QMC Manual

Release 0.0.1

QMCPACK Developers

Jun 08, 2020

CONTENTS:

1 Introduction **3**
- 1.1 Quickstart and a first QMCPACK calculation . . . 3
- 1.2 Authors and History . . . 4
- 1.3 Support and Contacting the Developers . . . 5
- 1.4 Performance . . . 5
- 1.5 Open Source License . . . 6
- 1.6 Contributing to QMCPACK . . . 7
- 1.7 QMCPACK Roadmap . . . 7

2 Features of QMCPACK **9**
- 2.1 Production-level features . . . 9
- 2.2 SoA optimizations and improved algorithms . . . 10
- 2.3 Supported GPU features . . . 10
- 2.4 Beta test features . . . 11

3 Obtaining, installing, and validating QMCPACK **13**
- 3.1 Installation steps . . . 13
- 3.2 Obtaining the latest release version . . . 14
- 3.3 Obtaining the latest development version . . . 14
- 3.4 Prerequisites . . . 14
- 3.5 C++ 14 standard library . . . 15
- 3.6 Building with CMake . . . 15
- 3.7 Installation instructions for common workstations and supercomputers . . . 22
- 3.8 Installing via Spack . . . 28
- 3.9 Testing and validation of QMCPACK . . . 34
- 3.10 Automated testing of QMCPACK . . . 38
- 3.11 Building ppconvert, a pseudopotential format converter . . . 38
- 3.12 Installing and patching Quantum ESPRESSO . . . 40
- 3.13 How to build the fastest executable version of QMCPACK . . . 40
- 3.14 Troubleshooting the installation . . . 41

4 Running QMCPACK **43**
- 4.1 Command line options . . . 43
- 4.2 Input files . . . 43
- 4.3 Output files . . . 44
- 4.4 Running in parallel with MPI . . . 44
- 4.5 Using OpenMP threads . . . 44
- 4.6 Running on GPU machines . . . 46

5 Units used in QMCPACK **49**

6 Input file overview **51**
6.1 Project . . . 53
6.2 Random number initialization . . . 53

7 Specifying the system to be simulated **55**
7.1 Specifying the Simulation Cell . . . 55
7.2 Specifying the particle set . . . 57

8 Trial wavefunction specificaion **61**
8.1 Introduction . . . 61
8.2 Single determinant wavefunctons . . . 61
8.3 Single-particle orbitals . . . 62
8.4 Jastrow Factors . . . 72
8.5 Multideterminant wavefunctions . . . 84
8.6 Backflow Wavefunctions . . . 84
8.7 Finite-difference linear response wave functions . . . 87
8.8 Gaussian Product Wavefunction . . . 89

9 Hamiltonian and Observables **91**
9.1 The Hamiltonian . . . 91
9.2 Pair potentials . . . 92
9.3 General estimators . . . 98
9.4 Forward-Walking Estimators . . . 114
9.5 “Force” estimators . . . 115

10 Quantum Monte Carlo Methods **117**
10.1 Variational Monte Carlo . . . 118
10.2 Wavefunction optimization . . . 120

Bibliography **125**

QMCPACK

CHAPTER

ONE

INTRODUCTION

QMCPACK is an open-source, high-performance electronic structure code that implements numerous Quantum Monte Carlo (QMC) 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 is designed with the modularity afforded by object-oriented programming. It makes extensive use of template metaprogramming to achieve high computational efficiency. Because of the modular architecture, the addition of new wavefunctions, algorithms, and observables is relatively straightforward. For parallelization, QMCPACK uses 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 uses 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 as a guide to installing and running it. Over time this manual will be expanded to include 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

In case you are keen to get started, this section describes how to quickly build and run QMCPACK on a 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 might require hints on the locations of libraries and which versions to use, typical of any code; see *Obtaining, installing, and validating QMCPACK*. *Installation instructions for common workstations and supercomputers* includes complete examples of installations 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 file.
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 *Obtaining, installing, and validating QMCPACK* for instructions and specific build instructions for common systems.

6. Build QMCPACK: `make` or `make -j 16`; use the latter for a faster parallel build on a system using, for example, 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/H2O`.
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.

If you have Python and matplotlib installed, you can use the analysis utility to produce statistics and plots of the data. See analyzing for information on analyzing QMCPACK data.

```
export PATH=$PATH:location-of-qmcpack/nexus/bin
export PYTHONPATH=$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, time step, 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 Professor David Ceperley at the University of Illinois at Urbana-Champaign, with later contributations being made at Oak Ridge National Laboratory (ORNL). Over the years, many others have contributed, particularly students and researchers in the groups of Professor David Ceperley and Professor Richard M. Martin, as well as staff at Lawrence Livermore National Laboratory, Sandia National Laboratories, Argonne National Laboratory, and ORNL.

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 the following:

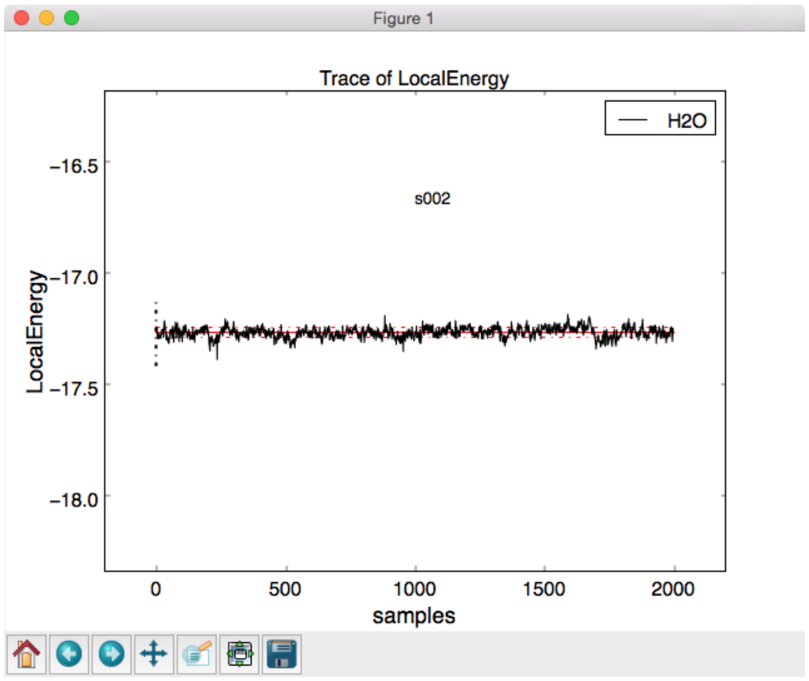


Fig. 1.1: Trace of walker energies produced by the qmca tool for a simple water molecule example.

- "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 US National Science Foundation
- Materials Computation Center (MCC), supported by the US 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 at 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 (MC) algorithms, is highly parallel, and is 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 central processing units (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 at Urbana-Champaign/National Center for Supercomputing Applications (UIUC/NCSA) Open Source License.

Copyright is generally believed to remain with the authors of the individual sections of code. See the various notations in the source code as well as the code history.

1.6 Contributing to QMCPACK

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

A guide to developing for QMCPACK, including instructions on how to work with GitHub and make pull requests (contributions) to the main source are listed on the QMCPACK GitHub wiki: https://github.com/QMCPACK/qmcpack/wiki.

Contributions are made under the same license as QMCPACK, the UIUC/NCSA open source license. If this is problematic, please discuss with a developer.

Please note the following guidelines for contributions:

- Additions should be fully synchronized with the latest release version and ideally the latest develop branch on github. Merging of code developed on older versions is error prone.
- Code should be cleanly formatted, commented, portable, and accessible to other programmers. That is, 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 in the following sections. 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 continue to improve the accessibility and usability of QMCPACK through 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 QMC-PACK in the near future.

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

1.7.2 Documentation

This manual 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.

Because of its history as a research code, QMCPACK contains a variety of additional QMC methods, trial wavefunction forms, potentials, etc., that, although not critical, might 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 a historical Jastrow form is needed for benchmarking).

CHAPTER

TWO

FEATURES OF QMCPACK

2.1 Production-level features

The following list contains the main production-level features of QMCPACK. If you do not see a specific feature that you are interested in, see the remainder of this manual and ask whether that specific feature is available or can be quickly brought to the full production level.

- Variational Monte Carlo (VMC)
- Diffusion Monte Carlo (DMC)
- Reptation Monte Carlo
- Single and multideterminant Slater Jastrow wavefunctions
- Wavefunction updates using optimized multideterminant algorithm of Clark et al.
- Backflow wavefunctions
- One, two, and three-body Jastrow factors
- Excited state calculations via flexible occupancy assignment of Slater determinants
- All electron and nonlocal pseudopotential calculations
- Casula T-moves for variational evaluation of nonlocal pseudopotentials (non-size-consistent and size-consistent variants)
- Wavefunction optimization using the "linear method" of Umrigar and coworkers, with arbitrary mix of variance and energy in the objective function
- Blocked, low memory adaptive shift optimizer of Zhao and Neuscamman
- Gaussian, Slater, plane-wave, and real-space spline basis sets for orbitals
- Interface and conversion utilities for plane-wave wavefunctions from Quantum Espresso (Plane-Wave Self-Consistent Field package [PWSCF])
- Interface and conversion utilities for Gaussian-basis wavefunctions from GAMESS
- Easy extension and interfacing to other electronic structure codes via standardized XML and HDF5 inputs
- MPI parallelism
- Fully threaded using OpenMP
- GPU (NVIDIA CUDA) implementation (limited functionality)
- HDF5 input/output for large data
- Nexus: advanced workflow tool to automate all aspects of QMC calculation from initial DFT calculations through to final analysis

- Analysis tools for minimal environments (Perl only) through to Python-based environments with graphs produced via matplotlib (included with Nexus)

2.2 SoA optimizations and improved algorithms

The Structure-of-Arrays (SoA) optimizations [MLC+17] are a set of improved data layouts facilitating vectorization on modern CPUs with wide SIMD units. **For many calculations and architectures, the SoA implementation more than doubles the speed of the code.** This so-called SoA implementation replaces the older, less efficient Array-of-Structures (AoS) code and can be enabled or disabled at compile time. The memory footprint is also reduced in the SoA implementation by better algorithms, enabling more systems to be run.

The SoA build was made the default for QMCPACK v3.7.0. As described in *Configuration Options*, the SoA implementation can be disabled by configuring with `-DENABLE_SOA=0`.

The SoA code path currently does *not* support:

- Backflow wavefunctions
- Many observables

The code should abort with a message referring to AoS vs SoA features if any unsupported feature is invoked. In this case the AoS build should be used by configuring with `-DENABLE_SOA=0`. In addition, please inform the developers via GitHub or Google Groups so that porting these features can be prioritized.

Core features are heavily tested in both SoA and AoS versions. If using untested and noncore features in the SoA code, please compare the AoS and SoA results carefully.

2.3 Supported GPU features

The GPU implementation supports multiple GPUs per node, with MPI tasks assigned in a round-robin order to the GPUs. Currently, for large runs, 1 MPI task should be used per GPU per node. For smaller calculations, use of multiple MPI tasks per GPU might yield improved performance. Supported GPU features:

- VMC, wavefunction optimization, DMC.
- Periodic and open boundary conditions. Mixed boundary conditions are not yet supported.
- Wavefunctions:
    1. Single Slater determinants with 3D B-spline orbitals. Twist-averaged boundary conditions and complex wavefunctions are fully supported. Gaussian type orbitals are not yet supported.
    2. Hybrid 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.
    3. One-body and two-body Jastrows represented as 1D B-splines. Three-body Jastrow functions are not yet supported.
- Semilocal (nonlocal and local) pseudopotentials, Coulomb interaction (electron-electron, electron-ion), and model periodic Coulomb (MPC) interaction.

2.4 Beta test features

This section describes developmental features in QMCPACK that might be ready for production but that require additional testing, features, or documentation to be ready for general use. We describe them here because they offer significant benefits and are well tested in specific cases.

2.4.1 Auxiliary-Field Quantum Monte Carlo

The orbital-space Auxiliary-Field Quantum Monte Carlo (AFMQC) method is now available in QMCPACK. The main input for the code is the matrix elements of the Hamiltonian in a given single particle basis set, which must be produced from mean-field calculations such as Hartree-Fock or density functional theory. The code and many features are in development. Check the latest version of QMCPACK for an up-to-date description of available features. A partial list of the current capabilities of the code follows. For a detailed description of the available features, see afqmc.

- Phaseless AFQMC algorithm of Zhang et al. (S. Zhang and H. Krakauer. 2003. "Quantum Monte Carlo Method using Phase-Free Random Walks with Slater Determinants." *PRL* 90: 136401).
- "Hybrid" and "local energy" propagation schemes.
- Hamiltonian matrix elements from (1) Molpro's FCIDUMP format (which can be produced by Molpro, PySCF, and VASP) and (2) internal HDF5 format produced by PySCF (see AFQMC section below).
- AFQMC calculations with RHF (closed-shell doubly occupied), ROHF (open-shell doubly occupied), and UHF (spin polarized broken symmetry) symmetry.
- Single and multideterminant trial wavefunctions. Multideterminant expansions with either orthogonal or nonorthogonal determinants.
- Fast update scheme for orthogonal multideterminant expansions.
- Distributed propagation algorithms for large systems. Enables calculations where data structures do not fit on a single node.
- Complex implementation for PBC calculations with complex integrals.
- Sparse representation of large matrices for reduced memory usage.
- Mixed and back-propagated estimators.
- Specialized implementation for solids with k-point symmetry (e.g. primitive unit cells with kpoints).
- Efficient GPU implementation (currently limited to solids with k-point symmetry).

2.4.2 Sharing of spline data across multiple GPUs

Sharing of GPU spline data enables distribution of the data across multiple GPUs on a given computational node. For example, on a two-GPU-per-node system, each GPU would have half of the orbitals. This allows use of larger overall spline tables than would fit in the memory of individual GPUs and potentially up to the total GPU memory on a node. To obtain high performance, large electron counts or a high-performing CPU-GPU interconnect is required.

To use this feature, the following needs to be done:

- The CUDA Multi-Process Service (MPS) needs to be used (e.g., on OLCF Summit/SummitDev use "-alloc_flags gpumps" for bsub). If MPI is not detected, sharing will be disabled.
- CUDA_VISIBLE_DEVICES needs to be properly set to control each rank's visible CUDA devices (e.g., on OLCF Summit/SummitDev create a resource set containing all GPUs with the respective number of ranks with "jsrun –task-per-rs Ngpus -g Ngpus").

- In the determinant set definition of the <wavefunction> section, the "gpusharing" parameter needs to be set (i.e., <determinantset gpusharing="yes">). See *Spline basis sets*.

CHAPTER

THREE

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.

3.1 Installation steps

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

1. Download the source code from *Obtaining the latest release version* or *Obtaining the latest development version*.
2. Verify that you have the required compilers, libraries, and tools installed (*Prerequisites*).
3. If you will use Quantum ESPRESSO, download and patch it. The patch adds the pw2qmcpack utility (*Installing and patching Quantum ESPRESSO*).
4. Run the cmake configure step and build with make (*Building with CMake* and *Quick build instructions (try first)*). Examples for common systems are given in *Installation instructions for common workstations and supercomputers*.
5. Run the tests to verify QMCPACK (*Testing and validation of QMCPACK*).
6. Build the ppconvert utility in QMCPACK (*Building ppconvert, a pseudopotential format converter*).

Hints for high performance are in *How to build the fastest executable version of QMCPACK*. Troubleshooting suggestions are in *Troubleshooting the installation*.

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

3.2 Obtaining the latest release version

Major releases of QMCPACK are distributed from http://www.qmcpack.org. Because these versions undergo the most testing, we encourage using them for all production calculations unless there are specific reasons not to do so.

Releases are usually compressed tar files that indicate the version number, date, and often the source code revision control number corresponding to the release. To obtain the latest release:

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

Releases can also be obtained from the 'master' branch of the QMCPACK git repository, similar to obtaining the development version (*Obtaining the latest development version*).

3.3 Obtaining the latest development version

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

```
git clone https://github.com/QMCPACK/qmcpack.git
```

Once checked out, updates can be made via the standard `git pull`.

The 'develop' branch of the git repository contains the day-to-day development source with the latest updates, bug fixes, etc. This version might 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 might not be fully consistent with the online documentation. We attempt to keep the development version fully working. However, please be sure to run 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.

3.4 Prerequisites

The following items 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 accessed 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 *How to build the fastest executable version of QMCPACK* for performance suggestions.

- C/C++ compilers such as GNU, Clang, Intel, and IBM XL. C++ compilers are required to support the C++ 14 standard. Use of recent ("current year version") compilers is strongly encouraged.
- An MPI library such as OpenMPI (http://open-mpi.org) or a vendor-optimized MPI.
- BLAS/LAPACK, numerical, and linear algebra libraries. Use platform-optimized libraries where available, such as Intel MKL. ATLAS or other optimized open source libraries can 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/). Good performance at large scale requires parallel version >= 1.10.
- BOOST, peer-reviewed portable C++ source libraries (http://www.boost.org). Minimum version is 1.61.0.

- 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.

3.5 C++ 14 standard library

The C++ standard consists of language features—which are implemented in the compiler—and library features—which are implemented in the standard library. GCC includes its own standard library and headers, but many compilers do not and instead reuse those from an existing GCC install. Depending on setup and installation, some of these compilers might not default to using a GCC with C++ 14 headers (e.g., GCC 4.8 is common as a base system compiler, but its standard library only supports C++ 11).

The symptom of having header files that do not support the C++ 14 standard is usually compile errors involving standard include header files. Look for the GCC library version, which should be present in the path to the include file in the error message, and ensure that it is 5.0 or greater. To avoid these errors occurring at compile time, QMCPACK tests for a C++ 14 standard library during configuration and will halt with an error if one is not found.

At sites that use modules, running is often sufficient to load a newer GCC and resolve the issue.

3.5.1 Intel compiler

The Intel compiler version must be 18 or newer. The version 17 compiler cannot compile some of the C++ 14 constructs in the code.

If a newer GCC is needed, the `-cxxlib` option can be used to point to a different GCC installation. (Alternately, the `-gcc-name` or `-gxx-name` options can be used.) Be sure to pass this flag to the C compiler in addition to the C++ compiler. This is necessary because CMake extracts some library paths from the C compiler, and those paths usually also contain to the C++ library. The symptom of this problem is C++ 14 standard library functions not found at link time.

3.6 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. The minimum required version of CMake is 3.6, which is the oldest version to support correct application of C++ 14 flags for the Intel compiler. Most computer installations have a sufficiently recent CMake, though it might not be the default.

If no appropriate version CMake is available, building it from source is straightforward. Download a version from https://cmake.org/download/ and unpack the files. Run `./bootstrap` from the CMake directory, and then run `make` when that finishes. The resulting CMake executable will be in the directory. The executable can be run directly from that location.

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 environment or CMake variables. When the libraries are installed in standard locations (e.g., /usr, /usr/local), there is no need to set environment or CMake variables for the packages.

3.6.1 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. For example, 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. That is, check the following details. We give examples for a number of common systems in *Installation instructions for common workstations and supercomputers*.

3.6.2 Environment variables

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

```
CXX                 C++ compiler
CC                  C Compiler
MKL_ROOT            Path for MKL
HDF5_ROOT           Path for HDF5
BOOST_ROOT          Path for Boost
FFTW_HOME           Path for FFTW
```

3.6.3 Configuration Options

In addition to reading the environment 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 environment 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)
QMC_MIXED_PRECISION   Build the mixed precision (mixing double/float) version
                      (1:yes (GPU default), 0:no (CPU default)).
                      The CPU support is experimental.
                      Use float and double for base and full precision.
                      The GPU support is quite mature.
                      Use always double for host side base and full precision
                      and use float and double for CUDA base and full precision.
ENABLE_SOA            Enable data layout and algorithm optimizations using
                      Structure-of-Array (SoA) datatypes (1:yes (default), 0:no).
ENABLE_TIMERS         Enable fine-grained timers (1:yes, 0:no (default)).
                      Timers are off by default to avoid potential slowdown in␣
↪small
                      systems. For large systems (100+ electrons) there is no␣
↪risk.
```

