSCF simulation object.
generate_vasp	Create VASP simulation object.
generate_gamess	Create GAMESS simulation object.
generate_qmcpack	Create QMCPACK simulation object.
generate_sqd	Create SQD simulation object.
input_template	Create generic input file object.
multi_input_template	Create generic input file object representing multiple files.
generate_pwscf_input	Create PWSCF input file object.
generate_vasp_input	Create VASP input file object.
generate_gamess_input	Create GAMESS input file object.
generate_qmcpack_input	Create QMCPACK input file object.
generate_sqd_input	Create SQD input file object.
Job	Provide job information for simulation run.
run_project	Initiate active workflow management.
obj	Generic container object. Store inputs for later use.

Table 4.1: Major Nexus components available for import.

4.2 Nexus settings: global state and user-specific information

Following imports, the next section of a Nexus script is dedicated to providing information regarding the local machine, the location of various files, and the desired runtime behavior. This information is communicated to Nexus through the `settings` function. To make `settings` available in your project script, use the following import statement:

```
from nexus import settings
```

In most cases, it is sufficient to supply only four pieces of information through the `settings` function: whether to run all jobs or just create the input files, how often to check jobs for completion, the location of pseudopotential files, and a description of the local machine.

```
settings(
    generate_only = True,                # only write input files, do not run
    sleep         = 3,                   # check on jobs every 3 seconds
```

```
    pseudo_dir    = './pseudopotentials', # path to PP file collection
    machine       = 'ws8'                # local machine is an 8 core workstation
    )
```

A few additional parameters are available in `settings` to control where runs are performed, where output data is gathered, and whether to print job status information. More detailed information about machines can be provided, such as allocation account numbers, filesystem structure, and where executables are located.

```
settings(
    status_only   = True,                # only show job status, do not write or run
    generate_only = True,                # only write input files, do not run
    sleep         = 3,                   # check on jobs every 3 seconds
    pseudo_dir    = './pseudopotentials', # path to PP file collection
    runs          = '',                  # base path for runs is local directory
    results       = '/home/jtk/results/', # light output data copied elsewhere
    machine       = 'titan',             # Titan supercomputer
    account       = 'ABC123',            # user account number
    )
```

4.3 Physical system specificaton

After providing settings information, the user often defines the atomic structure to be studied (whether generated or read in). The same structure can be used to form input to various simulations (*e.g.* DFT and QMC) performed on the same system. The examples below illustrate the main options for structure input.

Read structure from a file:

```
dia16 = generate_physical_system(
    structure = './dia16.POSCAR',  # load a POSCAR file
    C         = 4                  # pseudo-carbon (4 electrons)
    )
```

Generate structure directly:

```
dia16 = generate_physical_system(
    lattice   = 'cubic',           # cubic lattice
    cell      = 'primitive',       # primitive cell
    centering = 'F',               # face-centered
    constants = 3.57,              # a = 3.57
    units     = 'A',               # Angstrom units
    atoms     = 'C',               # monoatomic C crystal
    basis     = [[0,0,0],          # basis vectors
                 [.25,.25,.25]],   #  in lattice units
    tiling    = (2,2,2),           # tile from 2 to 16 atom cell
```

```
    C         = 4                   # pseudo-carbon (4 electrons)
    )
```

Provide cell, elements, and positions explicitly:

```
dia16 = generate_physical_system(
    units  = 'A',                       # Angstrom units
    axes   = [[1.785,1.785,0.   ],      # cell axes
              [0.   ,1.785,1.785],
              [1.785,0.   ,1.785]],
    elem   = ['C','C'],                 # atom labels
    pos    = [[0.    ,0.    ,0.    ],  # atomic positions
              [0.8925,0.8925,0.8925]],
    tiling = (2,2,2),                   # tile from 2 to 16 atom cell
    kgrid  = (4,4,4),                   # 4 by 4 by 4 k-point grid
    kshift = (0,0,0),                   #  centered at gamma
    C      = 4                          # pseudo-carbon (4 electrons)
    )
```

In each of these cases, the text "`C = 4`" refers to the number of electrons in the valence for a particular element. Here a pseudopotential is being used for carbon and so it effectively has four valence electrons. One line like this should be included for each element in the structure.

4.4 Workflow specification

The next section in a Nexus user script is the specification of simulation workflows. This stage can be logically decomposed into two sub-stages: (1) specifying inputs to each simulation individually, and (2) specifying the data dependencies between simulations.

Generating simulation objects

Simulation objects are created through calls to "`generate_xxxxxx`" functions, where "`xxxxxx`" represents the name of a particular simulation code, such as `pwscf`, `vasp`, or `qmcpack`. Each `generate` function shares certain inputs, such as the path where the simulation will be performed, computational resources required by the simulation job, an identifier to differentiate between simulations (must be unique only for simulations occurring in the same directory), and the atomic/electronic structure to simulate:

```
relax = generate_pwscf(
    identifier   = 'relax',                  # identifier for the run
    path         = 'diamond/relax',          # perform run at this location
    job          = Job(cores=16,app='pw.x'), # run on 16 cores using pw.x executable
    system       = dia16,                    # 16 atom diamond cell made earlier
    pseudos      = ['C.BFD.upf'],            # pseudopotential file
    files        = [],                       # any other files to be copied in
    ...                                      # PWSCF-specific inputs follow
    )
```

The simulation objects created in this way are just data. They represent requests for particular simulations to be carried out at a later time. No simulation runs are actually performed during the creation of these objects. A basic example of generation input for each of the four major codes currently supported by Nexus is given below.

Quantum Espresso (PWSCF) generation:

```
scf = generate_pwscf(
    identifier   = 'scf',
    path         = 'diamond/scf',
    job          = scf_job,
    system       = dia16,
    pseudos      = ['C.BFD.upf'],
    input_type   = 'generic',
    calculation  = 'scf',
    input_dft    = 'lda',
    ecutwfc      = 75,
    conv_thr     = 1e-7,
    kgrid        = (2,2,2),
    kshift       = (0,0,0),
    )
```

The keywords `calculation`, `input_dft`, `ecutwfc`, and `conv_thr` will be familiar to the casual user of PWSCF. Any input keyword that normally appears as part of a namelist in PWSCF input can be directly supplied here. The `generate_pwscf` function, like most of the others, actually takes an arbitrary number of keyword arguments. These are later screened against known inputs to PWSCF to avoid errors. The `kgrid` and `kshift` inputs inform the `KPOINTS` card in the PWSCF input file, overriding any similar information provided in `generate_physical_system`.

VASP generation:

```
relax = generate_vasp(
    identifier   = 'relax',
    path         = 'diamond/relax',
    job          = relax_job,
    system       = dia16,
    pseudos      = ['C.POTCAR'],
    input_type   = 'generic',
    istart       = 0,
    icharg       = 2,
    encut        = 450,
    nsw          = 5,
    ibrion       = 2,
    isif         = 2,
    kcenter      = 'monkhorst',
    kgrid        = (2,2,2),
```

```
    kshift       = (0,0,0),
    )
```

Similar to `generate_pwscf`,`generate_vasp` accepts an arbitrary number of keyword arguments and any VASP input file keyword is accepted (the VASP keywords provided here are `istart`, `icharg`, `encut`, `nsw`, `ibrion`, and `isif`). The `kcenter`, `kgrid`, and`kshift` keywords are used to form the `KPOINTS` input file. Pseudopotentials provided through the `pseudos` keyword will fused into a single `POTCAR` file following the order of the atoms created by `generate_physical_system`.

GAMESS generation:

```
uhf = generate_gamess(
    identifier = 'uhf',
    path       = 'water/uhf',
    job        = Job(cores=16,app='gamess.x'),
    system     = h2o,
    pseudos    = [H.BFD.gms,O.BFD.gms],
    symmetry   = 'Cnv  2',
    scftyp     = 'uhf',
    runtyp     = 'energy',
    ispher     = 1,
    exetyp     = 'run',
    maxit      = 200,
    memory     = 150000000,
    guess      = 'hcore',
    )
```

The `generate_gamess` function also accepts arbitrary GAMESS keywords (`symmetry`, `scftyp`, `runtyp`, `ispher`, `exetyp`, `maxit`, `memory`, and `guess` here). The pseudopotential files `H.BFD.gms` and `O.BFD.gms` include the gaussian basis sets as well as the pseudopotential channels (the two parts are just concatenated into the same file, commented lines are properly ignored). Nexus drives the GAMESS executable (`gamess.x` here) directly without the intermediate `rungms` script as is often done. To do this, the `ericfmt` keyword must be provided in `settings` specifying the path to `ericfmt.dat`.

QMCPACK generation:

```
qmc = generate_qmcpack(
    identifier   = 'vmc',
    path         = 'diamond/vmc',
    job          = Job(cores=16,threads=4,app='qmcapp'),
    input_type   = 'basic',
    system       = dia16,
    pseudos      = ['C.BFD.xml'],
    jastrows     = [],
    calculations = [
        vmc(
```

```
            walkers     =   1,
            warmupsteps =  20,
            blocks      = 200,
            steps       =  10,
            substeps    =   2,
            timestep    =  .4
            )
        ],
    dependencies = (conv,'orbitals')
    )
```

Unlike the other `generate` functions, `generate_qmcpack` takes only selected inputs. The reason for this is that QMCPACK's input file is highly structured (nested XML) and cannot be directly mapped to keyword-value pairs. The full set of allowed keywords is beyond the scope of this section. Please refer to the user walkthroughs provided in Sec. 5 for further examples.

