QMCPACK

User's Guide and Developer's Manual
v1.0
February 15, 2016

Contents

Chapter 1

Introduction

QMCPACK is an open-source, high-performance electronic structure code that implements numerous Quantum Monte Carlo algorithms. Its main applications are electronic structure calculations of molecular, periodic 2D and periodic 3D solid-state systems. Variational Monte Carlo (VMC), diffusion Monte Carlo (DMC) and a number of other advanced QMC algorithms are implemented. By directly solving the Schrodinger equation, QMC methods offer greater accuracy than methods such as density functional theory, but at a trade-off of much greater computational expense. Distinct from many other correlated many-body methods, QMC methods are readily applicable to both bulk (periodic) and isolated molecular systems.

QMCPACK is written in C++ and designed with the modularity afforded by object-oriented programming. It makes extensive use of template metaprogramming to achieve high computational efficiency. Due to the modular architecture, the addition of new wavefunctions, algorithms, and observables is relatively straightforward. For parallelization QMCPACK utilizes a fully hybrid (OpenMP,CUDA)/MPI approach to optimize memory usage and to take advantage of the growing number of cores per SMP node or graphical processing units (GPUs) and accelerators. High parallel and computational efficiencies are achievable on the largest supercomputers. Finally, QMCPACK utilizes standard file formats for input and output in XML and HDF5 to facilitate data exchange.

This manual currently serves as an introduction to the essential features of QMCPACK and a guide to installing and running it. Over time this manual will be expanded to including a fuller introduction to QMC methods in general and to include more of the specialized features in QMCPACK.

1.1 Quickstart and a first QMCPACK calculation

If you are keen to get started this section describes how to quickly build and run QMCPACK on standard UNIX or Linux-like system. The autoconfiguring build system usually works without much fuss on these systems. If C++, MPI, BLAS/LAPACK, FFTW, HDF5, and CMake are already installed, QMCPACK can be built and run within five minutes. For supercomputers, cross-compilation systems, and other computer clusters the build system may require hints on the locations of libraries and which versions to use, typical of any code, see Chapter 2. Section includes complete examples for common workstations and supercomputers that you can reuse.

To build QMCPACK:

1. Download the latest QMCPACK distribution from `http://www.qmcpack.org`
2. Untar the archive, e.g., `tar xvf qmcpack_v1.3.tar.gz`
3. Check the instructions in the README
4. Run CMake in a suitable build directory to configure QMCPACK for your system: `cd qmcpack/build; cmake ..`
5. If CMake is unable to find all needed libraries, see Chapter 2 for instructions and specific build instructions for common systems.
6. Build QMCPACK: `make` or `make -j 16`, the latter for a faster parallel build on a system using, e.g., 16 processes.
7. The QMCPACK executable is `bin/qmcpack`

QMCPACK is distributed with examples illustrating different capabilities. Most of the examples are designed to run quickly with modest resources. We'll run a short diffusion Monte Carlo calculation of a water molecule:

1. Go to the appropriate example directory: `cd ../examples/molecules`
2. (Optional) Put the QMCPACK binary on your path:
   `export PATH=$PATH:location-of-qmcpack/build/bin`
3. Run QMCPACK: `../../build/bin/qmcpack simple-H2O.xml` or `qmcpack simple-H2O.xml` if you followed the step above.
4. The run will output to the screen and generate a number of files:

   ```
   $ls H2O*
   H2O.HF.wfs.xml      H2O.s001.scalar.dat H2O.s002.cont.xml
   H2O.s002.qmc.xml    H2O.s002.stat.h5    H2O.s001.qmc.xml
   H2O.s001.stat.h5    H2O.s002.dmc.dat    H2O.s002.scalar.dat
   ```

5. Partially summarized results are in the standard text files with the suffixes scalar.dat and dmc.dat. They are viewable with any standard editor.

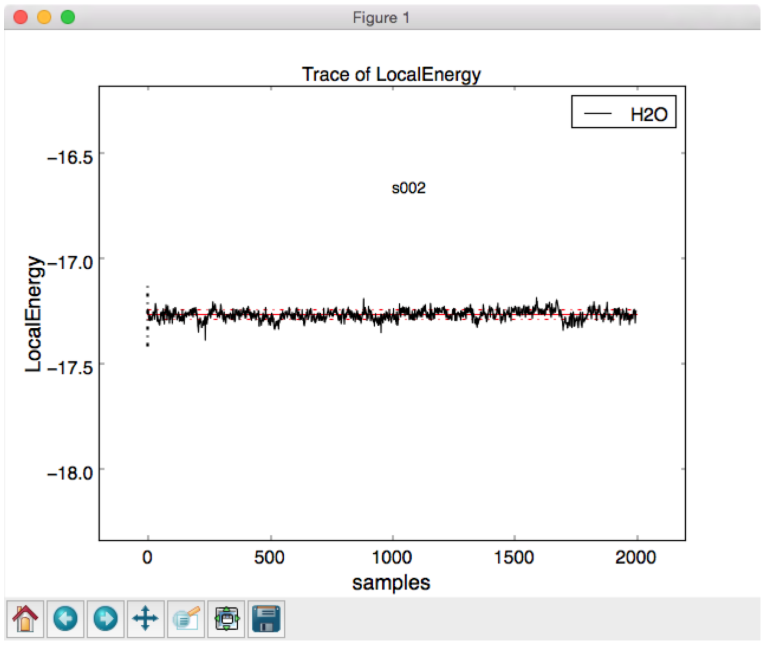


Figure 1.1: Trace of walker energies produced by qmca tool for simple water molecule example.

If you have python and matplotlib installed, you can use the `qmca` analysis utility to produce statistics and plots of the data. See Chapter 9 for information on analysing QMCPACK data.

```
export PATH=$PATH:location-of-qmcpack/nexus/executables
export PYTHONPATH=$PYTHONPATH:location-of-qmcpack/nexus/library
qmca H2O.s002.scalar.dat         # For statistical analysis of the DMC data
qmca -t -q e H2O.s002.scalar.dat # Graphical plot of DMC energy
```

The last command will produce a graph as per Fig. 1.1. This shows the average energy of the DMC walkers at each timestep. In a real simulation we would have to check equilibration, convergence with walker population, timestep etc.

Congratulations, you have completed a DMC calculation with QMCPACK!

1.2 Authors and History

QMCPACK was initially written by Jeongnim Kim while in the group of Prof. David Ceperley at the University of Illinois at Urbana-Champaign, with later contributations at Oak Ridge National Laboratory. Over the years, many others have contributed, particularly students and researchers in the groups of Prof. David Ceperley and Prof. Richard M. Martin, as well as staff at Lawrence

Livermore National Laboratory, Sandia National Laboratories, Argonne National Laboratory, and Oak Ridge National Laboratory.

The primary and original author of the code is Jeongnim Kim. Additional developers, contributors, and advisors include: Anouar Benali, Mark A. Berrill, David M. Ceperley, Simone Chiesa, Raymond C. III Clay, Bryan Clark, Kris T. Delaney, Kenneth P. Esler, Paul R. C. Kent, Jaron T. Krogel, Ying Wai Li, Ye Luo, Jeremy McMinis, Miguel A. Morales, William D. Parker, Nichols A. Romero, Luke Shulenburger, Norman M. Tubman, and Jordan E. Vincent.

If you should be added to this list please let us know.

Development of QMCPACK has been supported financially by several grants, including:

- "Network for ab initio many-body methods: development, education and training" supported through the Predictive Theory and Modeling for Materials and Chemical Science program by the U.S. Department of Energy Office of Science, Basic Energy Sciences.
- "QMC Endstation", supported by Accelerating Delivery of Petascale Computing Environment at the DOE Leadership Computing Facility at ORNL.
- PetaApps, supported by the U. S. National Science Foundation.
- Materials Computational Center, supported by the U.S. National Science Foundation.

1.3 Support and Contacting the Developers

Questions about installing, applying or extending QMCPACK can be posted on the QMCPACK Google group `https://groups.google.com/forum/#!forum/qmcpack`. You may also email any of the developers, but we recommend checking the group first. Particular attention is given to any problem reports.

1.4 Performance

QMCPACK implements modern Monte Carlo algorithms, is highly parallel, and is also written using very efficient code for high per-CPU or on node performance. In particular the code is highly vectorizable, giving high performance on modern CPUs and GPUs. We believe QMCPACK delivers performance either comparable to or better than other QMC codes when similar calculations are run, particularly for the most common QMC methods and for large systems. If you find a calculation where this is not the case, or you simply find performance slower than expected, please post on the Google group or contact one of the developers. These reports are valuable. If your calculation is sufficiently mainstream we will optimize QMCPACK to improve the performance.

1.5 Open source license

QMCPACK is distributed under the University of Illinois/NCSA Open Source License.

1.6 Contributing to QMCPACK

QMCPACK is fully open source and we welcome contributions. Please post on the QMCPACK Google group or contact the developers. If you are planning a development, early discussions are encouraged. We will be able to tell you if anyone else if working on a similar feature or if any related work has been done in the past. Credit for your contribution can be obtained, e.g., through citation of a paper, or becoming one of the authors on the next version of the standard QMCPACK reference citation.

Please note the following guidelines for a contributions:

- Additions should be fully synchronized with the latest release version and ideally the latest development SVN. Merging of code developed on older versions is error prone.
- Code should be cleanly formatted, commented, portable, and accessible to other programmers. i.e. If you need to use any clever tricks, add a comment to note this, why the trick is needed, how it works etc. Although we like high performance, ease of maintenance and accessibility are also considerations.
- Comment your code. You are not only writing it for the compiler for also for other humans! (We know this is a repeat of the previous point, but it is important enough to repeat.)
- Write a brief description of the method, algorithms and inputs and outputs suitable for inclusion in this manual.
- Develop some short tests that exercise the functionality that can be used for validation and for examples. We can help with this and their integration into the test system.

1.7 QMCPACK Roadmap

A general outline of the QMCPACK roadmap is given below. Suggestions for improvements are welcome, particularly those that would facilitate new scientific applications. For example, if an interface to a particular quantum chemical or density functional code would help, this would be given strong consideration.

1.7.1 Code

We will to continue improving the accessibility and usability of QMCPACK, by combinations of more convenient input parameters, improved workflow, integration with more quantum chemical and density functional codes, and a wider range of examples.

In terms of methodological development, we expect to significantly increase the range of QMC algorithms in QMCPACK in the near future.

Computationally, we are porting QMCPACK to the next generation of supercomputer systems. The internal changes required to run on these systems efficiently are expected to benefit *all* platforms due to improved vectorization, cache utilization and memory performance.

1.7.2 Documentation

This manual currently describes the core features of QMCPACK that are required for routine research calculations. i.e. the VMC and DMC methods, how to obtain and optimize trial wavefunctions, and simple observables. Over time this manual will be expanded to include a broader introduction to QMC methods and to describe more features of the code.

Due to its history as a research code, QMCPACK contains a variety of additional QMC methods, trial wavefunction forms, potentials (etc.) that, although not critical, may be very useful for specialized calculations or particular material or chemical systems. These “secret features” (every code has these) are not actually secret but simply lack descriptions, example inputs, and tests. You are encouraged to browse and read the source code to find them. New descriptions will be added over time, but can also be prioritized and added on request, e.g. if a specialized Jastrow factor would help or an historical Jastrow form is needed for benchmarking.

Chapter 2

Obtaining, installing and validating QMCPACK

This chapter describes how to obtain, build and validate QMCPACK. This process is designed to be as simple as possible and should be no harder than building a modern plane-wave density functional theory code such as Quantum Espresso, QBox, or VASP. Parallel builds enable a complete compilation in under 2 minutes on a fast multicore system. If you are unfamiliar with building codes we suggest working with your system administrator to install QMCPACK.

2.1 Installation steps

To install QMCPACK, follow the steps listed below. Full details of each step are given in the referenced sections.

1. Download the source code, Sections 2.2 or 2.3.
2. Verify that you have the required compilers, libraries and tools installed, Section 2.4.
3. Run the cmake configure step and build with make, Section 2.5 and 2.5.1. Some examples for common systems are given in Section 2.6.
4. Run the tests to verify QMCPACK, Section 2.7.
5. Build the ppconvert utility in QMCPACK, Section 2.8.
6. Download and patch Quantum Espresso. This patch adds the pw2qmcpack utility, Section 2.9.

Hints for high performance are in Section 2.10. Troubleshooting suggestions are in Section 2.11.

Note that there are two different QMCPACK executables that can be produced: the general one, which is the default, and the "complex" version which support periodic calculations at arbitrary twist angles and k-points. This second version is enabled via a cmake configuration parameter, see Section 2.5.3. The general version only supports wavefunctions that can be made real. If you run a calculation that needs the complex version, QMCPACK will stop and inform you.

2.2 Obtaining the latest release version

Major releases of QMCPACK are distributed from `http://www.qmcpack.org`. These releases undergo the most testing. Unless there are specific reasons we encourage all production calculations to use the latest release versions.

Releases are usually compressed tar files indicating the version number, date, and often the source code revision control number corresponding to the release.

- Download the latest QMCPACK distribution from `http://www.qmcpack.org`.
- Untar the archive, e.g., `tar xvf qmcpack_v1.3.tar.gz`

2.3 Obtaining the latest development version

The most recent development version of QMCPACK can be obtained anonymously via

```
svn checkout https://svn.qmcpack.org/svn/trunk
```

Once checked-out, updates can be made via the standard `svn update`.

The subversion repository contains the day-to-day development source with the latest updates, bugfixes etc. This may be useful for updates to the build system to support new machines, for support of the latest versions of Quantum Espresso, or for updates to the documentation. Note that the development version may not be fully consistent with the online documentation. We attempt to keep the development version fully working. However, please be sure to run the tests and compare with previous release versions before using for any serious calculations. We try to keep bugs out, but occasionally they crawl in! Reports of any breakages are appreciated.

2.4 Prerequisites

The following are required to build QMCPACK. For workstations, these are available via the standard package manager. On shared supercomputers this software is usually installed by default and is often access via a modules environment - check your system documentation.

Use of the latest versions of all compilers and libraries is strongly encouraged, but not absolutely essential. Generally newer versions are faster - see Section 2.10 for performance suggestions.

- C/C++ compilers such as GCC, Intel, IBM XLC. CLANG-based compilers are not yet supported by the build system, but the source code is ready.
- MPI library such at OpenMPI `http://open-mpi.org`
- BLAS/LAPACK, numerical and linear algebra libraries. Use platform-optimized libraries where available, such as Intel MKL. ATLAS or other optimized open-source libraries may also be used `http://math-atlas.sourceforge.net`
- CMake, build utility, `http://www.cmake.org`
- Libxml2, XML parser, `http://xmlsoft.org`
- HDF5, portable I/O library, `http://www.hdfgroup.org/HDF5/`
- BOOST, peer-reviewed portable C++ source libraries, `http://www.boost.org`
- FFTW, FFT library, `http://www.fftw.org/`

To build the GPU accelerated version of QMCPACK an installation of NVIDIA CUDA development tools is required. Ensure that this is compatible with the C and C++ compiler versions you plan to use. Supported versions are included in the NVIDIA release notes.

Many of the utilities provided with QMCPACK use python (v2). The numpy and matplotlib libraries are required for full functionality.

Note that the standalone einspline library used by previous versions of QMCPACK is no longer required. A more optimized version is included inside. The standalone version should *not* be on any standard search paths because conflicts between the old and new include files can result.

2.5 Building with CMake

The build system for QMCPACK is based on CMake. It will autoconfigure based on the detected compilers and libraries. The most recent version of CMake has the best detection for the greatest variety of systems - at the time of writing this means CMake 3.4.3. The much older CMake 2.8 is known to work, but might not work optimally on your system.

Previously QMCPACK made extensive use of toolchains, but the build system has since been updated to eliminate the use of toolchain files for most cases. The build system is verified to work with GNU, Intel, and IBM XLC compilers. Specific compile options can be specified either through specific environmental or CMake variables. When the libraries are installed in standard locations, e.g., /usr, /usr/local, there is no need to set environmental or cmake variables for the packages.

2.5.1 Quick build instructions (try first)

If you are feeling lucky and are on a standard UNIX-like system such as a Linux workstation, the following might quickly give a working QMCPACK:

The safest quick build option is to specify the C and C++ compilers through their MPI wrappers. Here we use Intel MPI and Intel compilers. Move to the build directory, run cmake and make

```
cd build
cmake -DCMAKE_C_COMPILER=mpiicc -DCMAKE_CXX_COMPILER=mpiicpc ..
make -j 8
```

You can increase the "8" to the number of cores on your system for faster builds. Substitute mpicc and mpicxx or other wrapped compiler names to suit your system. e.g. With OpenMPI use

```
cd build
cmake -DCMAKE_C_COMPILER=mpicc -DCMAKE_CXX_COMPILER=mpicxx ..
make -j 8
```

If you are feeling particularly lucky, you can skip the compiler specification:

```
cd build
cmake ..
make -j 8
```

The complexities of modern computer hardware and software systems are such that you should check that the autoconfiguration system has made good choices and picked optimized libraries and compiler settings before doing significant production. i.e. Check the details below. We give examples for a number of common systems in Section 2.6.

2.5.2 Environment variables

A number of environmental variables affect the build. In particular they can control the default paths for libraries, the default compilers, etc. The list of environmental variables is given below:

CXX	C++ compiler
CC	C Compiler
MKL_HOME	Path for MKL
LIBXML2_HOME	Path for libxml2
HDF5_ROOT	Path for HDF5
BOOST_ROOT	Path for Boost
FFTW_HOME	Path for FFTW

2.5.3 Configuration options

In addition to reading the environmental variables, CMake provides a number of optional variables that can be set to control the build and configure steps. When passed to CMake, these variables will take precedent over the environmental and default variables. To set them add -D FLAG=VALUE to the configure line between the cmake command and the path to the source directory.

- Key QMCPACK build options

```
QMC_CUDA           Enable CUDA and GPU acceleration (1:yes, 0:no)
QMC_COMPLEX        Build the complex (general twist/k-point) version (1:yes, 0:no)
```

- General build options

```
CMAKE_BUILD_TYPE   A variable which controls the type of build
                   (defaults to Release). Possible values are:
                   None (Do not set debug/optmize flags, use
                   CMAKE_C_FLAGS or CMAKE_CXX_FLAGS)
                   Debug (create a debug build)
                   Release (create a release/optimized build)
                   RelWithDebInfo (create a release/optimized build with debug info)
                   MinSizeRel (create an executable optimized for size)
CMAKE_C_COMPILER   Set the C compiler
CMAKE_CXX_COMPILER Set the C++ compiler
CMAKE_C_FLAGS      Set the C flags.  Note: to prevent default
                   debug/release flags from being used, set the CMAKE_BUILD_TYPE=None
                   Also supported: CMAKE_C_FLAGS_DEBUG,
                   CMAKE_C_FLAGS_RELEASE, and CMAKE_C_FLAGS_RELWITHDEBINFO
CMAKE_CXX_FLAGS    Set the C++ flags.  Note: to prevent default
                   debug/release flags from being used, set the CMAKE_BUILD_TYPE=None
                   Also supported: CMAKE_CXX_FLAGS_DEBUG,
                   CMAKE_CXX_FLAGS_RELEASE, and CMAKE_CXX_FLAGS_RELWITHDEBINFO
```

- Additional QMCPACK build options

```
QMC_DATA            Specify data directory for QMCPACK (currently
                    unused, but likely to be used for performance tests)
QMC_INCLUDE         Add extra include paths
QMC_EXTRA_LIBS      Add extra link libraries
QMC_BUILD_STATIC    Add -static flags to build
```

- libxml related

  ```
  Libxml2_INCLUDE_DIRS  Specify include directories for libxml2
  Libxml2_LIBRARY_DIRS  Specify library directories for libxml2
  ```

- FFTW related

  ```
  FFTW_INCLUDE_DIRS   Specify include directories for FFTW
  FFTW_LIBRARY_DIRS   Specify library directories for FFTW
  ```

2.5.4 Configure and build using cmake and make

To configure and build QMPACK, move to build directory, run cmake and make

```
cd build
cmake ..
make -j 8
```

As you will have gathered, cmake encourages "out of source" builds, where all the files for a specific build configuration reside in their own directory separate from the source files. This allows multiple builds to be created from the same source files which is very useful where the filesystem is shared between different systems. You can also build versions with different settings (e.g. QMC_COMPLEX) and different compiler settings. The build directory does not have to be called build - use something descriptive such as build_machinename or build_complex. The ".." in the cmake line refers to the directory containing CMakeLists.txt. Update the ".." for other build directory locations.

2.5.5 Example configure and build

- Set the environments (the examples below assume bash, Intel compilers and MKL library)

  ```
  export CXX=icpc
  export CC=icc
  export MKL_HOME=/usr/local/intel/mkl/10.0.3.020
  export LIBXML2_HOME=/usr/local
  export HDF5_ROOT=/usr/local
  export BOOST_ROOT=/usr/local/boost
  export FFTW_HOME=/usr/local/fftw
  ```

- Move to build directory, run cmake and make

  ```
  cd build
  cmake -D CMAKE_BUILD_TYPE=Release ..
  make -j 8
  ```

2.5.6 Build scripts

It is recommended to create a helper script that contains the configure line for CMake. This is particularly useful when avoiding environmental variables, packages are installed in custom locations, or if the configure line is long or complex. In this case it is also recommended to add ”rm -rf CMake*” before the configure line to remove existing CMake configure files to ensure a fresh configure each time that the script is called. Deleting all the files in the build directory is also acceptable. If you do so we recommend to add some sanity checks in case the script is run from the wrong directory, e.g., checking for the existence of some QMCPACK files.

Some build script examples for different systems are given in the config directory. For example, on Cray systems these scripts might load the appropriate modules to set the appropriate programming environment, specific library versions etc.

An example script build.sh is given below. It is overly complex for the sake of example:

```
export CXX=mpic++
export CC=mpicc
export ACML_HOME=/opt/acml-5.3.1/gfortran64
export HDF5_ROOT=/opt/hdf5
export BOOST_ROOT=/opt/boost

rm -rf CMake*

cmake                                                  \
  -D CMAKE_BUILD_TYPE=Debug                           \
  -D Libxml2_INCLUDE_DIRS=/usr/include/libxml2        \
  -D Libxml2_LIBRARY_DIRS=/usr/lib/x86_64-linux-gnu \
  -D FFTW_INCLUDE_DIRS=/usr/include                   \
  -D FFTW_LIBRARY_DIRS=/usr/lib/x86_64-linux-gnu      \
  -D QMC_EXTRA_LIBS="-ldl ${ACML_HOME}/lib/libacml.a -lgfortran" \
  -D QMC_DATA=/projects/QMCPACK/qmc-data              \
  ..
```

2.6 Installation instructions for common workstations and supercomputers

This section describes how to build QMCPACK on various common systems including multiple Linux distributions, Apple OS X, and various supercomputers. The examples should serve as good starting points for building QMCPACK on similar machines. For example, the software environment on modern Crays is very consistent. Note that updates to operating systems and system software may require small modifications to these recipes. See Section 2.10 for key points to check to obtain highest performance and Section 2.11 for troubleshooting hints.