- 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
CMAKE_INSTALL_PREFIX Set the install location (if using the optional install step)
INSTALL_NEXUS        Install Nexus alongside QMCPACK (if using the optional␣
↪install step)
```

- Additional QMCPACK build options

```
QE_BIN                     Location of Quantum Espresso binaries including␣
↪pw2qmcpack.x
```

(continues on next page)

(continued from previous page)

```
QMC_DATA                    Specify data directory for QMCPACK performance and␣
↪integration tests
QMC_INCLUDE                 Add extra include paths
QMC_EXTRA_LIBS              Add extra link libraries
QMC_BUILD_STATIC            Add -static flags to build
QMC_SYMLINK_TEST_FILES      Set to zero to require test files to be copied. Avoids␣
↪space
                            saving default use of symbolic links for test files.␣
↪Useful
                            if the build is on a separate filesystem from the␣
↪source, as
                            required on some HPC systems.
QMC_VERBOSE_CONFIGURATION Print additional information during cmake configuration
                            including details of which tests are enabled.
```

- Intel MKL related

```
ENABLE_MKL              Enable Intel MKL libraries (1:yes (default for intel␣
↪compiler),
                                                   0:no (default otherwise)).
MKL_ROOT                Path to MKL libraries (only necessary for non intel compilers
                        or intel without standard environment variables.)
                        One of the above environment variables can be used.
```

- libxml2 related

```
LIBXML2_INCLUDE_DIR   Include directory for libxml2

LIBXML2_LIBRARY       Libxml2 library
```

- HDF5 related

```
HDF5_PREFER_PARALLEL 1(default for MPI build)/0, enables/disable parallel HDF5␣
↪library searching.
ENABLE_PHDF5         1(default for parallel HDF5 library)/0, enables/disable␣
↪parallel collective I/O.
```

- FFTW related

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

- CTest related

```
MPIEXEC_EXECUTABLE     Specify the mpi wrapper, e.g. srun, aprun, mpirun, etc.
MPIEXEC_NUMPROC_FLAG   Specify the number of mpi processes flag,
                       e.g. "-n", "-np", etc.
MPIEXEC_PREFLAGS       Flags to pass to MPIEXEC_EXECUTABLE directly before the␣
↪executable to run.
```

- LLVM/Clang Developer Options

```
LLVM_SANITIZE_ADDRES  link with the Clang address sanitizer library
LLVM_SANITIZE_MEMORY  link with the Clang memory sanitizer library
```

Clang address sanitizer library

Clang memory sanitizer library

See LLVM-Sanitizer-Libraries for more information.

3.6.4 Installation from CMake

Installation is optional. The QMCPACK executable can be run from the `bin` directory in the build location. If the install step is desired, run the `make install` command to install the QMCPACK executable, the converter, and some additional executables. Also installed is the `qmcpack.settings` file that records options used to compile QMCPACK. Specify the `CMAKE_INSTALL_PREFIX` CMake variable during configuration to set the install location.

3.6.5 Role of QMC_DATA

QMCPACK includes a variety of optional performance and integration tests that use research quality wavefunctions to obtain meaningful performance and to more thoroughly test the code. The necessarily large input files are stored in the location pointed to by QMC_DATA (e.g., scratch or long-lived project space on a supercomputer). These files are not included in the source code distribution to minimize size. The tests are activated if CMake detects the files when configured. See tests/performance/NiO/README, tests/solids/NiO_afqmc/README, tests/performance/C-graphite/README, and tests/performance/C-molecule/README for details of the current tests and input files and to download them.

Currently the files must be downloaded via https://anl.box.com/s/yxz1ic4kxtdtgpva5hcmlom9ixfl3v3c.

The layout of current complete set of files is given below. If a file is missing, the appropriate performance test is skipped.

```
QMC_DATA/C-graphite/lda.pwscf.h5
QMC_DATA/C-molecule/C12-e48-pp.h5
QMC_DATA/C-molecule/C12-e72-ae.h5
QMC_DATA/C-molecule/C18-e108-ae.h5
QMC_DATA/C-molecule/C18-e72-pp.h5
QMC_DATA/C-molecule/C24-e144-ae.h5
QMC_DATA/C-molecule/C24-e96-pp.h5
QMC_DATA/C-molecule/C30-e120-pp.h5
QMC_DATA/C-molecule/C30-e180-ae.h5
QMC_DATA/C-molecule/C60-e240-pp.h5
QMC_DATA/NiO/NiO-fcc-supertwist111-supershift000-S1.h5
QMC_DATA/NiO/NiO-fcc-supertwist111-supershift000-S2.h5
QMC_DATA/NiO/NiO-fcc-supertwist111-supershift000-S4.h5
QMC_DATA/NiO/NiO-fcc-supertwist111-supershift000-S8.h5
QMC_DATA/NiO/NiO-fcc-supertwist111-supershift000-S16.h5
QMC_DATA/NiO/NiO-fcc-supertwist111-supershift000-S32.h5
QMC_DATA/NiO/NiO-fcc-supertwist111-supershift000-S64.h5
QMC_DATA/NiO/NiO-fcc-supertwist111-supershift000-S128.h5
QMC_DATA/NiO/NiO-fcc-supertwist111-supershift000-S256.h5
QMC_DATA/NiO/NiO_afm_fcidump.h5
QMC_DATA/NiO/NiO_afm_wfn.dat
QMC_DATA/NiO/NiO_nm_choldump.h5
```

3.6.6 Configure and build using CMake and make

To configure and build QMCPACK, 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 when the file system 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.

3.6.7 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_ROOT=/usr/local/intel/mkl/10.0.3.020
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
```

3.6.8 Build scripts

We recommended creating a helper script that contains the configure line for CMake. This is particularly useful when avoiding environment variables, packages are installed in custom locations, or 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 the script is called. Deleting all the files in the build directory is also acceptable. If you do so we recommend adding 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 much more complex than usually needed for comprehensiveness:

```
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*
```

(continues on next page)

(continued from previous page)

```
cmake                                                   \
  -D CMAKE_BUILD_TYPE=Debug                            \
  -D LIBXML2_INCLUDE_DIR=/usr/include/libxml2        \
  -D LIBXML2_LIBRARY=/usr/lib/x86_64-linux-gnu/libxml2.so \
  -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               \
  ..
```

3.6.9 Using vendor-optimized numerical libraries (e.g., Intel MKL)

Although QMC does not make extensive use of linear algebra, use of vendor-optimized libraries is strongly recommended for highest performance. BLAS routines are used in the Slater determinant update, the VMC wavefunction optimizer, and to apply orbital coefficients in local basis calculations. Vectorized math functions are also beneficial (e.g., for the phase factor computation in solid-state calculations). CMake is generally successful in finding these libraries, but specific combinations can require additional hints, as described in the following:

Using Intel MKL with non-Intel compilers

To use Intel MKL with, e.g. an MPICH wrapped gcc:

```
cmake \
  -DCMAKE_C_COMPILER=mpicc -DCMAKE_CXX_COMPILER=mpicxx \
  -DENABLE_MKL=1 -DMKL_ROOT=$MKLROOT/lib \
  ..
```

MKLROOT is the directory containing the MKL binary, examples, and lib directories (etc.) and is often /opt/intel/mkl.

Serial or multithreaded library

Vendors might provide both serial and multithreaded versions of their libraries. Using the right version is critical to QMCPACK performance. QMCPACK makes calls from both inside and outside threaded regions. When being called from outside an OpenMP parallel region, the multithreaded version is preferred for the possibility of using all the available cores. When being called from every thread inside an OpenMP parallel region, the serial version is preferred for not oversubscribing the cores. Fortunately, nowadays the multithreaded versions of many vendor libraries (MKL, ESSL) are OpenMP aware. They use only one thread when being called inside an OpenMP parallel region. This behavior meets exactly both QMCPACK needs and thus is preferred. If the multithreaded version does not provide this feature of dynamically adjusting the number of threads, the serial version is preferred. In addition, thread safety is required no matter which version is used.

3.6.10 Cross compiling

Cross compiling is often difficult but is required on supercomputers with distinct host and compute processor generations or architectures. QMCPACK tried to do its best with CMake to facilitate cross compiling.

- On a machine using a Cray programming environment, we rely on compiler wrappers provided by Cray to correctly set architecture-specific flags. The CMake configure log should indicate that a Cray machine was detected.
- If not on a Cray machine, by default we assume building for the host architecture (e.g., -xHost is added for the Intel compiler and -march=native is added for GNU/Clang compilers).
- If -x/-ax or -march is specified by the user in CMAKE_C_FLAGS and CMAKE_CXX_FLAGS, we respect the user's intention and do not add any architecture-specific flags.

The general strategy for cross compiling should therefore be to manually set CMAKE_C_FLAGS and CMAKE_CXX_FLAGS for the target architecture. Using `make VERBOSE=1` is a useful way to check the final compilation options. If on a Cray machine, selection of the appropriate programming environment should be sufficient.

3.7 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 might require small modifications to these recipes. See *How to build the fastest executable version of QMCPACK* for key points to check to obtain highest performance and *Troubleshooting the installation* for troubleshooting hints.

3.7.1 Installing on Ubuntu Linux or other apt-get–based distributions

The following is designed to obtain a working QMCPACK build on, for example, 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 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
```

3.7.2 Installing on CentOS Linux or other yum-based distributions

The following is designed to obtain a working QMCPACK build on, for example, 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 complicated only by the need to install another repository to obtain HDF5 packages that 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++ 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 set up repoforge as a source for the HDF5 package, go to http://repoforge.org/use. Install the appropriate up-to-date release package for your operating system. By default, CentOS Firefox will offer to run the installer. The CentOS 6.5 settings were still usable for HDF5 on CentOS 7 in 2016, 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
```

3.7.3 Installing on Mac OS X using Macports

These instructions assume a fresh installation of macports and use the gcc 6.1 compiler. Older versions are fine, but it is vital to ensure that matching compilers and libraries are used for all packages and to force use of what is installed in /opt/local. Performance should be very reasonable. Note that we use the Apple-provided Accelerate framework for optimized BLAS.

Follow the Macports install instructions at 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 gcc6
sudo port select gcc mp-gcc6
sudo port install openmpi-devel-gcc6
sudo port select --set mpi openmpi-devel-gcc61-fortran

sudo port install fftw-3 +gcc6
sudo port install libxml2
sudo port install cmake
sudo post install boost +gcc6
sudo port install hdf5 +gcc6
```

(continues on next page)

(continued from previous page)

```
sudo port select --set python python27
sudo port install py27-numpy +gcc6
sudo port install py27-matplotlib  #For graphical plots with qmca
```

QMCPACK build:

```
cd build
cmake -DCMAKE_C_COMPILER=mpicc -DCMAKE_CXX_COMPILER=mpiCXX ..
make -j 6 # Adjust for available core count
ls -l bin/qmcpack
```

Cmake should pickup the versions of HDF5 and libxml (etc.) installed in /opt/local by macports. If you have other copies of these libraries installed and wish to force use of a specific version, use the environment variables detailed in *Environment variables*.

This recipe was verified on July 1, 2016, on a Mac running OS X 10.11.5 "El Capitain."

3.7.4 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 10.12 "Sierra" in December 2017. 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 # installs gcc 7.2.0 on 2017-12-19
   export HOMEBREW_CXX=g++-7
   export HOMEBREW_CC=gcc-7
   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
   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, OSX will request approval of the network connection for each executable.

   ```
   ctest -R short -LE unstable
   ```

3.7.5 Installing on ALCF Theta, Cray XC40

Theta is a 9.65 petaflops system manufactured by Cray with 3,624 compute nodes. Each node features a second-generation Intel Xeon Phi 7230 processor and 192 GB DDR4 RAM.

```
export CRAYPE_LINK_TYPE=dynamic
# Do not use cmake 3.9.1, it causes trouble with parallel HDF5.
module load cmake/3.11.4
module unload cray-libsci
module load cray-hdf5-parallel
module load gcc   # Make C++ 14 standard library available to the Intel compiler
export BOOST_ROOT=/soft/libraries/boost/1.64.0/intel
cmake ..
make -j 24
ls -l bin/qmcpack
```

3.7.6 Installing on ORNL OLCF Summit

Summit is an IBM system at the ORNL OLCF built with IBM Power System AC922 nodes. They have two IBM Power 9 processors and six NVIDIA Volta V100 accelerators.

Building QMCPACK

Note that these build instructions are preliminary as the software environment is subject to change. As of December 2018, the IBM XL compiler does not support C++14, so we currently use the gnu compiler.

For ease of reproducibility we provide build scripts for Summit.

```
cd qmcpack
./config/build_olcf_summit.sh
ls bin
```

Building Quantum Espresso

We provide a build script for the v6.4.1 release of Quantum Espresso (QE). The following can be used to build a CPU version of QE on Summit, placing the script in the external_codes/quantum_espresso directory.

```
cd external_codes/quantum_espresso
./build_qe_olcf_summit.sh
```

Note that performance is not yet optimized although vendor libraries are used. Alternatively, the wavefunction files can be generated on another system and the converted HDF5 files copied over.

3.7.7 Installing on NERSC Cori, Haswell Partition, Cray XC40

Cori is a Cray XC40 that includes 16-core Intel "Haswell" nodes installed at NERSC. In the following example, the source code is cloned in $HOME/qmc/git_QMCPACK and QMCPACK is built in the scratch space.

```
mkdir $HOME/qmc
mkdir $HOME/qmc/git_QMCPACK
cd $HOME/qmc_git_QMCPACK
git clone https://github.com/QMCPACK/qmcpack.git
cd qmcpack
git checkout v3.7.0 # Edit for desired version
export CRAYPE_LINK_TYPE=dynamic
module unload cray-libsci
module load boost/1.70.0
module load cray-hdf5-parallel
module load cmake/3.14.0
module load gcc/7.3.0 # Make C++ 14 standard library available to the Intel compiler
cd $SCRATCH
mkdir build_cori_hsw
cd build_cori_hsw
cmake -DQMC_SYMLINK_TEST_FILES=0 $HOME/qmc/git_QMCPACK/qmcpack/
nice make -j 8
ls -l bin/qmcpack
```

When the preceding was tested on May 16, 2019, the following module and software versions were present:

```
build_cori_hsw> module list
Currently Loaded Modulefiles:
1) modules/3.2.10.6                                    14) alps/6.6.43-6.0.7.1_5.45__
↪ga796da32.ari
2) nsg/1.2.0                                           15) rca/2.2.18-6.0.7.1_5.47__
↪g2aa4f39.ari
3) intel/18.0.1.163                                    16) atp/2.1.3
4) craype-network-aries                                17) PrgEnv-intel/6.0.4
5) craype/2.5.15                                       18) craype-haswell
6) udreg/2.3.2-6.0.7.1_5.13__g5196236.ari              19) cray-mpich/7.7.3
7) ugni/6.0.14.0-6.0.7.1_3.13__gea11d3d.ari            20) gcc/7.3.0
8) pmi/5.0.14                                          21) altd/2.0
9) dmapp/7.1.1-6.0.7.1_5.45__g5a674e0.ari              22) darshan/3.1.4
10) gni-headers/5.0.12.0-6.0.7.1_3.11__g3b1768f.ari    23) boost/1.70.0
11) xpmem/2.2.15-6.0.7.1_5.11__g7549d06.ari            24) cray-hdf5-parallel/1.10.2.0
12) job/2.2.3-6.0.7.1_5.43__g6c4e934.ari               25) cmake/3.14.0
13) dvs/2.7_2.2.118-6.0.7.1_10.1__g58b37a2
```

The following slurm job file can be used to run the tests:

```
#!/bin/bash
#SBATCH --qos=debug
#SBATCH --time=00:10:00
#SBATCH --nodes=1
#SBATCH --tasks-per-node=32
#SBATCH --constraint=haswell
echo --- Start `date`
echo --- Working directory: `pwd`
ctest -VV -R deterministic
echo --- End `date`
```

3.7.8 Installing on NERSC Cori, Xeon Phi KNL partition, Cray XC40

Cori is a Cray XC40 that includes Intel Xeon Phi Knight's Landing (KNL) nodes. The following build recipe ensures that the code generation is appropriate for the KNL nodes. The source is assumed to be in $HOME/qmc/git_QMCPACK/qmcpack as per the Haswell example.

```
export CRAYPE_LINK_TYPE=dynamic
module swap craype-haswell craype-mic-knl # Only difference between Haswell and KNL␣
↪recipes
module unload cray-libsci
module load boost/1.70.0
module load cray-hdf5-parallel
module load cmake/3.14.0
module load gcc/7.3.0 # Make C++ 14 standard library available to the Intel compiler
cd $SCRATCH
mkdir build_cori_knl
cd build_cori_knl
cmake -DQMC_SYMLINK_TEST_FILES=0 $HOME/git_QMCPACK/qmcpack/
nice make -j 8
ls -l bin/qmcpack
```

When the preceding was tested on May 16, 2019, the following module and software versions were present:

```
build_cori_knl> module list
  Currently Loaded Modulefiles:
  1) modules/3.2.10.6                                  14) alps/6.6.43-6.0.7.1_5.45__
↪ga796da32.ari
  2) nsg/1.2.0                                         15) rca/2.2.18-6.0.7.1_5.47__
↪g2aa4f39.ari
  3) intel/18.0.1.163                                  16) atp/2.1.3
  4) craype-network-aries                              17) PrgEnv-intel/6.0.4
  5) craype/2.5.15                                     18) craype-mic-knl
  6) udreg/2.3.2-6.0.7.1_5.13__g5196236.ari            19) cray-mpich/7.7.3
  7) ugni/6.0.14.0-6.0.7.1_3.13__gea11d3d.ari          20) gcc/7.3.0
  8) pmi/5.0.14                                        21) altd/2.0
  9) dmapp/7.1.1-6.0.7.1_5.45__g5a674e0.ari            22) darshan/3.1.4
  10) gni-headers/5.0.12.0-6.0.7.1_3.11__g3b1768f.ari  23) boost/1.70.0
  11) xpmem/2.2.15-6.0.7.1_5.11__g7549d06.ari          24) cray-hdf5-parallel/1.10.2.0
  12) job/2.2.3-6.0.7.1_5.43__g6c4e934.ari             25) cmake/3.14.0
  13) dvs/2.7_2.2.118-6.0.7.1_10.1__g58b37a2
```

3.7.9 Installing on systems with ARMv8-based processors

The following build recipe was verified using the 'Arm Compiler for HPC' on the ANL JLSE Comanche system with Cavium ThunderX2 processors on November 6, 2018.