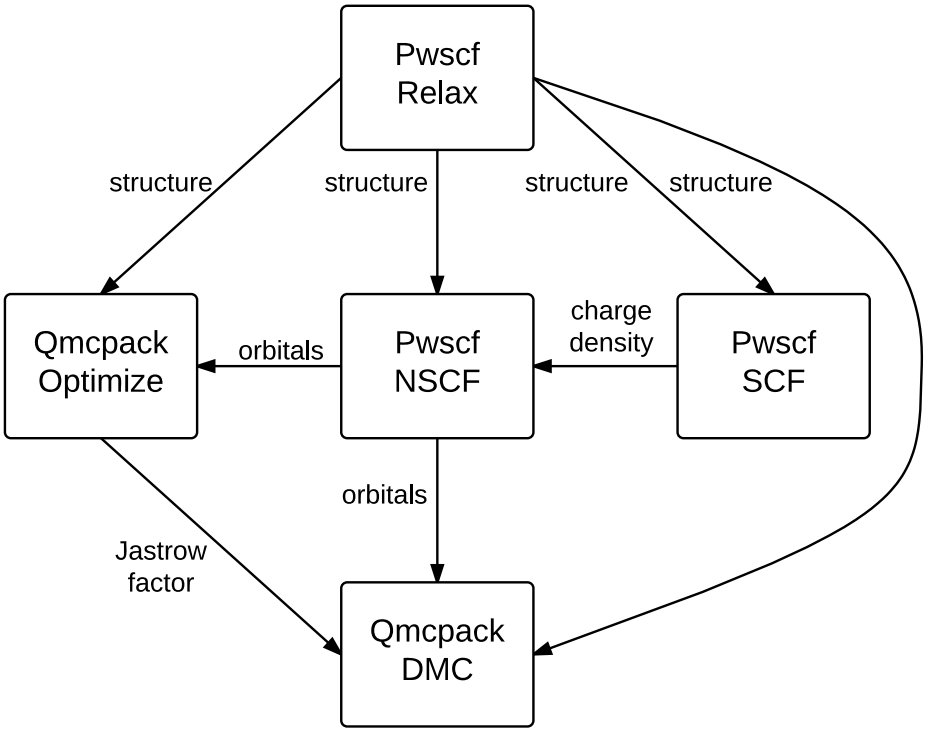

Figure 4.1: An example Nexus workflow/cascade involving QMCPACK and PWSCF. The arrows and labels denote the flow of information between the simulation runs.

Composing workflows from simulation objects

Simulation workflows are created by specifying the data dependencies between simulation runs. An example workflow is shown in Figure 4.1. In this case, a single relaxation calculation performed with PWSCF is providing a relaxed structure to each of the subsequent simulations. PWSCF is used to create a converged charge density (SCF) and then orbitals at specific k-points (NSCF). These orbitals are used by each of the two QMCPACK runs; the first optimization run provides a Jastrow factor to the final DMC run.

Below is an example of how this workflow can be created with Nexus. Most keywords to the `generate` functions have been omitted for brevity. The `conv` step listed below is implicit in Figure 4.1.

```
relax = generate_pwscf(
    ...
    )
```

```
scf = generate_pwscf(
    dependencies = (relax,'structure'),
    ...
    )

nscf = generate_pwscf(
    dependencies = [(relax,'structure'     ),
                    (scf  ,'charge_density')],
    ...
    )

conv = generate_pw2qmcpack(
    dependencies = (nscf ,'orbitals' ),
    ...
    )

opt = generate_qmcpack(
    dependencies = [(relax,'structure'),
                    (conv ,'orbitals' )],
    ...
    )

dmc = generate_qmcpack(
    dependencies = [(relax,'structure'),
                    (conv ,'orbitals' ),
                    (opt  ,'jastrow'  )],
    ...
    )
```

As suggested at the beginning of this section, workflow composition logically breaks into two parts: simulation generation and workflow dependency specification. This type of breakup can also be performed explicity within a Nexus user script, if desired:

```
# simulation generation
relax = generate_pwscf(...)
scf   = generate_pwscf(...)
nscf  = generate_pwscf(...)
conv  = generate_pw2qmcpack(...)
opt   = generate_qmcpack(...)
dmc   = generate_qmcpack(...)

# workflow dependency specification
scf.depends(relax,'structure')
nscf.depends((relax,'structure'     ),
             (scf  ,'charge_density'))
```

```
conv.depends(nscf ,'orbitals' )
opt.depends((relax,'structure'),
            (conv ,'orbitals' ))
dmc.depends((relax,'structure'),
            (conv ,'orbitals' ),
            (opt  ,'jastrow'  ))
```

More complicated workflows or scans over parameters of interest can be created with for loops and if-else logic constructs. This is fairly straightforward to accomplish because any keyword input can given a Python variable instead of a constant, as is mostly the case in the brief examples above.

4.5 Workflow execution

Simulation jobs are actually executed when the corresponding simulation objects are passed to the `run_project` function. Within the `run_project` function, most of the workflow management operations unique to Nexus are actually performed. The details of the management process is not the purpose of this section. This process is discussed in context in the example walkthroughs (Sec. 5).

The `run_project` function can be invoked in a couple of ways. The most straightforward is simply to provide all simulation objects directly as arguments to this function:

```
run_project(relax,scf,nscf,opt,dmc)
```

When complex workflows are being created (*e.g.* when the `generate` function appear in `for` loops and `if` statements), it is generally more convenient to accumulate a list of simulation objects and then pass the list to `run_project` as follows:

```
sims = []

relax = generate_pwscf(...)
sims.append(relax)

scf   = generate_pwscf(...)
sims.append(scf)

nscf  = generate_pwscf(...)
sims.append(nscf)

conv  = generate_pw2qmcpack(...)
sims.append(conv)

opt   = generate_qmcpack(...)
sims.append(opt)

dmc   = generate_qmcpack(...)
sims.append(dmc)
```

```
run_project(sims)
```

When the `run_project` function returns, all simulation runs should be finished.

4.6 Data analysis

Following the call to `run_project`, the user can perform data analysis tasks, if desired, as the analyzer object associated with each simulation contains a collection of post-processed output data rendered in numeric form (ints, floats, numpy arrays) and stored in a structured format. An interactive example for QMCPACK data analysis is shown below. Note that all analysis objects are interactively browsable in a similar manner.

```
>>> qa=dmc.load_analyzer_image()

>>> qa.qmc
  0                    VmcAnalyzer
  1                    DmcAnalyzer
  2                    DmcAnalyzer

>>> qa.qmc[2]
  dmc                  DmcDatAnalyzer
  info                 QAinformation
  scalars              ScalarsDatAnalyzer
  scalars_hdf          ScalarsHDFAnalyzer

>>> qa.qmc[2].scalars_hdf
  Coulomb              obj
  ElecElec             obj
  Kinetic              obj
  LocalEnergy          obj
  LocalEnergy_sq       obj
  LocalPotential       obj
  data                 QAHDFdata

>>> print qa.qmc[2].scalars_hdf.LocalEnergy
  error           = 0.0201256357883
  kappa           = 12.5422841447
  mean            = -75.0484800012
  sample_variance = 0.00645881103012
  variance        = 0.850521272106
```

5. Complete Examples

Disclaimer: Please note that the examples given here do not generally qualify as production calculations because the supercell size, optimization process, DMC timestep and other key parameters may not be converged. Pseudopotentials are provided "as is" and should not be trusted without explicit validation.

Complete examples of calculations performed with Nexus are provided in the following sections. These examples are intended to highlight basic features of Nexus and act as templates for future calculations. If there is an example you would like to contribute, or if you feel an example on a particular topic is needed, please contact the developer at krogeljt@ornl.gov to discuss the possibilities.

To perform the example calculations yourself, consult the `examples` directory in your Nexus installation:

```
/your_download_path/nexus/examples
```

The examples assume that you have working versions of `pw.x`, `pw2qmcpack.x`, `qmcapp` (real version), and `qmcapp_complex` (complex version) installed and in your `PATH`. A brief description of each example is given below.

Bulk Diamond VMC
: A representative bulk calculation. A simple workflow consisting of orbital generation with PWSCF, orbital conversion with pw2qmcpack, and a short VMC calculation with QMCPACK is performed.

Graphene Sheet DMC
: A representative slab calculation. The total DMC energy of a graphene "sheet" consisting of 8 atoms is computed. DFT is performed with PWSCF on the primitive cell followed by Jastrow optimization by QMCPACK and finally a supercell VMC+DMC calculation by QMCPACK.

C 20 Molecule DMC
: A representative molecular calculation. The total DMC energy of an ideal C 20 molecule is computed. DFT is performed with PWSCF on a periodic cell with some vacuum surrounding the molecule. QMCPACK optimization and VMC+DMC follow on the system with open boundary conditions.

  (Note that without the crystal field splitting afforded by the initial artificial periodicity, the Kohn-Sham HOMO would be degenerate, and so a production calculation would likely require more care in appropriately setting up the wavefunction.)

Oxygen Dimer DMC

An example demonstrating automation of a simple parameter scan, in this case the interparticle spacing in an oxygen dimer. The reader will gain some experience modifying Nexus user scripts to produce automated workflows.

5.1 Example 1: Bulk Diamond VMC

The files for this example are found in:

```
/your_download_path/nexus/examples/qmcpack/diamond
```

By following the instructions contained in this section the reader can execute a simple workflow with Nexus. The workflow presented here is intended to illustrate basic Nexus usage. The example workflow has three stages: (1) orbital generation in a primitive (2 atom) cell of diamond with PWSCF, (2) conversion of the orbitals from the native PWSCF format to the ESHDF format that QMCPACK reads, and (3) a minimal variational Monte Carlo (VMC) run of a 16 atom supercell of diamond with QMCPACK. The Nexus input script corresponding to this workflow is shown below.

The script is similar to the one discussed in section **??**, differing mainly in the name-by-name imports and the description of the physical system. Instead of reading an external VASP POSCAR file, the structure is specified in a format native to Nexus. The physical system is specified by providing the unit system (Angstrom in this case), the three vectors comprising the axes of the simulation cell, and the names and positions (Cartesian coordinates) of the atoms involved. The use of "`tiling=(2,2,2)`" communicates the request that a $2 \times 2 \times 2$ supercell be constructed out of the specified 2 atom primitive cell. The k-point grid applies to the supercell and is comprised of a single k-point. The eight corresponding primitive cell images of this k-point are determined automatically by Nexus. The text "`C = 4`" specifies the number of valence electrons for the carbon pseudopotential. The pseudopotential files used in this example (`C.BFD.*`) have been adapted from an open access pseudopotential database (see http://www.burkatzki.com/pseudos/index.2.html) for use in PWSCF and QMCPACK. Since the `PhysicalSystem` object, "`dia16`", contains both the supercell and its equivalent folded/primitive version, the PWSCF DFT calculation will be performed in the primitive cell to save memory for the subsequent VMC calculation of the full supercell performed with QMCPACK.