2.6.1 Installing on Ubuntu Linux or other apt-get based distributions

The following is designed to obtain a working QMCPACK build on e.g. a student laptop, starting from a basic Linux installation with none of the developer tools installed. Fortunately, all the required packages are available in the default repositories making for a quick installation. Note that for convenience we use a generic BLAS. For production a platform optimized BLAS should be used.

```
apt-get subversion cmake g++ openmpi-bin libopenmpi-dev libboost-dev
apt-get libatlas-base-dev liblapack-dev libhdf5-dev libxml2-dev fftw3-dev
export CXX=mpiCC
cd build
cmake ..
make -j 8
ls -l bin/qmcpack
```

For qmca and other tools to function, we install some python libraries:

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

2.6.2 Installing on CentOS Linux or other yum based distributions

The following is designed to obtain a working QMCPACK build on e.g. a student laptop, starting from a basic Linux installation with none of the developer tools installed. CentOS 7 (Red Hat compatible) is using gcc 4.8.2. The installation is only complicated by the need to install another repository to obtain HDF5 packages which are not available by default. Note that for convenience we use a generic BLAS. For production a platform optimized BLAS should be used.

```
sudo yum install make cmake gcc gcc-c++ subversion openmpi openmpi-devel fftw fftw-devel \
                  boost boost-devel libxml2 libxml2-devel
sudo yum install blas-devel lapack-devel atlas-devel
module load mpi
```

To setup repoforge as a source for the HDF5 package, go to `http://repoforge.org/use` . Install the appropriate up to date release package for your OS. By default the CentOS Firefox will offer to run the installer. The CentOS 6.5 settings were usable for HDF5 on CentOS 7 in July 2014, but use CentOS 7 versions when they become available.

```
sudo yum install hdf5 hdf5-devel
```

To build QMCPACK

```
module load mpi/openmpi-x86_64
which mpirun
# Sanity check; should print something like   /usr/lib64/openmpi/bin/mpirun
export CXX=mpiCC
cd build
cmake ..
make -j 8
ls -l bin/qmcpack
```

2.6.3 Installing on Mac OS X using Macports

These instructions assume a fresh installation of macports and for consistency with current Linux distributions, use the gcc 4.8.2 compiler. More recent versions are fine, but it is vital to ensure matching compilers/options for all packages and to force use of what is installed in /opt/local. As with the Linux examples above, this build is very good if not optimal, and is easily good enough to learn QMCPACK or experiment on a travel laptop.

Note that we utilize the Apple provided Accelerate framework for optimized BLAS.

Follow the Macports install instructions `https://www.macports.org/`

- Install Xcode and the Xcode Command Line Tools
- Agree to Xcode license in Terminal: sudo xcodebuild -license
- Install MacPorts for your version of OS X

Install the required tools:

```
sudo port install gcc48
sudo port select gcc mp-gcc48  # Set default

sudo port install openmpi-devel-gcc48
sudo port select set mpi openmpi-devel-gcc48-fortran  # Set default

# Sanity check
mpiCXX -v
#should return  gcc version 4.8.2 (MacPorts gcc48 4.8.2_2) or similar.

sudo port install fftw-3 +gcc48
sudo port install cmake    # already cmake 3 as of 2014/7/29
sudo port install boost +gcc48
sudo port install libxml2
sudo port install hdf5-18 +gcc48

sudo port select set python python27
sudo port install py27-matplotlib  # For qmca
```

QMCPACK build:

```
export CXX=mpiCXX
export CC=/opt/local/bin/gcc
export LIBXML2_HOME=/opt/local/
export HDF5_HOME=/opt/local
export BOOST_HOME=/opt/local
export FFTW_HOME=/opt/local
cd build
cmake ..
make -j 6 # Adjust for available core count
ls -l bin/qmcpack
```

2.6.4 Installing on Mac OS X using Homebrew (brew)

Homebrew is a package manager for OS X that provides a convenient route to install all the QMCPACK dependencies. The following recipe will install the latest available versions of each package. This was successfully tested under OS X 10.11 "El Capitain" in February 2016. Note that it is necessary to build the MPI software from source to use the brew-provided gcc instead of Apple CLANG.

1. Install Homebrew from `http://brew.sh/`

   ```
   /usr/bin/ruby -e "$(curl -fsSL
       https://raw.githubusercontent.com/Homebrew/install/master/install)"
   ```

2. Install the prerequisites

   ```
   brew install gcc # Builds full gcc 5 from scratch, will take 30 minutes
   export HOMEBREW_CXX=g++-5
   export HOMEBREW_CC=gcc-5
   brew install mpich2 --build-from-source
   # Build from source required to use homebrew compiled compilers as
   # opposed to Apple CLANG. Check "mpicc -v" indicates Homebrew gcc 5.x.x
   ```

```
brew install cmake
brew install fftw
brew install boost
brew install homebrew/science/hdf5
#Note: Libxml2 is not required via brew since OS X already includes it.
```

3. Configure and build QMCPACK

   ```
   cmake -DCMAKE_C_COMPILER=/usr/local/bin/mpicc \
         -DCMAKE_CXX_COMPILER=/usr/local/bin/mpicxx ..
   make -j 12
   ```

4. Run the short tests. When mpich is used for the first time, OS X will request approval of the network connection.

   ```
   ctest -R short
   ```

2.6.5 Installing on ANL ALCF Mira/Cetus IBM Blue Gene/Q

Mira/Cetus is a Blue Gene/Q supercomputer at Argonne National Laboratory's Argonne Leadership Computing Facility (ANL ALCF). Mira has 49152 compute nodes and each node has a 16-core PowerPC A2 processor with 16 GB DDR3 memory. Due to the fact that the login nodes and the compute nodes have different processors with distinct instruction sets, cross-compiling is required on this platform. See details about using Blue Gene/Q at `http://www.alcf.anl.gov/user-guides/compiling-linking`. On Mira, compilers are loaded via softenv and users need to add +mpiwrapper-xl and +cmake in $HOME/.soft. In order to build QMCPACK, a toolchain file is provided for setting up CMake and the cmake command should be executed twice.

```
cd build
cmake -DCMAKE_TOOLCHAIN_FILE=../config/BGQToolChain.cmake ..
cmake -DCMAKE_TOOLCHAIN_FILE=../config/BGQToolChain.cmake ..
make -j 16
ls -l bin/qmcpack
```

In addition, adding a very useful cmake option -DCMAKE_VERBOSE_MAKEFILE=TRUE allows printing all the build commands during the make step. Alternatively you can use make VERBOSE=1.

2.6.6 Installing on ORNL OLCF Titan Cray XK7 (NVIDIA GPU accelerated)

Titan is a GPU accelerated supercomputer at Oak Ridge National Laboratory's Oak Ridge Leadership Computing Facility (ORNL OLCF). Each compute node has a 16 core AMD 2.2GHz Opteron 6274 (Interlagos) and an NVIDIA Kepler accelerator. The standard Cray software environment is available, with libraries accessed via modules. The only extra settings required to build the GPU version are the cudatoolkit module and specifying -DQMC_CUDA=1 on the cmake configure line.

Note that on Crays the compiler wrappers "CC" and "cc" are used. The build system checks for these and does not (should not) use the compilers directly.

```
module swap PrgEnv-pgi PrgEnv-gnu # Use gnu compilers
module load cudatoolkit           # CUDA for GPU build
module load cray-hdf5
module load cmake
module load fftw
export FFTW_HOME=$FFTW_DIR/..
module load boost
mkdir build_titan_gpu
cd build_titan_gpu
cmake -DQMC_CUDA=1 ..             # Must enable CUDA capabilities
make -j 8
ls -l bin/qmcpack
```

2.6.7 Installing on ORNL OLCF Titan Cray XK7 (CPU version)

As noted in Section2.6.6 for the GPU, building on Crays requires only loading the appropriate library modules.

```
module swap PrgEnv-pgi PrgEnv-gnu # Use gnu compilers
module unload cudatoolkit         # No CUDA for CPU build
module load cray-hdf5
module load cmake
module load fftw
export FFTW_HOME=$FFTW_DIR/..
module load boost
mkdir build_titan_cpu
cd build_titan_cpu
cmake ..
make -j 8
ls -l bin/qmcpack
```

2.6.8 Installing on ORNL OLCF Eos Cray XC30

Eos is Cray XC30 with 16 core Intel Xeon E5-2670 processors connected by the Aries interconnect. The build process is identical to Titan, except that we use the default Intel programming environment. This is usually preferred to GNU.

```
module load cray-hdf5
module load cmake
module load fftw
export FFTW_HOME=$FFTW_DIR/..
module load boost
mkdir build_eos
cd build_eos
cmake ..
make -j 8
ls -l bin/qmcpack
```

2.6.9 Installing on NERSC Edison Cray XC30

Edison is a Cray XC30 with dual 12-core Intel ”Ivy Bridge” nodes installed at NERSC. The build settings are identical to eos.

```
module load cray-hdf5
module load cmake
module load fftw
export FFTW_HOME=$FFTW_DIR/..
module load boost
mkdir build_edison
cd build_edison
cmake ..
make -j 8
ls -l bin/qmcpack
```

When the above was tested on 1 February 2016, the following module and software versions were present:

```
qmcpack@edison04:trunk> module list
Currently Loaded Modulefiles:
  1) modules/3.2.10.3                     16) alps/5.2.3-2.0502.9295.14.14.ari
  2) nsg/1.2.0                            17) rca/1.0.0-2.0502.57212.2.56.ari
  3) eswrap/1.1.0-1.020200.1130.0         18) atp/1.8.3
  4) switch/1.0-1.0502.57058.1.58.ari     19) PrgEnv-intel/5.2.56
  5) craype-network-aries                    20) craype-ivybridge
  6) craype/2.5.0                            21) cray-shmem/7.3.0
  7) intel/15.0.1.133                        22) cray-mpich/7.3.0
  8) cray-libsci/13.3.0                      23) slurm/edison
  9) udreg/2.3.2-1.0502.9889.2.20.ari        24) altd/2.0
 10) ugni/6.0-1.0502.10245.9.9.ari           25) darshan/2.3.0
 11) pmi/5.0.10-1.0000.11050.0.0.ari         26) subversion/1.7.9
 12) dmapp/7.0.1-1.0502.10246.8.47.ari       27) cray-hdf5/1.8.14
 13) gni-headers/4.0-1.0502.10317.9.2.ari    28) cmake/2.8.11.2
 14) xpmem/0.1-2.0502.57015.1.15.ari         29) fftw/3.3.4.6
 15) dvs/2.5_0.9.0-1.0502.1958.2.55.ari      30) boost/1.54
```

2.6.10 Installing on NERSC Cori (Phase 1) Cray XC40

Cori is a Cray XC40 with 16-core Intel "Haswell" nodes installed at NERSC. The build settings are identical to eos.

```
module load cray-hdf5
module load cmake
module load fftw
export FFTW_HOME=$FFTW_DIR/..
module load boost
mkdir build_cori
cd build_cori
cmake ..
make -j 8
ls -l bin/qmcpack
```

When the above was tested on 1 February 2016, the following module and software versions were present:

```
qmcpack@cori05:trunk> module list
Currently Loaded Modulefiles:
  1) nsg/1.2.0                             15) dvs/2.5_0.9.0-1.0502.2188.1.116.ari
  2) modules/3.2.10.3                      16) alps/5.2.4-2.0502.9774.31.11.ari
  3) eswrap/1.1.0-1.020200.1231.0          17) rca/1.0.0-2.0502.60530.1.62.ari
  4) switch/1.0-1.0502.60522.1.61.ari      18) atp/1.8.3
  5) intel/16.0.0.109                      19) PrgEnv-intel/5.2.82
  6) craype-network-aries                  20) craype-haswell
  7) craype/2.4.2                          21) cray-shmem/7.2.5
  8) cray-libsci/13.2.0                    22) cray-mpich/7.2.5
  9) udreg/2.3.2-1.0502.10518.2.17.ari     23) slurm/cori
 10) ugni/6.0-1.0502.10863.8.29.ari        24) cray-hdf5/1.8.14
 11) pmi/5.0.9-1.0000.10911.0.0.ari        25) gcc/5.1.0
 12) dmapp/7.0.1-1.0502.11080.8.76.ari     26) cmake/3.3.2
 13) gni-headers/4.0-1.0502.10859.7.8.ari  27) fftw/3.3.4.5
 14) xpmem/0.1-2.0502.64982.5.3.ari        28) boost/1.59
```

2.7 Testing and validation of QMCPACK

We **strongly encourage** running the included tests each time QMCPACK is built. These compare the results from the executable with known-good mean-field, quantum chemical, and other QMC results.

The tests included with QMCPACK currently test only the VMC code with single determinant wavefunction and simple spline Jastrow wavefunctions, and for gaussian and periodic spline basis sets. Although not yet comprehensive, it is extremely unlikely that, e.g., DMC will be correct if the VMC tests do not pass. We check that the known mean field results are obtained with no Jastrow. When Jastrow functions are included we test against previous QMC data. The tests are statistical with a generous 3 σ tolerance, however the system sizes are small, typically < 10 electrons, so the error bars are typically small.

The "short" tests only take a few minutes on a 16 core machine. You can run these tests using the command below in the build directory:

```
ctest -R short   # Run the tests with "short" in their name
```

The output should be similar to the following:

```
Test project build_gcc
      Start  1: short-LiH_dimer_ae-vmc_hf_noj-16-1
 1/44 Test  #1: short-LiH_dimer_ae-vmc_hf_noj-16-1 .............  Passed   11.20 sec
      Start  2: short-LiH_dimer_ae-vmc_hf_noj-16-1-kinetic
 2/44 Test  #2: short-LiH_dimer_ae-vmc_hf_noj-16-1-kinetic ......  Passed    0.13 sec
..
42/44 Test #42: short-monoO_1x1x1_pp-vmc_sdj-1-16 ..............  Passed   10.02 sec
      Start 43: short-monoO_1x1x1_pp-vmc_sdj-1-16-totenergy
43/44 Test #43: short-monoO_1x1x1_pp-vmc_sdj-1-16-totenergy .....  Passed    0.08 sec
      Start 44: short-monoO_1x1x1_pp-vmc_sdj-1-16-samples
44/44 Test #44: short-monoO_1x1x1_pp-vmc_sdj-1-16-samples .......  Passed    0.08 sec

100% tests passed, 0 tests failed out of 44

Total Test time (real) = 167.14 sec
```

Note that the number of tests that are run varies between the standard, complex, and GPU compilations.

The full set of tests consist of significantly longer versions of the short tests. They require several hours each to run yielding a much more stringent test of the code. To run all the tests simply run ctest in the build directory:

```
ctest            # Run all the tests. This will take several hours.
```

You can also run verbose tests which direct the QMCPACK output to the standard output:

```
ctest -V -R short   # Verbose short tests
```

The test system includes specific tests for the complex version of the code.

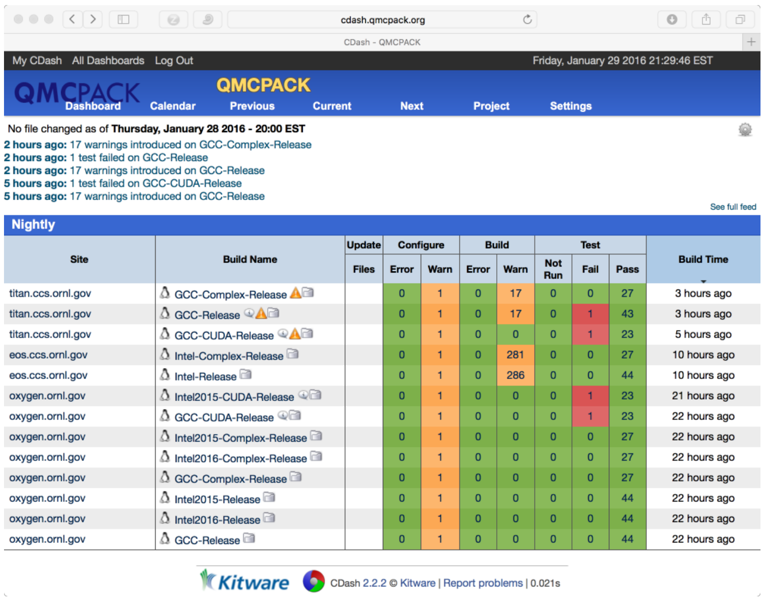
The data files for the tests are located in the tests directory. The runs occur in build/src/QMCApp/test/test_name. The numerical comparisons and test definitions are in src/QMCApp/test/CMakeLists.txt. If *all* the QMC tests fail it is likely that the appropriate mpiexec (or aprun, srun) is not being called or found. If the QMC runs appear to work but all the other tests fail it is possible that python is not working on your system - we suggest checking some of the test outputs in build/src/QMCApp/test/test_name.

Note that because the tests are very small, consisting of only a few electrons, the performance is not representative of larger calculations. For example, while the calculations might fit in cache, there will be essentially no vectorization due to the small electron counts. **The tests should not be used for any benchmarking or performance analysis**. Dedicated larger runs are required.

2.7.1 Automatic tests of QMCPACK

The QMCPACK developers run automatic tests of QMCPACK on several different computer systems, many on a continuous basis. We currently test the following combinations nights (workstation) or weekly (supercomputers):

- On a Red Hat Linux workstation:
  - GCC 4.8.2 with OpenMPI and CUDA 7.0 (GPU build, run on NVIDIA K40s)
  - GCC 4.8.2 with OpenMPI
  - Intel 2016 with Intel MPI and MKL
  - Intel 2015 with Intel MPI and MKL and CUDA 7.0 (GPU build, run on NVIDIA K40s)
  - Intel 2015 with Intel MPI and MKL
- On Eos, a Cray XC30 Intel machine:
  - The default Intel programming environment and compiler with Cray MPI and Intel MKL



cdash.qmcpack.org

CDash - QMCPACK

My CDash All Dashboards Log Out Friday, January 29 2016 21:29:46 EST

QMCPACK

Dashboard Calendar Previous Current Next Project Settings

No file changed as of **Thursday, January 28 2016 - 20:00 EST**

2 hours ago: 17 warnings introduced on GCC-Complex-Release
2 hours ago: 1 test failed on GCC-Release
2 hours ago: 17 warnings introduced on GCC-Release
5 hours ago: 1 test failed on GCC-CUDA-Release
5 hours ago: 17 warnings introduced on GCC-Release

See full feed

Nightly

Site	Build Name	Update	Configure		Build		Test			Build Time
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass	
titan.ccs.ornl.gov	GCC-Complex-Release		0	1	0	17	0	0	27	3 hours ago
titan.ccs.ornl.gov	GCC-Release		0	1	0	17	0	1	43	3 hours ago
titan.ccs.ornl.gov	GCC-CUDA-Release		0	1	0	0	0	1	23	5 hours ago
eos.ccs.ornl.gov	Intel-Complex-Release		0	1	0	281	0	0	27	10 hours ago
eos.ccs.ornl.gov	Intel-Release		0	1	0	286	0	0	44	10 hours ago
oxygen.ornl.gov	Intel2015-CUDA-Release		0	1	0	0	0	1	23	21 hours ago
oxygen.ornl.gov	GCC-CUDA-Release		0	1	0	0	0	1	23	22 hours ago
oxygen.ornl.gov	Intel2015-Complex-Release		0	1	0	0	0	0	27	22 hours ago
oxygen.ornl.gov	Intel2016-Complex-Release		0	1	0	0	0	0	27	22 hours ago
oxygen.ornl.gov	GCC-Complex-Release		0	1	0	0	0	0	27	22 hours ago
oxygen.ornl.gov	Intel2015-Release		0	1	0	0	0	0	44	22 hours ago
oxygen.ornl.gov	Intel2016-Release		0	1	0	0	0	0	44	22 hours ago
oxygen.ornl.gov	GCC-Release		0	1	0	0	0	0	44	22 hours ago

Kitware CDash 2.2.2 © Kitware | Report problems | 0.021s

Figure 2.1: Example test results for QMCPACK, showing data for a workstation (Intel, GCC, both CPU and GPU builds) and for two ORNL supercomputers. In this example, 4 errors were found.

- On Titan, a Cray XK7 CPU+GPU machine:
  - The GCC programming environment and compiler with Cray MPI and CUDA
  - The GCC programming environment and compiler with Cray MPI

2.8 Building ppconvert, a pseudopotential format converter

QMCPACK includes a utility, ppconvert, to convert between different pseudopotential formats. Examples include effective core potential formats (in gaussians), the UPF format used by Quantum Espresso, and the XML format used by QMCPACK itself. The utility also enables the atomic orbitals to recomputed via a numerical density functional calculation if they need to be reconstructed for use in an electronic structure calculation.

To build ppconvert follow the instructions in src/QMCTools/ppconvert/README. Currently ppconvert is not built automatically although we expect to automate it soon. The makefile must be updated to refer to suitable C++ compiler and link in BLAS. Due to the small size of the calculations, optimal settings are not essential.

2.9 Installing and patching Quantum Espresso

For trial wavefunctions obtained in a plane-wave basis we mainly support Quantum Espresso. Note that ABINIT and QBox were supported historically and could be reactivated.

Quantum Espresso currently stores wavefunctions in a non-standard internal "save" format. To convert these to a conventional HDF5 format file we have developed a converter, pw2qmcpack. This is an add on to the Quantum Espresso distribution.

To simplify the process of patching Quantum Espresso we have developed a script that will automatically download and patch the source code. The patches are specific to each version. e.g. To download and patch QE v5.3.0:

```
cd external_codes/quantum_espresso
./download_and_patch_qe5.3.0.sh
```

After running the patch, you must configure Quantum Espresso with the HDF5 capability enabled, i.e.

```
cd espresso-5.3.0
./configure --with-hdf5 HDF5_DIR=/opt/local   # Specify HDF5 base directory
```

The complete process is described in external_codes/quantum_espresso/README.

2.10 How to build the fastest executable version of QMCPACK

To build the fastest version of QMCPACK we recommend the following:

- Use the latest C++ compilers available for your system. Substantial gains have been made optimizing C++ in recent years.
- Use a vendor optimized BLAS library such as Intel MKL and AMD ACML. Although QMC does not make extensive use of linear algebra, it is used in the VMC wavefunction optimizer and also to apply the orbital coefficients in local basis calculations.
- Use a vector math library such as Intel VML. For periodic calculations, the calculation of the structure factor and Ewald potential benefit from vectorized evaluation of sin and cos. Currently we only autodetect Intel VML, as provided with MKL, but support for MASSV and AMD LibM is included via #defines. See, e.g. src/Numerics/e2iphi.h. For large supercells, this optimization can gain 10% in performance.

Note that greater speedups of QMC calculations can usually be obtained by carefully choosing the required statistics for each investigation. i.e. Do not compute smaller error bars than necessary.