```
# load armclang compiler
module load Generic-AArch64/RHEL/7/arm-hpc-compiler/18.4
# load Arm performance libraries
module load ThunderX2CN99/RHEL/7/arm-hpc-compiler-18.4/armpl/18.4.0
# define path to pre-installed packages
export HDF5_ROOT=</path/to/hdf5/install/>
export BOOST_ROOT=</path/to/boost/install> # header-only, no need to build
```

Then using the following command:

```
mkdir build_armclang
cd build_armclang
cmake -DCMAKE_C_COMPILER=armclang -DCMAKE_CXX_COMPILER=armclang++ -DQMC_MPI=0 \
    -DLAPACK_LIBRARIES="-L$ARMPL_DIR/lib -larmpl_mp" \
    -DFFTW_INCLUDE_DIR="$ARMPL_DIR/include" \
    -DFFTW_LIBRARIES="$ARMPL_DIR/lib/libarmpl_mp.a" \
    ..
make -j 56
```

Note that armclang is recognized as an `unknown' compiler by CMake v3.13* and below. In this case, we need to force it as clang to apply necessary flags. To do so, pass the following additionals option to CMake:

```
-DCMAKE_C_COMPILER_ID=Clang -DCMAKE_CXX_COMPILER_ID=Clang \
-DCMAKE_CXX_COMPILER_VERSION=5.0 -DCMAKE_CXX_STANDARD_COMPUTED_DEFAULT=98 \
```

3.7.10 Installing on Windows

Install the Windows Subsystem for Linux and Bash on Windows. Open a bash shell and follow the install directions for Ubuntu in *Installing on Ubuntu Linux or other apt-get–based distributions*.

3.8 Installing via Spack

Spack is a package manager for scientific software. One of the primary goals of Spack is to reduce the barrier for users to install scientific software. Spack is intended to work on everything from laptop computers to high-end supercomputers. More information about Spack can be found at https://spack.readthedocs.io/en/latest. The major advantage of installation with Spack is that all dependencies are automatically built, potentially including all the compilers and libraries, and different versions of QMCPACK can easily coexist with each other. The QMCPACK Spack package also knows how to automatically build and patch QE. In principle, QMCPACK can be installed with a single Spack command.

3.8.1 Known limitations

The QMCPACK Spack package inherits the limitations of the underlying Spack infrastructure and its dependencies. The main limitation is that installation can fail when building a dependency such as HDF5, MPICH, etc. For `spack install qmcpack` to succeed, it is very important to leverage preinstalled packages on your computer or supercomputer. The other frequently encountered challenge is that the compiler configuration is nonintuitive. This is especially the case with the Intel compiler. If you encounter any difficulties, we recommend testing the Spack compiler configuration on a simpler packages, e.g. HDF5.

Here are some additional limitations of the QMCPACK Spack package that will be resolved in future releases:

- CUDA support in Spack still has some limitations. It will catch only some compiler-CUDA conflicts.
- The Intel compiler must find a recent and compatible GCC compiler in its path or one must be explicity set with the `-gcc-name` and `-gxx-name` flags.

3.8.2 Setting up the Spack environment

Begin by cloning Spack from GitHub and configuring your shell as described at https://spack.readthedocs.io/en/latest/getting_started.html.

The goal of the next several steps is to set up the Spack environment for building. First, we highly recommend limiting the number of build jobs to a reasonable value for your machine. This can be accomplished by modifying your `~/.spack/config.yaml` file as follows:

```
config:
  build_jobs: 16
```

Make sure any existing compilers are properly detected. For many architectures, compilers are properly detected with no additional effort.

```
your-laptop> spack compilers
==> Available compilers
-- gcc sierra-x86_64 -------------------------------------------
gcc@7.2.0  gcc@6.4.0  gcc@5.5.0  gcc@4.9.4  gcc@4.8.5  gcc@4.7.4  gcc@4.6.4
```

However, if your compiler is not automatically detected, it is straightforward to add one:

```
your-laptop> spack compiler add <path-to-compiler>
```

The Intel compiler, and other commerical compilers like PGI, typically require extra environment variables to work properly. If you have an module environment set-up by your system administrators, it is recommended that you set the module name in `~/.spack/linux/compilers.yaml`. Here is an example for the Intel compiler:

```
- compiler:
  environment:{}
  extra_rpaths:  []
  flags: {}
  modules:
  - intel/18.0.3
  operating_system: ubuntu14.04
  paths:
    cc: /soft/com/packages/intel/18/u3/compilers_and_libraries_2018.3.222/linux/bin/
↪intel64/icc
    cxx: /soft/com/packages/intel/18/u3/compilers_and_libraries_2018.3.222/linux/bin/
↪intel64/icpc
    f77: /soft/com/packages/intel/18/u3/compilers_and_libraries_2018.3.222/linux/bin/
↪intel64/ifort
    fc: /soft/com/packages/intel/18/u3/compilers_and_libraries_2018.3.222/linux/bin/
↪intel64/ifort
  spec: intel@18.0.3
  target: x86_64
```

If a module is not available, you will have to set-up the environment variables manually:

```
- compiler:
  environment:
    set:
      INTEL_LICENSE_FILE: server@national-lab.doe.gov
  extra_rpaths:
  ['/soft/com/packages/intel/18/u3/compilers_and_libraries_2018.3.222/linux/compiler/
↪lib/intel64',
  '/soft/apps/packages/gcc/gcc-6.2.0/lib64']
```

(continues on next page)

(continued from previous page)

```
  flags:
    cflags: -gcc-name=/soft/apps/packages/gcc/gcc-6.2.0/bin/gcc
    fflags: -gcc-name=/soft/apps/packages/gcc/gcc-6.2.0/bin/gcc
    cxxflags: -gxx-name=/soft/apps/packages/gcc/gcc-6.2.0/bin/g++
  modules: []
  operating_system: ubuntu14.04
  paths:
    cc: /soft/com/packages/intel/18/u3/compilers_and_libraries_2018.3.222/linux/bin/
↪intel64/icc
    cxx: /soft/com/packages/intel/18/u3/compilers_and_libraries_2018.3.222/linux/bin/
↪intel64/icpc
    f77: /soft/com/packages/intel/18/u3/compilers_and_libraries_2018.3.222/linux/bin/
↪intel64/ifort
    fc: /soft/com/packages/intel/18/u3/compilers_and_libraries_2018.3.222/linux/bin/
↪intel64/ifort
  spec: intel@18.0.3
  target: x86_64
```

This last step is the most troublesome. Pre-installed packages are not automatically detected. If vendor optimized libraries are already installed, you will need to manually add them to your `~/.spack/packages.yaml`. For example, this works on Mac OS X for the Intel MKL package.

```
your-laptop> cat \~/.spack/packages.yaml
packages:
  intel-mkl:
      paths:
          intel-mkl@2018.0.128: /opt/intel/compilers_and_libraries_2018.0.104/mac/mkl
      buildable: False
```

Some trial-and-error might be involved to set the directories correctly. If you do not include enough of the tree path, Spack will not be able to register the package in its database. More information about system packages can be found at http://spack.readthedocs.io/en/latest/getting_started.html#system-packages.

Beginning with QMCPACK v3.9.0, Python 3.x is required. However, installing Python with a compiler besides GCC is tricky. We recommend leveraging your local Python installation by adding an entry in `~/.spack/packages.yaml`:

```
packages:
  python:
     modules:
       python@3.7.4: anaconda3/2019.10
```

Or if a module is not available

```
packages:
  python:
     paths:
        python@3.7.4: /nfs/gce/software/custom/linux-ubuntu18.04-x86_64/anaconda3/
↪2019.10/bin/python
   buildable: False
```

3.8.3 Building QMCPACK

The QMCPACK Spack package has a number of variants to support different compile time options and different versions of the application. A full list can be displayed by typing:

```
your laptop> spack info qmcpack
CMakePackage:   qmcpack

Description:
  QMCPACK, is a modern high-performance open-source Quantum Monte Carlo
  (QMC) simulation code.

Homepage: http://www.qmcpack.org/

Tags:
  ecp  ecp-apps

Preferred version:
  3.9.1      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.9.1

Safe versions:
  develop  [git] https://github.com/QMCPACK/qmcpack.git
  3.9.1      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.9.1
  3.9.0      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.9.0
  3.8.0      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.8.0
  3.7.0      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.7.0
  3.6.0      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.6.0
  3.5.0      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.5.0
  3.4.0      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.4.0
  3.3.0      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.3.0
  3.2.0      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.2.0
  3.1.1      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.1.1
  3.1.0      [git] https://github.com/QMCPACK/qmcpack.git at tag v3.1.0

Variants:
  Name [Default]         Allowed values          Description


  build_type [Release]    Debug, Release,         The build type to build
                          RelWithDebInfo
  afqmc [off]             True, False             Install with AFQMC support.
                                                  NOTE that if used in
                                                  combination with CUDA, only
                                                  AFQMC will have CUDA.
  complex [off]           True, False             Build the complex (general
                                                  twist/k-point) version
  cuda [off]              True, False             Build with CUDA
  cuda_arch [none]        none, 53, 20, 62,       CUDA architecture
                          60, 61, 50, 75, 70,
                          72, 32, 52, 30, 35
  da [off]                True, False             Install with support for basic
                                                  data analysis tools
  gui [off]               True, False             Install with Matplotlib (long
                                                  installation time)
  mixed [off]             True, False             Build the mixed precision
                                                  (mixture of single and double
                                                  precision) version for gpu and
                                                  cpu
```

(continues on next page)

(continued from previous page)

```
  mpi [on]                True, False                 Build with MPI support
  phdf5 [on]              True, False                 Build with parallel collective
                                                      I/O
  ppconvert [off]         True, False                 Install with pseudopotential
                                                      converter.
  qe [on]                 True, False                 Install with patched Quantum
                                                      Espresso 6.4.0
  soa [on]                True, False                 Build with Structure-of-Array
                                                      instead of Array-of-Structure
                                                      code. Only for CPU codeand
                                                      only in mixed precision
  timers [off]            True, False                 Build with support for timers

Installation Phases:
  cmake    build    install

Build Dependencies:
  blas  boost  cmake  cuda  fftw-api  hdf5  lapack  libxml2  mpi  python

Link Dependencies:
  blas  boost  cuda  fftw-api  hdf5  lapack  libxml2  mpi  python

Run Dependencies:
  py-matplotlib  py-numpy  quantum-espresso

Virtual Packages:
  None
```

For example, to install the complex-valued version of QMCPACK in mixed-precision use:

```
your-laptop> spack install qmcpack+mixed+complex%gcc@7.2.0 ^intel-mkl
```

where

```
%gcc@7.2.0
```

specifies the compiler version to be used and

```
^intel-mkl
```

specifies that the Intel MKL should be used as the BLAS and LAPACK provider. The ^ symbol indicates the the package to the right of the symbol should be used to fulfill the dependency needed by the installation.

It is also possible to run the QMCPACK regression tests as part of the installation process, for example:

```
your-laptop> spack install --test=root qmcpack+mixed+complex%gcc@7.2.0 ^intel-mkl
```

will run the unit and short tests. The current behavior of the QMCPACK Spack package is to complete the install as long as all the unit tests pass. If the short tests fail, a warning is issued at the command prompt.

For CUDA, you will need to specify and extra `cuda_arch` parameter otherwise, it will default to `cuda_arch=61`.

```
your-laptop> spack install qmcpack+cuda%intel@18.0.3 cuda_arch=61 ^intel-mkl
```

Due to limitations in the Spack CUDA package, if your compiler and CUDA combination conflict, you will need to set a specific verison of CUDA that is compatible with your compiler on the command line. For example,

```
your-laptop> spack install qmcpack+cuda%intel@18.0.3 cuda_arch=61 ^cuda@10.0.130 ^
↪intel-mkl
```

3.8.4 Loading QMCPACK into your environment

If you already have modules set-up in your enviroment, the Spack modules will be detected automatically. Otherwise, Spack will not automatically find the additional packages. A few additional steps are needed. Please see the main Spack documentation for additional details: https://spack.readthedocs.io/en/latest/module_file_support.html.

3.8.5 Dependencies that need to be compiled with GCC

Failing to compile a QMCPACK dependency is the most common reason that a Spack build fails. We recommend that you compile the following dependencies with GCC:

For MPI, using MPICH as the provider, try:

```
your-laptop> spack install qmcpack%intel@18.0.3 ^boost%gcc ^pkgconf%gcc ^perl%gcc ^
↪libpciaccess%gcc ^cmake%gcc ^findutils%gcc ^m4%gcc
```

For serial,

```
your-laptop> spack install qmcpack~mpi%intel@18.0.3 ^boost%gcc ^pkgconf%gcc ^perl%gcc␣
↪^cmake%gcc
```

3.8.6 Installing QMCPACK with Spack on Linux

Spack works robustly on the standard flavors of Linux (Ubuntu, CentOS, Ubuntu, etc.) using GCC, Clang, PGI, and Intel compilers.

3.8.7 Installing QMCPACK with Spack on Mac OS X

Spack works on Mac OS X but requires installation of a few packages using Homebrew. You will need to install at minimum the GCC compilers, CMake, and pkg-config. The Intel compiler for Mac on OS X is not well supported by Spack packages and will most likely lead to a compile time failure in one of QMCPACK's dependencies.

3.8.8 Installing QMCPACK with Spack on Cray Supercomputers

Spack now works with the Cray environment. To leverage the installed Cray environment, both a `compilers.yaml` and `packages.yaml` file should be provided by the supercomputing facility. Additionally, Spack packages compiled by the facility can be reused by chaining Spack installations https://spack.readthedocs.io/en/latest/chain.html.

Instructions for DOE supercomputing facilities that support Spack directly will be forthcoming.

3.8.9 Installing Quantum-Espresso with Spack

More information about the QE Spack package can be obtained directly from Spack

```
spack info quantum-espresso
```

There are many variants available for QE, most, but not all, are compatible with QMCPACK patch. Here is a minimalistic example of the Spack installation command that needs to be invoked:

```
your-laptop> spack install quantum-espresso+qmcpack~patch@6.4.1%gcc hdf5=parallel
```

The ~ decorator means deactivate the `patch` variant. This refers not to the QMCPACK patch, but to the upstream patching that is present for some versions of QE. These upstream QE patches fix specific critical autoconf/configure fixes. Unfortunately, some of these QE upstream patches are incompatible with the QMCPACK patch. Note that the Spack package will prevent you from installing incompatible variants and will emit an error message explaining the nature of the incompatibility.

A serial (no MPI) installation is also available, but the Spack installation command is non-intuitive for Spack newcomers:

```
your-laptop> spack install quantum-espresso+qmcpack~patch~mpi~scalapack@6.4.1%gcc␣
↪hdf5=serial
```

QE Spack package is well tested with GCC and Intel compilers, but will not work with the PGI compiler or in a cross-compile environment.

3.8.10 Reporting Bugs

Bugs with the QMCPACK Spack package should be filed at the main GitHub Spack repo https://github.com/spack/spack/issues.

In the GitHub issue, include `@naromero77` to get the attention of our developer.

3.9 Testing and validation of QMCPACK

We **strongly encourage** running the included tests each time QMCPACK is built. A range of unit and integration tests ensure that the code behaves as expected and that results are consistent with known-good mean-field, quantum chemical, and historical QMC results.

The tests include the following:

- Unit tests: to check fundamental behavior. These should always pass.
- Stochastic integration tests: to check computed results from the Monte Carlo methods. These might fail statistically, but rarely because of the use of three sigma level statistics. These tests are further split into "short" tests, which have just sufficient length to have valid statistics, and "long" tests, to check behavior to higher statistical accuracy.
- Converter tests: to check conversion of trial wavefunctions from codes such as QE and GAMESS to QMCPACK's formats. These should always pass.
- Workflow tests: in the case of QE, we test the entire cycle of DFT calculation, trial wavefunction conversion, and a subsequent VMC run.
- Performance: to help performance monitoring. Only the timing of these runs is relevant.

The test types are differentiated by prefixes in their names, for example, `short-LiH_dimer_ae_vmc_hf_noj_16-1` indicates a short VMC test for the LiH dime.

QMCPACK also includes tests for developmental features and features that are unsupported on certain platforms. To indicate these, tests that are unstable are labeled with the CTest label "unstable." For example, they are unreliable, unsupported, or known to fail from partial implementation or bugs.

When installing QMCPACK you should run at least the unit tests:

```
ctest -R unit
```

These tests take only a few seconds to run. All should pass. A failure here could indicate a major problem with the installation.

A wider range of deterministic integration tests are being developed. The goal is to test much more of QMCPACK than the unit tests do and to do so in a manner that is reproducible across platforms. All of these should eventually pass 100% reliably and quickly. At present, some fail on some platforms and for certain build types.

```
ctest -R deterministic -LE unstable
```

If time allows, the "short" stochastic tests should also be run. The short tests take a few minutes each on a 16-core machine—about 1 hour total depending on the platform. You can run these tests using the following command in the build directory:

```
ctest -R short -LE unstable  # Run the tests with "short" in their name.
                             # Exclude any known unstable tests.
```

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 run varies between the standard, complex, and GPU compilations. These tests should pass with three sigma reliability. That is, they should nearly always pass, and when rerunning a failed test it should usually pass. Overly frequent failures suggest a problem that should be addressed before any scientific production.

The full set of tests consist of significantly longer versions of the short tests, as well as tests of the conversion utilities. The runs require several hours each for improved statistics and a much more stringent test of the code. To run all the tests, simply run CTest in the build directory:

```
ctest -LE unstable           # Run all the stable 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 input data files for the tests are located in the `tests` directory. The system-level test directories are grouped into `heg`, `molecules`, and `solids`, with particular physical systems under each (for example `molecules/H4_ae`[1]). Under each physical system directory there might be tests for multiple QMC methods or parameter variations. The numerical comparisons and test definitions are in the `CMakeLists.txt` file in each physical system directory.

If *all* the QMC tests fail it is likely that the appropriate mpiexec (or mpirun, aprun, srun, jsrun) 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 console output in `build/Testing/Temporary/LastTest.log` or the output files under `build/tests/`.

Note that because most of the tests are very small, consisting of only a few electrons, the performance is not representative of larger calculations. For example, although the calculations might fit in cache, there will be essentially no vectorization because of the small electron counts. **These tests should therefore not be used for any benchmarking or performance analysis**. Example runs that can be used for testing performance are described in *Performance tests*.

3.9.1 Deterministic and unit tests

QMCPACK has a set of deterministic tests, predominantly unit tests. All of these tests can be run with the following command (in the build directory):

```
ctest -R deterministic -LE unstable
```

These tests should always pass. Failure could indicate a major problem with the compiler, compiler settings, or a linked library that would give incorrect results.

The output should look similar to the following:

```
Test project qmcpack/build
    Start  1: unit_test_numerics
1/11 Test  #1: unit_test_numerics ...............   Passed    0.06 sec
    Start  2: unit_test_utilities
2/11 Test  #2: unit_test_utilities ..............   Passed    0.02 sec
    Start  3: unit_test_einspline
...
10/11 Test #10: unit_test_hamiltonian ............   Passed    1.88 sec
    Start 11: unit_test_drivers
11/11 Test #11: unit_test_drivers ................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 11

Label Time Summary:
unit    =   2.20 sec

Total Test time (real) =   2.31 sec
```

Individual unit test executables can be found in `build/tests/bin`. The source for the unit tests is located in the `tests` directory under each directory in `src` (e.g. `src/QMCWavefunctions/tests`).

See unit-testing for more details about unit tests.

[1] The suffix “ae” is short for “all-electron,” and “pp” is short for “pseudopotential.”

3.9.2 Integration tests with Quantum Espresso

As described in *Installing and patching Quantum ESPRESSO*, it is possible to test entire workflows of trial wavefunction generation, conversion, and eventual QMC calculation. A patched QE must be installed so that the pw2qmcpack converter is available.

By adding `-D QE_BIN=your_QE_binary_path` in the CMake command line when building your QMCPACK, tests named with the "qe-" prefix will be included in the test set of your build. You can test the whole `pw > pw2qmcpack > qmcpack` workflow by

```
ctest -R qe
```

This provides a very solid test of the entire QMC toolchain for plane wave–generated wavefunctions.

3.9.3 Performance tests

Performance tests representative of real research runs are included in the tests/performance directory. They can be used for benchmarking, comparing machine performance, or assessing optimizations. This is in contrast to the majority of the conventional integration tests in which the particle counts are too small to be representative. Care is still needed to remove initialization, I/O, and compute a representative performance measure.