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

# nexus imports
from nexus import settings,Job,run_project
from nexus import generate_physical_system
from nexus import generate_pwscf
from nexus import generate_pw2qmcpack
from nexus import generate_qmcpack,vmc

# general settings for nexus
settings(
    pseudo_dir    = '../pseudopotentials',# directory with all pseudopotentials
```

```
    status_only   = 0,                    # only show status of runs
    generate_only = 0,                    # only make input files
    sleep         = 3,                    # check on runs every 3 seconds
    machine       = 'ws16'                # local machine is 16 core workstation
    )

# generate diamond structure
dia16 = generate_physical_system(
    units  = 'A',                         # Angstrom units
    axes   = [[1.785,1.785,0.   ],        # cell axes
              [0.   ,1.785,1.785],
              [1.785,0.   ,1.785]],
    elem   = ['C','C'],                   # 2 C atoms
    pos    = [[0.    ,0.    ,0.    ],    # atomic positions
              [0.8925,0.8925,0.8925]],
    tiling = (2,2,2),                     # tile to 16 atom cell
    kgrid  = (1,1,1),                     # single supercell k-point
    kshift = (0,0,0),                     #  at gamma
    C      = 4                            # pseudo-C (4 val. elec.)
    )

# scf run produces orbitals
scf = generate_pwscf(
    identifier   = 'scf',           # identifier/file prefix
    path         = 'diamond/scf',   # directory for scf run
    job          = Job(cores=16,app='pw.x'),
    input_type   = 'generic',
    calculation  = 'scf',           # perform scf calculation
    input_dft    = 'lda',           # dft functional
    ecutwfc      = 200,             # planewave energy cutoff
    conv_thr     = 1e-8,            # scf convergence threshold
    nosym        = True,            # don't use symmetry
    wf_collect   = True,            # write orbitals
    system       = dia16,           # run diamond system
    pseudos      = ['C.BFD.upf'],   # pwscf PP for C
    )

# convert orbitals for qmcpack
conv = generate_pw2qmcpack(
    identifier   = 'conv',          # identifier/file prefix
    path         = 'diamond/scf',   # directory for conv job
    job          = Job(cores=1,app='pw2qmcpack.x'),
    write_psir   = False,           # output in k-space
    dependencies = (scf,'orbitals') # get orbitals from scf
    )
```

```
# vmc run
qmc = generate_qmcpack(
    identifier   = 'vmc',           # identifier/file prefix
    path         = 'diamond/vmc',   # directory for vmc run
    job          = Job(cores=16,threads=4,app='qmcapp'),
    input_type   = 'basic',
    system       = dia16,           # run diamond system
    pseudos      = ['C.BFD.xml'],   # qmcpack PP for C
    jastrows     = [],              # no jastrows, test run
    calculations = [
        vmc(                        # vmc inputs
            walkers     =   1,      #  one walker per core
            warmupsteps =  20,      #  20 steps before measurement
            blocks      = 200,      #  200 blocks
            steps       =  10,      #   of 10 MC steps each
            substeps    =   2,      #   2 substeps w/o measurement
            timestep    =  .4       #  0.4/Ha timestep
            )
        ],
    dependencies = (conv,'orbitals')# get orbitals from conv job
    )

# nexus monitors all runs
run_project(scf,conv,qmc)
```

To fully execute the usage example provided here, copies of PWSCF, QMCPACK, and the orbital converter pw2qmcpack will need to be installed on the local machine. The example assumes that the executables are in the user's `PATH` and are named `pw.x`, `qmcapp`, and `pw2qmcpack.x`. See Sec. 6.2 for download and installation instructions for these codes. A test of Nexus including the generation of input files, but without actual job submission, can be performed without installing these codes. However, Python itself and NumPy are required to run Nexus (see Sec. **??**). The example also assumes the local machine is a workstation with 16 available cores ("`ws16`"). If fewer than 16 cores are available, *e.g.* 4, change the example files to reflect this: `ws16`→`ws4`, `Job(cores=16,...)`→`Job(cores=4,...)`.

In this example, we will run Nexus in three different modes:

1. status mode: print the status of each simulation and then exit (`status_only=1`).
2. generate mode: generate input files but do not execute workflows (`generate_only=1`).
3. execute mode: execute workflows by submitting jobs and monitoring simulation progress (`status_only=0, generate_only=0`).

Only the last mode requires executables for PWSCF and QMCPACK.

First, run Nexus in status mode. Enter the `examples/qmcpack/diamond` directory, open `diamond.py` with a text editor and set "`status_only`=1". Run the script by typing

"`./diamond.py`" at the command line and inspect the output. The output should be similar to the text below (without the comments):

```
  Pseudopotentials
    reading pp:  ../pseudopotentials/C.BFD.upf  # dft PP found
    reading pp:  ../pseudopotentials/C.BFD.xml  # qmc PP found

Project starting
  checking for file collisions        # files do not overlap
  loading cascade images              # load saved workflow state
    cascade 0 checking in             # only one workflow/cascade
  checking cascade dependencies       # match producers/consumers
    all simulation dependencies satisfied
  cascade status
    setup, sent_files, submitted, finished, got_output, analyzed
    000000  scf  ./runs/diamond/scf   # no work has been done yet
    000000  conv  ./runs/diamond/scf  # for any of the
    000000  vmc  ./runs/diamond/vmc   # three simulations
    setup, sent_files, submitted, finished, got_output, analyzed
```

The binary string "000000" indicates that none of the six stages of simulation progression have been completed. These stages correpond to the following actions/states: writing input files ("`setup`"), copying pseudopotential files ("`sent_files`"), submitting simulation jobs for execution ("`submitted`"), the completion of a simulation job ("`finished`"), collecting output files ("`got_output`"), and preprocessing output files for later analysis ("`analyzed`") . In a production setting, this mode is useful for checking the status of current workflows/cascades prior to adding new ones. It is also useful in general for detecting any problems with the Nexus input script itself.

Next, run the example in generate mode. Set "`status_only`=0" and "`generate_only`=1", then run the example script again. Instead of showing workflow status, Nexus will now perform a dry run of the workflows by generating all of the run directories and input files. The output should contain text similar to what is shown below:

```
  starting runs:                      # start submitting jobs
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  poll 0  memory 60.45 MB             # first poll cycle
    Entering ./runs/diamond/scf 0     # scf job
      writing input files  0 scf      # input file written
    Entering ./runs/diamond/scf 0
      sending required files  0 scf   # PP files copied
      submitting job  0 scf           # job is in virtual queue
```

```
  Entering ./runs/diamond/scf 0
    Would have executed:            # shows submission command
      export OMP_NUM_THREADS=1      # does not execute
      mpirun -np 16 pw.x -input scf.in

poll 1  memory 60.72 MB
  Entering ./runs/diamond/scf 0
    copying results  0 scf          # output file copying stage
  Entering ./runs/diamond/scf 0
    analyzing  0 scf                # output analysis stage

poll 2  memory 60.73 MB             # third poll cycle
  Entering ./runs/diamond/scf 1     # similar for conv job
    writing input files  1 conv
  Entering ./runs/diamond/scf 1
    sending required files  1 conv
    submitting job  1 conv
  Entering ./runs/diamond/scf 1
    Would have executed:
      export OMP_NUM_THREADS=1
      mpirun -np 1 pw2qmcpack.x<conv.in

poll 3  memory 60.73 MB
  Entering ./runs/diamond/scf 1
    copying results  1 conv
  Entering ./runs/diamond/scf 1
    analyzing  1 conv

poll 4  memory 60.73 MB             # fifth poll cycle
  Entering ./runs/diamond/vmc 2     # similar for vmc job
    writing input files  2 vmc
  Entering ./runs/diamond/vmc 2
    sending required files  2 vmc
    submitting job  2 vmc
  Entering ./runs/diamond/vmc 2
    Would have executed:
      export OMP_NUM_THREADS=4
      mpirun -np 4 qmcapp vmc.in.xml

poll 5  memory 60.78 MB
  Entering ./runs/diamond/vmc 2
    copying results  2 vmc
  Entering ./runs/diamond/vmc 2
    analyzing  2 vmc
```

```
Project finished                    # jobs finished
```

The output describes the progress of each simulation. The run submission commands are also clearly shown as well as the amount of memory used by Nexus. There should now be a "runs" directory containing the generated input files with the following structure:

```
runs/
\-- diamond               # main diamond directory
    |-- scf               #  scf directory
    |-- |-- C.BFD.upf     #   pwscf PP file
    |-- |-- conv.in       #   conv job input file
    |-- |-- pwscf_output  #   pwscf output directory
    |-- |-- scf.in        #   scf job input file
    |-- |-- sim_conv      #   nexus directory for conv
    |-- |-- |-- input.p   #    stored input object
    |-- |-- \-- sim.p     #    simulation status file
    |-- \-- sim_scf       #   nexus directory for scf
    |--     |-- input.p   #    stored input object
    |--     \-- sim.p     #    simulation status file
    \-- vmc               #  vmc directory
        |-- C.BFD.xml     #   qmcpack PP file
        |-- sim_vmc       #   nexus directory for vmc
        |-- |-- input.p   #    stored input object
        |-- \-- sim.p     #    simulation status file
        \-- vmc.in.xml    #   vmc job input file
```

The "sim.p" files record the state of each simulation. Inspect the input files generated by Nexus (scf.in, conv.in, and vmc.in.xml). Compare the files with the input provided to Nexus in diamond.py.

Finally, run the example in execute mode. Remove the "runs" and "results" directories, set "status_only=0" and "generate_only=0", and rerun the script. The output shown should be similar to what was seen for generate mode, only now there may be multiple workflow polls while a particular simulation is running. The "sleep" keyword controls how often the polls occur (every 3 seconds in this example). Note that the Nexus host process sleeps in between polls so that a minimum of computational resources are occupied. Once "Project finished" is displayed, all the simulation runs should be complete. Confirm the success of the runs by checking the output files. The text "JOB DONE." should appear near the end of the PWSCF output file scf.out. QMCPACK has completed successfully if "Total Execution time" appears near the end of the output in vmc.out.

5.2 Example 2: Graphene Sheet DMC

The files for this example are found in:

```
/your_download_path/nexus/examples/qmcpack/graphene
```

Take a moment to study the "input file" script (`graphene.py`) and the attendant comments (prefixed with #).

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

from nexus import settings,Job,run_project
from nexus import generate_physical_system
from nexus import generate_pwscf
from nexus import generate_pw2qmcpack
from nexus import generate_qmcpack
from nexus import loop,linear,vmc,dmc


# general settings for nexus
settings(
    pseudo_dir    = '../pseudopotentials',# directory with all pseudopotentials
    sleep         = 3,                    # check on runs every 'sleep' seconds
    generate_only = 0,                    # only make input files
    status_only   = 0,                    # only show status of runs
    machine       = 'ws16',               # local machine is 16 core workstation
    )



# generate the graphene physical system
graphene = generate_physical_system(
    lattice   = 'hexagonal',      # hexagonal cell shape
    cell      = 'primitive',      # primitive cell
    centering = 'P',              # primitive basis centering
    constants = (2.462,10.0),     # a,c constants
    units     = 'A',              # in Angstrom
    atoms     = ('C','C'),        # C primitive atoms
    basis     = [[ 0   , 0   , 0],  # basis vectors
                 [2./3,1./3, 0]],
    tiling    = (2,2,1),          # tiling of primitive cell
    kgrid     = (1,1,1),          # Monkhorst-Pack grid
```

```
    kshift    = (.5,.5,.5),       # and shift
    C         = 4                 # C has 4 valence electrons
    )


# list of simulations in workflow
sims = []

# scf run produces charge density
scf = generate_pwscf(
    # nexus inputs
    identifier   = 'scf',           # identifier/file prefix
    path         = 'graphene/scf',  # directory for scf run
    job          = Job(cores=16),   # run on 16 cores
    pseudos      = ['C.BFD.upf'],   # pwscf PP file
    system       = graphene,        # run graphene
    # input format selector
    input_type   = 'scf',           # scf, nscf, relax, or generic
    # pwscf input parameters
    input_dft    = 'lda',           # dft functional
    ecut         =  150 ,           # planewave energy cutoff (Ry)
    conv_thr     =  1e-6,           # scf convergence threshold (Ry)
    mixing_beta  =    .7,           # charge mixing factor
    kgrid        = (8,8,8),         # MP grid of primitive cell
    kshift       = (1,1,1),         #  to converge charge density
    wf_collect   = False,           # don't collect orbitals
    use_folded   = True             # use primitive rep of graphene
    )
sims.append(scf)

# nscf run to produce orbitals for jastrow optimization
nscf_opt = generate_pwscf(
    # nexus inputs
    identifier   = 'nscf',          # identifier/file prefix
    path         = 'graphene/nscf_opt', # directory for nscf run
    job          = Job(cores=16),   # run on 16 cores
    pseudos      = ['C.BFD.upf'],   # pwscf PP file
    system       = graphene,        # run graphene
    # input format selector
    input_type   = 'nscf',          # scf, nscf, relax, or generic
    # pwscf input parameters
    input_dft    = 'lda',           # dft functional
    ecut         =  150 ,           # planewave energy cutoff (Ry)
    conv_thr     =  1e-6,           # scf convergence threshold (Ry)
    mixing_beta  =    .7,           # charge mixing factor
    nosym        = True,            # don't symmetrize k-points
```

```
    use_folded   = True,            # use primitive rep of graphene
    wf_collect   = True,            # write out orbitals
    kgrid        = (1,1,1),         # single k-point for opt
    kshift       = (0,0,0),         # gamma point
    # workflow dependencies
    dependencies = (scf,'charge_density')
    )
sims.append(nscf_opt)

# orbital conversion job for jastrow optimization
p2q_opt = generate_pw2qmcpack(
    # nexus inputs
    identifier   = 'p2q',
    path         = 'graphene/nscf_opt',
    job          = Job(cores=1),
    # pw2qmcpack input parameters
    write_psir   = False,
    # workflow dependencies
    dependencies = (nscf_opt,'orbitals')
    )
sims.append(p2q_opt)

# Jastrow optimization
opt = generate_qmcpack(
    # nexus inputs
    identifier   = 'opt',           # identifier/file prefix
    path         = 'graphene/opt',  # directory for opt run
    job          = Job(cores=16,app='qmcapp'),
    pseudos      = ['C.BFD.xml'],   # qmcpack PP file
    system       = graphene,        # run graphene
    # input format selector
    input_type   = 'basic',
    # qmcpack input parameters
    corrections  = [],
    jastrows     = [('J1','bspline',8),   # 1 body bspline jastrow
                    ('J2','bspline',8)],  # 2 body bspline jastrow
    calculations = [
        loop(max = 6,                       # No. of loop iterations
             qmc = linear(                  # linearized optimization method
                energy               =  0.0, # cost function
                unreweightedvariance =  1.0, #   is all unreweighted variance
                reweightedvariance   =  0.0, #   no energy or r.w. var.
                timestep             =  0.5, # vmc timestep (1/Ha)
                warmupsteps          =  100, # MC steps before data collected
                samples              = 16000,# samples used for cost function
                stepsbetweensamples  =   10, # steps between uncorr. samples
```

```
                blocks              =   10, # ignore this
                minwalkers          =   0.1,#  and this
                bigchange           =  15.0,#  and this
                alloweddifference   =  1e-4 #  and this, for now
                )
            )
        ],
    # workflow dependencies
    dependencies = (p2q_opt,'orbitals')
    )
sims.append(opt)


# nscf run to produce orbitals for final dmc
nscf = generate_pwscf(
    # nexus inputs
    identifier   = 'nscf',          # identifier/file prefix
    path         = 'graphene/nscf', # directory for nscf run
    job          = Job(cores=16),   # run on 16 cores
    pseudos      = ['C.BFD.upf'],   # pwscf PP file
    system       = graphene,        # run graphene
    # input format selector
    input_type   = 'nscf',          # scf, nscf, relax, or generic
    # pwscf input parameters
    input_dft    = 'lda',           # dft functional
    ecut         =  150 ,           # planewave energy cutoff (Ry)
    conv_thr     =  1e-6,           # scf convergence threshold (Ry)
    mixing_beta  =    .7,           # charge mixing factor
    nosym        = True,            # don't symmetrize k-points
    use_folded   = True,            # use primitive rep of graphene
    wf_collect   = True,            # write out orbitals
    # workflow dependencies
    dependencies = (scf,'charge_density')
    )
sims.append(nscf)

# orbital conversion job for final dmc
p2q = generate_pw2qmcpack(
    # nexus inputs
    identifier   = 'p2q',
    path         = 'graphene/nscf',
    job          = Job(cores=1),
    # pw2qmcpack input parameters
    write_psir   = False,
    # workflow dependencies
    dependencies = (nscf,'orbitals')
```

```
    )
sims.append(p2q)

# final dmc run
qmc = generate_qmcpack(
    # nexus inputs
    identifier   = 'qmc',           # identifier/file prefix
    path         = 'graphene/qmc',  # directory for dmc run
    job          = Job(cores=16,app='qmcapp'),
    pseudos      = ['C.BFD.xml'],   # qmcpack PP file
    system       = graphene,        # run graphene
    # input format selector
    input_type   = 'basic',
    # qmcpack input parameters
    corrections  = [],              # no finite size corrections
    jastrows     = [],              # overwritten from opt
    calculations = [                # qmcpack input parameters for qmc
        vmc(                        # vmc parameters
            timestep      = 0.5,    # vmc timestep (1/Ha)
            warmupsteps   = 100,    # No. of MC steps before data is collected
            blocks        = 200,    # No. of data blocks recorded in scalar.dat
            steps         =  10,    # No. of steps per block
            substeps      =   3,    # MC steps taken w/o computing E_local
            samplesperthread = 40   # No. of dmc walkers per thread
            ),
        dmc(                        # dmc parameters
            timestep      = 0.01,   # dmc timestep (1/Ha)
            warmupsteps   =  50,    # No. of MC steps before data is collected
            blocks        = 400,    # No. of data blocks recorded in scalar.dat
            steps         =   5,    # No. of steps per block
            nonlocalmoves = True    # use Casula's T-moves
            ),                      #  (retains variational principle for NLPP's)
        ],
    # workflow dependencies
    dependencies = [(p2q,'orbitals'),
                    (opt,'jastrow')]
    )


# nexus monitors all runs
run_project(sims)


# print out the total energy
performed_runs = not settings.generate_only and not settings.status_only
if performed_runs:
```

```
    # get the qmcpack analyzer object
    # it contains all of the statistically analyzed data from the run
    qa = qmc.load_analyzer_image()
    # get the local energy from dmc.dat
    le = qa.dmc[1].dmc.LocalEnergy  # dmc series 1, dmc.dat, local energy
    #  print the total energy for the 8 atom system
    print 'The DMC ground state energy for graphene is:'
    print '    {0} +/- {1} Ha'.format(le.mean,le.error)
#end if
```

To run the example, navigate to the example directory and type

```
./graphene.py
```