2.11 Troubleshooting the installation

Some tips to help troubleshoot installations of QMCPACK:

- First, build QMCPACK on a workstation that you control, or on any system that has a simple and up-to-date set of development tools. You can compare the results of cmake and QMCPACK on this system with any more difficult systems you encounter.
- Use up to date development software, particularly a recent CMake.
- Verify that the compilers and libraries that you expect are being configured. It is common to have multiple versions installed. The configure system will stop at the first version it finds which might not be the most recent. If this occurs, specify the appropriate directories and files directly (Section 2.5.3). e.g. cmake -DCMAKE_C_COMPILER=/full/path/to/mpicc -DCMAKE_CXX_COMPILER=/full/path/to/mpicxx ..
- To monitor the compiler and linker settings, use a verbose build, "make VERBOSE=1". If an individual source file fails to compile you can experiment by hand using the output of the verbose build to reconstruct the full compilation line.

If you still have problems please post to the QMCPACK Google group with full details, or contact a developer.

Chapter 3

Running QMCPACK

3.1 Command line options

3.2 Input files

3.3 Output files

scalar.dat
dmc.dat
stat.h5
config.h5

3.4 Running in parallel

3.4.1 MPI

QMCPACK is fully parallelized with MPI. When performing an ensemble job, all the MPI ranks are first equally divided into groups which perform individual QMC calculations. Within one calculation, all the walkers are fully distributed across all the MPI ranks in the group. Since MPI requires distributed memory, there must be at least one MPI per node. To maximize the efficiency, more facts should be taken into account. When using MPI+threads on compute nodes with more than one NUMA domain (e.g., AMD Interlagos CPU on Titan or a node with multiple CPU sockets), it is recommended to place as many MPI ranks as the number of NUMA domains if the memory is sufficient. On clusters with more than just one GPU per node (NVIDIA Tesla K80), it requires to use the same number of MPI ranks as the number of GPUs per node in order to let each MPI rank take one GPU.

3.4.2 Use of OpenMP threads

Modern processors integrate multiple identical cores even with hardware threads on a single die to increase the total performance and maintain a reasonable power draw. QMCPACK takes advantage of all that compute capability on a processor by using threads via OpenMP programming model as well as threaded linear algebra libraries. By default, QMCPACK is always built with OpenMP enabled. When launching calculations, users should instruct QMCPACK to create the right number of threads per MPI rank by specifying environmental variable OMP_NUM_THREADS. Even in the GPU accelerated version, using threads significantly reduces the time spent on the calculations performed on CPU.

Performance consideration

As walkers are the basic units of workload in QMC algorithms, they are loosely coupled and distributed across all the threads. For this reason, the best strategy to run QMCPACK efficiently is to feed enough walkers to the available threads.

In a VMC calculation, the code automatically raises the actual number of walkers per MPI rank to the number of available threads if the user-specified number of walkers is smaller, see "walkers/mpi=XXX" in the VMC output. In a DMC calculation, the target number of walkers should be chosen to be slightly smaller than a multiple of the total number of available threads across all the MPI ranks belongs to this calculation. Since the number of walkers varies from generation to generation, its dynamical value should be slightly smaller or equal to that multiple most of the time.

Memory consideration

When using threads, some memory objects shared by all the threads. Usually these memory are read-only when the walkers are evolving, for instance the ionic distance table and wavefunction coefficients. If a wavefunction is represented by B-splines, the whole table is shared by all the threads. It usually takes a large chunk of memory when a large primitive cell was used in the simulation. Its actual size is reported as "MEMORY increase XXX MB BsplineSetReader" in the output file. See details about how to reduce it in section 6.1.1.

The other memory objects which are distinct for each walker during random walk need to be associated with individual walkers and can not be shared. This part of memory grows linearly as the number of walkers per MPI rank. Those objects include wavefunction values (Slater determinants) at given electronic configurations and electron related distance tables (electron-electron distance table). Those matrices dominate the N^2 scaling of the memory usage per walker.

3.4.3 Running on GPU machines

The GPU version on the NVIDIA CUDA platform is fully incorporated into the main trunk. Currently some commonly used functionalities for solid-state and molecular systems using B-spline single-particle orbitals is supported. A detailed description of the GPU implementation can be found in Ref. [1].

Current GPU implementation assumes one MPI process per GPU. Vectorization is achieved over walkers, that is, all walkers are propagated in parallel. In each GPU kernel, loops over electrons, atomic cores or orbitals are further vectorized to exploit an additional level of parallelism and to allow coalesced memory access.

Supported GPU features

1. Quantum Monte Carlo methods:
    (a) Variational Monte Carlo (VMC).
    (b) Diffusion Monte Carlo (DMC).
    (c) Limited support for wavefunction optimization.
2. Boundary conditions:
    (a) Periodic and open boundary conditions are fully supported.
    (b) Twist-averaged boundary condition is supported for only real-valued wavefunctions.
    (c) Mixed boundary conditions and complex wavefunctions (e.g. fixed phase) are not yet supported.
3. Wavefunctions:
    (a) Single Slater determinants with 3D B-spline orbitals. Only real-valued wavefunctions is supported, but tiling complex orbitals to supercells is supported as long as each k-point is a multiple of half a G-vector of the supercell.
    (b) Mixed basis representation in which orbitals are represented as 1D splines times spherical harmonics in spherical regions (muffin tins) around atoms, and 3D B-splines in the interstitial region.
    (c) One-body and two-body Jastrows represented as 1D B-splines are supported. Note that only single-precision arithmetic is fully functional at the time of writing.
4. Interaction types:
    (a) Semilocal (nonlocal and local) pseudopotentials.

(b) Coulomb interaction (electron-electron, electron-ion).

(c) Model periodic Coulomb (MPC) interaction.

Compiling the GPU code

To build the executable `qmcpack` with GPU support, follow these steps:

1. Make sure NVIDIA's CUDA compiler, nvcc, is in the search path. In most cases, CMake should be able to locate the nvcc compiler on the system automatically.
2. (a) Run CMake with the argument `QMC_CUDA` switched on:
   ```
   cd build
   cmake -D QMC_CUDA=1 ..
   make
   ```
   or

   (b) If a CMake toolchain file is used, switch on `QMC_CUDA` by including this line in the toolchain file:
   ```
   SET (QMC_CUDA 1)
   ```
   Then compile the code as before:
   ```
   cd build
   cmake -D CMAKE_TOOLCHAIN_FILE=[toolchain name] ..
   make
   ```

CMake variables for adjusting CUDA code build features

These values can be changed by passing them as CMake's command line options with the `-D` flag, or using a toolchain file to overwrite the default values.

1. `QMC_CUDA`

`=0` (default):	no GPU support, build QMCPACK as a CPU code
`=1` :	build QMCPACK with GPU support

2. `CUDA_PRECISION`

`=float` (default):	single precision arithmetics and data types will be used for GPU kernels
`=double` :	double precision arithmetics and data types will be used for GPU kernels (Warning: not fully functional!)

Performance consideration

The relative speedup of the GPU implementation increases with both the number of electrons and the number of walkers running on a GPU. Typically, 128-256 walkers per GPU utilize sufficient number of threads to operate the GPU efficiently and to hide memory-access latency.

To achieve better performance, current implementation utilizes single precision operations on most GPU calculations, except for matrix inversions where double precision is required to retain high accuracy. The single precision GPU code is as accurate as the double precision CPU code up to a certain system size. Cross checking and verification of accuracy are encouraged for systems with more than approximately 1500 electrons.

Memory consideration

In the GPU implementation, each walker has an anonymous buffer on the GPU's global memory to store temporary data associated with the wavefunctions. Therefore, the amount of memory available on a GPU limits the number of walkers and eventually the system size that it can process.

If the GPU memory is exhausted, reduce the number of walkers per GPU. Coarsening the grids of the B-splines representation (by decreasing the value of meshfactor in the input file) can also lower the memory usage, at the expense (risk) of obtaining inaccurate results. Proceed with caution if this option has to be considered.

Chapter 4

Units used in QMCPACK

Internally, QMCPACK uses atomic units throughout. Unless stated, all inputs and outputs are also in atomic units. For convenience the analysis tools offer conversions to eV, Ry, Angstrom, Bohr etc.

Chapter 5

Specifying the system to be simulated

5.1 Specifying the simulation cell

5.2 Specifying the particle set

The `particleset` blocks specify the particles in the QMC simulations: their types, attributes (mass, charge, valence), and positions.

5.2.1 Input specification

`particleset` element				
parent elements:	`simulation`			
child elements:	`group, attrib`			
attribute :				
name	**datatype**	**values**	**default**	**description**
`name/id`	text	*any*	e	Name of particle set
`size`^{o}	integer	*any*	0	Number of particles in set
`random`^{o}	text	yes/no	no	Randomize starting positions
`randomsrc/` `random_source`^{o}	text	`particleset.name`	*none*	Particle set to randomize

`group` element				
parent elements:	`particleset`			
child elements:	`parameter, attrib`			
attribute :				
name	**datatype**	**values**	**default**	**description**
`name`	text	*any*	e	Name of particle set
`size`^{o}	integer	*any*	0	Number of particles in set
`mass`^{o}	real	*any*	1	Mass of particles in set
`unit`^{o}	text	au/amu	au	Units for mass of particles
parameters				
name	**datatype**	**values**	**default**	**description**
`charge`	real	*any*	0	Charge of particles in set
`valence`	real	*any*	0	Valence charge of particles in set
`atomicnumber`	integer	*any*	0	Atomic number of particles in set

`attrib` element				
parent elements:	`particleset,group`			
attribute :				
name	**datatype**	**values**	**default**	**description**
`name`	string	*any*	*none*	Name of attrib
`datatype`	string	intArray, realArray, posArray, stringArray	*none*	Type of data in attrib
`size`^{o}	string	*any*	*none*	Size of data in attrib

5.2.2 Detailed attribute description

particleset required attributes

- `name/id`
  Unique name for the particle set. Default is “e” for electrons. “i” or “ion0” is typically used for ions.

particleset optional attributes

- `size`
  Number of particles in set

- `random`
  Randomize starting positions of particles. Each component of each particle’s position is

randomized independently in the range of the simulation cell in that component's direction.

- `randomsrc/random_source`
  Specify source particle set around which to randomize the initial positions of this particle set.

name required attributes

- `name/id`
  Unique name for the particle set group. Typically, element symbols are used for ions and "u" or "d" for spin-up and spin-down electron groups, respectively.

group optional attributes

- `mass`
  Mass of particles in set.

- `unit`
  Units for mass of particles in set (au$[m_e = 1]$ or amu$[\frac{1}{12}m_{^{12}\mathrm{C}} = 1]$).

5.2.3 Example use cases

Listing 5.1: particleset elements for ions and electrons randomizing electron start positions.

```
<particleset name="i" size="2">
  <group name="Li">
    <parameter name="charge">3.000000</parameter>
    <parameter name="valence">3.000000</parameter>
    <parameter name="atomicnumber">3.000000</parameter>
  </group>
  <group name="H">
    <parameter name="charge">1.000000</parameter>
    <parameter name="valence">1.000000</parameter>
    <parameter name="atomicnumber">1.000000</parameter>
  </group>
  <attrib name="position" datatype="posArray" condition="1">
  0.0 0.0 0.0
  0.5 0.5 0.5
  </attrib>
  <attrib name="ionid" datatype="stringArray">
     Li H
  </attrib>
</particleset>
<particleset name="e" random="yes" randomsrc="i">
  <group name="u" size="2">
    <parameter name="charge">-1</parameter>
  </group>
  <group name="d" size="2">
    <parameter name="charge">-1</parameter>
  </group>
</particleset>
```

Listing 5.2: particleset elements for ions and electrons specifying electron start positions

```
<particleset name="e">
  <group name="u" size="4">
    <parameter name="charge">-1</parameter>
    <attrib name="position" datatype="posArray">
      2.9151687332e-01 -6.5123272502e-01 -1.2188463918e-01
      5.8423636048e-01 4.2730406357e-01 -4.5964306231e-03
      3.5228575807e-01 -3.5027014639e-01 5.2644808295e-01
     -5.1686250912e-01 -1.6648002292e+00 6.5837023441e-01
    </attrib>
  </group>
  <group name="d" size="4">
    <parameter name="charge">-1</parameter>
    <attrib name="position" datatype="posArray">
      3.1443445436e-01 6.5068682609e-01 -4.0983449009e-02
     -3.8686061749e-01 -9.3744432997e-02 -6.0456005388e-01
      2.4978241724e-02 -3.2862514649e-02 -7.2266047173e-01
     -4.0352404772e-01 1.1927734805e+00 5.5610824921e-01
    </attrib>
  </group>
</particleset>
<particleset name="ion0" size="3">
  <group name="O">
    <parameter name="charge">6</parameter>
    <parameter name="valence">4</parameter>
    <parameter name="atomicnumber">8</parameter>
  </group>
  <group name="H">
    <parameter name="charge">1</parameter>
    <parameter name="valence">1</parameter>
    <parameter name="atomicnumber">1</parameter>
  </group>
  <attrib name="position" datatype="posArray">
    0.0000000000e+00 0.0000000000e+00 0.0000000000e+00
    0.0000000000e+00 -1.4308249289e+00 1.1078707576e+00
    0.0000000000e+00 1.4308249289e+00 1.1078707576e+00
  </attrib>
  <attrib name="ionid" datatype="stringArray">
    O H H
  </attrib>
</particleset>
```

Listing 5.3: particleset elements for ions specifying positions by ion type

```
<particleset name="ion0">
  <group name="O" size="1">
    <parameter name="charge">6</parameter>
    <parameter name="valence">4</parameter>
    <parameter name="atomicnumber">8</parameter>
    <attrib name="position" datatype="posArray">
      0.0000000000e+00 0.0000000000e+00 0.0000000000e+00
    </attrib>
  </group>
  <group name="H" size="2">
    <parameter name="charge">1</parameter>
    <parameter name="valence">1</parameter>
    <parameter name="atomicnumber">1</parameter>
    <attrib name="position" datatype="posArray">
      0.0000000000e+00 -1.4308249289e+00 1.1078707576e+00
      0.0000000000e+00 1.4308249289e+00 1.1078707576e+00
    </attrib>
  </group>
</particleset>
```

Chapter 6

Trial wavefunction specification

6.1 Single-particle orbitals

6.1.1 Spline basis sets

6.1.2 Gaussian basis sets

6.1.3 Plane-wave basis sets

6.1.4 Homogeneous electron gas

The interacting Fermi Liquid has its own special determinantset for filling up a Fermi surface. The shell number can be specified seperately for both spin up and spin down. This determines how many electrons to include of each time, only closed shells are currently implemented. The shells are filled according to the rules of a square box, if other lattice vectors are used, the electrons may not fill up a complete shell.

This following example can also be used for Helium simulations too, by specifying the proper pair interaction in the Hamiltonian section.

Listing 6.1: 2D Fermi Liquid example: particle specification

```
<qmcsystem>
<simulationcell name="global">
<parameter name="rs" pol="0" condition="74">6.5</parameter>
<parameter name="bconds">p p p</parameter>
<parameter name="LR_dim_cutoff">15</parameter>
</simulationcell>
<particleset name="e" random="yes">
<group name="u" size="37">
<parameter name="charge">-1</parameter>
<parameter name="mass">1</parameter>
</group>
<group name="d" size="37">
<parameter name="charge">-1</parameter>
<parameter name="mass">1</parameter>
</group>
</particleset>
</qmcsystem>
```

Listing 6.2: 2D Fermi Liquid example (Slater Jastrow wave function)

```
<qmcsystem>
  <wavefunction name="psi0" target="e">
      <determinantset type="electron-gas" shell="7" shell2="7" randomize="true">
  </determinantset>
      <jastrow name="J2" type="Two-Body" function="Bspline" print="no">
        <correlation speciesA="u" speciesB="u" size="8" cusp="0">
          <coefficients id="uu" type="Array" optimize="yes">
        </correlation>
        <correlation speciesA="u" speciesB="d" size="8" cusp="0">
          <coefficients id="ud" type="Array" optimize="yes">
        </correlation>
      </jastrow>
```

6.2 Jastrow Factors

6.3 One-body Jastrow functions

The text below is rough draft and to be replaced with a complete and accurate description! In particular, the table columns and entries require consideration.

The one-body Jastrow is designed to... and is normally used in conjunction with an additional two-body term. Many different one-body are implemented...

The jastrow function is specified within a `wavefunction` element and must contain one of more `correlation` elements specifying additional parameters as well as the actual coefficients. Section 6.3.2 gives examples of the typical nesting of `jastrow`, `correlation`, and `coefficient` elements.

6.3.1 Input Specification

Jastrow element

name	**datatype**	**values**	**defaults**	**description**
name	text	?	?	Unique name for this Jastrow function
type	text	One-body	(required)	Define a one-body function
function	text	Bspline	(required)	BSpline Jastrow
		Pade		Pade form
		...		...
source	text	?	?	
print	text	?	?	

6.3.2 Spline form

The one-body spline Jastrow function is the most commonly used one-body Jastrow for solids. This form was first described and used in [**?**].

$$J1 = \sum_{I}^{ion0} \sum_{i}^{e} u_{ab}(|r_r - R_I|) \tag{6.1}$$

where u_{ab} is an interpolating spline between zero distance and r_{cut}. In 3D periodic systems the default cutoff distance is the Wigner Seitz cell radius. For other periodicities including isolated molecules the r_{cut} must be specified. The gradient at zero distance is... cusp can be set.... r_i and R_I are most commonly the electron and ion positions, but any particlesets that can provide the needed centers can be used.

Input Specification

Additional information:

Correlation element				
name	**datatype**	**values**	**defaults**	**description**
elementType	fillmein			
size				
rcut				
cusp				
print	text	?	(optional)	
elements				
	Coefficients			
Contents				
	(None)			

- `rcut`. The cutoff distance for the function in atomic units. For 3D fully periodic systems this parameter is optional and a default of the Wigner Seitz cell radius is used. Otherwise this parameter is required.

Coefficients element				
name	**datatype**	**values**	**defaults**	**description**
id	text			Unique identifier
type	text	Array		
elements				
(None)				
Contents				
(no name)	real array		zeros	Jastrow coefficients

Example use cases

Specify a spin-independent function with four parameters. Because rcut is not specified, the default cutoff of the Wigner Seitz cell radius is used; this Jastrow must be used with a 3D periodic system such as a bulk solid. The source of the ionic positions is set "i".

```
<jastrow name="J1" type="One-Body" function="Bspline" print="yes" source="i">
 <correlation elementType="C" cusp="0.0" size="4">
   <coefficients id="C" type="Array"> 0 0 0 0 </coefficients>
 </correlation>
</jastrow>
```

Specify a spin-dependent function with seven upspin and seven downspin parameters. The cutoff distance is set to 6 atomic units.

```
<jastrow name="J1" type="One-Body" function="Bspline" source="ion0" spin="yes">
  <correlation speciesA="C" speciesB="u" size="7" rcut="6">
    <coefficients id="eCu" type="Array">
    0.0 0.0 0.0 0.0 0.0 0.0 0.0
    </coefficients>
  </correlation>
  <correlation speciesA="C" speciesB="d" size="7" rcut="6">
    <coefficients id="eCd" type="Array">
    0.0 0.0 0.0 0.0 0.0 0.0 0.0
    </coefficients>
  </correlation>
</jastrow>
```

6.4 Multideterminant wavefunctions

6.5 Backflow wavefunctions

One can perturb the nodal surface of a single-slater/multi-slater wavefunction through use of a backflow-transformation. Specifically, if we have an antisymmetric function $D(\mathbf{x}_{0\uparrow}, \cdots, \mathbf{x}_{N\uparrow}, \mathbf{x}_{0\downarrow}, \cdots, \mathbf{x}_{N\downarrow})$, and if i_α is the i-th particle of species type α, then the backflow transformation works by making the coordinate transformation $\mathbf{x}_{i_\alpha} \to \mathbf{x}'_{i_\alpha}$, and evaluating D at these new "quasiparticle" coordinates. QMCPACK currently supports quasiparticle transformations given by:

$$\mathbf{x}'_{i_\alpha} = \mathbf{x}_{i_\alpha} + \sum_{\alpha \leq \beta} \sum_{i_\alpha \neq j_\beta} \eta^{\alpha\beta}(|\mathbf{x}_{i_\alpha} - \mathbf{x}_{j_\beta}|)(\mathbf{x}_{i_\alpha} - \mathbf{x}_{j_\beta}) \tag{6.2}$$

Here, $\eta^{\alpha\beta}(|\mathbf{x}_{i_\alpha} - \mathbf{x}_{j_\beta}|)$ is a radially symmetric back flow transformation between species α and β. In QMCPACK, particle i_α is known as the "target" particle and j_β is known as the "source". The main types of transformations we'll talk about are so called one-body terms, which are between an electron and an ion $\eta^{eI}(|\mathbf{x}_{i_e} - \mathbf{x}_{j_I}|)$, and two-body terms. Two body terms are distinguished as those between like and opposite spin electrons: $\eta^{e(\uparrow)e(\uparrow)}(|\mathbf{x}_{i_e(\uparrow)} - \mathbf{x}_{j_e(\uparrow)}|)$ and $\eta^{e(\uparrow)e(\downarrow)}(|\mathbf{x}_{i_e(\uparrow)} - \mathbf{x}_{j_e(\downarrow)}|)$. Henceforth, we will assume that $\eta^{e(\uparrow)e(\uparrow)} = \eta^{e(\downarrow)e(\downarrow)}$.

In the following, I will explain how to describe general terms like Eq. 6.2 in a QMCPACK XML file. For specificity, I will consider a particle set consisting of H and He (in that order). This ordering will be important when we build the XML file, so you can find this out either through your specific declaration of ¡particleset¿, by looking at the hdf5 file in the case of plane waves, or by looking at the qmcpack output file in the section labelled "Summary of QMC systems".

6.5.1 Input Specifications

All backflow declarations occur within a single `<backflow> ... </backflow>` block. Backflow transformations occur in `<transformation>` blocks, and have the following input parameters

Transformation element				
name	**datatype**	**values**	**defaults**	**description**
name	text		(required)	Unique name for this Jastrow function
type	text	"e-I"	(required)	Define a one-body backflow transformation.
		"e-e"		Define a two-body backflow transformation.
function	text	Bspline	(required)	B-spline type transformation. (No other types supported)
source	text			"e" if two-body, ion particle set if one-body.

Just like one and two-body jastrows, parameterization of the backflow transformations are specified within the `<transformation>` blocks by `<correlation>` blocks. Please refer to 6.3.2 for more information.