The CTest integration is sufficient to run the benchmarks and measure relative performance from version to version of QMCPACK and to assess proposed code changes. Performance tests are prefixed with "performance." To obtain the highest performance on a particular platform, you must run the benchmarks in a standalone manner and tune thread counts, placement, walker count (etc.). This is essential to fairly compare different machines. Check with the developers if you are unsure of what is a fair change.

For the largest problem sizes, the initialization of spline orbitals might take a large portion of overall runtime. When QMCPACK is run at scale, the initialization is fast because it is fully parallelized. However, the performance tests usually run on a single node. Consider running QMCPACK once with `save_coefs="yes"` XML input tag added to the line of 'determinantset' to save the converted spline coefficients to the disk and load them for later runs in the same folder. See *Spline basis sets* for more information.

The delayed update algorithm in *Single determinant wavefunctons* significantly changes the performance characteristics of QMCPACK. A parameter scan of the maximal number of delays specific to every architecture and problem size is required to achieve the best performance.

NiO performance tests

Follow the instructions in tests/performance/NiO/README to enable and run the NiO tests.

The NiO tests are for bulk supercells of varying size. The QMC runs consist of short blocks of (1) VMC without drift (2) VMC with drift term included, and (3) DMC with constant population. The tests use spline wavefunctions that must be downloaded as described in the README file because of their large size. You will need to set `-DQMC_DATA=YOUR_DATA_FOLDER -DENABLE_TIMERS=1` when running CMake as described in the README file.

Two sets of wavefunction are tested: spline orbitals with one- and two-body Jastrow functions and a more complex form with an additional three-body Jastrow function. The Jastrows are the same for each run and are not reoptimized, as might be done for research purposes. Runs in the hundreds of electrons up to low thousands of electrons are representative of research runs performed in 2017. The largest runs target future machines and require very large memory.

Table 3.1: System sizes and names for NiO performance tests. GPU performance tests are named similarly but have different walker counts.

Performance test name	Historical name	Atoms	Electrons	Electrons/spin
performance-NiO-cpu-a32-e384	S8	32	384	192
performance-NiO-cpu-a64-e768	S16	64	768	384
performance-NiO-cpu-a128-e1536	S32	128	1536	768
performance-NiO-cpu-a256-e3072	S64	256	3072	1536
performance-NiO-cpu-a512-e6144	S128	512	6144	3072
performance-NiO-cpu-a1024-e12288	S256	1024	12288	6144

3.9.4 Troubleshooting tests

CTest reports briefly pass or fail of tests in printout and also collects all the standard outputs to help investigating how tests fail. If the CTest execution is completed, look at `Testing/Temporary/LastTest.log`. If you manually stop the testing (ctrl+c), look at `Testing/Temporary/LastTest.log.tmp`. You can locate the failing tests by searching for the key word “Fail.”

3.9.5 Slow testing with OpenMPI

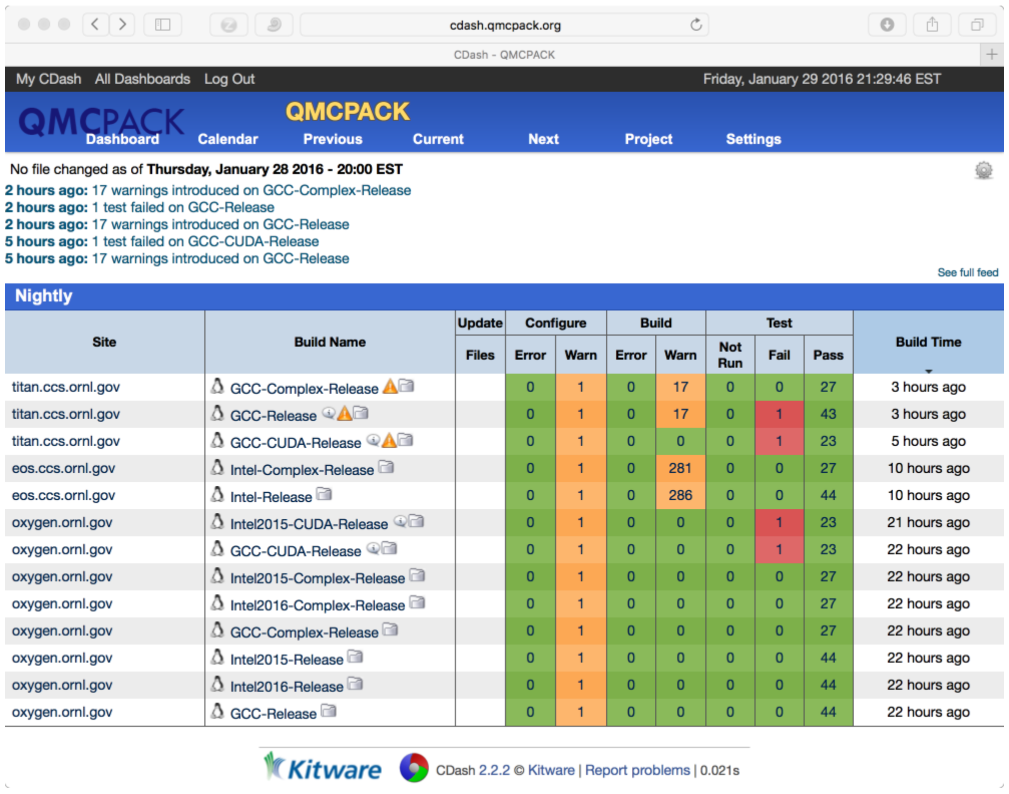
OpenMPI has a default binding policy that makes all the threads run on a single core during testing when there are two or fewer MPI ranks. This significantly increases testing time. If you are authorized to change the default setting, you can just add “hwloc_base_binding_policy=none” in /etc/openmpi/openmpi-mca-params.conf.

3.10 Automated testing of QMCPACK

The QMCPACK developers run automatic tests of QMCPACK on several different computer systems, many on a continuous basis. See the reports at https://cdash.qmcpack.org/CDash/index.php?project=QMCPACK. The combinations that are currently tested can be seen on CDash and are also listed in https://github.com/QMCPACK/qmcpack/blob/develop/README.md. They include GCC, Clang, Intel, and PGI compilers in combinations with various library versions and different MPI implementations. NVIDIA GPUs are also tested.

3.11 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 QE, and the XML format used by QMCPACK itself. The utility also enables the atomic orbitals to be recomputed via a numerical density functional calculation if they need to be reconstructed for use in an electronic structure calculation.



Friday, January 29 2016 21:29:46 EST

QMCPACK

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

CDash 2.2.2 © Kitware | Report problems | 0.021s

Fig. 3.1: 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, four errors were found. This dashboard is accessible at https://cdash.qmcpack.org

3.12 Installing and patching Quantum ESPRESSO

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

QE stores wavefunctions in a nonstandard internal "save" format. To convert these to a conventional HDF5 format file we have developed a converter—pw2qmcpack—which is an add-on to the QE distribution.

To simplify the process of patching QE we have developed a script that will automatically download and patch the source code. The patches are specific to each version. For example, to download and patch QE v6.3:

```
cd external_codes/quantum_espresso
./download_and_patch_qe6.3.sh
```

After running the patch, you must configure QE with the HDF5 capability enabled in either way:

- If your system already has HDF5 installed with Fortran, use the -{}-with-hdf5 configuration option.

  ```
  cd qe-6.3
  ./configure --with-hdf5=/opt/local    # Specify HDF5 base directory
  ```

  Check the end of the configure output if HDF5 libraries are found properly. If not, either install a complete library or use the other scheme. If using a parallel HDF5 library, be sure to use the same MPI with QE as used to build the parallel HDF5 library.

  Currently, HDF5 support in QE itself is preliminary. To enable use of pw2qmcpack but use the old non-HDF5 I/O within QE, replace `-D__HDF5` with `{-D__HDF5_C}` in make.inc.

- If your system has HDF5 with C only, manually edit make.inc by adding `-D__HDF5_C` and `-DH5_USE_16_API` in `DFLAGS` and provide include and library path in `IFLAGS` and `HDF5_LIB`.

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

The tests involving pw.x and pw2qmcpack.x have been integrated into the test suite of QMCPACK. By adding `-D QE_BIN=your_QE_binary_path` in the CMake command line when building your QMCPACK, tests named with the "qe-" prefix will be included in the test set of your build. You can test the whole `pw > pw2qmcpack > qmcpack workflow` by

```
ctest -R qe
```

See *Integration tests with Quantum Espresso* and the testing section for more details.

3.13 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 to apply the orbital coefficients in local basis calculations and in the Slater determinant update.
- 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, for example, 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. That is, do not compute smaller error bars than necessary.

3.14 Troubleshooting the installation

Some tips to help troubleshoot installations of QMCPACK:

- First, build QMCPACK on a workstation you control or on any system with 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 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, directly specify the appropriate directories and files (*Configuration Options*). For example,

```
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

FOUR

RUNNING QMCPACK

QMCPACK requires at least one xml input file, and is invoked via:

```
qmcpack [command line options] <XML input file(s)>
```

4.1 Command line options

QMCPACK offers several command line options that affect how calculations are performed. If the flag is absent, then the corresponding option is disabled:

- `--dryrun` Validate the input file without performing the simulation. This is a good way to ensure that QMC-PACK will do what you think it will.
- `--enable-timers=none|coarse|medium|fine` Control the timer granularity when the build option `ENABLE_TIMERS` is enabled.
- `help` Print version information as well as a list of optional command-line arguments.
- `noprint` Do not print extra information on Jastrow or pseudopotential. If this flag is not present, QMCPACK will create several `.dat` files that contain information about pseudopotentials (one file per PP) and Jastrow factors (one per Jastrow factor). These file might be useful for visual inspection of the Jastrow, for example.
- `--verbosity=low|high|debug` Control the output verbosity. The default low verbosity is concise and, for example, does not include all electron or atomic positions for large systems to reduce output size. Use "high" to see this information and more details of initialization, allocations, QMC method settings, etc.
- `version` Print version information and optional arguments. Same as `help`.

4.2 Input files

The input is one or more XML file(s), documented in *Input file overview*.

4.3 Output files

QMCPACK generates multiple files documented in output-overview.

4.4 Running in parallel with MPI

QMCPACK is fully parallelized with MPI. When performing an ensemble job, all the MPI ranks are first equally divided into groups that 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 (e.g., one MPI task per socket). On clusters with more than one GPU per node (NVIDIA Tesla K80), it is necessary to use the same number of MPI ranks as the number of GPUs per node to let each MPI rank take one GPU.

4.5 Using 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 this compute capability by using threads and the 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 environment variable OMP_NUM_THREADS. Assuming one MPI rank per socket, the number of threads should typically be the number of cores on that socket. Even in the GPU-accelerated version, using threads significantly reduces the time spent on the calculations performed by the CPU.

4.5.1 Nested OpenMP threads

Nested threading is an advanced feature requiring experienced users to finely tune runtime parameters to reach the best performance.

For small-to-medium problem sizes, using one thread per walker or for multiple walkers is most efficient. This is the default in QMCPACK and achieves the shortest time to solution.

For large problems of at least 1,000 electrons, use of nested OpenMP threading can be enabled to reduce the time to solution further, although at some loss of efficiency. In this scheme multiple threads are used in the computations of each walker. This capability is implemented for some of the key computational kernels: the 3D spline orbital evaluation, certain portions of the distance tables, and implicitly the BLAS calls in the determinant update. Use of the batched nonlocal pseudopotential evaluation is also recommended.

Nested threading is enabled by setting `OMP_NUM_THREADS=AA,BB`, `OMP_MAX_ACTIVE_LEVELS=2` and `OMP_NESTED=TRUE` where the additional `BB` is the number of second-level threads. Choosing the thread affinity is critical to the performance. QMCPACK provides a tool qmc-check-affinity (source file src/QMCTools/check-affinity.cpp for details), which might help users investigate the affinity. Knowledge of how the operating system logical CPU cores (/prco/cpuinfo) are bound to the hardware is also needed.

For example, on Blue Gene/Q with a Clang compiler, the best way to fully use the 16 cores each with 4 hardware threads is

```
OMP_NESTED=TRUE
OMP_NUM_THREADS=16,4
MAX_ACTIVE_LEVELS=2
```

(continues on next page)

(continued from previous page)

```
OMP_PLACES=threads
OMP_PROC_BIND=spread,close
```

On Intel Xeon Phi KNL with an Intel compiler, to use 64 cores without using hardware threads:

```
OMP_NESTED=TRUE
OMP_WAIT_POLICY=ACTIVE
OMP_NUM_THREADS=16,4
MAX_ACTIVE_LEVELS=2
OMP_PLACES=cores
OMP_PROC_BIND=spread,close
KMP_HOT_TEAMS_MODE=1
KMP_HOT_TEAMS_MAX_LEVEL=2
```

Most multithreaded BLAS/LAPACK libraries do not spawn threads by default when being called from an OpenMP parallel region. See the explanation in *Serial or multithreaded library*. This results in the use of only a single thread in each second-level thread team for BLAS/LAPACK operations. Some vendor libraries like MKL support using multiple threads when being called from an OpenMP parallel region. One way to enable this feature is using environment variables to override the default behavior. However, this forces all the calls to the library to use the same number of threads. As a result, small function calls are penalized with heavy overhead and heavy function calls are slow for not being able to use more threads. Instead, QMCPACK uses the library APIs to turn on nested threading only at selected performance critical calls. In the case of using a serial library, QMCPACK implements nested threading to distribute the workload wherever necessary. Users do not need to control the threading behavior of the library.

4.5.2 Performance considerations

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 DMC, for typical small to mid-sized calculations choose the total number of walkers to be a significant multiple of the total number of threads (MPI tasks * threads per task). This will ensure a good load balance. e.g., for a calculation on a few nodes with a total 512 threads, using 5120 walkers may keep the load imbalance around 10%. For the very largest calculations, 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. This will reduce occurrences worse-case load imbalance e.g. where one thread has two walkers while all the others have one.

To achieve better performance, a mixed-precision version (experimental) has been developed in the CPU code. The mixed-precision CPU code uses a mixed of single precision (SP) and double precision (DP) operations, while the default code use DP exclusively. This mixed precision version is more aggressive than the GPU CUDA version in using single precision (SP) operations. The Current implementation uses SP on most calculations, except for matrix inversions and reductions where double precision is required to retain high accuracy. All the constant spline data in wavefunction, pseudopotentials, and Coulomb potentials are initialized in double precision and later stored in single precision. The mixed-precision code is as accurate as the double-precision code up to a certain system size, and may have double the throughput. Cross checking and verification of accuracy is always required but is particularly important above approximately 1,500 electrons.

4.5.3 Memory considerations

When using threads, some memory objects are shared by all the threads. Usually these memory objects 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 *Spline basis sets*.

The other memory objects that are distinct for each walker during random walks need to be associated with individual walkers and cannot 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.

4.6 Running on GPU machines

The GPU version for the NVIDIA CUDA platform is fully incorporated into the main source code. Commonly used functionalities for solid-state and molecular systems using B-spline single-particle orbitals are supported. Use of Gaussian basis sets, three-body Jastrow functions, and many observables are not yet supported. A detailed description of the GPU implementation can be found in [EKCS12].

The current GPU implementation assumes one MPI process per GPU. To use nodes with multiple GPUs, use multiple MPI processes per node. 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.

4.6.1 Performance considerations

To run with high performance on GPUs it is crucial to perform some benchmarking runs: the optimum configuration is system size, walker count, and GPU model dependent. The GPU implementation vectorizes operations over multiple walkers, so generally the more walkers that are placed on a GPU, the higher the performance that will be obtained. Performance also increases with electron count, up until the memory on the GPU is exhausted. A good strategy is to perform a short series of VMC runs with walker count increasing in multiples of two. For systems with 100s of electrons, typically 128–256 walkers per GPU use a sufficient number of GPU threads to operate the GPU efficiently and to hide memory-access latency. For smaller systems, thousands of walkers might be required. For QMC algorithms where the number of walkers is fixed such as VMC, choosing a walker count the is a multiple of the number of streaming multiprocessors can be most efficient. For variable population DMC runs, this exact match is not possible.

To achieve better performance, the current GPU implementation uses single-precision operations for most of the calculations. Double precision is used in matrix inversions and the Coulomb interaction to retain high accuracy. The mixed-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 1,500 electrons. For typical calculations on smaller electron counts, the statistical error bars are much larger then the error introduced by mixed precision.

4.6.2 Memory considerations

In the GPU implementation, each walker has a buffer in 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. Additionally, for calculations using B-splines, this data is stored on the GPU in a shared read-only buffer. Often the size of the B-spline data limits the calculations that can be run on the GPU.

If the GPU memory is exhausted, first try reducing the number of walkers per GPU. Coarsening the grids of the B-splines representation (by decreasing the value of the mesh factor 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. It is also possible to distribute the B-spline coefficients table between the host and GPU memory, see option Spline_Size_Limit_MB in *Spline basis sets*.

CHAPTER

FIVE

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

SIX

INPUT FILE OVERVIEW

This chapter introduces XML as it is used in the QMCPACK 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.

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 previously shown.

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 as follows:

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

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

(continues on next page)

(continued from previous page)

```
<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.

6.1 Project

The `<project>` tag uses the `id` and `series` attributes. The value of `id` is the first part of the prefix for output file names.

Output file names also contain the series number, starting at the value given by the `series` tag. After every `<qmc>` section, the series value will increment, giving each section a unique prefix.

For the input file shown previously, the output files will start with `vmc.s000`, for example, `vmc.s000.scalar.dat`. If there were another `<qmc>` section in the input file, the corresponding output files would use the prefix `vmc.s001`.

6.2 Random number initialization

The random number generator state is initialized from the `random` element using the `seed` attribute:

```
<random seed="1000"/>
```

If the random element is not present, or the seed value is negative, the seed will be generated from the current time.

To initialize the many independent random number generators (one per thread and MPI process), the seed value is used (modulo 1024) as a starting index into a list of prime numbers. Entries in this offset list of prime numbers are then used as the seed for the random generator on each thread and process.

If checkpointing is enabled, the random number state is written to an HDF file at the end of each block (suffix: `.random.h5`). This file will be read if the `mcwalkerset` tag is present to perform a restart. For more information, see the `checkpoint` element in the QMC methods *Quantum Monte Carlo Methods* and checkpoint-files on checkpoint and restart files.

CHAPTER

SEVEN

SPECIFYING THE SYSTEM TO BE SIMULATED

7.1 Specifying the Simulation Cell

The `simulationcell` block specifies the geometry of the cell, how the boundary conditions should be handled, and how ewald summation should be broken up.

`simulationcell` Element:

Parent elements:	`qmcsystem`
Child elements:	None

Attribute:

parameter name	**datatype**	**values**	**default**	**description**
`lattice`	9 floats	any float	Must be specified	Specification of lattice vectors.
`bconds`	string	“p” or “n”	“n n n “	Boundary conditions for each axis.
`vacuum`	float	$\geq$ 1.0	1.0	Vacuum scale.
`LR_dim_cutoff`	float	float	15	Ewald breakup distance.
`LR_tol`	float	float	3e-4	Tolerance in Ha for Ewald ion-ion energy per atom.

An example of a block is given below:

```
<simulationcell>
    <parameter name="lattice">
      3.8       0.0       0.0
      0.0       3.8       0.0
      0.0       0.0       3.8
    </parameter>
    <parameter name="bconds">
       p p p
    </parameter>
    <parameter name="LR_dim_cutoff"> 20 </parameter>
  </simulationcell>
```

Here, a cubic cell 3.8 bohr on a side will be used. This simulation will use periodic boundary conditions, and the maximum k vector will be $20/r_{wigner-seitz}$ of the cell.

7.1.1 Lattice

The cell is specified using 3 lattice vectors.