or, alternatively,

```
python ./graphene.py
```

You should see output like this (without the added # comments):

```
  Pseudopotentials   # reading pseudopotential files
    reading pp:  ../pseudopotentials/C.BFD.upf
    reading pp:  ../pseudopotentials/C.BFD.xml

Project starting
  checking for file collisions  # ensure created files don't overlap
  loading cascade images        # load previous simulation state
    cascade 0 checking in
  checking cascade dependencies # ensure sim.'s have needed dep.'s
    all simulation dependencies satisfied

  starting runs:                # start submitting jobs
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  poll 0  memory 56.28 MB
    Entering ./runs/graphene/scf 0      # scf job
      writing input files  0 scf        # input file written
    Entering ./runs/graphene/scf 0
      sending required files  0 scf     # PP files copied
```

```
    submitting job  0 scf             # job is in virtual queue
  Entering ./runs/graphene/scf 0
    Executing:                        # job executed on workstation
      export OMP_NUM_THREADS=1
      mpirun -np 16 pw.x -input scf.in

poll 1  memory 56.30 MB               # waiting for job to finish
poll 2  memory 56.30 MB
poll 3  memory 56.30 MB
poll 4  memory 56.30 MB
  Entering ./runs/graphene/scf 0
    copying results  0 scf            # job is finished, copy results
  Entering ./runs/graphene/scf 0
    analyzing  0 scf                  # analyze output data

                                      # now do the same for
                                      # nscf job for Jastrow opt
                                      #   single k-point
                                      # nscf job for VMC/DMC
                                      #   multiple k-points

poll 5  memory 56.31 MB
  Entering ./runs/graphene/nscf 1     # nscf dmc
    writing input files  1 nscf
    ...
  Entering ./runs/graphene/nscfopt 4  # nscf opt
    writing input files  4 nscf
    ...

                                      # now convert KS orbitals
                                      # to eshdf format
                                      # with pw2qmcpack.x
                                      # for nscf opt & nscf dmc

poll 7  memory 56.32 MB
  Entering ./runs/graphene/nscf 2     # convert dmc orbitals
    sending required files  2 p2q
    ...
  Entering ./runs/graphene/nscfopt 4  # convert opt orbitals
    copying results  4 nscf
    ...

poll 10  memory 56.32 MB
  Entering ./runs/graphene/opt 6      # submit jastrow opt
```

```
    writing input files  6 opt        # write input file
  Entering ./runs/graphene/opt 6
    sending required files  6 opt     # copy PP files
    submitting job  6 opt             # job is in virtual queue
  Entering ./runs/graphene/opt 6
    Executing:                        # run qmcpack
      export OMP_NUM_THREADS=1        # w/ complex arithmetic
      mpirun -np 16 qmcapp_complex opt.in.xml

poll 11  memory 56.32 MB
poll 12  memory 56.32 MB
poll 13  memory 56.32 MB
...
...
...
poll 793  memory 56.32 MB   # qmcpack opt finishes
poll 794  memory 56.32 MB   # nearly an hour later
poll 795  memory 56.32 MB
  Entering ./runs/graphene/opt 6
    copying results  6 opt            # copy output files
  Entering ./runs/graphene/opt 6
    analyzing  6 opt                  # analyze the results

poll 796  memory 56.41 MB
  Entering ./runs/graphene/qmc 3      # submit dmc
    writing input files  3 qmc        # write input file
  Entering ./runs/graphene/qmc 3
    sending required files  3 qmc     # copy PP files
    submitting job  3 qmc             # job is in virtual queue
  Entering ./runs/graphene/qmc 3
    Executing:                        # run qmcpack
      export OMP_NUM_THREADS=1
      mpirun -np 16 qmcapp_complex qmc.in.xml

poll 797  memory 57.31 MB
poll 798  memory 57.31 MB
poll 799  memory 57.31 MB
...
...
...
poll 1041  memory 57.31 MB   # qmcpack dmc finishes
poll 1042  memory 57.31 MB   # about 15 minutes later
poll 1043  memory 57.31 MB
  Entering ./runs/graphene/qmc 3
```

```
      copying results  3 qmc            # copy output files
    Entering ./runs/graphene/qmc 3
      analyzing  3 qmc                  # analyze the results

Project finished                        # all jobs are finished

The DMC ground state energy for graphene is:
    -45.824960552 +/- 0.00498990689364 Ha    # one value from
                                             # qmcpack analyzer
```

If successful, you have just performed a start-to-finish DMC calculation. The total energy quoted above probably will not match the one you produce due to different compilation environments and the probabilistic nature of DMC. They should not, however differ by three sigma.

Take some time to inspect the input files generated by Nexus and the output files from PWSCF and QMCPACK. The runs were performed in sub-directories of the `runs` directory. The order of execution of the simulations is roughly `scf`, `nscf`, `nscfopt`, `opt`, then `qmc`.

```
runs
 graphene_test
     nscf
        nscf.in
        nscf.out
     nscfopt
        nscf.in
        nscf.out
     opt
        opt.in.xml
        opt.out
     qmc
        qmc.in.xml
        qmc.out
     scf
         scf.in
         scf.out
```

The directories above contain all the files generated by the simulations. Often one only wants to save the files with the most important data, which are generally small. These are copied to the `results` directory which mirrors the structure of `runs`.

```
results
 runs
     graphene_test
         nscf
            nscf.in
            nscf.out
         nscfopt
            nscf.in
            nscf.out
         opt
            opt.in.xml
            opt.out
         qmc
            qmc.in.xml
            qmc.out
         scf
             scf.in
             scf.out
```

Although this QMC run was performed at a single k-point, a twist-averaged run could be performed simply by changing `kgrid` in `generate_physical_system` from `(1,1,1)` to `(4,4,1)`, or similar.

5.3 Example 3: C 20 Molecule DMC

The files for this example are found in:

```
/your_download_path/nexus/examples/qmcpack/c20
```

Take a moment to study the "input file" script (`c20_example.py`) and the attendant comments (prefixed with #). The relevant differences from the graphene example mostly involve how the structure is procured (it is read from an XYZ file rather than being generated), the boundary conditions (open BC's, see `bconds` in the QMCPACK input parameters), and the workflow involved.

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

from nexus import settings,Job,run_project
from nexus import Structure,PhysicalSystem
from nexus import generate_pwscf
from nexus import generate_pw2qmcpack
from nexus import generate_qmcpack
from nexus import loop,linear,vmc,dmc


# general settings for nexus
settings(
    pseudo_dir    = '../pseudopotentials',# directory with all pseudopotentials
    sleep         = 3,                    # check on runs every 'sleep' seconds
    generate_only = 0,                    # only make input files
    status_only   = 0,                    # only show status of runs
    machine       = 'ws16',               # local machine is 16 core workstation
    )


#generate the C20 physical system
# specify the xyz file
structure_file = 'c20.cage.xyz'
# make an empty structure object
structure = Structure()
# read in the xyz file
structure.read_xyz(structure_file)
# place a bounding box around the structure
structure.bounding_box(
    box   = 'cubic',         # cube shaped cell
```

```
    scale = 1.5              # 50% extra space
    )
# make it a gamma point cell
structure.add_kmesh(
    kgrid      = (1,1,1),    # Monkhorst-Pack grid
    kshift     = (0,0,0)     # and shift
    )
# add electronic information
c20 = PhysicalSystem(
    structure = structure,   # C20 structure
    net_charge = 0,          # net charge in units of e
    net_spin   = 0,          # net spin in units of e-spin
    C          = 4           # C has 4 valence electrons
    )


# list of simulations in workflow
sims = []

# scf run produces charge density
scf = generate_pwscf(
    # nexus inputs
    identifier   = 'scf',           # identifier/file prefix
    path         = 'c20/scf',       # directory for scf run
    job          = Job(cores=16),   # run on 16 cores
    pseudos      = ['C.BFD.upf'],   # pwscf PP file
    system       = c20,             # run c20
    # input format selector
    input_type   = 'scf',           # scf, nscf, relax, or generic
    # pwscf input parameters
    input_dft    = 'lda',           # dft functional
    ecut         =  150 ,           # planewave energy cutoff (Ry)
    conv_thr     =  1e-6,           # scf convergence threshold (Ry)
    mixing_beta  =    .7,           # charge mixing factor
    nosym        = True,            # don't use symmetry
    wf_collect   = True,            # write out orbitals
    )
sims.append(scf)

# orbital conversion job for opt and dmc
p2q = generate_pw2qmcpack(
    # nexus inputs
    identifier   = 'p2q',
    path         = 'c20/nscf',
    job          = Job(cores=1),
    # pw2qmcpack input parameters
```

```
    write_psir   = False,
    # workflow dependencies
    dependencies = (scf,'orbitals')
    )
sims.append(p2q)


# Jastrow optimization
opt = generate_qmcpack(
    # nexus inputs
    identifier   = 'opt',           # identifier/file prefix
    path         = 'c20/opt',       # directory for opt run
    job          = Job(cores=16,app='qmcapp'),
    pseudos      = ['C.BFD.xml'],   # qmcpack PP file
    system       = c20,             # run c20
    # input format selector
    input_type   = 'basic',
    # qmcpack input parameters
    corrections  = [],
    jastrows     = [('J1','bspline',8,6),   # 1 body bspline jastrow
                    ('J2','bspline',8,8)],  # 2 body bspline jastrow
    calculations = [
        loop(max = 6,                       # No. of loop iterations
             qmc = linear(                  # linearized optimization method
                energy               =  0.0, # cost function
                unreweightedvariance =  1.0, #   is all unreweighted variance
                reweightedvariance   =  0.0, #   no energy or r.w. var.
                timestep             =  0.5, # vmc timestep (1/Ha)
                warmupsteps          =  100, # MC steps before data collected
                samples              = 16000,# samples used for cost function
                stepsbetweensamples  =   10, # steps between uncorr. samples
                blocks               =   10, # ignore this
                minwalkers           =   0.1,#  and this
                bigchange            =  15.0,#  and this
                alloweddifference    =  1e-4 #  and this, for now
                )
             )
        ],
    # workflow dependencies
    dependencies = (p2q,'orbitals')
    )
sims.append(opt)


# final dmc run
qmc = generate_qmcpack(
```

```
    # nexus inputs
    identifier   = 'qmc',           # identifier/file prefix
    path         = 'c20/qmc',  # directory for dmc run
    job          = Job(cores=16,app='qmcapp'),
    pseudos      = ['C.BFD.xml'],   # qmcpack PP file
    system       = c20,             # run c20
    # input format selector
    input_type   = 'basic',
    # qmcpack input parameters
    corrections  = [],              # no finite size corrections
    jastrows     = [],              # overwritten from opt
    calculations = [                # qmcpack input parameters for qmc
        vmc(                        # vmc parameters
            timestep      = 0.5,    # vmc timestep (1/Ha)
            warmupsteps   = 100,    # No. of MC steps before data is collected
            blocks        = 200,    # No. of data blocks recorded in scalar.dat
            steps         =  10,    # No. of steps per block
            substeps      =   3,    # MC steps taken w/o computing E_local
            samplesperthread = 40   # No. of dmc walkers per thread
            ),
        dmc(                        # dmc parameters
            timestep      = 0.01,   # dmc timestep (1/Ha)
            warmupsteps   =  50,    # No. of MC steps before data is collected
            blocks        = 400,    # No. of data blocks recorded in scalar.dat
            steps         =   5,    # No. of steps per block
            nonlocalmoves = True    # use Casula's T-moves
            ),                      #  (retains variational principle for NLPP's)
        ],
    # workflow dependencies
    dependencies = [(p2q,'orbitals'),
                    (opt,'jastrow')]
    )



# nexus monitors all runs
run_project(sims)



# print out the total energy
performed_runs = not settings.generate_only and not settings.status_only
if performed_runs:
    # get the qmcpack analyzer object
    # it contains all of the statistically analyzed data from the run
    qa = qmc.load_analyzer_image()
```

```
    # get the local energy from dmc.dat
    le = qa.dmc[1].dmc.LocalEnergy  # dmc series 1, dmc.dat, local energy
    #  print the total energy for the 20 atom system
    print 'The DMC ground state energy for C20 is:'
    print '    {0} +/- {1} Ha'.format(le.mean,le.error)
#end if
```

To run the example, navigate to the example directory and type