6.5.2 Example Use Case

Having specified the general form, we present a general example of one-body and two-body backflow transformations in a hydrogen-helium mixture. The H and He ions have independent backflow transformations, as do the like and unlike-spin two-body terms. One caveat is in order: ionic backflow transformations must be listed in the order that they appear in the particle set. If in our example, He is listed first, and H is listed second, the following example would be correct. However, switching backflow declaration to H first, then He, will result in an error. Outside of this, declaration of one-body blocks and two-body blocks aren't sensitive to ordering.

```
<backflow>
<!--The One-Body term with independent e-He and e-H terms. IN THAT ORDER -->
<transformation name="eIonB" type="e-I" function="Bspline" source="ion0">
    <correlation cusp="0.0" size="8" type="shortrange" init="no" elementType="He" rcut="3.0">
        <coefficients id="eHeC" type="Array" optimize="yes">
            0 0 0 0 0 0 0 0
        </coefficients>
    </correlation>
    <correlation cusp="0.0" size="8" type="shortrange" init="no" elementType="H" rcut="3.0">
        <coefficients id="eHC" type="Array" optimize="yes">
            0 0 0 0 0 0 0 0
        </coefficients>
    </correlation>
</transformation>

<!--The Two-Body Term with Like and Unlike Spins -->
<transformation name="eeB" type="e-e" function="Bspline" >
    <correlation cusp="0.0" size="7" type="shortrange" init="no" speciesA="u" speciesB="u" rcut="
        1.2">
        <coefficients id="uuB1" type="Array" optimize="yes">
            0 0 0 0 0 0 0
        </coefficients>
    </correlation>
    <correlation cusp="0.0" size="7" type="shortrange" init="no" speciesA="d" speciesB="u" rcut="
        1.2">
        <coefficients id="udB1" type="Array" optimize="yes">
            0 0 0 0 0 0 0
        </coefficients>
    </correlation>
</transformation>
</backflow>
```

Currently, backflow only works with single-slater determinant wavefunctions. When a backflow transformation has been declared, it should be placed within the `<determinantset>` block, but outside of the `<slaterdeterminant>` blocks, like so:

```
<determinantset ... >
   <!--basis set declarations go here, if there are any -->

   <backflow>
      <transformation ...>
        <!--Here's where discussed one and two-body terms are defined -->
       </transformation>
    </backflow>

    <slaterdeterminant>
       <!--Usual determinant definitions -->
    </slaterdeterminant>
 </determinantset>
```

6.5.3 Additional Information

- **Optimization**: Optimizable backflow transformation parameters are notoriously nonlinear, and so optimizing backflow wavefunctions can sometimes be difficult. We direct the reader to our provided backflow tutorials for more information.

Chapter 7

Hamiltonian and Observables

QMCPACK is capable of the simultaneous measurement of the Hamiltonian and many other quantum operators. The Hamiltonian attains a special status among the available operators (also referred to as observables) because it ultimately generates all available information regarding the quantum system. This is evident from an algorithmic standpoint as well since the Hamiltonian (embodied in the the projector) generates the imaginary time dynamics of the walkers in DMC and RMC.

This section covers how the Hamiltonian can be specified, component by component, by the user in the XML format native to QMCPACK . It also covers the input structure of statistical estimators corresponding to quantum observables such as the density, the static structure factor, and forces.

7.1 The Hamiltonian

The many-body Hamiltonian in Hartree units is given by

$$\hat{H} = -\sum_i \frac{1}{2m_i}\nabla_i^2 + \sum_i v^{ext}(r_i) + \sum_{i<j} v^{qq}(r_i, r_j) + \sum_{i\ell} v^{qc}(r_i, r_\ell) + \sum_{\ell<m} v^{cc}(r_\ell, r_m). \tag{7.1}$$

Here, the sums indexed by i/j are over quantum particles, while ℓ/m are reserved for classical particles. Often the quantum particles are electrons and the classical particles are ions, though QMCPACK is not limited in this way. The mass of each quantum particle is denoted m_i, $v^{qq}/v^{qc}/v^{cc}$ are pair potentials between quantum-quantum/quantum-classical/classical-classical particles, and v^{ext} denotes a purely external potential.

QMCPACK is designed modularly so that any potential can be supported with minimal additions to the code base. Potentials currently supported include Coulomb interactions in open and periodic boundary conditions, the modified periodic coulomb (MPC) potential, non-local pseudopotentials, helium pair potentials, and various model potentials such as hard sphere, gaussian, and modified Poschl-Teller.

Reference information and examples for the `<hamiltonian/>` XML element is provided below. Detailed descriptions of the input for individual potentials is given in the sections that follow.

`hamiltonian` element

parent elements: `simulation, qmcsystem`

child elements: `pairpot extpot estimator constant`(deprecated)

attributes

name	**datatype**	**values**	**default**	**description**
`name/id`^{o}	text	*anything*	h0	Unique id for this Hamiltonian instance
`type`^{o}	text		generic	*No current function*
`role`^{o}	text	primary/extra	extra	Designate as primary Hamiltonian or not
`source`^{o}	text	`particleset.name`	i	Identify classical particleset
`target`^{o}	text	`particleset.name`	e	Identify quantum particleset
`default`^{o}	boolean	yes/no	yes	Include kinetic energy term implicitly

Additional information:

- **target:** Must be set to the name of the quantum particeset. The default value is typically sufficient. In normal usage, no other attributes are provided.

Listing 7.1: All electron Hamiltonian XML element.

```
<hamiltonian target="e">
  <pairpot name="ElecElec" type="coulomb" source="e" target="e"/>
  <pairpot name="ElecIon" type="coulomb" source="i" target="e"/>
  <pairpot name="IonIon" type="coulomb" source="i" target="i"/>
</hamiltonian>
```

Listing 7.2: Pseudopotential Hamiltonian XML element.

```
<hamiltonian target="e">
  <pairpot name="ElecElec" type="coulomb" source="e" target="e"/>
  <pairpot name="PseudoPot" type="pseudo" source="i" wavefunction="psi0" format="xml">
    <pseudo elementType="Li" href="Li.xml"/>
    <pseudo elementType="H" href="H.xml"/>
  </pairpot>
  <pairpot name="IonIon" type="coulomb" source="i" target="i"/>
</hamiltonian>
```

7.2 Pair potentials

Many pair potentials are supported. Though only the most commonly used pair potentials are covered in detail in this section, all currently available potentials are listed briefly below. If a potential you desire is not covered below, or is not present at all, feel free to contact the developers.

`pairpot` factory element				
parent elements:	`hamiltonian`			
type selector:	`type` attribute			
type options:	coulomb		Coulomb/Ewald potential	
	pseudo		Semilocal pseudopotential	
	mpc		Modified Periodic Coulomb interaction/correction	
	cpp		Core polarization potential	
	numerical/*num*		Numerical radial potential	
	skpot		*Unknown*	
	vhxc		Exchange correlation potential (external)	
	jellium		Atom-centered spherical jellium potential	
	hardsphere		Hard sphere potential	
	gaussian		Gaussian potential	
	modpostel		Modified Poschl-Teller potential	
	huse		Huse quintic potential	
	modInsKE		Model insulator kinetic energy	
	oscillatory		*Unknown*	
	LJP_smoothed		Helium pair potential	
	HeSAPT_smoothed		Helium pair potential	
	HFDHE2_Moroni1995		Helium pair potential	
	HFDHE2		Helium pair potential	
	eHe		Helium-electron pair potential	
shared attributes:				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	*See above*	0	Select pairpot type
`name`^{r}	text	*anything*	any	Unique name for this pairpot
`source`^{r}	text	`particleset.name`	`hamiltonian.target`	Identify interacting particles
`target`^{r}	text	`particleset.name`	`hamiltonian.target`	Identify interacting particles
`units`^{o}	text		hartree	*No current function*

Additional information:

- **type:** Used to select the desired pair potential. Must be selected from the list of type options above.

- **name:** A unique name used to identify this pair potential. Block averaged output data will appear under this name in `scalar.dat` and/or `stat.h5` files.
- **source/target:** These specify the particles involved in a pair interaction. If an interaction is between classical (e.g. ions) and quantum (e.g. electrons), `source/target` should be the name of the classical/quantum particleset.
- Only `coulomb, pseudo, mpc` are described in detail below. The older or less used types (`cpp, numerical, jellium, hardsphere, gaussian, huse, modpostel, oscillatory, skpot, vhxc, modInsKE, LJP_smoothed, HeSAPT_smoothed, HFDHE2_Moroni1995, eHe, HFDHE2`) are not covered.
- Available only if `QMC_BUILD_LEVEL>2` and `QMC_CUDA` is not defined: `hardsphere, gaussian, huse, modpostel, oscillatory, skpot`.
- Available only if `OHMMS_DIM==3`: `mpc, vhxc, pseudo`.
- Available only if `OHMMS_DIM==3` and `QMC_BUILD_LEVEL>2` and `QMC_CUDA` is not defined: `cpp, LJP_smoothed, HeSAPT_smoothed, HFDHE2_Moroni1995, eHe, jellium, HFDHE2, modInsKE`.

7.2.1 Coulomb potentials

The bare Coulomb potential is used in open boundary conditions:

$$V_c^{open} = \sum_{i<j} \frac{q_i q_j}{|r_i - r_j|} \tag{7.2}$$

When periodic boundary conditions are selected, Ewald summation is used automatically:

$$V_c^{pbc} = \sum_{i<j} \frac{q_i q_j}{|r_i - r_j|} + \frac{1}{2}\sum_{L\neq 0}\sum_{i,j} \frac{q_i q_j}{|r_i - r_j + L|} \tag{7.3}$$

The sum indexed by L is over all non-zero simulation cell lattice vectors. In practice, the Ewald sum is broken into short and long ranged parts in a manner optimized for efficiency (see Ref. [2]) for details.

For information on how to set the boundary conditions, consult Sec. 5.1.

`pairpot type=coulomb` element				
parent elements:	`hamiltonian`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	**coulomb**		Must be coulomb
`name/id`^{r}	text	*anything*	ElecElec	Unique name for interaction
`source`^{r}	text	`particleset.name`	`hamiltonian.target`	Identify interacting particles
`target`^{r}	text	`particleset.name`	`hamiltonian.target`	Identify interacting particles
`pbc`^{o}	boolean	yes/no	yes	Use Ewald summation
`physical`^{o}	boolean	yes/no	yes	Hamiltonian(yes)/observable(no)
`forces`	boolean	yes/no	no	*Deprecated*

Additional information

- **type/source/target** See description for the generic `pairpot` factory element above.
- **name:** Traditional user-specified names for electron-electron, electron-ion, and ion-ion terms are `ElecElec`, `ElecIon`, and `IonIon`, respectively. While any choice can be used, the data analysis tools expect to find columns in `*.scalar.dat` with these names.
- **pbc**: Ewald summation will not be performed if `simulationcell.bconds== n n n`, regardless of the value of `pbc`. Similarly, the `pbc` attribute can only be used to turn off Ewald summation if `simulationcell.bconds!= n n n`. The default value is recommended.
- **physical**: If `physical==yes`, this pair potential is included in the Hamiltonian and will factor into the `LocalEnergy` reported by QMCPACK and also in the DMC branching weight. If `physical==no`, then the pair potential is treated as a passive observable but not as part of the Hamiltonian itself. As such it does not contribute to the outputted `LocalEnergy`. Regardless of the value of `physical` output data will appear in `scalar.dat` in a column headed by `name`.

Listing 7.3: XML element for Coulomb interaction between electrons.

```
<pairpot name="ElecElec" type="coulomb" source="e" target="e"/>
```

Listing 7.4: XML element for Coulomb interaction between electrons and ions (all-electron only).

```
<pairpot name="ElecIon" type="coulomb" source="i" target="e"/>
```

Listing 7.5: XML element for Coulomb interaction between ions.

```
<pairpot name="IonIon" type="coulomb" source="i" target="i"/>
```

7.2.2 Pseudopotentials

QMCPACK supports pseudopotentials in semilocal form, which is local in the radial coordinate and non-local in angular coordinates. When all angular momentum channels above a certain threshold (ℓ_{max}) are well approximated by the same potential ($V_{\bar{\ell}} \equiv V_{loc}$), the pseudpotential separates into a fully local channel and an angularly-nonlocal component:

$$V^{PP} = \sum_{ij} \Big(V_{\bar{\ell}}(|r_i - \tilde{r}_j|) + \sum_{\ell \neq \bar{\ell}}^{\ell_{max}} \sum_{m=-\ell}^{\ell} |Y_{\ell m}\rangle \big[V_{\ell}(|r_i - \tilde{r}_j|) - V_{\bar{\ell}}(|r_i - \tilde{r}_j|) \big] \langle Y_{\ell m}| \Big) \tag{7.4}$$

Here the electron/ion index is i/j and only one type of ion is shown for simplicity.

Evaluation of the localized pseudopotential energy $\Psi_T^{-1} V^{PP} \Psi_T$ requires additional angular integrals. These integrals are evaluated on a randomly shifted angular grid. The size of this grid is determined by ℓ_{max}. See Ref. [3] for further detail.

QMCPACK uses the FSAtom pseudopotential file format associated with the "Free Software Project for Atomic-scale Simulations" initiated in 2002 (see `http://www.tddft.org/fsatom/manifest.php` for general information). The FSAtom format uses XML for structured data. Files in this format do not use a specific identifying file extension; they are simply suffixed with "`.xml`". The tabular data format of CASINO is also supported.

`pairpot type=pseudo` element				
parent elements:	`hamiltonian`			
child elements:	`pseudo`			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	**pseudo**		Must be pseudo
`name/id`^{r}	text	*anything*	PseudoPot	*No current function*
`source`^{r}	text	`particleset.name`	i	Ion particleset name
`target`^{r}	text	`particleset.name`	`hamiltonian.target`	Electron particleset name
`pbc`^{o}	boolean	yes/no	yes*	Use Ewald summation
`forces`	boolean	yes/no	no	*Deprecated*
`wavefunction`^{r}	text	`wavefunction.name`	invalid	Identify wavefunction
`format`^{r}	text	xml/table	table	Select file format

Additional information:

- **type/source/target** See description for the generic `pairpot` factory element above.
- **name:** Ignored. Instead default names will be present in `*scalar.dat` output files when pseudopotentials are used. The field `LocalECP` refers to the local part of the pseudopotential.

If non-local channels are present, a `NonLocalECP` field will be added that contains the non-local energy summed over all angular momentum channels.

- **pbc:** Ewald summation will not be performed if `simulationcell.bconds== n n n`, regardless of the value of `pbc`. Similarly, the `pbc` attribute can only be used to turn off Ewald summation if `simulationcell.bconds!= n n n`.
- **format:** If `format`==table, QMCPACK looks for `*.psf` files containing pseudopotential data in a tabular format. The files must be named after the ionic species provided in `particleset` (*e.g.* `Li.psf` and `H.psf`). If `format`==xml, additional `pseudo` child XML elements must be provided (see below). These elements specify individual file names and formats (both the FSAtom XML and CASINO tabular data formats are supported).

Listing 7.6: XML element for pseudopotential electron-ion interaction (psf files).

```
<pairpot name="PseudoPot" type="pseudo" source="i" wavefunction="psi0" format="psf"/>
```

Listing 7.7: XML element for pseudopotential electron-ion interaction (xml files).

```
<pairpot name="PseudoPot" type="pseudo" source="i" wavefunction="psi0" format="xml">
  <pseudo elementType="Li" href="Li.xml"/>
  <pseudo elementType="H" href="H.xml"/>
</pairpot>
```

Details of `<pseudo/>` input elements are given below. It is possible to include (or construct) a full pseudopotential directly in the input file without providing an external file via `href`. The full XML format for pseudopotentials is not yet covered.

`pseudo` element				
parent elements:	`pairpot type=pseudo`			
child elements:	`header local grid`			
attributes				
name	**datatype**	**values**	**default**	**description**
`elementType/symbol`^{r}	text	`group.name`	none	Identify ionic species
`href`^{r}	text	*filepath*	none	Pseudopotential file path
`format`^{r}	text	xml/casino	xml	Specify file format
`cutoff`^{o}	real			Non-local cutoff radius
`lmax`^{o}	integer			Largest angular momentum
`nrule`^{o}	integer			Integration grid order

Listing 7.8: XML element for pseudopotential of single ionic species.

```
<pseudo elementType="Li" href="Li.xml"/>
```

7.2.3 Modified periodic Coulomb interaction/correction

The modified periodic Coulomb (MPC) interaction is an alternative to direct Ewald summation. The MPC corrects the exchange correlation hole to more closely match its thermodynamic limit. Because of this, the MPC exhibits smaller finite size errors than the bare Ewald interaction, though a few alternative and competitive finite size correction schemes now exist. The MPC is itself often used just as a finite size correction in postprocessing (set `physical=false` in the input).

`pairpot type=mpc` element				
parent elements:	`hamiltonian`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	**mpc**		Must be mpc
`name/id`^{r}	text	*anything*	MPC	Unique name for interaction
`source`^{r}	text	`particleset.name`	`hamiltonian.target`	Identify interacting particles
`target`^{r}	text	`particleset.name`	`hamiltonian.target`	Identify interacting particles
`physical`^{o}	boolean	yes/no	no	Hamiltonian(yes)/observable(no)
`cutoff`	real	> 0	30.0	Kinetic energy cutoff

Remarks

- `physical`: Typically set to `no`, meaning the standard Ewald interaction will be used during sampling and MPC will be measured as an observable for finite-size post correction. If `physical` is `yes`, the MPC interaction will be used during sampling. In this case an electron-electron Coulomb `pairpot` element should not be supplied.
- Developer note: Currently the `name` attribute for the mpc interaction is ignored. The name is always reset to `MPC`.

Listing 7.9: Modified periodic coulomb for finite size post-correction.

```
<pairpot type="MPC" name="MPC" source="e" target="e" ecut="60.0" physical="no"/>
```

7.3 General estimators

A broad range of estimators for physical observables are available in QMCPACK . The sections below contain input details for the total number density (`density`), number density resolved by particle spin (`spindensity`), spherically averaged pair correlation function (`gofr`), static structure factor (`sk`), energy density (`energydensity`), one body reduced density matrix (`dm1b`), $S(k)$

based kinetic energy correction (`chiesa`), forward walking (`ForwardWalking`), and force (`Force`) estimators. Other estimators are not yet covered.

When an `<estimator/>` element appears in `<hamiltonian/>`, it is evaluated for all applicable chained QMC runs (*e.g.* VMC$\rightarrow$DMC$\rightarrow$DMC). Estimators are generally not accumulated during wavefunction optimization sections. If an `<estimator/>` element is instead provided in a particular `<qmc/>` element, that estimator is only evaluated for that specific section (*e.g.* during VMC only).

`estimator` factory element				
parent elements:	`hamiltonian, qmc`			
type selector:	`type` attribute			
type options:	density		Density on a grid	
	spindensity		Spin density on a grid	
	gofr		Pair correlation function (quantum species)	
	sk		Static structure factor	
	structurefactor		Species resolved structure factor	
	momentum		Momentum distribution	
	energydensity		Energy density on uniform or Voronoi grid	
	dm1b		One body density matrix in arbitrary basis	
	chiesa		Chiesa-Ceperley-Martin-Holzmann kinetic energy correction	
	Force		Family of “force” estimators (see 7.5)	
	ForwardWalking		Forward walking values for existing estimators	
	orbitalimages		Create image files for orbitals, then exit	
	flux		Checks sampling of kinetic energy	
	localmoment		Atomic spin polarization within cutoff radius	
	numberfluctuations		Spatial number fluctuations	
	HFDHE2		Helium pressure	
	NearestNeighbors		Trace nearest neighbor indices	
	Kinetic		*No current function*	
	Pressure		*No current function*	
	ZeroVarObs		*No current function*	
	DMCCorrection		*No current function*	
shared attributes:				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	*See above*	0	Select estimator type
`name`^{r}	text	*anything*	any	Unique name for this estimator

7.3.1 Chiesa-Ceperley-Martin-Holzmann kinetic energy correction

This estimator calculates a finite size correction to the kinetic energy following the formalism laid out in Ref. [4]. The total energy can be corrected for finite size effects by using this estimator in conjuction with the MPC correction.

`estimator type=chiesa` element				
parent elements:	`hamiltonian, qmc`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	**chiesa**		Must be chiesa
`name`^{o}	text	*anything*	KEcorr	Always reset to KEcorr
`source`^{o}	text	`particleset.name`	e	Identify quantum particles
`psi`^{o}	text	`wavefunction.name`	psi0	Identify wavefunction

Listing 7.10: “Chiesa” kinetic energy finite size post-correction.

```
<estimator name="KEcorr" type="chiesa" source="e" psi="psi0"/>
```

7.3.2 Density estimator

The particle number density operator is given by

$$\hat{n}_r = \sum_i \delta(r - r_i) \tag{7.5}$$

The `density` estimator accumulates the number density on a uniform histogram grid over the simulation cell. The value obtained for a grid cell c with volume Ω_c is then the average number of particles in that cell:

$$n_c = \int dR|\Psi|^2 \int_{\Omega_c} dr \sum_i \delta(r - r_i) \tag{7.6}$$

`estimator type=density` element

parent elements:	`hamiltonian, qmc`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	**density**		Must be density
`name`^{r}	text	*anything*	any	Unique name for estimator
`delta`^{o}	real array(3)	$0 \leq v_i \leq 1$	0.1 0.1 0.1	Grid cell spacing, unit coords
`x_min`^{o}	real	> 0	0	Grid starting point in x (Bohr)
`x_max`^{o}	real	> 0	|`lattice[0]`|	Grid ending point in x (Bohr)
`y_min`^{o}	real	> 0	0	Grid starting point in y (Bohr)
`y_max`^{o}	real	> 0	|`lattice[1]`|	Grid ending point in y (Bohr)
`z_min`^{o}	real	> 0	0	Grid starting point in z (Bohr)
`z_max`^{o}	real	> 0	|`lattice[2]`|	Grid ending point in z (Bohr)
`potential`^{o}	boolean	yes/no	no	Accumulate local potential, *Deprecated*
`debug`^{o}	boolean	yes/no	no	*No current function*

Additional information:

- `name`: The name provided will be used as a label in the `stat.h5` file for the blocked output data. Post-processing tools expect `name="Density"`.
- `delta`: This sets the histogram grid size used to accumulate the density: `delta="0.1 0.1 0.05"`$\rightarrow$ $10 \times 10 \times 20$ grid, `delta="0.01 0.01 0.01"`$\rightarrow$ $100 \times 100 \times 100$ grid. The density grid is written to a `stat.h5` file at the end of each Monte Carlo block. If you request many *blocks* in a `<qmc/>` element, or select a large grid, the resulting `stat.h5` file may be many GB in size.
- `*_min/*_max`: Can be used to select a subset of the simulation cell for the density histogram grid. For example if a (cubic) simulation cell is 20 Bohr on a side, setting `*_min=5.0` and `*_max=15.0` will result in a density histogram grid spanning a $10 \times 10 \times 10$ Bohr cube about the center of the box. Use of `x_min, x_max, y_min, y_max, z_min, z_max` is only appropriate for orthorhombic simulation cells with open boundary conditions.
- When open boundary conditions are used, a `<simulationcell/>` element must be explicitly provided as the first sub-element of `<qmcsystem/>` for the density estimator to work. In this case the molecule should be centered around the middle of the simulation cell ($L/2$) and not the origin (0 since the space within the cell, and hence the density grid, is defined from 0 to L.

Listing 7.11: Density estimator (uniform grid).

```
<estimator name="Density" type="density" delta="0.05 0.05 0.05"/>
```

7.3.3 Spin density estimator

The spin density is similar to the total density described above. In this case, the sum over particles is performed independently for each spin component.