7.1.2 Boundary conditions

QMCPACK offers the capability to use a mixture of open and periodic boundary conditions. The parameter expects a single string of three characters separated by spaces, *e.g.* "p p p" for purely periodic boundary conditions. These characters control the behavior of the x, y, and z, axes, respectively. Non periodic directions must be placed after the periodic ones. Examples of valid include:

"p p p" Periodic boundary conditions. Corresponds to a 3D crystal.

"p p n" Slab geometry. Corresponds to a 2D crystal.

"p n n" Wire geometry. Corresponds to a 1D crystal.

"n n n" Open boundary conditions. Corresponds to an isolated molecule in a vacuum.

7.1.3 Vacuum

The vacuum option allows adding a vacuum region in slab or wire boundary conditions (`bconds= p p n` or `bconds= p n n`, respectively). The main use is to save memory with spline or plane-wave basis trial wavefunctions, because no basis functions are required inside the vacuum region. For example, a large vacuum region can be added above and below a graphene sheet without having to generate the trial wavefunction in such a large box or to have as many splines as would otherwise be required. Note that the trial wavefunction must still be generated in a large enough box to sufficiently reduce periodic interactions in the underlying electronic structure calculation.

With the vacuum option, the box used for Ewald summation increases along the axis labeled by a factor of `vacuum`. Note that all the particles remain in the original box without altering their positions. i.e. Bond lengths are not changed by this option. The default value is 1, no change to the specified axes.

An example of a `simulationcell` block using is given below. The size of the box along the z-axis increases from 12 to 18 by the vacuum scale of 1.5.

```
<simulationcell>
    <parameter name="lattice">
      3.8       0.0       0.0
      0.0       3.8       0.0
      0.0       0.0      12.0
    </parameter>
    <parameter name="bconds">
       p p n
    </parameter>
    <parameter name="vacuum"> 1.5 </parameter>
    <parameter name="LR_dim_cutoff"> 20 </parameter>
  </simulationcell>
```

7.1.4 LR_dim_cutoff

When using periodic boundary conditions direct calculation of the Coulomb energy is not well behaved. As a result, QMCPACK uses an optimized Ewald summation technique to compute the Coulomb interaction. [NC95]

In the Ewald summation, the energy is broken into short- and long-ranged terms. The short-ranged term is computed directly in real space, while the long-ranged term is computed in reciprocal space. controls where the short-ranged term ends and the long-ranged term begins. The real-space cutoff, reciprocal-space cutoff, and are related via:

$$\mathrm{LR\_dim\_cutoff} = r_c \times k_c$$

where r_c is the Wigner-Seitz radius, and k_c is the length of the maximum k-vector used in the long-ranged term. Larger values of increase the accuracy of the evaluation. A value of 15 tends to be conservative.

7.2 Specifying the particle set

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

7.2.1 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/randomsrc`^{o}	Text	`particleset.name`	*None*	Particle set to randomize

7.2.2 Detailed attribute description

Required particleset attributes

- `name/id`
  Unique name for the particle set. Default is "e" for electrons. "i" or "ion0" is typically used for ions. For special cases where an empty particle set is needed, the special name "empty" can be used to bypass the zero-size error check.

Optional particleset attributes

- `size`
  Number of particles in set.

`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

- `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.

Required name 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.

Optional group 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]).

7.2.3 Example use cases

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>
```

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
```

(continues on next page)

(continued from previous page)

```
    -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>
```

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

EIGHT

TRIAL WAVEFUNCTION SPECIFICAION

8.1 Introduction

This section describes the input blocks associated with the specification of the trial wavefunction in a QMCPACK calculation. These sections are contained within the `<wavefunction>` ... `</wavefunction>` xml blocks. **Users are expected to rely on converters to generate the input blocks described in this section.** The converters and the workflows are designed such that input blocks require minimum modifications from users. Unless the workflow requires modification of wavefunction blocks (e.g., setting the cutoff in a multideterminant calculation), only expert users should directly alter them.

The trial wavefunction in QMCPACK has a general product form:

$$\Psi_T(\vec{r}) = \prod_k \Theta_k(\vec{r}), \tag{8.1}$$

where each $\Theta_k(\vec{r})$ is a function of the electron coordinates (and possibly ionic coordinates and variational parameters). For problems involving electrons, the overall trial wavefunction must be antisymmetric with respect to electron exchange, so at least one of the functions in the product must be antisymmetric. Notice that, although QMCPACK allows for the construction of arbitrary trial wavefunctions based on the functions implemented in the code (e.g., slater determinants, jastrow functions), the user must make sure that a correct wavefunction is used for the problem at hand. From here on, we assume a standard trial wavefunction for an electronic structure problem

$$Psi_T(\vec{r}) = A(\vec{r}) \prod_k J_k(\vec{r}), \tag{8.2}$$

where $A(\vec{r})$ is one of the antisymmetric functions: (1) slater determinant, (2) multislater determinant, or (3) pfaffian and J_k is any of the Jastrow functions (described in *Jastrow Factors*). The antisymmetric functions are built from a set of single particle orbitals `(sposet)`. QMCPACK implements four different types of `sposet`, described in the following section. Each `sposet` is designed for a different type of calculation, so their definition and generation varies accordingly.

8.2 Single determinant wavefunctons

Placing a single determinant for each spin is the most used ansatz for the antisymmetric part of a trial wavefunction. The input xml block for `slaterdeterminant` is given in *Listing 1*. A list of options is given in Table 8.2.

`slaterdeterminant` element:

Parent elements	`determinantset`
Child elements	`determinant`

Attribute:

Name	Datatype	Values	Default	Description
`delay_rank`	Integer	>=0	1	Number of delayed updates.
`optimize`	Text	yes/no	yes	Enable orbital optimization.

Table 2 Options for the `slaterdeterminant` xml-block.

Listing 8.1: Slaterdeterminant set XML element.

```
<slaterdeterminant delay_rank="32">
   <determinant id="updet" size="208">
     <occupation mode="ground" spindataset="0">
     </occupation>
   </determinant>
   <determinant id="downdet" size="208">
     <occupation mode="ground" spindataset="0">
     </occupation>
   </determinant>
 </slaterdeterminant>
```

Additional information:

- `delay_rank` This option enables delayed updates of the Slater matrix inverse when particle-by-particle move is used. By default or if `delay_rank=0` given in the input file, QMCPACK sets 1 for Slater matrices with a leading dimension $<$ 192 and 32 otherwise. `delay_rank=1` uses the Fahy's variant [FWL90] of the Sherman-Morrison rank-1 update, which is mostly using memory bandwidth-bound BLAS-2 calls. With `delay_rank>1`, the delayed update algorithm [LK18][MDAzevedoL+17] turns most of the computation to compute bound BLAS-3 calls. Tuning this parameter is highly recommended to gain the best performance on medium-to-large problem sizes ($>$ 200 electrons). We have seen up to an order of magnitude speedup on large problem sizes. When studying the performance of QMCPACK, a scan of this parameter is required and we recommend starting from 32. The best `delay_rank` giving the maximal speedup depends on the problem size. Usually the larger `delay_rank` corresponds to a larger problem size. On CPUs, `delay_rank` must be chosen as a multiple of SIMD vector length for good performance of BLAS libraries. The best `delay_rank` depends on the processor microarchitecture. GPU support is under development.

8.3 Single-particle orbitals

8.3.1 Spline basis sets

In this section we describe the use of spline basis sets to expand the `sposet`. Spline basis sets are designed to work seamlessly with plane wave DFT code (e.g.,Quantum ESPRESSO as a trial wavefunction generator).

In QMC algorithms, all the SPOs $\{\phi(\vec{r})\}$ need to be updated every time a single electron moves. Evaluating SPOs takes a very large portion of computation time. In principle, PW basis set can be used to express SPOs directly in QMC, as in DFT. But it introduces an unfavorable scaling because the basis set size increases linearly as the system size. For this reason, it is efficient to use a localized basis with compact support and a good transferability from the plane wave basis.

In particular, 3D tricubic B-splines provide a basis in which only 64 elements are nonzero at any given point in [AlfeG04]. The 1D cubic B-spline is given by

$$f(x) = \sum_{i'=i-1}^{i+2} b^{i',3}(x)\; p_{i'}, \tag{8.3}$$

where $b^i(x)$ is the piecewise cubic polynomial basis functions and $i = \text{floor}(\Delta^{-1}x)$ is the index of the first grid point $\leq x$. Constructing a tensor product in each Cartesian direction, we can represent a 3D orbital as

$$\phi_n(x,y,z) = \sum_{i'=i-1}^{i+2} b_x^{i',3}(x) \sum_{j'=j-1}^{j+2} b_y^{j',3}(y) \sum_{k'=k-1}^{k+2} b_z^{k',3}(z)\; p_{i',j',k',n}. \tag{8.4}$$

This allows the rapid evaluation of each orbital in constant time. Furthermore, this basis is systematically improvable with a single spacing parameter so that accuracy is not compromised compared with the plane wave basis.

The use of 3D tricubic B-splines greatly improves computational efficiency. The gain in computation time from a plane wave basis set to an equivalent B-spline basis set becomes increasingly large as the system size grows. On the downside, this computational efficiency comes at the expense of increased memory use, which is easily overcome, however, by the large aggregate memory available per node through OpenMP/MPI hybrid QMC.

The input xml block for the spline SPOs is given in *Listing 2*. A list of options is given in Table 8.3.1.

Listing 8.2: Determinant set XML element.

```
<determinantset type="bspline" source="i" href="pwscf.h5"
                tilematrix="1 1 3 1 2 -1 -2 1 0" twistnum="-1" gpu="yes" meshfactor=
↪"0.8"
                twist="0  0  0" precision="double">
  <slaterdeterminant>
    <determinant id="updet" size="208">
      <occupation mode="ground" spindataset="0">
      </occupation>
    </determinant>
    <determinant id="downdet" size="208">
      <occupation mode="ground" spindataset="0">
      </occupation>
    </determinant>
  </slaterdeterminant>
</determinantset>
```

`determinantset` element:

Parent elements	`wavefunction`
Child elements	`slaterdeterminant`

attribute:

Name	Datatype	Values	Default	Description
`type`	Text	Bspline		Type of `sposet`
`href`	Text			Path to hdf5 file from pw2qmcpack.x.
`tilematrix`	9 integers			Tiling matrix used to expand supercell.
`twistnum`	Integer			Index of the super twist.
`twist`	3 floats			Super twist.
`meshfactor`	Float	$\leq$ 1.0		Grid spacing ratio.
`precision`	Text	Single/double		Precision of spline coefficients
`gpu`	Text	Yes/no		GPU switch.
`gpusharing`	Text	Yes/no	No	Share B-spline table across GPUs.
`Spline_Size_Limit_MB`	Integer			Limit B-spline table size on GPU.
`check_orb_norm`	Text	Yes/no	Yes	Check norms of orbitals from h5 file.
`save_coefs`	Text	Yes/no	No	Save the spline coefficients to h5 file.
`source`	Text	Any	Ion0	Particle set with atomic positions.

Table 3 Options for the `determinantset` xml-block associated with B-spline single particle orbital sets.

Additional information:

- `precision`. Only effective on CPU versions without mixed precision, "single" is always imposed with mixed precision. Using single precision not only saves memory use but also speeds up the B-spline evaluation. We recommend using single precision since we saw little chance of really compromising the accuracy of calculation.
- `meshfactor`. The ratio of actual grid spacing of B-splines used in QMC calculation with respect to the original one calculated from h5. A smaller meshfactor saves memory use but reduces accuracy. The effects are similar to reducing plane wave cutoff in DFT calculations. Use with caution!
- `twistnum`. If positive, it is the index. We recommend not taking this way since the indexing might show some uncertainty. If negative, the super twist is referred by `twist`.
- `save_coefs`. If yes, dump the real-space B-spline coefficient table into an h5 file on the disk. When the orbital transformation from k space to B-spline requires more than the available amount of scratch memory on the compute nodes, users can perform this step on fat nodes and transfer back the h5 file for QMC calculations.
- `gpusharing`. If enabled, spline data is shared across multiple GPUs on a given computational node. For example, on a two-GPU-per-node system, each GPU would have half of the orbitals. This enables larger overall spline tables than would normally fit in the memory of individual GPUs to be used, potentially up to the total GPU memory on a node. To obtain high performance, large electron counts or a high-performing CPU-GPU interconnect is required. To use this feature, the following needs to be done:

    - The CUDA Multi-Process Service (MPS) needs to be used (e.g., on

  Summit/SummitDev use "-alloc_flags gpumps" for bsub). If MPS is not detected, sharing will be disabled.

    - CUDA_VISIBLE_DEVICES needs to be properly set to control each rank's

  visible CUDA devices (e.g., on OLCF Summit/SummitDev one needs to create a resource set containing all GPUs with the respective number of ranks with "jsrun –task-per-rs Ngpus -g Ngpus").

- `Spline_Size_Limit_MB`. Allows distribution of the B-spline coefficient table between the host and GPU memory. The compute kernels access host memory via zero-copy. Although the performance penalty introduced by it is significant, it allows large calculations to go through.

8.3.2 Gaussian basis tests

In this section we describe the use of localized basis sets to expand the `sposet`. The general form of a single particle orbital in this case is given by:

$$\phi_i(\vec{r}) = \sum_k C_{i,k}\ \eta_k(\vec{r}), \tag{8.5}$$

where $\{\eta_k(\vec{r})\}$ is a set of M atom-centered basis functions and $C_{i,k}$ is a coefficient matrix. This should be used in calculations of finite systems employing an atom-centered basis set and is typically generated by the *convert4qmc* converter. Examples include calculations of molecules using Gaussian basis sets or Slater-type basis functions. Initial support for periodic systems is described in LCAO. Even though this section is called "Gaussian basis sets" (by far the most common atom-centered basis set), QMCPACK works with any atom-centered basis set based on either spherical harmonic angular functions or Cartesian angular expansions. The radial functions in the basis set can be expanded in either Gaussian functions, Slater-type functions, or numerical radial functions.

In this section we describe the input sections for the atom-centered basis set and the `sposet` for a single Slater determinant trial wavefunction. The input sections for multideterminant trial wavefunctions are described in *Multideterminant wavefunctions*. The basic structure for the input block of a single Slater determinant is given in *Listing 3*. A list of options for `determinantset` associated with this `sposet` is given in Table 8.3.2.

Listing 8.3: Basic input block for a single determinant trial wavefunction using a `sposet` expanded on an atom-centered basis set.

```
<wavefunction id="psi0" target="e">
  <determinantset>
    <basisset>
      ...
    </basisset>
    <slaterdeterminant>
      ...
    </slaterdeterminant>
  </determinantset>
</wavefunction>
```

The definition of the set of atom-centered basis functions is given by the `basisset` block, and the `sposet` is defined within `slaterdeterminant`. The `basisset` input block is composed from a collection of `atomicBasisSet` input blocks, one for each atomic species in the simulation where basis functions are centered. The general structure for `basisset` and `atomicBasisSet` are given in *Listing 4*, and the corresponding lists of options are given in Table 8.3.2 and Table 8.3.2.

`determinantset` element:

Parent elements	`wavefunction`
Child elements	`basisset` , `slaterdeterminant` , `sposet` , `multideterminant`

Attribute:

Name	**Datatype**	**Values**	**Default**	**Description**
`name/id`	Text	*Any*	‘’ ‘’	Name of determinant set
`type`	Text	See below	‘’ ‘’	Type of `sposet`
`keyword`	Text	NMO, GTO, STO	NMO	Type of orbital set generated
`transform`	Text	Yes/no	Yes	Transform to numerical radial functions?
`source`	Text	*Any*	Ion0	Particle set with the position of atom centers
`cuspCorrection`	Text	Yes/no	No	Apply cusp correction scheme to `sposet`?

Table 4 Options for the `determinantset` xml-block associated with atom-centered single particle orbital sets.

Listing 8.4: Basic input block for `basisset`.

```
<basisset name="LCAOBSet">
  <atomicBasisSet name="Gaussian-G2" angular="cartesian" elementType="C" normalized=
↪"no">
    <grid type="log" ri="1.e-6" rf="1.e2" npts="1001"/>
    <basisGroup rid="C00" n="0" l="0" type="Gaussian">
      <radfunc exponent="5.134400000000e-02" contraction="1.399098787100e-02"/>
      ...
    </basisGroup>
    ...
  </atomicBasisSet>
  <atomicBasisSet name="Gaussian-G2" angular="cartesian" type="Gaussian" elementType=
↪"C" normalized="no">
    ...
  </atomicBasisSet>
```

(continues on next page)

(continued from previous page)

```
  ...
</basisset>
```

`basisset` element:

Parent elements	`determinantset`
Child elements	`atomicBasisSet`

Attribute:

Name	Datatype	Values	Default	Description
`name` / `id`	Text	*Any*	” “	Name of atom-centered basis set

Table 5 Options for the `basisset` xml-block associated with atom-centered single particle orbital sets.

`AtomicBasisSet` element:

Parent elements	`basisset`
Child elements	`grid`, `basisGroup`

Attribute:

Name	Datatype	Values	Default	Description
`name` / `id`	Text	*Any*	” “	Name of atomic basis set
`angular`	Text	See below	Default	Type of angular functions
`expandYlm`	Text	See below	Yes	Expand Ylm shells?
`expM`	Text	See below	Yes	Add sign for $(-1)^m$?
`elementType/species`	Text	*Any*	e	Atomic species where functions are centered
`normalized`	Text	Yes/no	Yes	Are single particle functions normalized?

Table 6 Options for the `atomicBasisSet` xml-block.

`basicGroup` element:

Parent elements	`AtomicBasisSet`
Child elements	`radfunc`

Attribute:

Name	Datatype	Values	Default	Description
`rid/id`	Text	*Any*	‘’ ‘’	Name of the basisGroup
`type`	Text	*Any*	‘’ ‘’	Type of basisGroup
`n/l/m/s`	Integer	*Any*	0	Quantum numbers of basisGroup

Table 8.3.2 Options for the `basisGroup` xml-block.

Listing 8.5: Basic input block for `slaterdeterminant` with an atom-centered `sposet`.

```
<slaterdeterminant>
</slaterdeterminant>
```

element:

Parent elements:	
Child elements:	

Attribute:

Name	Datatype	Values	Default	Description
`name/id`	Text	*Any*	‘‘ ‘‘	Name of determinant set
	Text	*Any*	‘‘ ‘‘	

Detailed description of attributes:

In the following, we give a more detailed description of all the options presented in the various xml-blocks described in this section. Only nontrivial attributes are described. Those with simple yes/no options and whose previous description is enough to explain the intended behavior are not included.

`determinantset` attributes:

- **`type`** Type of `sposet`. For atom-centered based `sposets`, use type=”MolecularOrbital” or type=“MO.” Other options described elsewhere in this manual are “spline,” “composite,” “pw,” “heg,” “linearopt,” etc.
- **`keyword/key`** Type of basis set generated, which does not necessarily match the type of basis set on the input block. The three possible options are: NMO (numerical molecular orbitals), GTO (Gaussian-type orbitals), and STO (Slater-type orbitals). The default option is NMO. By default, QMCPACK will generate numerical orbitals from both GTO and STO types and use cubic or quintic spline interpolation to evaluate the radial functions. This is typically more efficient than evaluating the radial functions in the native basis (Gaussians or exponents) and allows for arbitrarily large contractions without any additional cost. To force use of the native expansion (not recommended), use GTO or STO for each type of input basis set.
- **`transform`** Request (or avoid) a transformation of the radial functions to NMO type. The default and recommended behavior is to transform to numerical radial functions. If `transform` is set to *yes*, the option `keyword` is ignored.
- **`cuspCorrection`** Enable (disable) use of the cusp correction algorithm (CASINO REFERENCE) for a `basisset` built with GTO functions. The algorithm is implemented as described in (CASINO REFERENCE) and works only with transform=”yes” and an input GTO basis set. No further input is needed.