```
./c20.py
```

or, alternatively,

```
python ./c20.py
```

You should see output like this (without the added # comments):

```
  Pseudopotentials   # reading pseudopotential files
    reading pp:  ../pseudopotentials/C.BFD.upf
    reading pp:  ../pseudopotentials/C.BFD.xml

Project starting
  checking for file collisions  # ensure created files don't overlap
  loading cascade images        # load previous simulation state
    cascade 0 checking in
  checking cascade dependencies # ensure sim.'s have needed dep.'s
    all simulation dependencies satisfied

  starting runs:                # start submitting jobs
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  poll 0  memory 56.21 MB
    Entering ./runs/c20/scf 0       # scf job
      writing input files  0 scf    # input file written
    Entering ./runs/c20/scf 0
      sending required files  0 scf # PP files copied
      submitting job  0 scf         # job is in the virtual queue
    Entering ./runs/c20/scf 0
      Executing:                    # job executed on workstation
```

```
      export OMP_NUM_THREADS=1
      mpirun -np 16 pw.x -input scf.in

poll 1  memory 56.23 MB          # waiting for job to finish
poll 2  memory 56.23 MB
poll 3  memory 56.23 MB
poll 4  memory 56.23 MB
poll 5  memory 56.23 MB
poll 6  memory 56.23 MB
poll 7  memory 56.23 MB
poll 8  memory 56.23 MB
  Entering ./runs/c20/scf 0
    copying results  0 scf       # job is finished, copy results
  Entering ./runs/c20/scf 0
    analyzing  0 scf             # analyze output data

poll 9  memory 56.23 MB          # now convert KS orbitals
  Entering ./runs/c20/scf 1      # to eshdf format
    writing input files  1 p2q   # with pw2qmcpack.x
    ...

poll 12  memory 56.23 MB
  Entering ./runs/c20/opt 3      # submit jastrow opt
    writing input files  3 opt   # write input file
  Entering ./runs/c20/opt 3
    sending required files  3 opt # copy PP files
    submitting job  3 opt        # job is in virtual queue
  Entering ./runs/c20/opt 3
    Executing:                   # run qmcpack
      export OMP_NUM_THREADS=1   # w/ real arithmetic
      mpirun -np 16 qmcapp opt.in.xml

poll 13  memory 56.24 MB
poll 14  memory 56.24 MB
poll 15  memory 56.24 MB
...
...
...
poll 204  memory 56.24 MB   # qmcpack opt finishes
poll 205  memory 56.24 MB   # about 10 minutes later
poll 206  memory 56.24 MB
  Entering ./runs/c20/opt 3
    copying results  3 opt       # copy output files
  Entering ./runs/c20/opt 3
```

```
      analyzing  3 opt              # analyze the results

  poll 207  memory 56.27 MB
    Entering ./runs/c20/qmc 2       # submit dmc
      writing input files  2 qmc    # write input file
    Entering ./runs/c20/qmc 2
      sending required files  2 qmc # copy PP files
      submitting job  2 qmc         # job is in virtual queue
    Entering ./runs/c20/qmc 2
      Executing:                    # run qmcpack
        export OMP_NUM_THREADS=1
        mpirun -np 16 qmcapp qmc.in.xml

  poll 208  memory 56.49 MB
  poll 209  memory 56.49 MB
  poll 210  memory 56.49 MB
  ...
  ...
  ...
  poll 598  memory 56.49 MB   # qmcpack dmc finishes
  poll 599  memory 56.49 MB   # about 20 minutes later
  poll 600  memory 56.49 MB
    Entering ./runs/c20/qmc 2
      copying results  2 qmc        # copy output files
    Entering ./runs/c20/qmc 2
      analyzing  2 qmc              # analyze the results

Project finished                    # all jobs are finished

The DMC ground state energy for C20 is:
    -112.890695404 +/- 0.0151688786226 Ha  # one value from
                                           # qmcpack analyzer
```

Again, the total energy quoted above probably will not match the one you produce due to different compilation environments and the probabilistic nature of QMC. The results should still be statistically comparable.

The directory trees generated by Nexus for C 20 have a similar structure to the graphene example. Note the absence of the `nscf` runs. The order of execution of the simulations is `scf`, `opt`, then `qmc`.

```
runs
 c20_test
```

```
     opt
        opt.in.xml
        opt.out
     qmc
        qmc.in.xml
        qmc.out
     scf
         scf.in
         scf.out
results
 runs
     c20_test
         opt
            opt.in.xml
            opt.out
         qmc
            qmc.in.xml
            qmc.out
         scf
             scf.in
             scf.out
```

5.4 Example 4: Automated oxygen dimer binding curve

The files for this example are found in:

```
/your_download_path/nexus/examples/qmcpack/oxygen_dimer
```

Enter the `examples/qmcpack/oxygen_dimer` directory. Open `oxygen_dimer.py` with a text editor. The overall format is similar to the example file shown in the prior sections. The header material, including Nexus imports, settings, and the job parameters for QMC are nearly identical.

Following the job parameters, inputs for the optimization method are given. The keywords are identical to the parameters of QMCPACK's XML input file.

```
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 the native XML input file:

```
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 also mirror the XML:

```
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 prior examples, 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 DFT calculations and orbital conversions are constructed with the `generate_pwscf` and `generate_pw2qmcpack` functions.

```
scf = generate_pwscf(
    identifier   = 'scf',
    ...
    )
sims.append(scf)

p2q = generate_pw2qmcpack(
    identifier   = 'p2q',
    ...
    dependencies = (scf,'orbitals')
    )
sims.append(p2q)
```

Finally, 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,
    dependencies = (p2q,'orbitals')
    )
sims.append(opt)

qmc = generate_qmcpack(
    identifier   = 'qmc',
    ...
    jastrows     = [],
    calculations = qmc_calcs,
    dependencies = [(p2q,'orbitals'),
                    (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. Nexus 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. No input files are written or job submissions made until the "`run_project`" function is reached.

As written, `oxygen_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).

Change the "`status_only`" parameter in the "`settings`" function to `1` and type "./oxygen_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  scf  ./scale_1.0
    000000  scf  ./scale_0.9
    000000  scf  ./scale_0.95
    000000  scf  ./scale_1.05
    000000  scf  ./scale_1.1
    000000  p2q  ./scale_1.0
    000000  p2q  ./scale_0.9
    000000  p2q  ./scale_0.95
    000000  p2q  ./scale_1.05
    000000  p2q  ./scale_1.1
    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 `oxygen_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
```

Nexus 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 `oxygen_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 `oxygen_dimer.py` file (*e.g.* adding more separation distances) without worrying about duplicate job submissions.

Now actually submit the optimization and DMC jobs. Reset the state of the simulations by removing the `sim.p` files ("`rm ./scale*/sim*/sim.p`"), set "`generate_only`" to `0`, and rerun `oxygen_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 `oxygen_dimer.py` in the foreground.

After completion, try expanding to the full set of "`scale`" values: `[0.90,0.925,0.95,0.975,1.00,1.025,1.05,1.075,1.10]`. Nexus should only run the workflows that correspond to new values. In this way, Nexus scripts can be expanded over time to include new aspects of growing projects.

6. Recommended Reading

The sections below contain information, or at least links to information, that should be helpful for anyone who wants to use Nexus, but who is not an expert in one of the following areas: installing python and related modules, installing PWSCF and QMCPACK, the Python programming language, and the theory and practice of Quantum Monte Carlo.

6.1 Helpful Links for Installing Python Modules

Python itself
Download: http://www.python.org/download/
Be sure to get Python 2.x, not 3.x.

Numpy and Scipy
Download and installation: http://www.scipy.org/Installing_SciPy.

Matplotlib
Download: http://matplotlib.org/downloads.html
Installation: http://matplotlib.org/users/installing.html

H5py
Download and installation: http://www.h5py.org/

6.2 Helpful Links for Installing Electronic Structure Codes

PWSCF: pw.x, pw2qmcpack.x, pw2casino.x

Download: http://www.quantum-espresso.org/ , http://qmcpack.org/downloads/
Installation instructions: See the `README` file in `qmcpack/external_codes/quantum_espresso` for instructions on patching Quantum Espresso version 5.1.

QMCPACK: qmcapp, qmcapp_complex, convert4qmc, ppconvert, sqd

Download: http://qmcpack.org/downloads/
Install: http://docs.qmcpack.org/ug/a00002.html

Wfconvert: wfconvert

Download: svn co http://qmctools.googlecode.com/svn/trunk/wfconvert
See also: https://code.google.com/p/qmctools/

VASP

Download: Proprietary, see https://www.vasp.at/
Install: http://cms.mpi.univie.ac.at/vasp/vasp/vasp.html

GAMESS

Download: http://www.msg.ameslab.gov/gamess/License_Agreement.html Install: See the "readme.unix" file in the GAMESS source distribution (gamess/machines/readme.unix).

6.3 Brushing Up On Python

Python

Python is a flexible, multi-paradigm, interpreted programming language with powerful intrinsic datatypes and a large library of modules that greatly expand its functionality. A good way to learn the language is through the extensive Documentation provided on the python.org website. If you have never worked with Python before, be sure to go through the Tutorial. To learn more about the intrinsic data types and standard libraries look at Library Reference. A very short introduction to Python is in Appendix A.

Documentation	http://docs.python.org/2/
Tutorial	http://docs.python.org/2/tutorial/index.html
Library Reference	http://docs.python.org/2/library/index.html

NumPy

Other than the Python Standard Library, the main library/module Nexus makes heavy use of is NumPy. NumPy provides a convenient and fairly fast implementation of multi-dimensional arrays and related functions, much like MATLAB. If you want to learn about NumPy arrays, the NumPy Tutorial is recommended. For more detailed information, see the NumPy User Guide and the NumPy Reference Manual. If MATLAB is one of your native languages, check out NumPy for MATLAB Users.

Tutorial	http://www.scipy.org/Tentative_NumPy_Tutorial
User Guide	http://docs.scipy.org/doc/numpy/user/index.html#user
Reference	http://docs.scipy.org/doc/numpy/reference/
MATLAB	http://www.scipy.org/NumPy_for_Matlab_Users

Matplotlib

Plotting in Nexus is currently handled by Matplotlib. If you want to learn more about plotting with Matplotlib, the Pyplot Tutorial is a good place to start. More detailed information is in the User's Guide. Sometimes Examples provide the fastest way to learn.

Tutorial	http://matplotlib.org/users/pyplot_tutorial.html
User's Guide	http://matplotlib.org/users/index.html
Examples	http://matplotlib.org/examples/

Scipy and H5Py

Nexus also occasionally uses functionality from SciPy and H5Py. Learning more about them is unlikely to help you interact with Nexus. However, they are quite valuable on their own. SciPy provides access to special functions, numerical integration, optimization, interpolation, fourier transforms, eigenvalue solvers, and statistical analysis. To get an overview, try the SciPy Tutorial. More detailed material is found in the Scipy Reference. H5Py provides a NumPy-like interface to HDF5 data files, which QMCPACK creates. To learn more about interacting with HDF5 files through H5Py, try the Quick Start Guide. For more information, see the General Documentation.

SciPy Tutorial	http://docs.scipy.org/doc/scipy/reference/tutorial/index.html
SciPy Reference	http://docs.scipy.org/doc/scipy/reference/
H5Py Quick Guide	http://www.h5py.org/docs/intro/quick.html#quick
H5Py General Docs	http://www.h5py.org/docs/

6.4 Quantum Monte Carlo: Theory and Practice

Currently, review articles may be the best way to get an overview of Quantum Monte Carlo methods and practices. The review article by Foulkes, *et al.* from 2001 remains quite relevant and is lucidly written. Other review articles also provide a broader perspective on QMC, including more recent developments. Another resource that can be useful for newcomers (and needs to be updated) is the QMC Wiki. If you are aware of resources that fill a gap in the information presented here (almost a certainty), please contact the developer at krogeljt@ornl.gov to add your contribution.

QMC Review Articles	
Foulkes, 2001	http://rmp.aps.org/abstract/RMP/v73/i1/p33_1
Bajdich, 2009	http://www.physics.sk/aps/pub.php?y=2009&pub=aps-09-02
Needs, 2010	http://iopscience.iop.org/0953-8984/22/2/023201/
Kolorenc, 2011	http://iopscience.iop.org/0034-4885/74/2/026502/
Online Resources	
QMCWiki	www.qmcwiki.org
QMC Summer School 2012	http://www.mcc.uiuc.edu/summerschool/2012/program.html

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/.

A.1 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'
```