`estimator type=spindensity` element				
parent elements:	`hamiltonian, qmc`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	**spindensity**		Must be spindensity
`name`^{r}	text	*anything*	any	Unique name for estimator
`report`^{o}	boolean	yes/no	no	Write setup details to stdout
parameters				
name	**datatype**	**values**	**default**	**description**
`grid`^{o}	integer array(3)	$v_i > 0$		Grid cell count
`dr`^{o}	real array(3)	$v_i > 0$		Grid cell spacing (Bohr)
`cell`^{o}	real array(3,3)	*anything*		Volume grid exists in
`corner`^{o}	real array(3)	*anything*		Volume corner location
`center`^{o}	real array(3)	*anything*		Volume center/origin location
`voronoi`^{o}	text	`particleset.name`		*Under development*
`test_moves`^{o}	integer	>= 0	0	Test estimator with random moves

Additional information:

- `name`: The name provided will be used as a label in the `stat.h5` file for the blocked output data. Post-processing tools expect `name="SpinDensity"`.
- `grid`: Sets the dimension of the histogram grid. Input like `<parameter name="grid"> 40 40 40 </parameter>` requests a $40 \times 40 \times 40$ grid. The shape of individual grid cells is commensurate with the supercell shape.
- `dr`: Real space dimensions of grid cell edges (Bohr units). Input like `<parameter name="dr"> 0.5 0.5 0.5 </parameter>` in a supercell with axes of length 10 Bohr each (but of arbitrary shape) will produce a $20 \times 20 \times 20$ grid. The inputted `dr` values are rounded to produce an integer number of grid cells along each supercell axis. Either `grid` or `dr` must be provided, but not both.

- `cell`: When `cell` is provided, a user defined grid volume is used instead of the global supercell. This must be provided if open boundary conditions are used. Additionally, if `cell` is provided, the user must specify where the volume is located in space in addition to its size/shape (`cell`) using either the `corner` or `center` parameters.

- `corner`:The grid volume is defined as $corner + \sum_{d=1}^{3} u_d cell_d$ with $0 < u_d < 1$ ("cell" refers to either the supercell or user provided cell).

- `center`:The grid volume is defined as $center + \sum_{d=1}^{3} u_d cell_d$ with $-1/2 < u_d < 1/2$ ("cell" refers to either the supercell or user provided cell). `corner/center` can be used to shift the grid even if `cell` is not specified. Simultaneous use of `corner` and `center` will cause QMCPACK to abort.

Listing 7.12: Spin density estimator (uniform grid).

```
<estimator type="spindensity" name="SpinDensity" report="yes">
  <parameter name="grid"> 40 40 40 </parameter>
</estimator>
```

Listing 7.13: Spin density estimator (uniform grid centered about origin).

```
<estimator type="spindensity" name="SpinDensity" report="yes">
  <parameter name="grid">
    20 20 20
  </parameter>
  <parameter name="center">
    0.0 0.0 0.0
  </parameter>
  <parameter name="cell">
    10.0 0.0 0.0
     0.0 10.0 0.0
     0.0 0.0 10.0
  </parameter>
</estimator>
```

7.3.4 Pair correlation function, $g(r)$

The functional form of the species resolved radial pair correlation function operator is

$$g_{ss'}(r) = \frac{V}{4\pi r^2 N_s N_{s'}} \sum_{i_s=1}^{N_s} \sum_{j_{s'}=1}^{N_{s'}} \delta(r - |r_{i_s} - r_{j_{s'}}|). \tag{7.7}$$

Here N_s is the number of particles of species s and V is the supercell volume. If $s = s'$, then the sum is restricted so that $i_s \neq j_s$.

In QMCPACK, an estimate of $g_{ss'}(r)$ is obtained as a radial histogram with a set of N_b uniform bins of width δr. This can be expressed analytically as

$$\tilde{g}_{ss'}(r) = \frac{V}{4\pi r^2 N_s N_{s'}} \sum_{i=1}^{N_s} \sum_{j=1}^{N_{s'}} \frac{1}{\delta r} \int_{r-\delta r/2}^{r+\delta r/2} dr' \delta(r' - |r_{si} - r_{s'j}|), \tag{7.8}$$

where the radial coordinate r is restricted to reside at the bin centers, $\delta r/2, 3\delta r/2, 5\delta r/2, \ldots$.

`estimator type=gofr` element

parent elements:	`hamiltonian, qmc`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	**gofr**		Must be gofr
`name`^{o}	text	*anything*	any	*No current function*
`num_bin`^{r}	integer	> 1	20	# of histogram bins
`rmax`^{o}	real	> 0	10	Histogram extent (Bohr)
`dr`^{o}	real	> 0	0.5	*No current function*
`debug`^{o}	boolean	yes/no	no	*No current function*
`target`^{o}	text	`particleset.name`	`hamiltonian.target`	Quantum particles
`source/sources`^{o}	text array	`particleset.name`	`hamiltonian.target`	Classical particles

Additional information:

- **num_bin:** The number of bins in each species pair radial histogram.
- **rmax:** Maximum pair distance included in the histogram. The uniform bin width is $\delta r =$ `rmax/num_bin`. If periodic boundary conditions are used for any dimension of the simulation cell, then the default value of `rmax` is the simulation cell radius instead of 10 Bohr. For open boundary conditions the volume (V) used is 1.0 Bohr^3.
- **source/sources:** If unspecified, only pair correlations between each species of quantum particle will be measured. For each classical particleset specified by `source/sources`, additional pair correlations between each quantum and classical species will be measured. Typically there is only one classical particleset (*e.g.* `source="ion0"`), but there can be several in principle (*e.g.* `sources="ion0 ion1 ion2"`).
- **target:** The default value is the preferred usage (*i.e.* `target` does not need to be provided).
- Data is outputted to the `stat.h5` for each QMC sub-run. Individual histograms are named according to the quantum particleset and index of the pair. For example, if the quantum

particleset is named ”e” and there are two species (up and down electrons, say), then there will be three sets of histogram data in each `stat.h5` file named `gofr_e_0_0`, `gofr_e_0_1`, and `gofr_e_1_1` for up-up, up-down, and down-down correlations, respectively.

Listing 7.14: Pair correlation function estimator element.

```
<estimator type="gofr" name="gofr" num_bin="200" rmax="3.0" />
```

Listing 7.15: Pair correlation function estimator element with additional electron-ion correlations.

```
<estimator type="gofr" name="gofr" num_bin="200" rmax="3.0" source="ion0" />
```

7.3.5 Static structure factor, $S(k)$

Let $\rho^e_{\mathbf{k}} = \sum_j e^{i\mathbf{k}\cdot\mathbf{r}^e_j}$ be the Fourier space electron density, with $\mathbf{r}^e_j$ being the coordinate of the j-th electron. $\mathbf{k}$ is a wavevector commensurate with the simulation cell. QMCPACK allows the user to accumulate the static electron structure factor $S(\mathbf{k})$ at all commensurate $\mathbf{k}$ such that $|\mathbf{k}| \leq (LR\_DIM\_CUTOFF)r_c$. N^e is the number of electrons, `LR_DIM_CUTOFF` is the optimized breakup parameter, and r_c is the Wigner-Seitz radius. It is defined as follows:

$$S(\mathbf{k}) = \frac{1}{N^e}\langle\rho^e_{-\mathbf{k}}\rho^e_{\mathbf{k}}\rangle \tag{7.9}$$

`estimator type=sk` element				
parent elements:	`hamiltonian, qmc`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	sk		Must be sk
`name`^{r}	text	*anything*	any	Unique name for estimator
`hdf5`^{o}	boolean	yes/no	no	Output to `stat.h5` (yes) or `scalar.dat` (no)

Additional information:

- **name:** Unique name for estimator instance. A data structure of the same name will appear in `stat.h5` output files.
- **hdf5:** If `hdf5==yes` output data for $S(k)$ is directed to the `stat.h5` file (recommended usage). If `hdf5==no`, the data is instead routed to the `scalar.dat` file resulting in many columns of data with headings prefixed by `name` and postfixed by the k-point index (*e.g.* `sk_0 sk_1 ...sk_1037 ...`).

- This estimator only works in periodic boundary conditions. Its presence in the input file is ignored otherwise.
- This is not a species resolved structure factor. Additionally, for $\mathbf{k}$ vectors commensurate with the unit cell, $S(\mathbf{k})$ will include contributions from the static electronic density, thus meaning it won't accurately measure the electron-electron density response.

Listing 7.16: Static structure factor estimator element.

```
<estimator type="sk" name="sk" hdf5="yes"/>
```

7.3.6 Energy density estimator

An energy density operator, $\hat{\mathcal{E}}_r$, satisfies

$$\int dr\hat{\mathcal{E}}_r = \hat{H}, \tag{7.10}$$

where the integral is over all space and $\hat{H}$ is the Hamiltonian. In QMCPACK , the energy density is split into kinetic and potential components

$$\hat{\mathcal{E}}_r = \hat{\mathcal{T}}_r + \hat{\mathcal{V}}_r \tag{7.11}$$

with each component given by

$$\begin{aligned}
\hat{\mathcal{T}}_r &= \frac{1}{2}\sum_i \delta(r-r_i)\hat{p}_i^2 \qquad (7.12)\\
\hat{\mathcal{V}}_r &= \sum_{i<j}\frac{\delta(r-r_i)+\delta(r-r_j)}{2}\hat{v}^{ee}(r_i,r_j) + \sum_{i\ell}\frac{\delta(r-r_i)+\delta(r-\tilde{r}_\ell)}{2}\hat{v}^{eI}(r_i,\tilde{r}_\ell)\\
&\quad + \sum_{\ell<m}\frac{\delta(r-\tilde{r}_\ell)+\delta(r-\tilde{r}_m)}{2}\hat{v}^{II}(\tilde{r}_\ell,\tilde{r}_m).
\end{aligned}$$

Here r_i and $\tilde{r}_\ell$ represent electon and ion positions, respectively, $\hat{p}_i$ is a single electron momentum operator, and $\hat{v}^{ee}(r_i,r_j)$, $\hat{v}^{eI}(r_i,\tilde{r}_\ell)$, $\hat{v}^{II}(\tilde{r}_\ell,\tilde{r}_m)$ are the electron-electron, electron-ion, and ion-ion pair potential operators (including non-local pseudopotentials, if present). This form of the energy density is size consistent, *i.e.* the partially integrated energy density operators of well separated atoms gives the isolated Hamiltonians of the respective atoms. For periodic systems with twist averaged boundary conditions, the energy density is formally correct only for either a set of supercell k-points that correspond to real valued wavefunctions, or a k-point set that has inversion symmetry around a k-point having a real valued wavefunction. For more information about the energy density, see Ref. [5].

In QMCPACK , the energy density can be accumulated on piecewise uniform three dimensional grids in generalized cartesian, cylindrical, or spherical coordinates. The energy density integrated within Voronoi volumes centered on ion positions is also available. The total particle number density is also accumulated on the same grids by the energy density estimator for convenience so that related quantities, such as the regional energy per particle, can be computed easily.

`estimator type=EnergyDensity` element				
parent elements:	`hamiltonian, qmc`			
child elements:	`reference_points, spacegrid`			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	**EnergyDensity**		Must be EnergyDensity
`name`^{r}	text	*anything*		Unique name for estimator
`dynamic`^{r}	text	`particleset.name`		Identify electrons
`static`^{o}	text	`particleset.name`		Identify ions

Additional information:

- `name`: Must be unique. A dataset with blocked statistical data for the energy density will appear in the `stat.h5` files labeled as `name`.

Listing 7.17: Energy density estimator accumulated on a 20x10x10 grid over the simulation cell.

```
<estimator type="EnergyDensity" name="EDcell" dynamic="e" static="ion0">
  <spacegrid coord="cartesian">
    <origin p1="zero"/>
    <axis p1="a1" scale=".5" label="x" grid="-1 (.05) 1"/>
    <axis p1="a2" scale=".5" label="y" grid="-1 (.1) 1"/>
    <axis p1="a3" scale=".5" label="z" grid="-1 (.1) 1"/>
  </spacegrid>
</estimator>
```

Listing 7.18: Energy density estimator accumulated within spheres of radius 6.9 Bohr centered on the first and second atoms in the ion0 particleset.

```
<estimator type="EnergyDensity" name="EDatom" dynamic="e" static="ion0">
  <reference_points coord="cartesian">
    r1 1 0 0
    r2 0 1 0
    r3 0 0 1
  </reference_points>
  <spacegrid coord="spherical">
    <origin p1="ion01"/>
    <axis p1="r1" scale="6.9" label="r" grid="0 1"/>
    <axis p1="r2" scale="6.9" label="phi" grid="0 1"/>
    <axis p1="r3" scale="6.9" label="theta" grid="0 1"/>
  </spacegrid>
  <spacegrid coord="spherical">
    <origin p1="ion02"/>
    <axis p1="r1" scale="6.9" label="r" grid="0 1"/>
    <axis p1="r2" scale="6.9" label="phi" grid="0 1"/>
    <axis p1="r3" scale="6.9" label="theta" grid="0 1"/>
  </spacegrid>
</estimator>
```

Listing 7.19: Energy density estimator accumulated within Voronoi polyhedra centered on the ions.

```
<estimator type="EnergyDensity" name="EDvoronoi" dynamic="e" static="ion0">
  <spacegrid coord="voronoi"/>
</estimator>
```

The `<reference_points/>` element provides a set of points for later use in specifying the origin and coordinate axes needed to construct a spatial histogramming grid. Several reference points on the surface of the simulation cell (see Table 7.1) as well as the positions of the ions (see the `energydensity.static` attribute) are made available by default. The reference points can be used, for example, to construct a cylindrical grid along a bond with the origin on the bond center.

`reference_points` element				
parent elements:	`estimator type=EnergyDensity`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`coord`^{r}	text	cartesian/cell		Specify coordinate system
body text				
	The body text is a line formatted list of points with labels			

Additional information

- **coord:** If `coord=cartesian`, labeled points are in cartesian (x,y,z) format in units of Bohr. If `coord=cell`, then labeled points are in units of the simulation cell axes.
- **body text:** The list of points provided in the body text are line formatted, with four entries per line (*label coor1 coor2 coor3*). A set of points referenced to the simulation cell are available by default (see table 7.1). If `energydensity.static` is provided, the location of each individual ion is also available (*e.g.* if `energydensity.static=ion0`, then the location of the first atom is available with label ion01, the second with ion02, etc.). All points can be used by label when constructing spatial histogramming grids (see the `spacegrid` element below) used to collect energy densities.

`label`	`point`	`description`
`zero`	0 0 0	Cell center
`a1`	a_1	Cell axis 1
`a2`	a_2	Cell axis 2
`a3`	a_3	Cell axis 3
`f1p`	$a_1/2$	Cell face 1+
`f1m`	-$a_1/2$	Cell face 1-
`f2p`	$a_2/2$	Cell face 2+
`f2m`	-$a_2/2$	Cell face 2-
`f3p`	$a_3/2$	Cell face 3+
`f3m`	-$a_3/2$	Cell face 3-
`cppp`	$(a_1 + a_2 + a_3)/2$	Cell corner +,+,+
`cppm`	$(a_1 + a_2 - a_3)/2$	Cell corner +,+,-
`cpmp`	$(a_1 - a_2 + a_3)/2$	Cell corner +,-,+
`cmpp`	$(-a_1 + a_2 + a_3)/2$	Cell corner -,+,+
`cpmm`	$(a_1 - a_2 - a_3)/2$	Cell corner +,-,-
`cmpm`	$(-a_1 + a_2 - a_3)/2$	Cell corner -,+,-
`cmmp`	$(-a_1 - a_2 + a_3)/2$	Cell corner -,-,+
`cmmm`	$(-a_1 - a_2 - a_3)/2$	Cell corner -,-,-

Table 7.1: Reference points available by default. The vectors a_1, a_2, and a_3 refer to the simulation cell axes. The representation of the cell is centered around `zero`.

The `<spacegrid/>` element is used to specify a spatial histogramming grid for the energy density. Grids are constructed based on a set of, potentially non-orthogonal, user provided coordinate axes. The axes are based on information available from `reference_points`. Voronoi grids are based only on nearest neighbor distances between electrons and ions. Any number of space grids can be provided to a single energy density estimator.

`spacegrid` element				
parent elements:	`estimator type=EnergyDensity`			
child elements:	`origin, axis`			
attributes				
name	**datatype**	**values**	**default**	**description**
`coord`^{r}	text	cartesian		Specify coordinate system
		cylindrical		
		spherical		
		voronoi		

The `<origin/>` element gives the location of the origin for a non-Voronoi grid.

`origin` element				
parent elements:	`spacegrid`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`p1`^{r}	text	`reference_point.label`		Select end point
`p2`^{o}	text	`reference_point.label`		Select end point
`fraction`^{o}	real		0	Interpolation fraction

Additional information:

- **p1/p2/fraction:** The location of the origin is set to `p1+fraction*(p2-p1)`. If only `p1` is provided, the origin is at `p1`.

The `<axis/>` element represents a coordinate axis used to construct the, possibly curved, coordinate system for the histogramming grid. Three `<axis/>` elements must be provided to a non-Voronoi `<spacegrid/>` element.

`axis` element				
parent elements:	`spacegrid`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`label`^{r}	text	*See below*		Axis/dimension label
`grid`^{r}	text		"0 1"	Grid ranges/intervals
`p1`^{r}	text	`reference_point.label`		Select end point
`p2`^{o}	text	`reference_point.label`		Select end point
`scale`^{o}	real			Interpolation fraction

Additional information:

- **label:** The allowed set of axis labels depends on the coordinate system (*i.e.* `spacegrid.coord`). Labels are `x/y/z` for `coord=cartesian`, `r/phi/z` for `coord=cylindrical`, `r/phi/theta` for `coord=spherical`.
- **p1/p2/scale:** The axis vector is set to `p1+scale*(p2-p1)`. If only `p1` is provided, the axis vector is `p1`.
- **grid:** Specifies the histogram grid along the direction specified by `label`. The allowed grid points fall in the range [-1,1] for `label=x/y/z` or [0,1] for `r/phi/theta`. A grid of 10 evenly spaced points between 0 and 1 can be requested equivalently by `grid="0 (0.1) 1"` or `grid="0 (10) 1"`. Piecewise uniform grids covering portions of the range are supported, *e.g.* `grid="-0.7 (10) 0.0 (20) 0.5"`.
- Note that `grid` specifies the histogram grid along the (curved) coordinate given by `label`. The axis specified by `p1/p2/scale` does not correspond one-to-one with `label` unless `label=x/y/z`, but the full set of axes provided define the (sheared) space on top of which the curved (*e.g.* spherical) coordinate system is built.

7.3.7 One body density matrix

The N-body density matrix in DMC is $\hat{\rho}_N = |\Psi_T\rangle\langle\Psi_{FN}|$ (for VMC, substitute Ψ_T for Ψ_{FN}). The one body reduced density matrix (1RDM) is obtained by tracing out all particle coordinates but one:

$$\hat{n}_1 = \sum_n Tr_{R_n} |\Psi_T\rangle\langle\Psi_{FN}| \tag{7.13}$$

In the formula above, the sum is over all electron indices and $Tr_{R_n}(*) \equiv \int dR_n \langle R_n| * |R_n\rangle$ with $R_n = [r_1, ..., r_{n-1}, r_{n+1}, ..., r_N]$. When the sum is restricted over spin up or down electrons, one obtains a density matrix for each spin species. The 1RDM computed by QMCPACK is partitioned in this way.

In real space, the matrix elements of the 1RDM are

$$n_1(r,r') = \langle r|\hat{n}_1|r'\rangle = \sum_n \int dR_n \Psi_T(r,R_n)\Psi_{FN}^*(r',R_n) \tag{7.14}$$

A more efficient and compact representation of the 1RDM is obtained by expanding in the single particle orbitals obtained from a Hartree-Fock or DFT calculation, $\{\phi_i\}$:

$$\begin{aligned} n_1(i,j) &= \langle \phi_i|\hat{n}_1|\phi_j\rangle \\ &= \int dR \Psi_{FN}^*(R)\Psi_T(R) \sum_n \int dr_n' \frac{\Psi_T(r_n',R_n)}{\Psi_T(r_n,R_n)}\phi_i(r_n')^*\phi_j(r_n) \end{aligned} \tag{7.15}$$

The integration over r' in Eq. 7.15 is inefficient when one is also interested in obtaining matrices involving energetic quantities, such as the energy density matrix of Ref. [6] or the related (and more well known) Generalized Fock matrix. For this reason, an approximation is introduced as follows:

$$n_1(i,j) \approx \int dR \Psi_{FN}(R)^*\Psi_T(R) \sum_n \int dr_n' \frac{\Psi_T(r_n',R_n)^*}{\Psi_T(r_n,R_n)^*}\phi_i(r_n)^*\phi_j(r_n') \tag{7.16}$$

For VMC, FN-DMC, FP-DMC, and RN-DMC the formula above represents an exact sampling of the 1RDM corresponding to $\hat{\rho}_N^\dagger$ (see appendix A of Ref. [6] for more detail).

`estimator type=dm1b` element				
parent elements:	`hamiltonian, qmc`			
child elements:	*none*			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	**dm1b**		Must be dm1b
`name`^{r}	text	*anything*		Unique name for estimator
parameters				
name	**datatype**	**values**	**default**	**description**
`basis`^{r}	text array	sposet.name(s)		Orbital basis
`integrator`^{o}	text	uniform_grid uniform density	uniform_grid	Integration method
`evaluator`^{o}	text	loop/matrix	loop	Evaluation method
`scale`^{o}	real	$0 < scale < 1$	1.0	Scale integration cell
`center`^{o}	real array(3)	*any point*		Center of cell
`points`^{o}	integer	> 0	10	Grid points in each dim
`samples`^{o}	integer	> 0	10	MC samples
`warmup`^{o}	integer	> 0	30	MC warmup
`timestep`^{o}	real	> 0	0.5	MC time step
`use_drift`^{o}	boolean	yes/no	no	Use drift in VMC
`check_overlap`^{o}	boolean	yes/no	no	Print overlap matrix
`check_derivatives`^{o}	boolean	yes/no	no	Check density derivatives
`acceptance_ratio`^{o}	boolean	yes/no	no	Print accept ratio
`rstats`^{o}	boolean	yes/no	no	Print spatial stats
`normalized`^{o}	boolean	yes/no	no	`basis` comes norm'ed
`energy_matrix`^{o}	boolean	yes/no	no	Energy density matrix

Additional information

- `name:` Density matrix results appear in `stat.h5` files labeled according to `name`.
- `basis:` List of `sposet.name`'s. The total set of orbitals contained in all `sposet`'s comprises the basis (subspace) the one body density matrix is projected onto. This set of orbitals generally includes many virtual orbitals that are not occupied in a single reference Slater determinant.
- `integrator:` This selects the method used to perform the additional single particle integration. Options are `uniform_grid` (uniform grid of points over the cell), `uniform` (uniform random sampling over the cell), and `density` (Metropolis sampling of approximate density: $\sum_{b\in\texttt{basis}}|\phi_b|^2$, not well tested, please check results carefully!). Depending on the integrator selected, different subsets of the other input parameters are active.
- `evaluator:` Select for-loop or matrix multiply implementations. Matrix is preferred for speed. Both implementations should give the same results, but please check as this has not been exhaustively tested.
- `scale:` Resize the simulation cell by scale for use as an integration volume (active for `integrator=uniform/uniform_grid`).
- `center:` Translate the integration volume to center at this point (active for `integrator=uniform/uniform_g`
If `center` is not provided, the scaled simulation cell is used as is.
- `points:` The number of grid points in each dimension for `integrator=uniform_grid`. For example, `points=10` results in a uniform 10x10x10 grid over the cell.
- `samples:` Sets the number of Monte Carlo samples collected each step (active for `integrator=uniform/dens`
- `warmup:` Number of warmup Metropolis steps at the start of the run, prior to data collection (active for `integrator=density`).
- `timestep:` Drift-diffusion timestep used in Metropolis sampling (active for `integrator=density`).
- `use_drift:` Enable drift in Metropolis sampling (active for `integrator=density`).
- `check_overlap:` Print the overlap matrix (computed via simple Riemann sums) to the log and then abort. Note that subsequent analysis based on the 1RDM is simplest if the input orbitals are orthogonal.
- `check_derivatives:` Print analytic and numerical derivatives of the approximate (sampled) density for several sample points, then abort.
- `acceptance_ratio:` Print the acceptance ratio of the density sampling to the log each step.