`atomicBasisSet` attributes:

- **`name/id`** Name of the basis set. Names should be unique.
- **`angular`** Type of angular functions used in the expansion. In general, two angular basis functions are allowed: “spherical” (for spherical Ylm functions) and “Cartesian” (for functions of the type $x^n y^m z^l$).
- **`expandYlm`** Determines whether each basis group is expanded across the corresponding shell of m values (for spherical type) or consistent powers (for Cartesian functions). Options:
  - “No”: Do not expand angular functions across corresponding angular shell.

  - "Gaussian": Expand according to Gaussian03 format. This function is compatible only with angular="spherical." For a given input (l,m), the resulting order of the angular functions becomes (1,-1,0) for l=1 and (0,1,-1,2,-2,. . . ,l,-l) for general l.
  - "Natural": Expand angular functions according to (-l,-l+1,. . . ,l-1,l).
  - "Gamess": Expand according to Gamess' format for Cartesian functions. Notice that this option is compatible only with angular="Cartesian." If angular="Cartesian" is used, this option is not necessary.

- **expM** Determines whether the sign of the spherical Ylm function associated with m (-1^m) is included in the coefficient matrix or not.
- **elementType/species** Name of the species where basis functions are centered. Only one `atomicBasisSet` block is allowed per species. Additional blocks are ignored. The corresponding species must exist in the `particleset` given as the `source` option to `determinantset`. Basis functions for all the atoms of the corresponding species are included in the basis set, based on the order of atoms in the `particleset`.

`basisGroup` attributes:

- **type** Type of input basis radial function. Note that this refers to the type of radial function in the input xml-block, which might not match the radial function generated internally and used in the calculation (if `transform` is set to "yes"). Also note that different `basisGroup` blocks within a given `atomicBasisSet` can have different `types`.
- **n/l/m/s** Quantum numbers of the basis function. Note that if `expandYlm` is set to *"yes"* in `atomicBasisSet`, a full shell of basis functions with the appropriate values of *"m"* will be defined for the corresponding value of *"l."* Otherwise a single basis function will be given for the specific combination of *"(l,m)."*

`radfunc` attributes for `type` = *"Gaussian"*:

- `TBDoc`

`slaterdeterminant` attributes:

- `TBDoc`

8.3.3 Hybrid orbital representation

The hybrid representation of the single particle orbitals combines a localized atomic basis set around atomic cores and B-splines in the interstitial regions to reduce memory use while retaining high evaluation speed and either retaining or increasing overall accuracy. Full details are provided in [LEKS18], and **users of this feature are kindly requested to cite this paper**. In practice, we have seen that using a meshfactor=0.5 is often possible and achieves huge memory savings. Fig. 8.1 illustrates how the regions are assigned.

Orbitals within region A are computed as

$$\phi^A_n(\mathbf{r}) = R_{n,l,m}(r)Y_{l,m}(\hat{r})$$

Orbitals in region C are computed as the regular B-spline basis described in *Spline basis sets* above. The region B interpolates between A and C as

$$\phi^B_n(\mathbf{r}) = S(r)\phi^A_n(\mathbf{r}) + (1 - S(r))\phi^C_n(r) \tag{8.6}$$

$$(S(r) = \frac{1}{2} - \frac{1}{2}tanh\left[\alpha\left(\frac{r - r_{\mathrm{A/B}}}{r_{\mathrm{B/C}} - r_{\mathrm{A/B}}} - \frac{1}{2}\right)\right] \tag{8.7}$$

To enable hybrid orbital representation, the input XML needs to see the tag `hybridrep="yes"` shown in *Listing 6*.

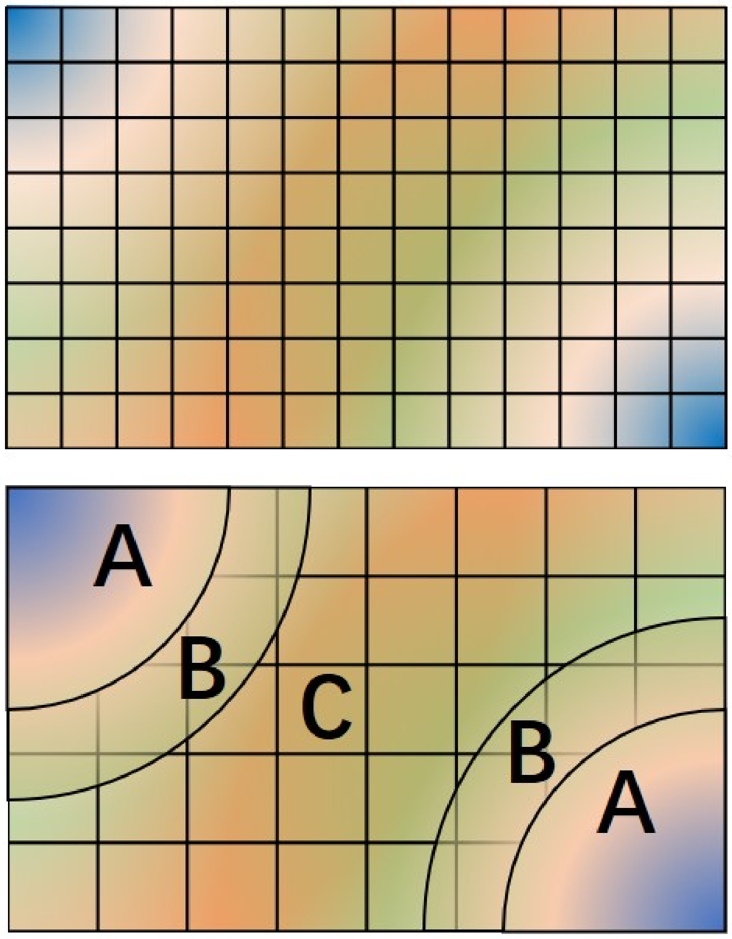


Fig. 8.1: Regular and hybrid orbital representation. Regular B-spline representation (left panel) contains only one region and a sufficiently fine mesh to resolve orbitals near the nucleus. The hybrid orbital representation (right panel) contains near nucleus regions (A) where spherical harmonics and radial functions are used, buffers or interpolation regions (B), and an interstitial region (C) where a coarse B-spline mesh is used.

Listing 8.6: Hybrid orbital representation input example.

```
<determinantset type="bspline" source="i" href="pwscf.h5"
              tilematrix="1 1 3 1 2 -1 -2 1 0" twistnum="-1" gpu="yes" meshfactor="0.8
↪"
              twist="0  0  0" precision="single" hybridrep="yes">
  ...
</determinantset>
```

Second, the information describing the atomic regions is required in the particle set, shown in *Listing 7*.

Listing 8.7: particleset elements for ions with information needed by hybrid orbital representation.

```
<group name="Ni">
  <parameter name="charge">          18 </parameter>
  <parameter name="valence">         18 </parameter>
  <parameter name="atomicnumber" >   28 </parameter>
  <parameter name="cutoff_radius" > 1.6 </parameter>
  <parameter name="inner_cutoff" >  1.3 </parameter>
  <parameter name="lmax" >            5 </parameter>
  <parameter name="spline_radius" > 1.8 </parameter>
  <parameter name="spline_npoints">  91 </parameter>
</group>
```

The parameters specific to hybrid representation are listed as

`attrib` element

Attribute:

Name	Datatype	Values	Default	Description
`cutoff_radius`	Real	>=0.0	*None*	Cutoff radius for B/C boundary
`lmax`	Integer	>=0	*None*	Largest angular channel
`inner_cutoff`	Real	>=0.0	Dep.	Cutoff radius for A/B boundary
`spline_radius`	Real	>0.0	Dep.	Radial function radius used in spine
`spline_npoints`	Integer	>0	Dep.	Number of spline knots

- `cutoff_radius` is required for every species. If a species is intended to not be covered by atomic regions, setting the value 0.0 will put default values for all the reset parameters. A good value is usually a bit larger than the core radius listed in the pseudopotential file. After a parametric scan, pick the one from the flat energy region with the smallest variance.
- `lmax` is required if `cutoff_radius` > 0.0. This value usually needs to be at least the highest angular momentum plus 2.
- `inner_cutoff` is optional and set as `cutoff_radius` -0.3 by default, which is fine in most cases.
- `spline_radius` and `spline_npoints` are optional. By default, they are calculated based on `cutoff_radius` and a grid displacement 0.02 bohr. If users prefer inputing them, it is required that `cutoff_radius` <= `spline_radius` $- 2 \times$ `spline_radius`/(`spline_npoints` $- 1$).

In addition, the hybrid orbital representation allows extra optimization to speed up the nonlocal pseudopotential evaluation using the batched algorithm listed in *Pseudopotentials*.

8.3.4 Plane-wave basis sets

8.3.5 Homogeneous electron gas

The interacting Fermi liquid has its own special `determinantset` for filling up a Fermi surface. The shell number can be specified separately 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 might not fill up a complete shell.

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

Listing 8.8: 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 8.9: 2D Fermi liquid example (Slater Jastrow wavefunction)

```
<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>
```

8.4 Jastrow Factors

Jastrow factors are among the simplest and most effective ways of including dynamical correlation in the trial many body wavefunction. The resulting many body wavefunction is expressed as the product of an antisymmetric (in the case of Fermions) or symmetric (for Bosons) part and a correlating Jastrow factor like so:

$$\Psi(\vec{R}) = \mathcal{A}(\vec{R}) \exp\left[J(\vec{R})\right] \tag{8.8}$$

In this section we will detail the types and forms of Jastrow factor used in QMCPACK. Note that each type of Jastrow factor needs to be specified using its own individual `jastrow` XML element. For this reason, we have repeated the specification of the `jastrow` tag in each section, with specialization for the options available for that given type of Jastrow.

8.4.1 One-body Jastrow functions

The one-body Jastrow factor is a form that allows for the direct inclusion of correlations between particles that are included in the wavefunction with particles that are not explicitly part of it. The most common example of this are correlations between electrons and ions.

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

Input Specification

Jastrow element:

name	**datatype**	**values**	**defaults**	**description**
name	text		(required)	Unique name for this Jastrow function
type	text	One-body	(required)	Define a one-body function
function	text	Bspline	(required)	BSpline Jastrow
	text	pade2		Pade form
	text	…		…
source	text	name	(required)	Name of attribute of classical particle set
print	text	yes / no	yes	Jastrow factor printed in external file?

elements				
	Correlation			
Contents				
	(None)			

To be more concrete, the one-body Jastrow factors used to describe correlations between electrons and ions take the form below:

$$J1 = \sum_{I}^{ion0} \sum_{i}^{e} u_{ab}(|r_i - R_I|) \tag{8.9}$$

where I runs over all of the ions in the calculation, i runs over the electrons and u_{ab} describes the functional form of the correlation between them. Many different forms of u_{ab} are implemented in QMCPACK. We will detail two of the most common ones below.

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 [EKCS12]. Here u_{ab} is an interpolating 1D B-spline (tricublc spline on a linear grid) 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 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.

Correlation element:

Name	**Datatype**	**Values**	**Defaults**	**Description**
ElementType	Text	Name	See below	Classical particle target
SpeciesA	Text	Name	See below	Classical particle target
SpeciesB	Text	Name	See below	Quantum species target
Size	Integer	> 0	(Required)	Number of coefficients
Rcut	Real	> 0	See below	Distance at which the correlation goes to 0
Cusp	Real	≥ 0	0	Value for use in Kato cusp condition
Spin	Text	Yes or no	No	Spin dependent Jastrow factor

Elements				
	Coefficients			
Contents				
	(None)			

Additional information:

- **elementType, speciesA, speciesB, spin** For a spin-independent Jastrow factor (spin = "no"), elementType should be the name of the group of ions in the classical particleset to which the quantum particles should be correlated. For a spin-dependent Jastrow factor (spin = "yes"), set speciesA to the group name in the classical particleset and speciesB to the group name in the quantum particleset.
- **rcut** The cutoff distance for the function in atomic units (bohr). 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.
- **cusp** The one-body Jastrow factor can be used to make the wavefunction satisfy the electron-ion cusp condition :cite:`kato`. In this case, the derivative of the Jastrow factor as the electron approaches the nucleus will be given by

$$\left(\frac{\partial J}{\partial r_{iI}}\right)_{r_{iI}=0} = -Z. \tag{8.10}$$

Note that if the antisymmetric part of the wavefunction satisfies the electron-ion cusp condition (for instance by using single-particle orbitals that respect the cusp condition) or if a nondivergent pseudopotential is used, the Jastrow should be cuspless at the nucleus and this value should be kept at its default of 0.

Coefficients element:

Name	Datatype	Values	Defaults	Description
Id	Text		(Required)	Unique identifier
Type	Text	Array	(Required)	
Optimize	Text	Yes or no	Yes	if no, values are fixed in optimizations
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 name of the particleset holding the ionic positions is “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 up-spin and seven down-spin parameters. The cutoff distance is set to 6 atomic units. Note here that the particleset holding the ions is labeled as ion0 rather than “i,” as in the other example. Also in this case, the ion is lithium with a coulomb potential, so the cusp condition is satisfied by setting cusp=”d.”

```
<jastrow name="J1" type="One-Body" function="Bspline" source="ion0" spin="yes">
  <correlation speciesA="Li" speciesB="u" size="7" rcut="6">
    <coefficients id="eLiu" cusp="3.0" 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="eLid" cusp="3.0" type="Array">
    0.0 0.0 0.0 0.0 0.0 0.0 0.0
    </coefficients>
  </correlation>
</jastrow>
```

Pade form

Although the spline Jastrow factor is the most flexible and most commonly used form implemented in QMCPACK, there are times where its flexibility can make it difficult to optimize. As an example, a spline Jastrow with a very large cutoff can be difficult to optimize for isolated systems such as molecules because of the small number of samples present in the tail of the function. In such cases, a simpler functional form might be advantageous. The second-order Pade Jastrow factor, given in (8.11), is a good choice in such cases.

$$u_{ab}(r) = \frac{a*r+c*r^2}{1+b*r} \tag{8.11}$$

Unlike the spline Jastrow factor, which includes a cutoff, this form has an infinite range and will be applied to every particle pair (subject to the minimum image convention). It also is a cuspless Jastrow factor, so it should be used either in combination with a single particle basis set that contains the proper cusp or with a smooth pseudopotential.

Correlation element:

Name	Datatype	Values	Defaults	Description
ElementType	Text	Name	See below	Classical particle target
Elements				
	Coefficients			
Contents				
	(None)			

Parameter element:

Name	Datatype	Values	Defaults	Description
Id	String	Name	(Required)	Name for variable
Name	String	A or B or C	(Required)	See (8.11)
Optimize	Text	Yes or no	Yes	If no, values are fixed in optimizations

Elements				
(None)				
Contents				
(No name)	Real	Parameter value	(Required)	Jastrow coefficients

Example use case

Specify a spin-independent function with independent Jastrow factors for two different species (Li and H). The name of the particleset holding the ionic positions is "i."

```
<jastrow name="J1" function="pade2" type="One-Body" print="yes" source="i">
  <correlation elementType="Li">
    <var id="LiA" name="A">  0.34 </var>
    <var id="LiB" name="B"> 12.78 </var>
    <var id="LiC" name="C">  1.62 </var>
  </correlation>
  <correlation elementType="H"">
    <var id="HA" name="A">  0.14 </var>
    <var id="HB" name="B"> 6.88 </var>
    <var id="HC" name="C"> 0.237 </var>
  </correlation>
</jastrow>
```

Short Range Cusp Form

The idea behind this functor is to encode nuclear cusps and other details at very short range around a nucleus in the region that the Gaussian orbitals of quantum chemistry are not capable of describing correctly. The functor is kept short ranged, because outside this small region, quantum chemistry orbital expansions are already capable of taking on the correct shapes. Unlike a pre-computed cusp correction, this optimizable functor can respond to changes in the wave function during VMC optimization. The functor's form is

$$u(r) = -\exp\left(-r/R_0\right)\left(AR_0 + \sum_{k=0}^{N-1} B_k \frac{(r/R_0)^{k+2}}{1+(r/R_0)^{k+2}}\right) \tag{8.12}$$

in which R_0 acts as a soft cutoff radius ($u(r)$ decays to zero quickly beyond roughly this distance) and A determines the cusp condition.

$$\lim_{r\to 0} \frac{\partial u}{\partial r} = A \tag{8.13}$$

The simple exponential decay is modified by the N coefficients B_k that define an expansion in sigmoidal functions, thus adding detailed structure in a short-ranged region around a nucleus while maintaining the correct cusp condition at the nucleus. Note that sigmoidal functions are used instead of, say, a bare polynomial expansion, as they trend to unity past the soft cutoff radius and so interfere less with the exponential decay that keeps the functor short ranged. Although A, R_0, and the B_k coefficients can all be optimized as variational parameters, A will typically be fixed as the desired cusp condition is known.

To specify this one-body Jastrow factor, use an input section like the following.

```
<jastrow name="J1Cusps" type="One-Body" function="shortrangecusp" source="ion0" print=
↪"yes">
  <correlation rcut="6" cusp="3" elementType="Li">
    <var id="LiCuspR0" name="R0" optimize="yes"> 0.06 </var>
    <coefficients id="LiCuspB" type="Array" optimize="yes">
      0 0 0 0 0 0 0 0 0 0
    </coefficients>
  </correlation>
  <correlation rcut="6" cusp="1" elementType="H">
    <var id="HCuspR0" name="R0" optimize="yes"> 0.2 </var>
    <coefficients id="HCuspB" type="Array" optimize="yes">
      0 0 0 0 0 0 0 0 0 0
    </coefficients>
  </correlation>
</jastrow>
```

Here "rcut" is specified as the range beyond which the functor is assumed to be zero. The value of A can either be specified via the "cusp" option as shown above, in which case its optimization is disabled, or through its own "var" line as for R_0, in which case it can be specified as either optimizable ("yes") or not ("no"). The coefficients B_k are specified via the "coefficients" section, with the length N of the expansion determined automatically based on the length of the array.

Note that this one-body Jastrow form can (and probably should) be used in conjunction with a longer ranged one-body Jastrow, such as a spline form. Be sure to set the longer-ranged Jastrow to be cusp-free!

8.4.2 Two-body Jastrow functions

The two-body Jastrow factor is a form that allows for the explicit inclusion of dynamic correlation between two particles included in the wavefunction. It is almost always given in a spin dependent form so as to satisfy the Kato cusp condition between electrons of different spins [Kat51].

The two body Jastrow function is specified within a `wavefunction` element and must contain one or more correlation elements specifying additional parameters as well as the actual coefficients. *Example use cases* gives examples of the typical nesting of `jastrow`, `correlation` and `coefficient` elements.

Input Specification

Jastrow element:

name	datatype	values	defaults	description
name	text		(required)	Unique name for this Jastrow function
type	text	Two-body	(required)	Define a one-body function
function	text	Bspline	(required)	BSpline Jastrow
print	text	yes / no	yes	Jastrow factor printed in external file?
elements				
	Correlation			
Contents				
	(None)			

The two-body Jastrow factors used to describe correlations between electrons take the form

$$J2 = \sum_{i}^{e}\sum_{j>i}^{e} u_{ab}(|r_i - r_j|) \tag{8.14}$$

The most commonly used form of two body Jastrow factor supported by the code is a splined Jastrow factor, with many similarities to the one body spline Jastrow.

Spline form

The two-body spline Jastrow function is the most commonly used two-body Jastrow for solids. This form was first described and used in [EKCS12]. Here u_{ab} is an interpolating 1D B-spline (tricublc spline on a linear grid) 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. r_i and r_j are typically electron positions. The cusp condition as r_i approaches r_j is set by the relative spin of the electrons.