A.2 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.

A.3 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.

A.4 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}
```

A.5 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}
```

B. QMC Practice in a Nutshell

The aim of this section is to provide a very brief overview of the essential concepts undergirding Quantum Monte Carlo calculations of electronic structure with a particular focus on the key approximations and quantities to converge to achieve high accuracy. The discussion here is not intended to be comprehensive. For deeper perspectives on QMC, please see the review articles listed in the "Recommended Reading" section (6.4).

B.1 VMC and DMC in the abstract

Ground state QMC methods, such as Variational (VMC) and Diffusion (DMC) Monte Carlo, attempt to obtain the ground state energy of a many-body quantum system.

$$E_0 = \frac{\langle \Psi_0 | \hat{H} | \Psi_0 \rangle}{\langle \Psi_0 | \Psi_0 \rangle} \tag{B.1.1}$$

The VMC method obtains an upper bound on the ground state energy (guaranteed by the Variational Principle) by introducing a guess at the ground state wavefunction, known as the trial wavefunction Ψ_T:

$$E_{VMC} = \frac{\langle \Psi_T | \hat{H} | \Psi_T \rangle}{\langle \Psi_T | \Psi_T \rangle} \geq E_0 \tag{B.1.2}$$

The DMC method improves on this variational bound by projecting out component eigenstates of the trial wavefunction lying higher in energy than the ground state. The operator that acts as a projector is the imaginary time, or thermodynamic, density matrix:

$$\begin{aligned} |\Psi_t\rangle &= e^{-t\hat{H}} |\Psi_T\rangle \\ &= e^{-tE_0} \left(|\Psi_0\rangle + \sum_{n>0} e^{-t(E_n - E_0)} |\Psi_n\rangle \right) \\ &\xrightarrow[t\to\infty]{} e^{-tE_0} |\Psi_0\rangle \end{aligned} \tag{B.1.3}$$

The DMC energy approaches the ground state energy from above as the imaginary time becomes large.

$$E_{DMC} = \lim_{t\to\infty} \frac{\langle \Psi_t | \hat{H} | \Psi_t \rangle}{\langle \Psi_t | \Psi_t \rangle} = E_0 \tag{B.1.4}$$

However from the equations above, one can already anticipate that the DMC method will struggle in the face of degeneracy or near-degeneracy.

In principle, the DMC method is exact for the ground state, but further complications arise for systems that are extended, comprised of fermions, or contain heavy nuclei, pseudized or otherwise. Approximations arising from the numerical implementation of the method also require care to keep under control.

B.2 From expectation values to random walks

Evaluating expectation values of a many-body system involves performing high dimensional integrals (the dimensionality is at least the dimensions of the physical space times the number of particles). In VMC, for example, the expectation value of the total energy is represented succinctly as:

$$E_{VMC} = \int dR |\Psi_T|^2 E_L \tag{B.2.1}$$

where E_L is the local energy $E_L = \Psi_T^{-1} \hat{H} \Psi_T$. The other factor in the integral $|\Psi_T|^2$ can clearly be thought of as a probability distribution and can therefore be sampled by Monte Carlo methods (such as the Metropolis algorithm) to evaluate the integral exactly.

The sampling procedure takes the form of random walks. A "walker" is just a set of particle positions, along with a weight, that evolves (or moves) to new positions according to a set of statisical rules. In VMC as few walkers are used as possible to reduce the equilibration time (the number of steps or moves required to lose a memory of the potentially poor starting guess for particle posistions). In DMC, the walker population is a dynamic feature of the calculation and must be large enough to avoid introducing bias in expectation values.

The tradeoff of moving to a the sampling procedure for the integration is that it introduces statistical error into the calculation which diminishes slowly with the number of samples (it falls off like $1/(\#ofsamples)$ by the Central Limit Theorem). The good news for ground state QMC is that this error can be reduced more rapidly through the discovery of better guesses at the detailed nature of the many-body wavefunction.

B.3 Quality orbitals: planewaves, cutoffs, splines, and meshes

Acting on an understanding of perturbation theory, the zeroth order representation of the wavefunction of an interacting system takes the form of a Slater determinant of single particle orbitals. In practice, QMC calculations often obtain a starting guess at these orbitals from Hartree-Fock or Density Functional Theory calculations (which already contain non-perturbative contributions from correlation). An important factor in the generation and use of these orbitals is to ensure that they are described to high accuracy within the parent theory.

For example, when taking orbitals from a planewave DFT calculation, one must take care to converge the planewave energy cutoff to a sufficient level of accuracy (usually far beyond what is required to obtain an accurate DFT energy). One criterion to use it to converge the kinetic energy of the Kohn-Sham wavefunction with respect to the planewave energy cutoff until it is accurate to the energy scale you care about in your production QMC calcuation. For systems with a small number of valence electrons, a cutoff of around 200 Ry is often sufficient. To obtain the kinetic energy from a PWSCF calculation the `pw2casino.x` post-processing tool can be used. In Nexus one has the option to compute the kinetic energy by setting the `kinetic_E` flag in the `standard_qmc` or `basic_qmc` convenience functions.

For efficiency reasons, QMC codes often use a real-space representation of the wavefunction. It is common to represent the orbitals in terms of B-splines which have control points, or knots, that fall on a regular 3-D mesh. Analogous to the planewave cutoff, the fineness of the B-spline mesh controls the quality of the represented orbitals. To verify that the quality of the orbitals has not been compromised during the conversion process from planewave to B-spline, one often performs a VMC calculation with the B-spline Slater determinant wavefunction to obtain the kinetic energy. This value should agree with the kinetic energy of the planewave representation within the energy scale of interest.

In QMCPACK, the B-spline mesh is controlled with the `meshfactor` keyword. Larger values correspond to finer meshes. A value of 1.0 usually gives a similar quality representation as the original planewave calculation. Control of this parameter is made available in Nexus through the `meshfactor` keyword in the `standard_qmc` or `basic_qmc` convenience functions.

B.4 Quality Jastrows: less variance = more efficient

Taking a further que from perturbation theory, the first order correction to the Slater determinant wavefunction is the Jastrow correlation prefactor.

$$\Psi_T \approx e^{-J}\Psi_{Slater\,Det.} \tag{B.4.1}$$

In a quantum liquid, an appropriate form for the Jastrow factor is:

$$J = \sum_{i<j} u_{ij}(|r_i - r_j|) \tag{B.4.2}$$

This form is often used without modification in electronic structure calculations. Note that the correlation factors u_{ij} can be different for particles of differing species, or, if one of the particles in the pair is classical (such as a heavy atomic nucleus), the local electronic environment varies across the system.

The primary role of the Jastrow factor is to increase the efficiency of the QMC calculation. The variance of the local energy across all samples of the random walk is directly related to the statistical error of the final results:

$$v_{\Psi_T} = \frac{1}{N_{samples}} \sum_{s\in samples} E_L(s)^2 - \left[\frac{1}{N_{samples}} \sum_{s\in samples} E_L(s)\right]^2 \tag{B.4.3}$$

$$\sigma_{error} \approx \sqrt{\frac{v_{\Psi_T}}{N_{samples}}} \tag{B.4.4}$$

The variance of local energy is usually minimized by performing a statistical optimization of the Jastrow factor with QMC.

In addition to selecting a good form for the pair correlation functions u_{ij} (which are represented in QMCPACK as 1-D B-spline functions with a finite cutoff radius), the (iterative) optimization procedure must be performed with a sufficient number of samples to converge all the free parameters. Starting with a small number of samples ($\approx 20,000$) is usually preferable for early iterations, followed by a larger number for later iterations. This larger

number is something close to $100,000 \times (\#\ of\ free\ parameters)^2$. For B-spline functions, the number of free parameters is the number of control points, or knots.

The number of samples is controlled with the `samples` keyword in QMCPACK. Control of this parameter is made available in Nexus through the `samples` keyword in the `linear` or `cslinear` convenience functions (Which are often used in conjunction with `standard_qmc` or `basic_qmc`). For a B-spline correlation factor, the number of free parameters/knots is indicated by the `size` keyword in either QMCPACK or Nexus.