- `rstats`: Print statistical information about the spatial motion of the sampled points to the log each step.
- `normalized`: Declare whether the inputted orbitals are normalized or not. If `normalized=no`, direct Riemann integration over a 200x200x200 grid will be used to compute the normalizations prior to use.
- `energy_matrix`: Also accumulate the one body reduced energy density matrix and write it to `stat.h5`. This matrix is not covered in any detail here; the interested reader is referred to Ref. [6].

Listing 7.20: One body density matrix with uniform grid integration.

```
<estimator type="dm1b" name="DensityMatrices">
  <parameter name="basis" > spo_u spo_uv </parameter>
  <parameter name="evaluator" > matrix </parameter>
  <parameter name="integrator" > uniform_grid </parameter>
  <parameter name="points" > 4 </parameter>
  <parameter name="scale" > 1.0 </parameter>
  <parameter name="center" > 0 0 0 </parameter>
</estimator>
```

Listing 7.21: One body density matrix with uniform sampling.

```
<estimator type="dm1b" name="DensityMatrices">
  <parameter name="basis" > spo_u spo_uv </parameter>
  <parameter name="evaluator" > matrix </parameter>
  <parameter name="integrator" > uniform </parameter>
  <parameter name="samples" > 64 </parameter>
  <parameter name="scale" > 1.0 </parameter>
  <parameter name="center" > 0 0 0 </parameter>
</estimator>
```

Listing 7.22: One body density matrix with density sampling.

```
<estimator type="dm1b" name="DensityMatrices">
  <parameter name="basis" > spo_u spo_uv </parameter>
  <parameter name="evaluator" > matrix </parameter>
  <parameter name="integrator" > density </parameter>
  <parameter name="samples" > 64 </parameter>
  <parameter name="timestep" > 0.5 </parameter>
  <parameter name="use_drift" > no </parameter>
</estimator>
```

Listing 7.23: Example sposet initialization for density matrix use. Occupied and virtual orbital sets are created separately, then joined (`basis="spo_u spo_uv"`).

```
<sposet_builder type="bspline" href="../dft/pwscf_output/pwscf.pwscf.h5" tilematrix="1 0 0 0 1 0
    0 0 1" twistnum="0" meshfactor="1.0" gpu="no" precision="single">
  <sposet type="bspline" name="spo_u" group="0" size="4"/>
  <sposet type="bspline" name="spo_d" group="0" size="2"/>
  <sposet type="bspline" name="spo_uv" group="0" index_min="4" index_max="10"/>
</sposet_builder>
```

Listing 7.24: Example sposet initialization for density matrix use. Density matrix orbital basis created separately (`basis="dm_basis"`).

```
<sposet_builder type="bspline" href="../dft/pwscf_output/pwscf.pwscf.h5" tilematrix="1 0 0 0 1 0
    0 0 1" twistnum="0" meshfactor="1.0" gpu="no" precision="single">
  <sposet type="bspline" name="spo_u" group="0" size="4"/>
  <sposet type="bspline" name="spo_d" group="0" size="2"/>
  <sposet type="bspline" name="dm_basis" size="50" spindataset="0"/>
</sposet_builder>
```

7.4 Forward Walking Estimators

Forward walking is a method by which one can sample the pure fixed-node distribution $\langle\Phi_0|\Phi_0\rangle$. Specifically, one multiplies each walker's DMC mixed estimate for the observable $\mathcal{O}$, $\frac{\mathcal{O}(\mathbf{R})\Psi_T(\mathbf{R})}{\Psi_T(\mathbf{R})}$, by the weighting factor $\frac{\Phi_0(\mathbf{R})}{\Psi_T(\mathbf{R})}$. As it turns out, this weighting factor for any walker $\mathbf{R}$ is proportional to the total number of descendants the walker will have after a sufficiently long projection time β.

To forward walk on an observable, one declares a generic forward walking estimator within a `<hamiltonian>` block, and then specifies the observables to forward walk on and forward walking parameters. Here is a summary.

`estimator type=ForwardWalking` element				
parent elements:	`hamiltonian, qmc`			
child elements:	`Observable`			
attributes				
name	**datatype**	**values**	**default**	**description**
`type`^{r}	text	**ForwardWalking**		Must be "ForwardWalking"
`name`^{r}	text	*anything*	any	Unique name for estimator

Additional information:

- **Cost**: Due to having to store histories of observables up to `max` time-steps, one should multiply the memory cost of storing the non-forward walked observables variables by `max`. Not an issue for things like the potential energy, but can be prohibitive for observables like density, forces, etc.

`Observable` element				
parent elements:	`estimator, hamiltonian, qmc`			
child elements:	*None*			
attributes				
name	**datatype**	**values**	**default**	**description**
`name`^{r}	text	*anything*	any	Registered name of existing estimator on which to forward
`max`^{r}	integer	> 0		The maximum projection time in steps (`max`= β/τ).
`frequency`^{r}	text	≥ 1		Dump data only for every `frequency`-th to `scalar.dat` file

- **Naming Convention**: Forward walked observables are automatically named `FWE_name_i`, where `i` is the forward walked expectation value at time step `i`, and `name` is whatever name appears in the `<Observable>` block. This is also how it will appear in the `scalar.dat` file.

In the following example case, QMCPACK forward walks on the potential energy for 300 time steps, and dumps the forward walked value at every time step.

Listing 7.25: Forward walking estimator element.

```
<estimator name="fw" type="ForwardWalking">
    <Observable name="LocalPotential" max="300" frequency="1"/>
     <!--- Additional Observable blocks go here -->
  </estimator>
```

7.5 "Force" estimators

QMCPACK supports force estimation by use of the Chiesa-Ceperly-Zhang (CCZ) estimator. Currently, open and periodic boundary conditions are supported, but for all-electron calculations only.

Without loss of generality, the CCZ estimator for the z-component of the force on an ion centered at the origin is given by the following expression:

$$F_z = -Z \sum_{i=1}^{N_e} \frac{z_i}{r_i^3} [\theta(r_i - \mathcal{R}) + \theta(\mathcal{R} - r_i) \sum_{\ell=1}^{M} c_\ell r_i^\ell] \tag{7.17}$$

Z is the ionic charge, M is the degree of the smoothing polynomial, $\mathcal{R}$ is a real-space cutoff of the sphere within which the bare-force estimator is smoothed, and c_ℓ are predetermined coefficients. These coefficients are chosen to minimize the weighted mean square error between the bare force estimate and the s-wave filtered estimator. Specifically,

$$\chi^2 = \int_0^{\mathcal{R}} dr\, r^m\, [f_z(r) - \tilde{f}_z(r)]^2 \tag{7.18}$$

Here, m is the weighting exponent, $f_z(r)$ is the unfiltered radial force density for the z force component, and $\tilde{f}_z(r)$ smoothed polynomial function for the same force density. The reader is invited to refer to the original paper for a more thorough explanation of the methodology, but with the notation in hand, QMCPACK takes the following parameters.

`estimator type=Force` element				
parent elements:	`hamiltonian, qmc`			
child elements:	`parameter`			
attributes				
name	**datatype**	**values**	**default**	**description**
`mode`^{o}	text	*See above*	bare	Select estimator type
`type`^{r}	text	Force		Must be "Force"
`name`^{o}	text	*anything*	ForceBase	Unique name for this estimator
`pbc`^{o}	boolean	yes/no	yes	Using periodic BC's or not
`addionion`^{o}	boolean	yes/no	no	Add the ion-ion force contribution to output force estim
parameters				
name	**datatype**	**values**	**default**	**description**
`rcut`^{o}	real	> 0	1.0	Real space cutoff $\mathcal{R}$ in bohr.
`nbasis`^{o}	integer	> 0	2	Degree of smoothing polynomial M
`weightexp`^{o}	integer	> 0	2	χ^2 weighting exponent m.

Additional information:

- **Naming Convention**: The unique identifier `name` is appended with `name_X_Y` in the `scalar.dat` file, where `X` is the ion ID number, and `Y` is the component ID (an integer with x=0, y=1, z=2). All force components for all ions are computed and dumped to the `scalar.dat` file.
- **Miscellaneous**: Usually, the default choice of `weightexp` is sufficient. Different combinations of `rcut` and `nbasis` should be tested though to minimize variance and bias. There is of course a tradeoff, with larger `nbasis` and smaller `rcut` leading to smaller biases and larger variances.

The following is an example use case.

```
<estimator name="myforce" type="Force" mode="cep" addionion="yes">
   <parameter name="rcut">0.1</parameter>
   <parameter name="nbasis">4</parameter>
   <parameter name="weightexp">2</parameter>
</estimator>
```

Chapter 8

Quantum Monte Carlo Methods

`qmc` factory element				
parent elements:	`simulation, loop`			
type selector:	`method` attribute			
type options:	vmc	Variational Monte Carlo		
	linear	Wavefunction optimization with linear method		
	dmc	Diffusion Monte Carlo		
	rmc	Reptation Monte Carlo		
shared attributes:				
name	**datatype**	**values**	**default**	**description**
`method`	text	listed above	invalid	QMC driver
`move`	text	pbyp, alle	pbyp	method used to move electrons
`gpu`	text	yes, no	dep.	use the GPU
`trace`	text		no	???
`checkpoints`	integer	-1, 0, n	-1	checkpoint frequency
`target`	text			???
`completed`	text			???
`append`	text	yes, no	yes	???

Additional information:

- `move`. There are two ways implemented to move electrons. The more used method is the particle-by-particle move. In this method, only one electron is moved for acception or rejection. The other method is the all-electron move, namely all the electrons are moved once for testing acception or rejection.

- `gpu`. When the executable is compiled with CUDA, the target computing device can be chosen by this switch. With a regular CPU only compilation, this option is not effective.

- `checkpoints`. If Checkpoint="-1" no checkpoint will be done (default setting). If Checkpoint="0" dump after the completion of a qmc section. If Checkpoint="n" where n is an integer¿0, walkers will be dumped into a *.config.h5 file every n block. The config.h5 file will contain the state of a population to continue a run including the random number sequences; the list of what is included in the .congig.h5 is: number of walkers, status of the run, branch mode, energy dataset, ratio to accepted moves, ratio to proposed moves, variance dataset, vParam{tau, taueff. E_trial, E_ref, Branch_Max, BranchCutOff, BranchFilter, Sigma, Accepted_Energy, Accepted_Samples}, IParamwarmumSteps, Energy_Update_Interval, Counter, targetwalkers, Maxwalkers, MinWalkers, Branching Interval, Walker coordinates, Random number size, Random number sequence, version of the code.

8.1 Variational Monte Carlo

`vmc` method

parameters

name	**datatype**	**values**	**default**	**description**
`walkers`	integer	> 0	dep.	number of walkers per node
`blocks`	integer	≥ 0	1	number of blocks
`steps`	integer	≥ 0	1	number of steps per block
`warmupsteps`	integer	≥ 0	0	number of steps for warming up
`substeps`	integer	≥ 0	1	number of substeps per step
`usedrift`	text	yes, no	no	use the algorithm with drift
`timestep`	real	> 0	0.1	time step for each electron move
`samples`	integer	≥ 0	0	total number of samples
`stepsbetweensamples`	integer	> 0	1	period of the sample accumulation
`samplesperthread`	integer	≥ 0	0	number of samples per thread
`storeconfigs`	integer	all values	0	store configurations

Additional information:

- `walkers`. The initial default number of walkers is 1 but in the CPU branch this number will be overwritten as the number of OpenMP threads if the user requested number is smaller than the number of threads.

- `blocks`. This parameter is universal for all the method. At the end of each block, all the statistics accumulated in the block is dumped in to files, e.g. scalar.dat.

- `warmupsteps`. Warm-up steps are steps used only for equilibration. All the samples generated by warm-up steps are discarded. In practice, there's no need to use many walm-up steps because we can always discard more statistics when we perform the post-process.

- `substeps`. In a substep, each of the electrons is moved only once by either particle-by-particle or all-electron move. Because the local energy is evaluated not at each substep but at each step, increasing the number of substeps doesn't accumulate more samples. But in order to reduce the correlation between consecutive samples, increasing substeps is a very good option for its cheaper computational cost.

- `usedrift`. The VMC is implemented in two algorithms with or without drift. In the no-drift algorithm, the move of each electron is proposed with a Gaussian distribution. The standard deviation is chosen as the timestep input. In the drift algorithm, electrons are moved by langevin dynamics.

- `timestep`. The meaning of timestep depends on whether the drift is used or not. In general, larger timestep reduces the time correlation but might also reduces the accept ratio. Users are required to check the accept ratio of the calculation and make sure it's larger than 0.9 or between 0.2 and 0.8 with or without the drift.

- `stepsbetweensamples`. Due to the fact that samples generated by consecutive steps might be still correlated. Having stepsbetweensamples larger than 1 reduces that correlation. In practice, using larger substeps is cheaper than using stepsbetweensamples to decorrelate samples.

- `samples`. This is the total amount of samples generated in the current VMC session. This parameter is not important for VMC only calculation but necessary if optimization or DMC follows.
$$\text{samples} = \frac{\text{blocks} \cdot \text{steps} \cdot \text{walkers}}{\text{stepsbetweensamples}} \cdot \text{number of MPI tasks}$$

- `samplesperthread`. This is an alternative way to set the target amount of samples. More useful in the VMC session preparing the population for the following DMC calculation.
$$\text{samplesperthread} = \frac{\text{blocks} \cdot \text{steps}}{\text{stepsbetweensamples}}$$

- `storeconfigs`. If storeconfigs is set to a non-zero value, then electron configurations during the VMC run will be saved to the files.

  The following is an example of VMC section.

```
<qmc method="vmc" move="pbyp" gpu="yes">
  <estimator name="LocalEnergy" hdf5="no"/>
  <parameter name="walkers"> 256 </parameter>
  <parameter name="samples"> 2867200 </parameter>
  <parameter name="stepsbetweensamples"> 1 </parameter>
  <parameter name="substeps"> 5 </parameter>
  <parameter name="warmupSteps"> 5 </parameter>
  <parameter name="blocks"> 70 </parameter>
  <parameter name="timestep"> 1.0 </parameter>
  <parameter name="usedrift"> no </parameter>
</qmc>
```

8.2 Wavefunction Optimization

8.3 Diffusion Monte Carlo

`dmc` method

parameters

name	datatype	values	default	description
`targetwalkers`	integer	> 0	dep.	number of walkers per node
`blocks`	integer	≥ 0	1	number of blocks
`steps`	integer	≥ 0	1	number of steps per block
`warmupsteps`	integer	≥ 0	0	number of steps for warming up
`timestep`	real	> 0	0.1	time step for each electron move
`checkproperties`	integer	≥ 0	100	number of steps between walker updat
`maxcpusecs`	real	≥ 0	3.6e5	maximum allowed walltime in seconds
`energyUpdateInterval`	integer	≥ 0	0	trial energy update interval
`refEnergy`	AU	all values	dep.	reference energy
`feedback`	double	≥ 0	1.0	population feedback on the trial energ
`useBareTau`	option	yes,no	0	do not use effective time step
`warmupByReconfiguration`	option	yes,no	0	warm up with a fixed population
`sigmaBound`	double	≥ 0	10	parameter to cutoff large weights
`killnode`	string	yes/other	no	kill or reject walkers that cross nodes
`reconfiguration`	string	yes/pure/other	no	fixed population t:qechnique
`branchInterval`	integer	≥ 0	1	branching interval
`substeps`	integer	≥ 0	1	branching interval
`nonlocalmoves`	string	yes/other	no	run with tmoves
`scaleweight`	string	yes/other	yes	scale weights (CUDA only)
`MaxAge`	double	≥ 0	10	kill persistent walkers
`MaxCopy`	double	≥ 0	2	limit population growth
`fastgrad`	text	yes/other	yes	fast gradients
`maxDisplSq`	real	all values	-1	maximum particle move
`storeconfigs`	integer	all values	0	store configurations

Additional information:

- `targetwalkers`. A DMC run can be considered a restart run or a new run. A restart run is considered to be any method block beyond the first one, such as when a DMC method block that follows a VMC block. Alternatively, if the user reads in configurations from disk it is also considered a restart run. In the case of a restart run, the DMC driver will use the configurations from the previous run, and this variable will not be used. For a new run, if the number of walkers is less than the number of threads, then the number of walkers will be set equal to the number of threads.
- `blocks`. Number of blocks run during an DMC method block. A block consists of a number of DMC steps (steps), after which all the statistics accumulated in the block are written to

disk.

- `steps`. Number of diffusion Monte Carlo steps in a block.
- `warmupsteps`. Warm-up steps are steps at the beginning of a DMC run in which the instantaneous average energy is used to update the trial energy. During regular steps, E_{ref} is used.
- `timestep`. The timestep determines the accuracy of the imaginary time propagator. Generally, multiple time steps are used to extrapolate to the infinite time step limit. A good range of timesteps in which to perform time step extrapolation will typically have a minimum of 99% acceptance probability for each step.
- `checkproperties`. When using particle by particle driver, this variable specifies how often to reset all the variables kept in the buffer.
- `maxcpusecs`. The default is 100 hours. Once the specified time has elapsed, the program will finalize the simulation even if not all blocks are completed.
- `energyUpdateInterval`. The default is to update the trial energy at every step. Otherwise the trial energy is updated every `energyUpdateInterval` steps.

$$E_{\text{trial}} = \text{refEnergy} + \text{feedback} \cdot (\ln \text{targetWalkers} - \ln N)$$

  where N is the current population.
- `refEnergy`. The default reference energy is taken from the VMC run that precedes the DMC run. This value is updated to the current mean whenever branching happens.
- `feedback`. Variable used to determine how strong to react to population flucttutations when doing population control. See the equation in energyUpdateInterval for more details.
- `useBareTau`. The same time step is used whether a move is rejected to not. The default is to use an effective time step when a move is rejected.
- `warmupByReconfiguration`. Warmup DMC is done with a fixed population
- `sigmaBound` . Determine the branch cutoff to limit wild weights based on the sigma and sigmaBound
- `killnode` . When running fixed-node, if a walker attempts to cross a node, the move will normally be rejected. If killnode = ”yes”, then walkers are destroyed when they cross a node.
- `reconfiguration`. If reconfiguration is ”yes”, then run with a fixed walker population using the reconfiguration technique.

- `branchInterval`. Number of steps between branching. The total number of DMC steps in a block will be BranchInterval*Steps.
- `substeps`. Same as BranchInterval.
- `nonlocalmoves`. DMC driver for running Hamiltonians with non-local moves. An typical usage is to simulate Hamitonians with non-local psuedopotentials with T-Moves. Setting this equal to false will impose the locality approximation.
- `scaleweight`. Scaling weight per Umrigar/Nightengale. CUDA only.
- `MaxAge`. Set the weight of a walker to min(currentweight,0.5) after a walker has not moved for MaxAge steps. Needed if persistent walkers appear during the course of a run.
- `MaxCopy`. When determining the number of copies of a walker to branch, set the number of copies equal to min(Multiplicity,MaxCopy).
- `fastgrad`. Calculates gradients with either the fast version or the full-ratio version.
- `maxDisplSq` . When running a DMC calculation with particle by particle, this sets the maximum displacement allowed for a single particle move. All distance displacements larger than the max is rejected. If initialized to a negative value, it becomes equal to Lattice(LR/rc).
- `sigmaBound` . Determine the branch cutoff to limit wild weights based on the sigma and sigmaBound
- `storeconfigs`. If storeconfigs is set to a non-zero value, then electron configurations during the DMC run will be saved. This option is disabled for the OpenMP version of DMC.

Listing 8.1: The following is an example of a very simple DMC section.

```
<qmc method="dmc" move="pbyp" target="e">
  <parameter name="blocks">100</parameter>
  <parameter name="steps">400</parameter>
  <parameter name="timestep">0.010</parameter>
  <parameter name="warmupsteps">100</parameter>
</qmc>
```

The time step should be adjusted for each problem individually. Please refer to the theory section on diffusion Monte Carlo.

Listing 8.2: The following is an example of running a simulation that can be restarted .

```
<qmc method="dmc" move="pbyp" checkpoint="0" dumpconfig="5">
  <parameter name="timestep"> 0.004 </parameter>
  <parameter name="blocks"> 100 </parameter>
  <parameter name="steps"> 400 </parameter>
</qmc>
```

The flags checkpoint and dumpconfig instructs qmcpack to output walker configurations. This also works in variational Monte Carlo. This will output an h5 file with the name "projectid"."run-number".config.h5. Check that this file exists before attempting a restart. To read in this file for a continuation run, specify the following:

Listing 8.3: Restart (read wakers from previous run)

```
<mcwalkerset fileroot="BH.s002" node="-1" nprocs="1" version="0 6" collected="yes"/>
```

where, BH is the project id and s002 is the calculation number to read in the walkers from the previous run.

Combining VMC and DMC in a single run (and wave function optimization can be combined in this way too) is the standard way in which QMCPACK is typical run. There is no need to run two separate jobs, as method sections can be stacked, and walkers are transfered between them.

Listing 8.4: Combined VMC and DMC run

```
<qmc method="vmc" move="pbyp" target="e">
  <parameter name="blocks">100</parameter>
  <parameter name="steps">4000</parameter>
  <parameter name="warmupsteps">100</parameter>
  <parameter name="samples">1920</parameter>
  <parameter name="walkers">1</parameter>
  <parameter name="timestep">0.5</parameter>
</qmc>
<qmc method="dmc" move="pbyp" target="e">
  <parameter name="blocks">100</parameter>
  <parameter name="steps">400</parameter>
  <parameter name="timestep">0.010</parameter>
  <parameter name="warmupsteps">100</parameter>
</qmc>
<qmc method="dmc" move="pbyp" target="e">
  <parameter name="warmupsteps">500</parameter>
  <parameter name="blocks">50</parameter>
  <parameter name="steps">100</parameter>
  <parameter name="timestep">0.005</parameter>
</qmc>
```

8.4 Reptation Monte Carlo

Like diffusion monte carlo, reptation monte carlo (RMC) is a projector based method, allowing us the ability to sample the fixed-node wavefunciton. However, by exploiting the path-integral formulation of Schrödinger's equation, the RMC algorithm can offer some advantages over traditional DMC, such as sampling both the mixed and pure fixed-node distributions in polynomial time, as well as not having population fluctuations and biases. The current implementation does not work with T-moves.