Correlation element:

Name	Datatype	Values	Defaults	Description
SpeciesA	Text	U or d	(Required)	Quantum species target
SpeciesB	Text	U or d	(Required)	Quantum species target
Size	Integer	> 0	(Required)	Number of coefficients
Rcut	Real	> 0	See below	Distance at which the correlation goes to 0
Spin	Text	Yes or no	No	Spin-dependent Jastrow factor
Elements				
	Coefficients			
Contents				
	(None)			

Additional information:

- `speciesA, speciesB` The scale function u(r) is defined for species pairs uu and ud.

There is no need to define ud or dd since uu=dd and ud=du. The cusp condition is computed internally based on the charge of the quantum particles.

Coefficients element:

Name	Datatype	Values	Defaults	Description
Id	Text		(Required)	Unique identifier
Type	Text	Array	(Required)	
Optimize	Text	Yes or no	Yes	If no, values are fixed in optimizations
Elements				
(None)				
Contents				
(No name)	Real array		Zeros	Jastrow coefficients

Example use cases

Specify a spin-dependent function with four parameters for each channel. In this case, the cusp is set at a radius of 4.0 bohr (rather than to the default of the Wigner Seitz cell radius). Also, in this example, the coefficients are set to not be optimized during an optimization step.

```
<jastrow name="J2" type="Two-Body" function="Bspline" print="yes">
  <correlation speciesA="u" speciesB="u" size="8" rcut="4.0">
    <coefficients id="uu" type="Array" optimize="no"> 0.2309049836 0.1312646071 0.
↪05464141356 0.01306231516</coefficients>
  </correlation>
  <correlation speciesA="u" speciesB="d" size="8" rcut="4.0">
    <coefficients id="ud" type="Array" optimize="no"> 0.4351561096 0.2377951747 0.
↪1129144262 0.0356789236</coefficients>
  </correlation>
</jastrow>
```

8.4.3 User defined functional form

To aid in implementing different forms for $u_{ab}(r)$, there is a script that uses a symbolic expression to generate the appropriate code (with spatial and parameter derivatives). The script is located in `src/QMCWaveFunctions/Jastrow/codegen/user_jastrow.py`. The script requires Sympy (www.sympy.org) for symbolic mathematics and code generation.

To use the script, modify it to specify the functional form and a list of variational parameters. Optionally, there may be fixed parameters - ones that are specified in the input file, but are not part of the variational optimization. Also one symbol may be specified that accepts a cusp value in order to satisfy the cusp condition. There are several example forms in the script. The default form is the simple Padé.

Once the functional form and parameters are specified in the script, run the script from the `codegen` directory and recompile QMCPACK. The main output of the script is the file `src/QMCWaveFunctions/Jastrow/UserFunctor.h`. The script also prints information to the screen, and one section is a sample XML input block containing all the parameters.

There is a unit test in `src/QMCWaveFunctions/test/test_user_jastrow.cpp` to perform some minimal testing of the Jastrow factor. The unit test will need updating to properly test new functional forms. Most of the changes relate to the number and name of variational parameters.

Jastrow element:

name	datatype	values	defaults	description
name	text		(required)	Unique name for this Jastrow function
type	text	One-body	(required)	Define a one-body function
		Two-body	(required)	Define a two-body function
function	text	user	(required)	User-defined functor

See other parameters as appropriate for one or two-body functions

elements				
	Correlation			
Contents				
	(None)			

8.4.4 Long-ranged Jastrow factors

While short-ranged Jastrow factors capture the majority of the benefit for minimizing the total energy and the energy variance, long-ranged Jastrow factors are important to accurately reproduce the short-ranged (long wavelength) behavior of quantities such as the static structure factor, and are therefore essential for modern accurate finite size corrections in periodic systems.

Below two types of long-ranged Jastrow factors are described. The first (the k-space Jastrow) is simply an expansion of the one and/or two body correlation functions in plane waves, with the coefficients comprising the optimizable parameters. The second type have few variational parameters and use the optimized breakup method of Natoli and Ceperley [NC95] (the Yukawa and Gaskell RPA Jastrows).

Long-ranged Jastrow: k-space Jastrow

The k-space Jastrow introduces explicit long-ranged dependence commensurate with the periodic supercell. This Jastrow is to be used in periodic boundary conditions only.

The input for the k-space Jastrow fuses both one and two-body forms into a single element and so they are discussed together here. The one- and two-body terms in the k-Space Jastrow have the form:

$$J_1 = \sum_{G\neq 0} b_G \rho_G^I \rho_{-G} \tag{8.15}$$

$$J_2 = \sum_{G\neq 0} a_G \rho_G \rho_{-G} \tag{8.16}$$

Here ρ_G is the Fourier transform of the instantaneous electron density:

$$\rho_G = \sum_{n\in electrons} e^{iG\cdot r_n} \tag{8.17}$$

and ρ_G^I has the same form, but for the fixed ions. In both cases the coefficients are restricted to be real, though in general the coefficients for the one-body term need not be. See feature-kspace-jastrow for more detail.

Input for the k-space Jastrow follows the familar nesting of `jastrow-correlation-coefficients` elements, with attributes unique to the k-space Jastrow at the `correlation` input level.

`jastrow type=kSpace` element:

parent elements:	`wavefunction`
child elements:	`correlation`

attributes:

Name	Datatype	Values	Default	Description
`type`^{r}	text	**kSpace**		must be kSpace
`name`^{r}	text	*anything*	0	Unique name for Jastrow
`source`^{r}	text	`particleset.name`		Ion particleset name

`correlation` element:

parent elements:	`jastrow type=kSpace`
child elements:	`coefficients`

attributes:

Name	Datatype	Values	Default	Description
`type`^{r}	text	**One-body, Two-Body**		Must be One-body/Two-body
`kc`^{r}	real	kc ≥ 0	0.0	k-space cutoff in a.u.
`symmetry`^{o}	text	crystal,isotropic,none	crystal	symmetry of coefficients
`spinDependent`^{o}	boolean	yes,no	no	*No current function*

`coefficients` element:

parent elements:	`correlation`
child elements:	*None*

attributes:

Name	Datatype	Values	Default	Description
`id`^{r}	text	*anything*	cG1/cG2	Label for coeffs
`type`^{r}	text	`Array`	0	Must be Array

body text: The body text is a list of real values for the parameters.

Additional information:

- It is normal to provide no coefficients as an initial guess. The number of coefficients will be automatically calculated according to the k-space cutoff + symmetry and set to zero.
- Providing an incorrect number of parameters also results in all parameters being set to zero.
- There is currently no way to turn optimization on/off for the k-space Jastrow. The coefficients are always optimized.
- Spin dependence is currently not implemented for this Jastrow.
- `kc`: Parameters with G vectors magnitudes less than `kc` are included in the Jastrow. If `kc` is zero, it is the same as excluding the k-space term.
- `symmetry=crystal`: Impose crystal symmetry on coefficients according to the structure factor.
- `symmetry=isotropic`: Impose spherical symmetry on coefficients according to G-vector magnitude.
- `symmetry=none`: Impose no symmetry on the coefficients.

Listing 8.10: k-space Jastrow with one- and two-body terms.

```
<jastrow type="kSpace" name="Jk" source="ion0">
  <correlation kc="4.0" type="One-Body" symmetry="cystal">
    <coefficients id="cG1" type="Array">
    </coefficients>
  </correlation>
  <correlation kc="4.0" type="Two-Body" symmetry="crystal">
    <coefficients id="cG2" type="Array">
    </coefficients>
 </correlation>
</jastrow>
```

Listing 8.11: k-space Jastrow with one-body term only.

```
<jastrow type="kSpace" name="Jk" source="ion0">
   <correlation kc="4.0" type="One-Body" symmetry="crystal">
      <coefficients id="cG1" type="Array">
      </coefficients>
   </correlation>
</jastrow>
```

Listing 8.12: k-space Jastrow with two-body term only.

```
<jastrow type="kSpace" name="Jk" source="ion0">
   <correlation kc="4.0" type="Two-Body" symmetry="crystal">
      <coefficients id="cG2" type="Array">
      </coefficients>
   </correlation>
</jastrow>
```

Long-ranged Jastrows: Gaskell RPA and Yukawa forms

NOTE: The Yukawa and RPA Jastrows do not work at present and are currently being revived. Please contact the developers if you are interested in using them.

The exact Jastrow correlation functions contain terms which have a form similar to the Coulomb pair potential. In periodic systems the Coulomb potential is replaced by an Ewald summation of the bare potential over all periodic image cells. This sum is often handled by the optimized breakup method [NC95] and this same approach is applied to the long-ranged Jastrow factors in QMCPACK.

There are two main long-ranged Jastrow factors of this type implemented in QMCPACK: the Gaskell RPA [Gas61][Gas62] form and the [Cep78] form. Both of these forms were used by Ceperley in early studies of the electron gas [Cep78], but they are also appropriate starting points for general solids.

The Yukawa form is defined in real space. It's long-range form is formally defined as

$$u_Y^{PBC}(r) = \sum_{L\neq 0}\sum_{i<j} u_Y(|r_i - r_j + L|) \tag{8.18}$$

with $u_Y(r)$ given by

$$u_Y(r) = \frac{a}{r}\left(1 - e^{-r/b}\right) \tag{8.19}$$

In QMCPACK a slightly more restricted form is used:

$$u_Y(r) = \frac{r_s}{r}\left(1 - e^{-r/\sqrt{r_s}}\right) \tag{8.20}$$

here "r_s" is understood to be a variational parameter.

The Gaskell RPA form—which contains correct short/long range limits and minimizes the total energy of the electron gas within the RPA—is defined directly in k-space:

$$u_{RPA}(k) = -\frac{1}{2S_0(k)} + \frac{1}{2}\left(\frac{1}{S_0(k)^2} + \frac{4m_e v_k}{\hbar^2 k^2}\right)^{1/2} \tag{8.21}$$

where \$v_k\$ is the Fourier transform of the Coulomb potential and $S_0(k)$ is the static structure factor of the non-interacting electron gas:

$$S_0(k) = \begin{cases} 1 & k > 2k_F \\ \frac{3k}{4k_F} - \frac{1}{2}\left(\frac{k}{2k_F}\right)^3 & k < 2k_F \end{cases}$$

When written in atomic units, RPA Jastrow implemented in QMCPACK has the form

$$u_{RPA}(k) = \frac{1}{2N_e}\left(-\frac{1}{S_0(k)} + \left(\frac{1}{S_0(k)^2} + \frac{12}{r_s^3 k^4}\right)^{1/2}\right) \tag{8.22}$$

Here "r_s" is again a variational parameter and $k_F \equiv (\frac{9\pi}{4r_s^3})^{1/3}$.

For both the Yukawa and Gaskell RPA Jastrows, the default value for r_s is $r_s = (\frac{3\Omega}{4\pi N_e})^{1/3}$.

`jastrow type=Two-Body function=rpa/yukawa` element:

parent elements:	`wavefunction`
child elements:	`correlation`

attributes:

Name	Datatype	Values	Default	Description
`type`^{r}	text	**Two-body**		Must be two-body
`function`^{r}	text	**rpa/yukawa**		Must be rpa or yukawa
`name`^{r}	text	*anything*	RPA_Jee	Unique name for Jastrow
`longrange`^{o}	boolean	yes/no	yes	Use long-range part
`shortrange`^{o}	boolean	yes/no	yes	Use short-range part

parameters:

Name	Datatype	Values	Default	Description
`rs`^{o}	rs	$r_s > 0$	$\frac{3\Omega}{4\pi N_e}$	Avg. elec-elec distance
`kc`^{o}	kc	$k_c > 0$	$2\left(\frac{9\pi}{4}\right)^{1/3}\frac{4\pi N_e}{3\Omega}$	k-space cutoff

Listing 8.13: Two body RPA Jastrow with long- and short-ranged parts.

```
<jastrow name=''Jee'' type=''Two-Body'' function=''rpa''>
</jastrow>
```

8.4.5 Three-body Jastrow functions

Explicit three-body correlations can be included in the wavefunction via the three-body Jastrow factor. The three-body electron-electron-ion correlation function ($u_{\sigma\sigma' I}$) currently used in is identical to the one proposed in [DTN04]:

$$\begin{aligned} u_{\sigma\sigma' I}(r_{\sigma I}, r_{\sigma' I}, r_{\sigma\sigma'}) = &\sum_{\ell=0}^{M_{eI}}\sum_{m=0}^{M_{eI}}\sum_{n=0}^{M_{ee}}\gamma_{\ell mn} r_{\sigma I}^{\ell} r_{\sigma' I}^{m} r_{\sigma\sigma'}^{n} \\ &\times \left(r_{\sigma I} - \frac{r_c}{2}\right)^3 \Theta\left(r_{\sigma I} - \frac{r_c}{2}\right) \\ &\times \left(r_{\sigma' I} - \frac{r_c}{2}\right)^3 \Theta\left(r_{\sigma' I} - \frac{r_c}{2}\right) \end{aligned}$$

Here M_{eI} and M_{ee} are the maximum polynomial orders of the electron-ion and electron-electron distances, respectively, $\{\gamma_{\ell mn}\}$ are the optimizable parameters (modulo constraints), r_c is a cutoff radius, and r_{ab} are the distances between electrons or ions a and b. i.e. The correlation function is only a function of the interparticle distances and not a more complex function of the particle positions, $\mathbf{r}$. As indicated by the Θ functions, correlations are set to zero beyond a distance of $r_c/2$ in either of the electron-ion distances and the largest meaningful electron-electron distance is r_c. This is the highest-order Jastrow correlation function currently implemented.

Today, solid state applications of QMCPACK usually utilize one and two-body B-spline Jastrow functions, with calculations on heavier elements often also using the three-body term described above.

Example use case

Here is an example of H2O molecule. After optimizing one and two body Jastrow factors, add the following block in the wavefunction. The coefficients will be filled zero automatically if not given.

```
<jastrow name="J3" type="eeI" function="polynomial" source="ion0" print="yes">
  <correlation ispecies="O" especies="u" isize="3" esize="3" rcut="10">
    <coefficients id="uuO" type="Array" optimize="yes"> </coefficients>
  </correlation>
  <correlation ispecies="O" especies1="u" especies2="d" isize="3" esize="3" rcut="10">
    <coefficients id="udO" type="Array" optimize="yes"> </coefficients>
  </correlation>
  <correlation ispecies="H" especies="u" isize="3" esize="3" rcut="10">
    <coefficients id="uuH" type="Array" optimize="yes"> </coefficients>
  </correlation>
  <correlation ispecies="H" especies1="u" especies2="d" isize="3" esize="3" rcut="10">
    <coefficients id="udH" type="Array" optimize="yes"> </coefficients>
  </correlation>
</jastrow>
```

8.5 Multideterminant wavefunctions

Multiple schemes to generate a multideterminant wavefunction are possible, from CASSF to full CI or selected CI. The QMCPACK converter can convert MCSCF multideterminant wavefunctions from GAMESS [SBB+93] and CIPSI [EG13] wavefunctions from Quantum Package [Sce17] (QP). Full details of how to run a CIPSI calculation and convert the wavefunction for QMCPACK are given in cipsi.

The script `utils/determinants_tools.py` can be used to generate useful information about the multideterminant wavefunction. This script takes, as a required argument, the path of an h5 file corresponding to the wavefunction. Used without optional arguments, it prints the number of determinants, the number of CSFs, and a histogram of the excitation degree.

```
> determinants_tools.py ./tests/molecules/C2_pp/C2.h5
Summary:
excitation degree 0 count: 1
excitation degree 1 count: 6
excitation degree 2 count: 148
excitation degree 3 count: 27
excitation degree 4 count: 20

n_det 202
n_csf 104
```

If the `--verbose` argument is used, the script will print each determinant, the associated CSF, and the excitation degree relative to the first determinant.

```
> determinants_tools.py -v ./tests/molecules/C2_pp/C2.h5 | head
1
alpha  1111000000000000000000000000000000000000000000000000000000
beta   1111000000000000000000000000000000000000000000000000000000
scf    2222000000000000000000000000000000000000000000000000000000
excitation degree  0

2
alpha  1011100000000000000000000000000000000000000000000000000000
beta   1011100000000000000000000000000000000000000000000000000000
scf    2022200000000000000000000000000000000000000000000000000000
excitation degree  2
```

8.6 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

$$\mathbf{x}'_{i_\alpha}=\mathbf{x}_{i_\alpha}+\sum_{\alpha\leq\beta}\sum_{i_\alpha\neq j_\beta}\eta^{\alpha\beta}(|\mathbf{x}_{i_\alpha}-\mathbf{x}_{j_\beta}|)(\mathbf{x}_{i_\alpha}-\mathbf{x}_{j_\beta})\,. \tag{8.23}$$

Here, $\eta^{\alpha\beta}(|\mathbf{x}_{i_\alpha}-\mathbf{x}_{j_\beta}|)$ is a radially symmetric backflow 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 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, we explain how to describe general terms such as (8.23) in a QMCPACK XML file. For specificity, we 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 labeled "Summary of QMC systems."

8.6.1 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	Default	Description
name	Text		(Required)	Unique name for this Jastrow function.
type	Text	"e-I"	(Required)	Define a one-body backflow transformation.
	Text	"e-e"		Define a two-body backflow transformation.
function	Text	B-spline	(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 *Spline form* for more information.

8.6.2 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 hydrogen and helium 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 they appear in the particle set. If in our example, helium is listed first and hydrogen is listed second, the following example would be correct. However, switching backflow declaration to hydrogen first then helium, will result in an error. Outside of this, declaration of one-body blocks and two-body blocks are not 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>
```

(continues on next page)

(continued from previous page)

```
<!--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 works only 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 is where one and two-body terms are defined -->
         </transformation>
     </backflow>

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

8.6.3 Optimization Tips

Backflow is notoriously difficult to optimize—it is extremely nonlinear in the variational parameters and moves the nodal surface around. As such, it is likely that a full Jastrow+Backflow optimization with all parameters initialized to zero might not converge in a reasonable time. If you are experiencing this problem, the following pointers are suggested (in no particular order).

Get a good starting guess for Ψ_T:

1. Try optimizing the Jastrow first without backflow.
2. Freeze the Jastrow parameters, introduce only the e-e terms in the backflow transformation, and optimize these parameters.
3. Freeze the e-e backflow parameters, and then optimize the e-I terms.
    - If difficulty is encountered here, try optimizing each species independently.
4. Unfreeze all Jastrow, e-e backflow, and e-I backflow parameters, and reoptimize.

Optimizing Backflow Terms

It is possible that the previous prescription might grind to a halt in steps 2 or 3 with the inability to optimize the e-e or e-I backflow transformation independently, especially if it is initialized to zero. One way to get around this is to build a good starting guess for the e-e or e-I backflow terms iteratively as follows:

1. Start off with a small number of knots initialized to zero. Set r_{cut} to be small (much smaller than an interatomic distance).
2. Optimize the backflow function.
3. If this works, slowly increase r_{cut} and/or the number of knots.
4. Repeat steps 2 and 3 until there is no noticeable change in energy or variance of Ψ_T.

Tweaking the Optimization Run

The following modifications are worth a try in the optimization block:

- Try setting “useDrift” to “no.” This eliminates the use of wavefunction gradients and force biasing in the VMC algorithm. This could be an issue for poorly optimized wavefunctions with pathological gradients.
- Try increasing “exp0” in the optimization block. Larger values of exp0 cause the search directions to more closely follow those predicted by steepest-descent than those by the linear method.

Note that the new adaptive shift optimizer has not yet been tried with backflow wavefunctions. It should perform better than the older optimizers, but a considered optimization process is still recommended.

8.7 Finite-difference linear response wave functions

The finite-difference linear response wavefunction (FDLR) is an experimental wavefunction type described in detail in [BN17]. In this method, the wavefunction is formed as the linear response of some existing trial wavefunction in QMCPACK. This derivatives of this linear response are approximated by a simple finite difference.