B.5 Finite size effects: k-points, supercells, and corrections

For extended systems, finite size errors are a key consideration. In addition to the finite size effects that are typically seen in DFT (k-points related). Correlated, many-body methods such as QMC also must contend with correlation-related finite size effects. Both types of finite-size effects are reduced by simply using larger supercells. The complete elimination of finite size effects using this approach can be prohibitively costly since the finite size error typically falls off like $1/\Omega_C$, where Ω_C is the volume of the supercell. A more sophisticated approach involves a combination of the supercell size, k-point grid, and additional estimated corrections for correlation finite size effects.

Although there is no firm rule on the selection of these three elements, adhering to some general guidelines is usually helpful. For a production calculation of an extended system, the minimum supercell size is around 50 atoms. The size of the supercell k-point grid can then be determined by proxy with a DFT calculation (converge the energy down to the scale of interest). Note that although the cost of a DFT calculation scales linearly with the number of k-points, the cost of the corresponding QMC calculation is hardly increased due to the statistical averaging of the results (the QMC calculation at each separate supercell k-point is simply performed with fewer samples so that the total number of samples remains fixed w.r.t. the number of k-points). Finally, corrections for correlation-related finite size effects are computed during the QMC run and added to the result by hand in post-processing the data.

In Nexus, the supercell size is controlled through the `tiling` parameter in the `generate_physical_system`, `generate_structure`, `Structure`, or `Crystal` convenience functions. Supercells can also be constructed by tiling exising structures through the `tile` member function of `Structure` or `PhysicalSystem` objects. The k-point grid is controlled through the `kgrid` parameter in the `generate_physical_system`, `generate_structure`, `Structure`, or `Crystal` convenience functions. K-point grids can also be added to existing structures through the `add_kmesh` member function of `Structure` or `PhysicalSystem` objects.

B.6 Imaginary time discretization: the DMC timestep

An analytic form for the imaginary time projection operator is not known, but real-space approximations to it can be obtained in the small time limit. With importance sampling included (not covered here), the short-time projector splits into two parts, known as the

drift-diffusion and branching factors (shown below in atomic units):

$$\rho(R', R; t) = \langle R'|\hat{\Psi}_T e^{-t\hat{H}} \hat{\Psi}_T{}^{-1}|R\rangle \tag{B.6.1}$$

$$= G_d(R', R; t) G_b(R', R, t) + \mathcal{O}(t^2) \tag{B.6.2}$$

$$G_d(R', R; t) \equiv \exp\left(-\tfrac{1}{2t}\left[R' - R - t\nabla_R \log \Psi_T(R)\right]^2\right) \tag{B.6.3}$$

$$G_b(R', R; t) \equiv \exp\left(\tfrac{1}{2}\left[E_L(R') + E_L(R)\right]\right) \tag{B.6.4}$$

The long-time projector is found as the product of many approximate short-time solutions, which takes the form of a many-body path integral in real space:

$$\rho(R_M, R_0; M\tau) = \int dR_1 dR_{M-1} \ldots \prod_{m=0}^{M-1} \rho(R_{m+1}, R_m; \tau) \tag{B.6.5}$$

The short-time parameter τ is known as the DMC timestep and accurate quantities are obtained only in the limit as τ approaches zero.

Ensuring that the timestep error is sufficiently small usually involves performing many DMC calculations over a range of timesteps (sometimes on a smaller supercell than the production calculation). The largest timestep is chosen that produces a bias smaller than the energy scale of interest. For very high accuracy, one uses the total energy as a function of timestep to extrapolate to the zero time limit.

The DMC timestep is made available in Nexus through the `timestep` parameter of the `dmc` convenience function (which is often used in conjuction with the `standard_qmc`, `basic_qmc`, `generate_qmcpack`, or `Qmcpack` functions).

B.7 Population control bias: safety in numbers

While the drift-diffusion factor $G_d(R', R; \tau)$ can be sampled exactly using Gaussian distributed random numbers (this generates the DMC random walk), the branching factor $G_b(R', R; \tau)$ is handled a different way for efficiency. The product of branching factors over an imaginary time trajectory (random walk) serves as a statistical weight for each walker. The fluctuations in this weight rapidly become quite large as the random walk progresses (because it approaches an infinite product of real numbers). As its name suggests, this weight factor is used to "branch" walkers every few steps. If the weight is small the walker is deleted, but if the weight is large the walker is copied many times ("branched") with each copy carrying a weight close to unity. This is more efficient because more walkers are created (and thus more statistics are gathered) in the high weight regions of phase space that contribute most to the integral.

The branching process in DMC naturally leads to a fluctuating population of walkers. The fluctuations in the walker population, if left to its own dynamics, are unbounded. This means that the walker population can grow very large, or even become zero. To prevent collapse of the walker population, population control techniques (not covered here) are added to the algorithm. The practical upshot of population control is that it introduces a systematic bias in the DMC results that scales like $1/(\#ofwalkers)$ (Although note that another route to reduce the population control bias is to improve the trial wavefunction, since the fluctuations in the branching weights will become zero for the exact ground state).

For many production calculations, population control bias is not much of an issue because the simulations are performed on supercomputers with thousands of cores per run, and thus tens of thousands of walkers. As a rule of thumb, the walker population should at least number in the thousands. One should occasionally explicitly check the magnitude of the population control bias for the system under study since predictions have been made that it will eventually diverge exponentially with the number of particles in the system.

The DMC walker population can be directly controlled in QMCPACK or Nexus through the `samples` (total walker population) or `samplesperthread` (walkers per OpenMP thread) keywords in the VMC block directly proceeding DMC (`vmc` convenience function in Nexus). If you opt to use the `samples` keyword, check that each thread in the calculation will have at least a few walkers.

B.8 The fixed node/phase approximation: varying the nodes/phase

For every fermionic system, the bosonic ground state lies lower in energy than the fermionic ground state. This means that projection methods like DMC will approach the bosonic ground state exponentially fast in imaginary time if unconstrained (this would show up as an exponentially diverging statistical error). In order to guarantee that the projected wavefunction remains in the space of fermionic functions (and consequently that the projected energy remains an upper bound to the fermionic ground state energy), the projected wavefunction is constrained to share the nodes (if it is real-valued) or the phase (if it is complex-valued) of the trial wavefunction. The fixed node/phase approximation represents one of the two most important approximations for electronic structure calculations (the other is the pseudopotential approximation covered in the next section).

The fixed node/phase error can be reduced, but it cannot be completely eliminated unless the exact nodes/phase is known. A common approach to reduce the fixed node/phase error is to perform several DMC calculations (sometimes on a smaller supercell) with different sets of orbitals (perhaps generated with different functionals). Another, more expensive approach, is to include the backflow transformation (this is the second order correction to the wavefunction; it is not covered in any detail here) to get a lower bound on how large the fixed node error is in standard Slater-Jastrow calculations.

To perform a calculation of this type (scanning over orbitals from different functionals) with Nexus, the DFT functional can be selected with the `functional` keyword in the `standard_qmc` or `basic_qmc` convenience functions. If you are using pseudopotentials generated for use in DFT, you should maintain consistency between the functional and pseudopotential. Even if such consistency is maintained, the impact of using DFT pseudopotentials (or those made with many other theories) in QMC can be significant.

B.9 Pseudopotentials: theoretical dissonance, the locality approximation, and T-moves

The accurate use of pseudopotentials in electronic structure QMC calculations remains one of the largest challenges in current practice. The necessity for pseudopotentials arises

from the rapidly increasing computational cost with increasing nuclear charge (it scales like Z^6, compared with the $N^3_{electrons}$ scaling with Z fixed). The challenge in using pseudopotentials in QMC is that practically no pseudopotentials exist that have been generated self-consistently with QMC. In other words, QMC is currently reliant on other theories to provide the pseudopotentials, which can be a critical source of error.

The current state-of-the-art is not without rigor, however. One source of Dirac-Fock based pseudopotentials, the Burkatzki-Filippi-Dolg database (see http://www.burkatzki.com/pseudos/index.2.html), has been explicitly vetted against quantum chemistry calculations of atoms (a higher-fidelity proxy for QMC calculations of small systems). It must be stressed that these pseudopotentials should still be validated for use in a particular target system. Another collection of Dirac-Fock pseudopotentials that have been created for use in QMC can be found in the Trail-Needs database (see http://www.tcm.phy.cam.ac.uk/~mdt26/casino2_pseudopotentials.html). Many current calculations also use the OPIUM package (see http://opium.sourceforge.net/) to generate DFT pseudopotentials and then port them directly to QMC.

Whatever the source of pseudopotentials (but perhaps especially so for those derived from DFT), testing and validation remains an important step preceeding production calculations. One option is to perform parallel pseudopotential and all-electron DMC calculations of atoms with varying electron count (*i.e.* ionization potential/electron affinity calculations). As with any electronic structure calculation, it is also advisable to devise a test in or close to the target host environment. Validating pseudopotentials remains a difficult task, and while the suggestions presented here may be of some help, they do not amount to a panacea for the issue.

Beyond the central approximation of using a pseudopotential at all, two approximations unique to pseudopotential use in DMC merit discussion. The direct use of non-local pseudopotentials in DMC leads to a second sign-problem (akin to the fixed-node issue) in the imaginary time projector. One solution, devised first, is known as the locality approximation. In the locality approximation, the non-local pseudopotential is replaced by a "localized" form: $V_{NLPP} \rightarrow \Psi_T^{-1} V_{NLPP} \Psi_T$. This approximation becomes exact as the trial wavefunction approaches the pseudo ground state, however the Variational Principle of the pseudo-system is lost (though it should be acknowledged that a non-variational portion of the energy has been discarded by using pseudopotentials at all). The Variational Principle for the pseudo-system can be restored with an advanced sampling technique known as T-moves (although the first incarnation of the technique reduces to the locality approximation as the system becomes larger than several atoms, the second version fixes this oversight).

One can select whether to use the locality approximation or T-moves (version 1!) in QMCPACK from within Nexus by setting the parameter `nonlocalmoves` to True or False in the `dmc` convenience function.

B.10 Other approximations: what else is missing?

Though a few points could be selected for mention at this point, only one additional approximation will be highlighted here. In most modern QMC calculations of electronic structure, relativistic effects have been neglected entirely (there have been a few exceptions) or simply assumed to be covered by the pseudopotential. Clearly this will become an issue for systems

with large effective core charges. At present, relativistic corrections are not available within QMCPACK.