There are two adjustable parameters that affect the quality of the RMC projection: imaginary projection time β of the sampling path (commonly called a "reptile"), and the Trotter time step τ. β must be chosen to be large enough such that $e^{-\beta\hat{H}}|\Psi_T\rangle \approx |\Phi_0\rangle$ for mixed observables, and $e^{-\frac{\beta}{2}\hat{H}}|\Psi_T\rangle \approx |\Phi_0\rangle$ for pure observables. The reptile is discretized into $M = \beta/\tau$ beads at the cost of an $\mathcal{O}(\tau)$ time-step error for observables arising from the Trotter-Suzuki breakup of the short-time propagator.

The following table lists some of the more practical

`vmc` method

parameters

name	datatype	values	default	description
`beta`	real	> 0	dep.	reptile projection time β
`timestep`	real	> 0	0.1	Trotter time step τ for each electron move
`beads`	int	> 0	1	Number of reptile beads $M = \beta/\tau$
`blocks`	integer	≥ 0	1	number of blocks
`steps`	integer	≥ 0	1	number of steps per block
`vmcpresteps`	integer	≥ 0	0	propagates reptile using VMC for given number of steps
`warmupsteps`	integer	≥ 0	0	number of steps for warming up
`MaxAge`	integer	≥ 0	0	force accept for stuck reptile if age exceeds MaxAge.

Additional information:

Because of the sampling differences between DMC ensembles of walkers and RMC reptiles, the RMC block should contain the following estimator declaration to ensure correct sampling: `<estimator name="RMC" hdf5="no">`.

- `beta` or `beads`? One can specify one or the other, and from the Trotter time-step, the code will construct an appropriately sized reptile. If both are given, `beta` overrides `beads`.

- **Mixed vs. Pure observables?** For all observables appearing in the `scalar.dat` file in either VMC or DMC, RMC appends the suffix `_m` or `_p` for mixed and pure estimates respectively.

- **Sampling**. For pure estimators, one should check the traces of both pure and mixed estimates. Ergodicity is a known problem in RMC. Because we use the bounce algorithm, it is possible for the reptile to bounce back and forth without changing the electron coordinates of the central beads. This might not easily show up with mixed estimators, since these are accumulated at constantly regrown ends, but pure estimates are accumulated on these central beads, and so can exhibit strong autocorrelations in pure estimate traces.

- **Propagator**: Our implementation of RMC uses Moroni's DMC link action (symmetrized), with Umrigar's scaled drift near nodes. In this regard, the propagator is identical to the one QMCPACK uses in DMC.
- **Sampling**: We use Ceperley's bounce algorithm. MaxAge is used in case the reptile gets stuck, at which point the code forces move acceptance, stops accumulating statistics, and requilibrates the reptile. Very rarely will this be required. For move proposals, we use particle-by-particle VMC a total of N_e times to generate a new all-electron configuration, at which point the action is computed and the move is either accepted or rejected.

Chapter 9

Analysing QMCPACK data

9.1 Using the qmca tool

9.2 Densities and spin-densities

9.3 Energy densities

Chapter 10

Examples

WARNING: THESE EXAMPLES ARE NOT CONVERGED! YOU MUST CONVERGE PARAMETERS (SIMULATION CELL SIZE, JASTROW PARAMETER NUMBER/CUTOFF, TWIST NUMBER, DMC TIME STEP, DFT PLANE WAVE CUTOFF, DFT K-POINT MESH, ETC.) FOR REAL CALCUATIONS!

The following examples should run in serial on a modern workstation in a few hours.

10.1 Using Nexus

10.1.1 H_2O Molecule with Quantum ESPRESSO Orbitals

With BFD pseudopotentials (see Section ??) for O and H in a subdirectory named `pseudopotentials` (both UPF and FSAtom xml formats, named `O.BFD.upf`, `O.BFD.xml`, `H.BFD.upf`, and `H.BFD.xml`, respectively) and the following XYZ file (named `H2O.xyz`)

```
3

O 0.0000000000e+00 0.0000000000e+00 0.0000000000e+00
H 0.0000000000e+00 -1.4308249289e+00 1.1078707576e+00
H 0.0000000000e+00 1.4308249289e+00 1.1078707576e+00
```

in the working directory, a Python script using Nexus to generate the orbitals using Quantum ESPRESSO, then run QMCPACK to optimize the Jastrow and then do DMC for H_2O in a box is:

Listing 10.1: Nexus example for H_2O using Quantum ESPRESSO orbitals and BFD pseudopotentials

```
#! /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,vmc,loop,linear,dmc

# General Settings (Directories For I/O, Machine Type, etc.)
settings(
   pseudo_dir = 'pseudopotentials',
   runs = 'runs',
   results = 'results',
   sleep = 3,
   generate_only = 0,
   status_only = 0,
   machine = 'ws1',
   )

# Executables (Indicate Path If Needed)
pwscf = 'pw.x'
pw2qmcpack = 'pw2qmcpack.x'
qmcpack = 'qmcapp'

# Pseudopotentials
dft_pps = ['O.BFD.upf','H.BFD.upf']
qmc_pps = ['O.BFD.xml','H.BFD.xml']

# Job Definitions (MPI Tasks, MP Threading, PBS Queue, Time, etc.)
scf_job = Job(app=pwscf,serial=True)
p2q_job = Job(app=pw2qmcpack,serial=True)
opt_job = Job(threads=4,app=qmcpack,serial=True)
dmc_job = Job(threads=4,app=qmcpack,serial=True)

# System To Be Simulated
structure = Structure()
structure.read_xyz('H2O.xyz')
structure.bounding_box(
   box = 'cubic',
   scale = 1.5
   )
structure.add_kmesh(
   kgrid = (1,1,1),
   kshift = (0,0,0)
)
H2O_molecule = PhysicalSystem(
   structure = structure,
   net_charge = 0,
   net_spin = 0,
   O = 6,
   H = 1,
   )
```

```
sims = []

# DFT SCF To Generate Converged Density
scf = generate_pwscf(
    identifier = 'scf',
    path = '.',
    job = scf_job,
    input_type = 'scf',
    system = H2O_molecule,
    pseudos = dft_pps,
    ecut = 50,
    ecutrho = 400,
    conv_thr = 1.0e-5,
    mixing_beta = 0.7,
    mixing_mode = 'local-TF',
    degauss = 0.001
    )
sims.append(scf)

# Convert DFT Wavefunction Into HDF5 File For QMCPACK
p2q = generate_pw2qmcpack(
    identifier = 'p2q',
    path = '.',
    job = p2q_job,
    write_psir = False,
    dependencies = (scf,'orbitals')
    )
sims.append(p2q)

# QMC Optimization Parameters - Coarse Sampling Set
linopt1 = linear(
    energy = 0.0,
    unreweightedvariance = 1.0,
    reweightedvariance = 0.0,
    timestep = 0.4,
    samples = 8192,
    warmupsteps = 50,
    blocks = 64,
    substeps = 4,
    nonlocalpp = True,
    usebuffer = True,
    walkers = 1,
    minwalkers = 0.5,
    maxweight = 1e9,
    usedrift = True,
    minmethod = 'quartic',
    beta = 0.0,
    exp0 = -16,
    bigchange = 15.0,
    alloweddifference = 1e-4,
    stepsize = 0.2,
    stabilizerscale = 1.0,
    nstabilizers = 3
    )

# QMC Optimization Parameters - Finer Sampling Set
linopt2 = linopt1.copy()
linopt2.samples = 16384

# QMC Optimization
opt = generate_qmcpack(
    identifier = 'opt',
    path = '.',
    job = opt_job,
    input_type = 'basic',
    system = H2O_molecule,
    twistnum = 0,
    bconds = 'nnn',
    pseudos = qmc_pps,
    jastrows = [('J1','bspline',8,4),
                  ('J2','bspline',8,4)],
    calculations = [loop(max=4,qmc=linopt1),
                  loop(max=4,qmc=linopt2)],
    dependencies = (p2q,'orbitals')
    )
sims.append(opt)

# QMC VMC/DMC With Optimized Jastrow Parameters
qmc = generate_qmcpack(
    identifier = 'dmc',
    path = '.',
    job = dmc_job,
    input_type = 'basic',
    system = H2O_molecule,
    pseudos = qmc_pps,
    bconds = 'nnn',
    jastrows = [],
    calculations = [
        vmc(
            walkers = 1,
            samplesperthread = 64,
            stepsbetweensamples = 1,
            substeps = 5,
            warmupsteps = 100,
            blocks = 1,
            timestep = 1.0,
            usedrift = False
           ),
        dmc(
            minimumtargetwalkers = 128,
            reconfiguration = 'no',
            warmupsteps = 100,
            timestep = 0.005,
            steps = 10,
            blocks = 200,
            nonlocalmoves = True
           )
        ],
    dependencies = [(p2q,'orbitals'),(opt,'jastrow')]
    )
sims.append(qmc)

run_project(sims)
```

10.1.2 LiH Crystal with Quantum ESPRESSO Orbitals

With CASINO-formatted Trail-Needs pseudopotentials (see Section **??**) for O and H in a subdirectory named `pseudopotentials` (both UPF and CASINO formats, named `O.TN-DF.upf`, `O.pp.data`, `H.TN-DF.upf`, and `H.pp.data`, respectively), a Python script using Nexus to generate the orbitals using Quantum ESPRESSO, then run QMCPACK to optimize the Jastrow and then do DMC for LiH with periodic boundary conditions is:

Listing 10.2: Nexus example for bulk LiH using Quantum ESPRESSO orbitals and CASINO pseudopotentials

```
#! /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,vmc,loop,linear,dmc

# General Settings (Directories For I/O, Machine Type, etc.)
settings(
    pseudo_dir = 'pseudopotentials',
    runs = 'runs',
    results = 'results',
    sleep = 3,
    generate_only = 1,
    status_only = 0,
    machine = 'ws1',
    )

# Executables (Indicate Path If Needed)
```

```
pwscf = 'pw.x'
pw2qmcpack = 'pw2qmcpack.x'
qmcpack = 'qmcapp'

# Pseudopotentials
dft_pps = ['Li.TN-DF.upf','H.TN-DF.upf']
qmc_pps = ['Li.pp.data','H.pp.data']

# Job Definitions (MPI Tasks, MP Threading, PBS Queue, Time, etc.)
scf_job = Job(app=pwscf,serial=True)
nscf_job = Job(app=pwscf,serial=True)
p2q_job = Job(app=pw2qmcpack,serial=True)
opt_job = Job(threads=4,app=qmcpack,serial=True)
dmc_job = Job(threads=4,app=qmcpack,serial=True)

# System To Be Simulated
rocksalt_LiH = generate_physical_system(
    lattice = 'cubic',
    cell = 'primitive',
    centering = 'F',
    atoms = ('Li','H'),
    basis = [[0.0,0.0,0.0],
                     [0.5,0.5,0.5]],
    basis_vectors = 'conventional',
    constants = 7.1,
    units = 'B',
    kgrid = (17,17,17),
    kshift = (1,1,1),
    net_charge = 0,
    net_spin = 0,
    Li = 1,
    H = 1,
    )

sims = []

# DFT SCF To Generate Converged Density
scf = generate_pwscf(
    identifier = 'scf',
    path = '.',
    job = scf_job,
    input_type = 'scf',
    system = rocksalt_LiH,
    pseudos = dft_pps,
    ecut = 450,
    ecutrho = 1800,
    conv_thr = 1.0e-10,
    mixing_beta = 0.7,
    )
```

```
sims.append(scf)

# DFT NSCF To Generate Wave Function At Specified K-points
nscf = generate_pwscf(
    identifier = 'nscf',
    path = '.',
    job = nscf_job,
    input_type = 'nscf',
    system = rocksalt_LiH,
    pseudos = dft_pps,
    ecut = 450,
    ecutrho = 1800,
    conv_thr = 1.0e-10,
    mixing_beta = 0.7,
    kgrid = (1,1,1),
    kshift = (0,0,0),
    dependencies = (scf,'charge-density')
    )
sims.append(nscf)

# Convert DFT Wavefunction Into HDF5 File For QMCPACK
p2q = generate_pw2qmcpack(
    identifier = 'p2q',
    path = '.',
    job = p2q_job,
    write_psir = False,
    dependencies = (nscf,'orbitals')
    )
sims.append(p2q)

# QMC Optimization Parameters - Coarse Sampling Set
linopt1 = linear(
    energy = 0.0,
    unreweightedvariance = 1.0,
    reweightedvariance = 0.0,
    timestep = 0.4,
    samples = 8192,
    warmupsteps = 50,
    blocks = 64,
    substeps = 4,
    nonlocalpp = True,
    usebuffer = True,
    walkers = 1,
    minwalkers = 0.5,
    maxweight = 1e9,
    usedrift = True,
    minmethod = 'quartic',
    beta = 0.0,
    exp0 = -16,
    bigchange = 15.0,
    alloweddifference = 1e-4,
    stepsize = 0.2,
    stabilizerscale = 1.0,
    nstabilizers = 3

# QMC Optimization Parameters - Finer Sampling Set
linopt2 = linopt1.copy()
linopt2.samples = 16384

# QMC Optimization
opt = generate_qmcpack(
    identifier = 'opt',
    path = '.',
    job = opt_job,
    input_type = 'basic',
    system = rocksalt_LiH,
    twistnum = 0,
    bconds = 'ppp',
    pseudos = qmc_pps,
    jastrows = [('J1','bspline',8),
                 ('J2','bspline',8)],
    calculations = [loop(max=4,qmc=linopt1),
                 loop(max=4,qmc=linopt2)],
    dependencies = (p2q,'orbitals')
    )
pp = opt.input.get('pseudos')
pp.Li.format='casino'
pp.Li['l-local']='s'
pp.Li.nrule=2
pp.Li.lmax=2
pp.Li.cutoff=2.19
pp.H.format='casino'
pp.H['l-local']='s'
pp.H.nrule=2
pp.H.lmax=2
pp.H.cutoff=0.50
sims.append(opt)

# QMC VMC/DMC With Optimized Jastrow Parameters
qmc = generate_qmcpack(
    identifier = 'dmc',
    path = '.',
    job = dmc_job,
    input_type = 'basic',
    system = rocksalt_LiH,
    pseudos = qmc_pps,
    bconds = 'ppp',
    jastrows = [],
    calculations = [
        vmc(
            walkers = 1,
            samplesperthread = 64,
            stepsbetweensamples = 1,
            substeps = 5,
            warmupsteps = 100,
            blocks = 1,
            timestep = 1.0,
            usedrift = False
           ),
        dmc(
            minimumtargetwalkers = 128,
            reconfiguration = 'no',
            warmupsteps = 100,
            timestep = 0.005,
            steps = 10,
            blocks = 200,
            nonlocalmoves = True
           )
        ],
    dependencies = [(p2q,'orbitals'),(opt,'jastrow')]
    )
pp = qmc.input.get('pseudos')
pp.Li.format='casino'
pp.Li['l-local']='s'
pp.Li.nrule=2
pp.Li.lmax=2
pp.Li.cutoff=2.37
pp.H.format='casino'
pp.H['l-local']='s'
pp
pp.H.lmax=2
pp.H.cutoff=0.50
sims.append(qmc)

run_project(sims)
```

Chapter 11

Lab 2: QMC Basics

11.1 Overview

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

11.1.1 Getting the most out of this lab

The outline below shows the overall structure of the lab. Those who are new to QMC or QMCPACK should probably work through the contents in order. If you have a specific application/target system you would like to explore with QMCPACK, be sure to leave enough time for the optional material in section 11.4. Feel free to discuss your answers/results to the questions/exercises at the end of each section with a lab instructor.

1. **Overview**
   Overview of lab content.

   1.1 Getting the most out of this lab
   This section.

   1.2 Lab directories and files
   Description of directories and files used in the lab.

1.3 The QMCPACK input file and XML
XML as used by QMCPACK. Reduced example of input file structure.

2. Testing PP atomic properties: optimization, diffusion Monte Carlo
Calculate ionization potential of oxygen using pre-generated QMCPACK input files.

2.1 Getting and converting a pseudopotential
Download PP from BFD database. Convert to QMCPACK format with `ppconvert`.

2.2 Optimization walkthrough: neutral O atom
Theoretical background on trial wavefunction & optimization. QMCPACK optimization walkthrough. Fully annotated input file (`O.q0.opt.in.xml`) and explanation of Jastrow & optimization inputs.

2.3 DMC timestep extrapolation I: neutral O atom
Theoretical background on timestep & population control biases. DMC timestep extrapolation walkthrough w/ QMCPACK and explanation of DMC inputs.

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

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

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

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

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

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

11.1.2 Lab directories and files

```
Lab_2_QMC_Basics/

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

 oxygen_atom               - oxygen atom calculations
    ip_conv.py               - tool to fit oxygen IP vs timestep
    O.q0.dmc.in.xml          - neutral O DMC input file
    O.q0.dmc.qsub.in         -    "    "  "  submission file
    O.q0.opt.in.xml          -    "    " optimization input file
    O.q0.opt.qsub.in         -    "    "  "  submission file
    O.q0.pwscf.h5            -    "    "  orbitals file
    O.q1.dmc.in.xml          - charged O DMC input file
    O.q1.dmc.qsub.i          -    "    "  "  submission file    n
    O.q1.opt.in.xml          -    "    " optimization input file
    O.q1.opt.qsub.i          -    "    "  "  submission file    n
    O.q1.pwscf.h5            -    "    "  orbitals file
    reference                - directory w/ completed runs
    submit_O_q0_dmc          - executable to submit neutral DMC
    submit_O_q0_opt          -    "       "    "       "    optimization
    submit_O_q1_dmc          -    "       "    "    charged DMC
    submit_O_q1_opt          -    "       "    "       "    optimization

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

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

11.1.3 The QMCPACK input file and XML

This section introduces XML as it is used in QMCPACK's input file. The focus is on the XML file format itself and the general structure of the input file rather than an exhaustive discussion of all keywords and structure elements. Specific keywords and the relevant XML elements are discussed in context as they are encountered in the lab. Participants will work with a complete, annotated input file for the oxygen atom during the wavefunction optimization walkthrough in section 11.2.2.

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

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

begin vmc
  move     = pbyp
  blocks   = 200
  steps    =  10
  timestep = 0.4
end vmc
```

QMCPACK input looks like

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

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

XML elements start with `<element_name>`, end with `</element_name>`, and can be nested within each other to denote substructure (the trial wavefunction is composed of a Slater determinant and a Jastrow factor, which are each further composed of . . .). `id` and `series` are attributes of the

`<project/>` element. XML attributes are generally used to represent simple values, like names, integers, or real values. Similar functionality is also commonly provided by `<parameter/>` elements like those shown above.

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

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

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

  <qmcsystem>

    <simulationcell>
      ...
    </simulationcell>

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

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

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

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

   </qmcsystem>

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

</simulation>
```

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

11.2 Testing PP atomic properties: optimization, diffusion Monte Carlo

11.2.1 Getting and converting a pseudopotential

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

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

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

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

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

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

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

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

```
1
38.41914135 2 8.71924452
```

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

11.2.2 Optimization walkthrough: neutral O atom

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

Background on trial wavefunction and optimization

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

$$\Psi_T = e^{-(J_1+J_2)} D^{\uparrow}(\{\phi_u^{\uparrow}\}_{u=1}^{N^{\uparrow}}) D^{\downarrow}(\{\phi_d^{\downarrow}\}_{d=1}^{N^{\uparrow}}) \tag{11.1}$$

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

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

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

$$J_1 = \sum_{e=1}^{N^{\uparrow}+N^{\downarrow}} U_1^{\uparrow/\downarrow}(|r_e - r_O|) \tag{11.2}$$

where r_e refers to the electron positions and r_O is the position of the oxygen ion. The $U_1^{\uparrow/\downarrow}$ term is a one-dimensional radial function represented with piecewise continuous cubic polynomials (B-splines). The adjustable parameters to be optimized are the "knots" of the B-splines which are

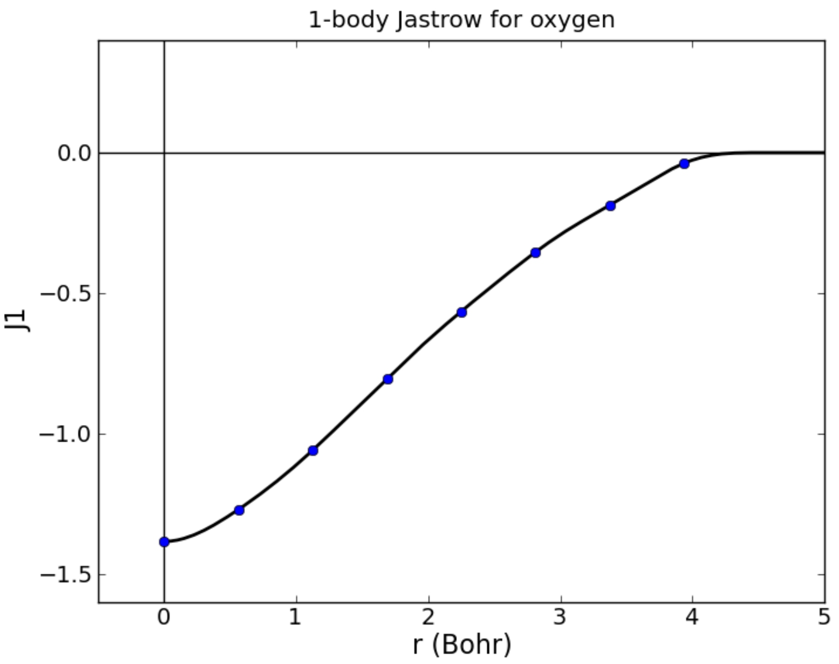


Figure 11.1: Optimized U_1 function for 1-body Jastrow factor of an oxygen atom.

simply the values of the U_1 function at uniformly spaced grid points (See fig. 11.1 for an example of a U_1 spline function with 8 knots).

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

$$J_2 = \sum_{u<u'} U_2^{\uparrow\uparrow/\downarrow\downarrow}(|r_u^\uparrow - r_{u'}^\uparrow|) + \sum_{d<d'} U_2^{\uparrow\uparrow/\downarrow\downarrow}(|r_d^\downarrow - r_{d'}^\downarrow|) + \sum_{u,d} U_2^{\uparrow\downarrow}(|r_u^\uparrow - r_d^\downarrow|) \tag{11.3}$$

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

$$U_2(r) = \frac{Ar}{1+Br} \tag{11.4}$$

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

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

$$E_T(P) = \frac{\langle\Psi_T(P)|\hat{H}|\Psi_T(P)\rangle}{\langle\Psi_T(P)|\Psi_T(P)\rangle} \geq E_0 \tag{11.5}$$

$$V_T(P) = \frac{\langle\Psi_T(P)|\hat{H}^2|\Psi_T(P)\rangle}{\langle\Psi_T(P)|\Psi_T(P)\rangle} - \left(\frac{\langle\Psi_T(P)|H|\Psi_T(P)\rangle}{\langle\Psi_T(P)|\Psi_T(P)\rangle}\right)^2 \geq 0 \tag{11.6}$$

Here E_0 is the ground state energy, $E_T(P)$ is the trial energy, $V_T(P)$ is the trial variance, and P denotes the set of adjustable parameters in the trial wavefunction. Equality is reached only for the true ground state wavefunction and so the trial wavefunction can be improved by attempting to minimize a chosen cost function:

$$C(P) = \alpha E_T(P) + (1-\alpha)V_T(P). \tag{11.7}$$

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

Optimization walkthrough with QMCPACK

Enter the `oxygen_atom` directory and copy over the oxygen pseudopotential (`O.BFD.xml`) you downloaded and converted (section 11.2.1). Alternatively, the already converted pseudopotential is located in the `oxygen_atom/reference` directory. All files prefixed with "`O.q0`" relate to the neutral oxygen atom.

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

The most important parts to focus on for the purposes of this exercise are the Jastrow factors and the inputs to the linear optimization method. Input specifying the one-body electron-ion Jastrow factor corresponding to eq. 11.2 is

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

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

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

The XML describes $U_2^{\uparrow\uparrow/\downarrow\downarrow}(r)$ and $U_2^{\uparrow\downarrow}(r)$ from eq. 11.3 as Padé functions with initial guesses of $B^{\uparrow\uparrow} = 0.6$ Bohr^{-1} and $B^{\uparrow\downarrow} = 1.0$ Bohr^{-1} for the adjustable parameters.

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

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

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

energy Fraction of trial energy in the cost function.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
OldCost: 7.4701713964e-01 NewCost: 7.4681622535e-01 Delta Cost:-2.0091428584e-04
...
  <optVariables href="O_q0_opt.s011.opt.xml">
```

```
e0_0 -9.5623201640e-01 1 1  ON 0
e0_1 -8.4728730387e-01 1 1  ON 1
e0_2 -6.8954452383e-01 1 1  ON 2
e0_3 -4.9327199567e-01 1 1  ON 3
e0_4 -3.2560096773e-01 1 1  ON 4
e0_5 -1.9567566480e-01 1 1  ON 5
e0_6 -1.2940405487e-01 1 1  ON 6
e0_7 -9.5221474839e-02 1 1  ON 7
uu_b 4.2002038228e-01 0 1  ON 8
ud_b 6.3472757070e-01 0 1  ON 9
  </optVariables>
...
  QMC Execution time = 7.0060820112e+00 secs
```

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

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

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

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

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

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

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

Questions and Exercises

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

11.2.3 DMC timestep extrapolation I: neutral O atom

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

Background on timestep and population control bias

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

$$\hat{\rho} = e^{-t\hat{H}} \tag{11.8}$$

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

$$\langle R|e^{-t\hat{H}}|\Psi_T\rangle = \int dR'\rho(R,R';t)\Psi_T(R') \tag{11.9}$$

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

$$\begin{aligned}\langle R|e^{-tH}|\Psi_T\rangle &= \langle R|e^{-\tau\hat{H}}e^{-\tau\hat{H}}\cdots e^{-\tau\hat{H}}|\Psi_T\rangle & (11.10)\\ &= \int dR_1dR_2\cdots dR_M\rho(R,R_1;\tau)\rho(R_1,R_2;\tau)\cdots\rho(R_{M-1},R_M;\tau)\Psi_T(R_M) & (11.11)\end{aligned}$$

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

$$\rho(R,R';\tau) = e^{D(R,R';\tau)}e^{B(R,R';\tau)} + \mathcal{O}(\tau^2) \tag{11.12}$$

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

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

Timestep extrapolation with QMCPACK

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

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

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

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

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

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

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

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

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

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

The purpose of the VMC run is to provide initial electron positions for each DMC walker. Setting `walkers` = 1 in the VMC block ensures there will be only one VMC walker per execution thread. There will be a total of 512 VMC walkers in this case (see `O.q0.dmc.qsub.in`). We want the electron positions used to initialize the DMC walkers to be decorrelated from one another. A

VMC walker will often decorrelate from its current position after propagating for a few Ha^{-1} in imaginary time (in general this is system dependent). This leads to a rough rule of thumb for choosing `blocks` and `steps` for the VMC run ($\texttt{VWALKERS} = 512$ here):

$$\texttt{VBLOCKS} \times \texttt{VSTEPS} \geq \frac{\texttt{DWALKERS}}{\texttt{VWALKERS}} \frac{5\ \text{Ha}^{-1}}{\texttt{VTIMESTEP}} \tag{11.13}$$

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

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

$$\begin{aligned}
\texttt{timestep}_n &= \texttt{timestep}_{n-1}/2 \\
\texttt{warmupSteps}_n &= \texttt{warmupSteps}_{n-1} \times 2 \\
\texttt{blocks}_n &= \texttt{blocks}_{n-1} \\
\texttt{steps}_n &= \texttt{steps}_{n-1} \times 2
\end{aligned} \tag{11.14}$$

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

Choose an initial DMC timestep and create a sequence of N timesteps according to 11.14. Make N copies of the DMC XML block in the input file

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

Fill in `DWARMUP`, `DBLOCKS`, `DSTEPS`, and `DTIMESTEP` for each DMC run according to 11.14. Submit the DMC timestep extrapolation run to the queue with `submit_O_q0_dmc`. The run should take only a few minutes to complete.

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

Use the `PlotTstepConv.pl` to obtain a linear fit to the timestep data (type "`PlotTstepConv.pl O.q0.dmc.in.xml 40`"). You should see a plot similar to fig. 11.2. The tail end of the text output displays the parameters for the linear fit. The "`a`" parameter is the total energy extrapolated to zero timestep in Hartree units.

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

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

Questions and Exercises

1. What is the $\tau \to 0$ extrapolated value for the total energy?

2. What is the maximum timestep you should use if you want to calculate the total energy to an accuracy of 0.05 eV? For convenience, 1 Ha = 27.2113846 eV.

3. What is the acceptance ratio for this (bias$<$ 0.05 eV) run? Does it follow the rule of thumb for sensible DMC (acceptance ratio $>$ 99%) ?

4. Check the fluctuations in the walker population (`qmca -t -q nw O_q0_dmc*dmc.dat --noac`). Does the population seem to be stable?

5. (Optional) Study population control bias for the oxygen atom. Select a few population sizes (use multiples of 512 to fit cleanly on a single Vesta partition). Copy `O.q0.dmc.in.xml` to a new file and remove all but one DMC run (select a single timestep). Make one copy of the new file for each population, set "`samples`", and choose a unique `id` in `<project/>`. Make submission files similar to `submit_O_q0_dmc` and `O.q0.dmc.qsub.in` and run one job at a time to avoid crowding the lab allocation. Use `qmca` to study the dependence of the

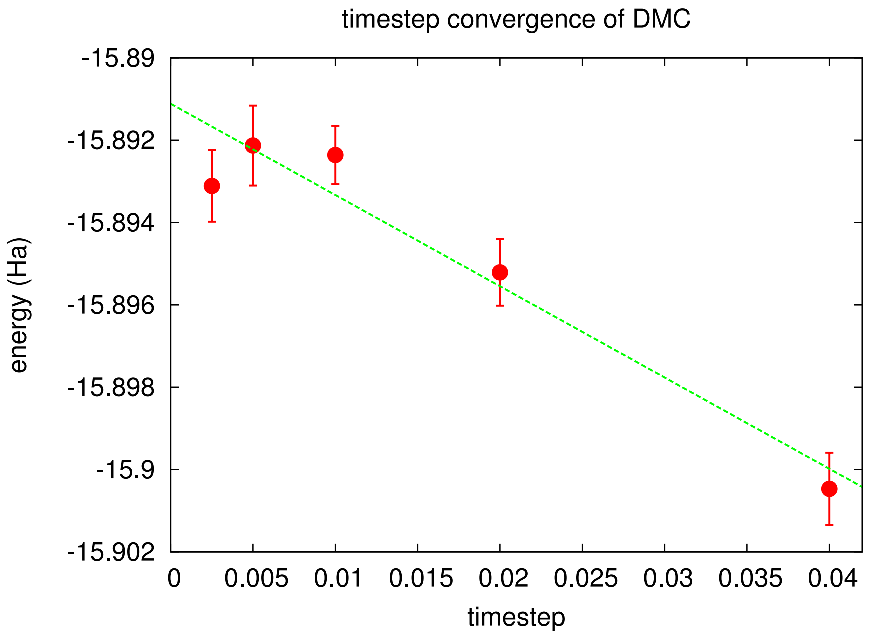


Figure 11.2: Linear fit to DMC timestep data from `PlotTstepConv.pl`.

DMC total energy on the walker population. How large is the bias compared to timestep error? What bias is incurred by following the "rule of thumb" of a couple thousand walkers? Will population control bias generally be an issue for production runs on modern parallel machines?

11.2.4 DMC timestep extrapolation II: IP of oxygen

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

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

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

The process listed above, which excludes additional steps for orbital generation and conversion, can become tedious to perform by hand in production settings where many calculations are often required. For this reason automation tools are introduced for calculations involving the oxygen dimer in section 11.3 of the lab.

Questions and Exercises

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

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

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

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

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

11.3 Testing PP dimer properties: DMC workflow automation

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

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

1. Orbital generation via Quantum Espresso or GAMESS.

2. Conversion of orbital data via `pw2qmcpack.x` or `convert4qmc`.

3. Optimization of Jastrow factors via QMCPACK.

4. DMC calculation via QMCPACK.

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

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

11.3.1 Example Project Suite input

The Project Suite (PS) is driven by simple user-defined scripts that resemble keyword-driven input files. An example PS input file that performs a single VMC calculation is shown below. Take a moment to read it over and especially note the comments (prefixed with "#") explaining most of the contents. If the input syntax is unclear you may want to consult portions of appendix .1, which gives a condensed summary of Python constructs. For more information about the functionality and effective use of the Project Suite, consult `docs/Project_Suite.pdf` first. More information can be found in the user guide distributed with QMCPACK, although examples in this lab series and `Project_Suite.pdf` are more up to date (if `qmcpack` is the location of your QMCPACK distribution, the user guide can be found at `qmcpack/project_suite/documentation/project_suite_user_guide.pdf`).

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

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

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

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

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

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

dimer = generate_physical_system(      # make a dimer system
    type       = 'dimer',                  # system type is dimer
    dimer      = ('O','O'),                # dimer is two oxygen atoms
    separation = 1.2074,                   # separated by 1.2074 Angstrom
    Lbox       = 15.0,                     # simulation box is 15 Angstrom
    units      = 'A',                      # Angstrom is dist. unit
    net_spin   = 2,                        # nup-ndown is 2
    O          = 6                         # pseudo-oxygen has 6 valence el.
    )

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

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

11.3.2 Automated binding curve of the oxygen dimer

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

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

```
linopt1 = linear(
    energy               = 0.0,
    unreweightedvariance = 1.0,
    reweightedvariance   = 0.0,
    timestep             = 0.4,
    samples              = 5000,
    warmupsteps          = 50,
    blocks               = 200,
    substeps             = 1,
    nonlocalpp           = True,
    usebuffer            = True,
    walkers              = 1,
    minwalkers           = 0.5,
    maxweight            = 1e9,
    usedrift             = True,
    minmethod            = 'quartic',
    beta                 = 0.025,
    exp0                 = -16,
    bigchange            = 15.0,
    alloweddifference    = 1e-4,
    stepsize             = 0.2,
    stabilizerscale      = 1.0,
    nstabilizers         = 3
    )
```

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

```
linopt1 = ...

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

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

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

The VMC/DMC method inputs should also look familiar:

```
qmc_calcs = [
    vmc(
        walkers     =   1,
        warmupsteps =  30,
        blocks      =  20,
        steps       =  10,
        substeps    =   2,
        timestep    =  .4,
        samples     = 2048
        ),
    dmc(
        warmupsteps   = 100,
        blocks        = 400,
        steps         =  32,
        timestep      = 0.01,
        nonlocalmoves = True
        )
    ]
```

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

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

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

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

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

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

Shared details such as the run directory, job, pseudopotentials, and orbital file have been omitted

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

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

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

Note that the text inside the `for` loop and the `if` block must be indented by precisely four spaces. If you use Emacs, changes in indentation can be performed easily with `Cntrl-C >` and `Cntrl-C <` after highlighting a block of text (other editors should have similar functionality). If you see something like "`SyntaxError:  invalid syntax`" print to the screen when you run `O_dimer.py` later on, consult the completed file in `oxygen_dimer/reference`.

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

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

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

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

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

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

  starting runs:
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  poll 0  memory 88.54 MB
```

```
    Entering ./scale_1.0 0
      writing input files  0 opt
    Entering ./scale_1.0 0
      sending required files  0 opt
      submitting job  0 opt
    Entering ./scale_1.0 1
      Would have executed:  qsub --mode script --env BG_SHAREDMEMSIZE=32 opt.qsub.in

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

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

Project finished
```

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

```
scale_1.0
 O2.pwscf.h5
 O.BFD.xml
 opt.in.xml
 opt.qsub.in
 qmc.in.xml
 qmc.qsub.in
 sim_opt
    analyzer.p
    input.p
    sim.p
 sim_qmc
     analyzer.p
     input.p
     sim.p
```

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

```

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

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

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

Let's actually submit the optimization and DMC jobs now. Reset the state of the simulations by removing the `sim.p` files ("`rm ./scale*/sim*/sim.p`"), set "`generate_only`" to `0`, and rerun

`O_dimer.py`. It should take about 15 minutes for all the jobs to complete. You may wish to open another terminal to monitor the progress of the individual jobs while the current terminal runs `O_dimer.py` in the foreground. You can begin the first exercise below once the optimization job completes.

Questions and Exercises

1. Evaluate the quality of the optimization at `scale=1.0` using the `qmca` tool. Did the optimization succeed? How does the variance compare with the neutral oxygen atom? Is the wavefunction of similar quality to the atomic case?
2. Evaluate the traces of the local energy and the DMC walker population for each separation distance with the `qmca` tool. Are there any anomalies in the runs? Is the acceptance ratio reasonable? Is the wavefunction of similar quality across all separation distances?
3. Use the `dimer_fit.py` tool located in `oxygen_dimer` to fit the oxygen dimer binding curve. To get the binding energy of the dimer, we will need the DMC energy of the atom. Before performing the fit, answer: What DMC timestep should be used for the oxygen atom results? The tool accepts three arguments ("`O_dimer.py P N E Eerr`"), `P` is the prefix of the DMC input files (should be "`qmc`" at this point), `N` is the order of the fit (use 2 to start), `E` and `Eerr` are your DMC total energy and error bar, respectively for the oxygen atom (in eV). A plot of the dimer data will be displayed and text output will show the DMC equilibrium bond length and binding energy as well as experimental values. How accurately does your fit to the DMC data reproduce the experimental values? What factors affect the accuracy of your results?
4. Refit your data with a fourth-order polynomial. How do your predictions change with a fourth-order fit? Is a fourth-order fit appropriate for the available data?
5. Add the four remaining "`scale`" values to the list in `O_dimer.py` that interpolate between the original set. Perform the DMC calculations and redo the fits. How accurately does your fit to the DMC data reproduce the experimental values? Should this pseudopotential be used in production calculations?
6. (Optional) Perform optimization runs at the extremal separation distances corresponding to `scale=[0.90,1.10]`. Are the individually optimized wavefunctions of significantly better quality than the one imported from `scale=1.00`? Why? What form of Jastrow factor might give an even better improvement?

11.4 (Optional) Running your system with QMCPACK

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

To start out with, you will need pseudopotentials (PP's) for each element in your system in both the UPF (PWSCF) and FSATOM/XML (QMCPACK) formats. A good place to start is the Burkatzki-Filippi-Dolg (BFD) pseudopotential database (http://www.burkatzki.com/pseudos/index.2.html), which we have already used in our study of the oxygen atom. The database does not contain PP's for the 4th and 5th row transition metals or any of the lanthanides or actinides. If you need a PP that is not in the BFD database, you may need to generate and test one manually (*e.g.* with OPIUM, http://opium.sourceforge.net/). Otherwise, use `ppconvert` as outlined in section 11.2.1 to obtain PP's in the formats used by PWSCF and QMCPACK. Enter the `your_system` lab directory and place the converted PP's in `your_system/pseudopotentials`.

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

Beyond pseudopotentials, all that is required to get started are the atomic positions and the dimensions/shape of the simulation cell. The PS file `example.py` illustrates how to setup PWSCF and QMCPACK input files by providing minimal information regarding the physical system (an 8-atom cubic cell of diamond in the example). Most of the contents should be familiar from your experience with the automated calculations of the oxygen dimer binding curve in section 11.3 (if you've skipped ahead you may want to skim that section for relevant information). The most important change is the expanded description of the physical system:

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

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

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

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

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

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

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

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

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

Alternatively the runs can be submitted by hand:

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

(wait until JOB DONE appears in scf.output)

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

Once the conversion process has finished the orbitals should be located in the file `diamond_vmc/pwscf_output/pw` Open `diamond_vmc/vmc.in.xml` and replace "`MISSING.h5`" with "`./pwscf_output/pwscf.pwscf.h5`". Next submit the VMC run:

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

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

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

.1 Basic Python constructs

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

Intrinsic types: `int, float, str`

```
#this is a comment
i=5                     # integer
f=3.6                   # float
s='quantum/monte/carlo' # string
```

```
n=None                  # represents "nothing"

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>>
>>> p=o
>>>
>>> p.a=7
>>>
>>> print o
  a               = 7
  b               = 6

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

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

Conditional Statements: `if/elif/else`

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

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

Iteration: `for`

```
from generic import obj

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

print s                  # s is 21

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

print s                  # s is 6

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

print s                  # s is 11

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

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

Functions: `def`, argument syntax

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

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


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

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

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

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

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

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

Appendix A

Contributing to the Manual

This section briefly describes how to contribute to the manual. It is primarily "by developers, for developers". This section should iterate until a consistent view on style/contents is reached.

Desirable:

- Use the table templates below when describing XML input.
- Place unformatted text targeted at developers in comments. Include generously.
- Encapsulate formatted text aimed at developers (like this entire chapter), in `\dev{}`. Text encapsulated in this way will be removed from the user version of the manual by editing the definition of `\dev` in `qmcpack_manual.tex`. Existing but deprecated or partially functioning features fall in this category.

Missing sections (these are opinions, not decided priorities):

- Description of XML input in general. Discuss XML format, use of attributes and `<parameter/>`'s in general, case sensitivity (input is generally case sensitive), and behavior of QMCPACK when unrecognized XML elements are encountered (they are generally ignored without notification).
- Overview of the input file in general, broad structure, and at least one full example that works in isolation.

Information currently missing for a complete reference specification:

- Noting how many instances of each child element are allowed. Examples: `simulation`–1 only, `method`–1 or more, `jastrow`–0 or more.

Below are template tables for describing XML elements in reference fashion. A number of examples can be found in *e.g.* Chapter 7. Preliminary style is (please weigh in with opinions): typewriter text (`\texttt{}`) for XML element, attribute, and parameter names, normal text for literal information in datatype/values/default columns, bold (`\textbf{}`) text if an attribute/parameter must take on a particular value (values column), italics (`\textit{}`) for descriptive (non-literal) information in the values column (e.g. *anything*, *non-zero*, etc.), required/optional attributes/parameters noted by `some_attr`^{r}/`some_attr`^{o} superscripts. Valid datatypes are text, integer, real, boolean, and arrays of each. Fixed lengh arrays can be noted, *e.g.* by "real array(3)".

Template for a generic XML element:

`generic` element				
parent elements:	`parent1 parent2`			
child elements:	`child1 child2 child3 ...`			
attributes				
name	**datatype**	**values**	**default**	**description**
`attr1`^{r}	text			
`attr2`^{r}	integer			
`attr3`^{o}	real			
`attr4`^{o}	boolean			
`attr5`^{o}	text array			
`attr6`^{o}	integer array			
`attr7`^{o}	real array			
`attr8`^{o}	boolean array			
parameters				
name	**datatype**	**values**	**default**	**description**
`param1`^{r}	text			
`param2`^{r}	integer			
`param3`^{o}	real			
`param4`^{o}	boolean			
`param5`^{o}	text array			
`param6`^{o}	integer array			
`param7`^{o}	real array			
`param8`^{o}	boolean array			
body text				
	Long form description of body text format			

"Factory" elements are XML elements that share a tag, but whose contents change based on the value an attribute (or sometimes multiple attributes take). The attribute(s) that determine the allowed contents is referred to below as the "type selector" (*e.g.* for `<estimator/>` elements, the type selector is usually the `type` attribute). These types of elements are frequently encountered as they correspond (sometimes loosely, sometimes literally) to polymorphic classes in QMCPACK that are built in “factories”. This name is true to the underlying code, but may be obscure to the general user (is there a better name to retain the general meaning?).

The template below should be provided each time a new “factory” type is encountered (like `<estimator/>`). The table lists all types of possible elements (see “type options” below) and any attributes that are common to all possible related elements. Specific “derived” elements are then described one at a time with the template above, noting the type selector in addition to the XML tag (*e.g.* “`estimator type=density` element”).

Template for shared information about “factory” elements.

`generic` factory element				
parent elements:	`parent1 parent2`			
child elements:	`child1 child2 child3 ...`			
type selector:	`some` attribute			
type options :	Selection1			
	Selection2			
	Selection3			
	...			
shared attributes:				
name	**datatype**	**values**	**default**	**description**
`attr1`	text			
`attr2`	integer			
...				

Appendix B

References

Bibliography

[1] K. P. Esler, J. Kim, D. M. Ceperley, and L. Shulenburger, “Accelerating quantum monte carlo simulations of real materials on gpu clusters,” *Computing in Science and Engineering*, vol. 14, no. 1, pp. 40–51, 2012.

[2] V. Natoli and D. M. Ceperley, “An optimized method for treating long-range potentials,” *Journal of Computational Physics*, vol. 117, no. 1, pp. 171 – 178, 1995.

[3] L. Mitas, E. L. Shirley, and D. M. Ceperley, “Nonlocal pseudopotentials and diffusion monte carlo,” *The Journal of Chemical Physics*, vol. 95, no. 5, pp. 3467–3475, 1991.

[4] S. Chiesa, D. M. Ceperley, R. M. Martin, and M. Holzmann, “Finite-size error in many-body simulations with long-range interactions,” *Phys. Rev. Lett.*, vol. 97, p. 076404, Aug 2006.

[5] J. T. Krogel, M. Yu, J. Kim, and D. M. Ceperley, “Quantum energy density: Improved efficiency for quantum monte carlo calculations,” *Phys. Rev. B*, vol. 88, p. 035137, Jul 2013.

[6] J. T. Krogel, J. Kim, and F. A. Reboredo, “Energy density matrix formalism for interacting quantum systems: Quantum monte carlo study,” *Phys. Rev. B*, vol. 90, p. 035125, Jul 2014.