Forming a wavefunction within the linear response space of an existing ansatz can be very powerful. For example, a configuration interaction singles (CIS) wavefunction can be formed as a linear combination of the first derivatives of a Slater determinant (with respect to its orbital rotation parameters). Thus, in this sense, CIS is the linear response of Hartree–Fock theory.

Forming a CIS wavefunction as the linear response of an optimizable Slater determinant is where all testing of this wavefunction has been performed. In theory, the implementation is flexible and can be used with other trial wavefunctions in QMCPACK, but this has not been tested; the FDLR trial wavefunction is experimental.

Mathematically, the FDLR wavefunction has the form

$$\Psi_{\text{FDLR}}(\mu, \mathbf{X}) = \Psi(\mathbf{X} + \mu) - \Psi(\mathbf{X} - \mu)\,, \tag{8.24}$$

where $\Psi(\mathbf{P})$ is some trial wavefunction in QMCPACK, and $\mathbf{P}$ is its optimizable parameters. $\mathbf{X}$ is the “base” parameters about which the finite difference is performed (for example, an overall orbital rotation). μ is the “finite-difference” parameters, which define the direction of the derivative, and whose magnitude determines the magnitude of the finite difference. In the limit that the magnitude of $\mathbf{mu}$ goes to 0, the Ψ_{FDLR} object just defined becomes equivalent to

$$\Psi_{\text{FDLR}}(\mu, \mathbf{X}) = \sum_{pq} \mu_{pq} \frac{\partial \Psi_{\text{det}}(\mathbf{X})}{\partial X_{pq}}\,, \tag{8.25}$$

which is the desired linear response wavefunction we are approximating. In the case that $\Psi(\mathbf{P})$ is a determinant with orbital rotation parameters $\mathbf{P}$, the previous equation is a CIS wavefunction with CIS expansion coefficients μ and orbital rotation $\mathbf{X}$.

8.7.1 Input specifications

An FDLR wavefunction is specified within a `<fdlr> ... </fdlr>` block.

To fully specify an FDLR wavefunction as done previously, we require the initial parameters for both $\mathbf{X}$ and μ to be input. This therefore requires two trial wavefunctions to be provided on input. Each of these is best specified in its own XML file. The names of these two files are provided in an `<include>` tag via `<include wfn_x_href=" ... " wfn_d_href=" ... ">`. `wfn_x_href` specifies the file that will hold the $\mathbf{X}$ parameters. `wfn_d_href` specifies the file that will hold the μ parameters.

Other options inside the `<include>` tag are `opt_x` and `opt_d`, which specify whether or not $\mathbf{X}$ and μ parameters are optimizable, respectively.

8.7.2 Example Use Case

```
<fdlrwfn name="FDLR">
  <include wfn_x_href="h2.wfn_x.xml" wfn_d_href="h2.wfn_d.xml" opt_x="yes" opt_d="yes
↪"/>
</fdlrwfn>
```

with the `h2.wfn_x.xml` file containing one of the wavefunctions and corresponding set of $\mathbf{X}$ parameters, such as:

```
<?xml version="1.0"?>
<wfn_x>
    <determinantset name="LCAOBSet" type="MolecularOrbital" transform="yes" source=
↪"ion0">
      <basisset name="LCAOBSet">
        <atomicBasisSet name="Gaussian-G2" angular="cartesian" type="Gaussian"␣
↪elementType="H" normalized="no">
          <grid type="log" ri="1.e-6" rf="1.e2" npts="1001"/>
          <basisGroup rid="H00" n="0" l="0" type="Gaussian">
            <radfunc exponent="1.923840000000e+01" contraction="3.282799101900e-02"/>
            <radfunc exponent="2.898720000000e+00" contraction="2.312039367510e-01"/>
            <radfunc exponent="6.534720000000e-01" contraction="8.172257764360e-01"/>
          </basisGroup>
          <basisGroup rid="H10" n="1" l="0" type="Gaussian">
            <radfunc exponent="1.630642000000e-01" contraction="1.000000000000e+00"/>
          </basisGroup>
        </atomicBasisSet>
      </basisset>

    <slaterdeterminant optimize="yes">
      <determinant id="det_up" sposet="spo-up">
        <opt_vars size="3">
          0.0 0.0 0.0
        </opt_vars>
      </determinant>

      <determinant id="det_down" sposet="spo-dn">
        <opt_vars size="3">
          0.0 0.0 0.0
        </opt_vars>
      </determinant>
    </slaterdeterminant>

      <sposet basisset="LCAOBSet" name="spo-up" size="4" optimize="yes">
```

(continues on next page)

(continued from previous page)

```
        <occupation mode="ground"/>
        <coefficient size="4" id="updetC">
  2.83630000000000e-01  3.35683000000000e-01  2.83630000000000e-01  3.35683000000000e-
↪01
  1.66206000000000e-01  1.22367400000000e+00 -1.66206000000000e-01 -1.
↪22367400000000e+00
  8.68279000000000e-01 -6.95081000000000e-01  8.68279000000000e-01 -6.95081000000000e-
↪01
 -9.77898000000000e-01  1.19682400000000e+00  9.77898000000000e-01 -1.
↪19682400000000e+00
</coefficient>
      </sposet>
      <sposet basisset="LCAOBSet" name="spo-dn" size="4" optimize="yes">
        <occupation mode="ground"/>
        <coefficient size="4" id="downdetC">
  2.83630000000000e-01  3.35683000000000e-01  2.83630000000000e-01  3.35683000000000e-
↪01
  1.66206000000000e-01  1.22367400000000e+00 -1.66206000000000e-01 -1.
↪22367400000000e+00
  8.68279000000000e-01 -6.95081000000000e-01  8.68279000000000e-01 -6.95081000000000e-
↪01
 -9.77898000000000e-01  1.19682400000000e+00  9.77898000000000e-01 -1.
↪19682400000000e+00
</coefficient>
      </sposet>

    </determinantset>
</wfn_x>
```

and similarly for the `h2.wfn_d.xml` file, which will hold the initial μ parameters.

This use case is a wavefunction file for an optimizable determinant wavefunction for H_2, in a double zeta valence basis set. Thus, the FDLR wavefunction here would perform CIS on H_2 in a double zeta basis set.

8.8 Gaussian Product Wavefunction

The Gaussian Product wavefunction implements (8.26)

$$\Psi(\vec{R}) = \prod_{i=1}^{N} \exp\left[-\frac{(\vec{R}_i - \vec{R}_i^o)^2}{2\sigma_i^2}\right] \tag{8.26}$$

where $\vec{R}_i$ is the position of the i^{th} quantum particle and $\vec{R}_i^o$ is its center. σ_i is the width of the Gaussian orbital around center i.

This variational wavefunction enhances single-particle density at chosen spatial locations with adjustable strengths. It is useful whenever such localization is physically relevant yet not captured by other parts of the trial wavefunction. For example, in an electron-ion simulation of a solid, the ions are localized around their crystal lattice sites. This single-particle localization is not captured by the ion-ion Jastrow. Therefore, the addition of this localization term will improve the wavefunction. The simplest use case of this wavefunction is perhaps the quantum harmonic oscillator (please see the “tests/models/sho” folder for examples).

Input specification

Gaussian Product Wavefunction (ionwf):

Name	Datatype	Values	Default	Description
Name	Text	ionwf	(Required)	Unique name for this wavefunction
Width	Floats	1.0 -1	(Required)	Widths of Gaussian orbitals
Source	Text	ion0	(Required)	Name of classical particle set

Additional information:

- `width` There must be one width provided for each quantum particle. If a negative width is given, then its corresponding Gaussian orbital is removed. Negative width is useful if one wants to use Gaussian wavefunction for a subset of the quantum particles.
- `source` The Gaussian centers must be specified in the form of a classical particle set. This classical particle set is likely the ion positions “ion0,” hence the name “ionwf.” However, arbitrary centers can be defined using a different particle set. Please refer to the examples in “tests/models/sho.”

8.8.1 Example Use Case

```
<qmcsystem>
  <simulationcell>
    <parameter name="bconds">
          n n n
    </parameter>
  </simulationcell>
  <particleset name="e">
    <group name="u" size="1">
      <parameter name="mass">5.0</parameter>
      <attrib name="position" datatype="posArray" condition="0">
        0.0001 -0.0001 0.0002
      </attrib>
    </group>
  </particleset>
  <particleset name="ion0" size="1">
    <group name="H">
      <attrib name="position" datatype="posArray" condition="0">
        0 0 0
      </attrib>
    </group>
  </particleset>
  <wavefunction target="e" id="psi0">
    <ionwf name="iwf" source="ion0" width="0.8165"/>
  </wavefunction>
  <hamiltonian name="h0" type="generic" target="e">
    <extpot type="HarmonicExt" mass="5.0" energy="0.3"/>
    <estimator type="latticedeviation" name="latdev"
      target="e"    tgroup="u"
      source="ion0" sgroup="H"/>
  </hamiltonian>
</qmcsystem>
```

CHAPTER

NINE

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 projector) generates the imaginary time dynamics of the walkers in DMC and reptation Monte Carlo (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, static structure factor, and forces.

9.1 The Hamiltonian

The many-body Hamiltonian in Hartree units is given by

$$\hat{H} = -\sum_i \frac{1}{2m_i}\nabla_i^2 + \sum_i v^{ext}(r_i) + \sum_{i<j} v^{qq}(r_i, r_j) + \sum_{i\ell} v^{qc}(r_i, r_\ell) + \sum_{\ell<m} v^{cc}(r_\ell, r_m) \,. \tag{9.1}$$

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 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 MPC potential, nonlocal 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 are provided subsequently. 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 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 `particleset`. The default value is typically sufficient. In normal usage, no other attributes are provided.

Listing 9.1: 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 9.2: 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>
```

9.2 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 subsequently. If a potential you desire is not listed, or is not present at all, feel free to contact the developers.

`pairpot` factory element:

parent elements:	`hamiltonian`
child elements:	`type` attribute

type options	coulomb	Coulomb/Ewald potential
	pseudo	Semilocal pseudopotential
	mpc	Model periodic Coulomb interaction/correction
	cpp	Core polarization potential
	skpot	*Unknown*

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.
- **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`, and `mpc` are described in detail in the following subsections. The older or less-used types (`cpp, skpot`) are not covered.
- Available only if `QMC_CUDA` is not defined: `skpot`.
- Available only if `OHMMS_DIM==3`: `mpc, vhxc, pseudo`.
- Available only if `OHMMS_DIM==3` and `QMC_CUDA` is not defined: `cpp`.

9.2.1 Coulomb potentials

The bare Coulomb potential is used in open boundary conditions:

$$V_c^{open} = \sum_{i<j} \frac{q_i q_j}{|r_i - r_j|} \,. \tag{9.2}$$

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

$$V_c^{pbc} = \sum_{i<j} \frac{q_i q_j}{|r_i - r_j|} + \frac{1}{2}\sum_{L\neq 0}\sum_{i,j} \frac{q_i q_j}{|r_i - r_j + L|} \,. \tag{9.3}$$

The sum indexed by L is over all nonzero simulation cell lattice vectors. In practice, the Ewald sum is broken into short- and long-range parts in a manner optimized for efficiency (see [NC95]) for details.

For information on how to set the boundary conditions, consult *Specifying the system to be simulated*.

`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 previous generic `pairpot` factory element.
- **name:** Traditional user-specified names for electron-electron, electron-ion, and ion-ion terms are `ElecElec`, `ElecIon`, and `IonIon`, respectively. Although 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 9.3: QMCPXML element for Coulomb interaction between electrons.

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

Listing 9.4: QMCPXML element for Coulomb interaction between electrons and ions (all-electron only).

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

Listing 9.5: QMCPXML element for Coulomb interaction between ions.

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

9.2.2 Pseudopotentials

QMCPACK supports pseudopotentials in semilocal form, which is local in the radial coordinate and nonlocal 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 pseudopotential separates into a fully local channel and an angularly nonlocal component:

$$V^{PP} = \sum_{ij} \Big(V_{\bar{\ell}}(|r_i - \tilde{r}_j|) + \sum_{\ell \neq \bar{\ell}}^{\ell_{max}} \sum_{m=-\ell}^{\ell} |Y_{\ell m}\rangle \big[V_\ell(|r_i - \tilde{r}_j|) - V_{\bar{\ell}}(|r_i - \tilde{r}_j|) \big] \langle Y_{\ell m}| \Big) . \tag{9.4}$$

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 [MSC91] for further detail.

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 more information. The FSAtom format uses XML for structured data. Files in this format do not use a specific identifying file extension; instead 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
`algorithm`^{o}	text	batched/default	default	Choose NLPP algorithm
`DLA`^{o}	text	yes/no	no	Use determinant localization approximation

Additional information:

- **type/source/target** See description for the generic `pairpot` factory element.

- **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 nonlocal channels are present, a `NonLocalECP` field will be added that contains the nonlocal 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 the following). These elements specify individual file names and formats (both the FSAtom XML and CASINO tabular data formats are supported).
- **algorithm** The default algorithm evaluates the ratios of wavefunction components together for each quadrature point and then one point after another. The batched algorithm evaluates the ratios of quadrature points together for each wavefunction component and then one component after another. Internally, it uses `VirtualParticleSet` for quadrature points. Hybrid orbital representation has an extra optimization enabled when using the batched algorithm.
- **DLA** Determinant localization approximation (DLA) [ZBMAlfe19] uses only the fermionic part of the wavefunction when calculating NLPP.

Listing 9.6: QMCPXML element for pseudopotential electron-ion interaction (psf files).

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

Listing 9.7: QMCPXML 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 shown in the following. 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	`groupe.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			Nonlocal cutoff radius
`lmax`^{o}	integer			Largest angular momentum
`nrule`^{o}	integer			Integration grid order

Listing 9.8: QMCPXML element for pseudopotential of single ionic species.

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

9.2.3 MPC Interaction/correction

The 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 post-processing (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 9.9: MPC for finite-size postcorrection.

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

9.3 General estimators

A broad range of estimators for physical observables are available in QMCPACK. The following sections 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`), static structure factor (`skall`), 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→DMC→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
	SkAll	Static structure factor needed for finite size correction
	structurefactor	Species resolved structure factor
	species kinetic	Species resolved kinetic energy
	latticedeviation	Spatial deviation between two particlesets
	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 *“Force” estimators*)
	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
	Pressure	*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

9.3.1 Chiesa-Ceperley-Martin-Holzmann kinetic energy correction

This estimator calculates a finite-size correction to the kinetic energy following the formalism laid out in [CCMH06]. The total energy can be corrected for finite-size effects by using this estimator in conjunction 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 9.10: “Chiesa” kinetic energy finite-size postcorrection.

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

9.3.2 Density estimator

The particle number density operator is given by

$$\hat{n}_r = \sum_i \delta(r - r_i)\,. \tag{9.5}$$

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:

$$n_c = \int dR\,|\Psi|^2 \int_{\Omega_c} dr \sum_i \delta(r - r_i)\,. \tag{9.6}$$

`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. Postprocessing 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 MC block. If you request many *blocks* in a `<qmc/>` element, or select a large grid, the resulting `stat.h5` file could be many gigabytes 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 subelement 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 9.11: QMCPXML,caption=Density estimator (uniform grid).

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

9.3.3 Spin density estimator

The spin density is similar to the total density described previously. 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)	$v_i >$		Grid cell count
`dr`^{o}	real array(3)	$v_i >$		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. Postprocessing tools expect `name="SpinDensity."`
- `grid`: The 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`: The `dr` sets the 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 9.12: Spin density estimator (uniform grid).

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

Listing 9.13: 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>
```

9.3.4 Pair correlation function, $g(r)$

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

$$g_{ss'}(r) = \frac{V}{4\pi r^2 N_s N_{s'}} \sum_{i_s=1}^{N_s} \sum_{j_{s'}=1}^{N_{s'}} \delta(r - |r_{i_s} - r_{j_{s'}}|) \, , \tag{9.7}$$

where 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

$$\tilde{g}_{ss'}(r) = \frac{V}{4\pi r^2 N_s N_{s'}} \sum_{i=1}^{N_s} \sum_{j=1}^{N_{s'}} \frac{1}{\delta r} \int_{r-\delta r/2}^{r+\delta r/2} dr' \delta(r' - |r_{si} - r_{s'j}|) \, , \tag{9.8}$$

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`: This is the number of bins in each species pair radial histogram.
- `rmax`: This is the 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 output to the `stat.h5` for each QMC subrun. 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 9.14: Pair correlation function estimator element.

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

Listing 9.15: Pair correlation function estimator element with additional electron-ion correlations.

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

9.3.5 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:

$$S(\mathbf{k}) = \frac{1}{N^e}\langle \rho^e_{-\mathbf{k}}\rho^e_{\mathbf{k}}\rangle \ . \tag{9.9}$$

`estimator type=sk` element:

parent elements:	`hamiltonian, qmc`
child elements:	*None*

attributes:

Name	Datatype	Values	Default	Description
`type`^{r}	text	sk		Must 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`: This is the 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 will not accurately measure the electron-electron density response.

Listing 9.16: Static structure factor estimator element.

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

9.3.6 Static structure factor, `SkAll`

In order to compute the finite size correction to the potential energy, records of $\rho(\mathbf{k})$ is required. What sets `SkAll` apart from `sk` is that `SkAll` records $\rho(\mathbf{k})$ in addition to $s(\mathbf{k})$.

`estimator type=SkAll` 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
`source`^{r}	text	Ion ParticleSet name	None	-
`target`^{r}	text	Electron ParticleSet name	None	-
`hdf5`^{o}	boolean	yes/no	no	Output to `stat.h5` (yes) or `scalar.dat` (no)
`writeionion`^{o}	boolean	yes/no	no	Writes file rhok_IonIon.dat containing $s(\mathbf{k})$ for the ions

Additional information:

- `name`: This is the 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 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 `rhok` and postfixed by the k-point index.
- 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 wil not accurately measure the electron-electron density response.

Listing 9.17: SkAll estimator element.

```
<estimator type="skall" name="SkAll" source="ion0" target="e" hdf5="yes"/>
```

9.3.7 Species kinetic energy

Record species-resolved kinetic energy instead of the total kinetic energy in the `Kinetic` column of scalar.dat. `SpeciesKineticEnergy` is arguably the simplest estimator in QMCPACK. The implementation of this estimator is detailed in `manual/estimator/estimator_implementation.pdf`.

`estimator type=specieskinetic` element:

parent elements:	`hamiltonian, qmc`
child elements:	*None*

attributes:

Name	Datatype	Values	Default	Description
`type`^{r}	text	specieskinetic		Must be specieskinetic
`name`^{r}	text	*anything*	any	Unique name for estimator
`hdf5`^{o}	boolean	yes/no	no	Output to `stat.h5` (yes)

Listing 9.18: Species kinetic energy estimator element.

```
<estimator type="specieskinetic" name="skinetic" hdf5="no"/>
```

9.3.8 Lattice deviation estimator

Record deviation of a group of particles in one particle set (target) from a group of particles in another particle set (source).

`estimator type=latticedeviation` element:

parent elements:	`hamiltonian, qmc`
child elements:	*None*

attributes:

Name	Datatype	Values	Default	Description
`type`^{r}	text	latticedeviation		Must be latticedeviation
`name`^{r}	text	*anything*	any	Unique name for estimator
`hdf5`^{o}	boolean	yes/no	no	Output to `stat.h5` (yes)
`per_xyz`^{o}	boolean	yes/no	no	Directionally resolved (yes)
`source`^{r}	text	e/ion0/...	no	source particleset
`sgroup`^{r}	text	u/d/...	no	source particle group
`target`^{r}	text	e/ion0/...	no	target particleset
`tgroup`^{r}	text	u/d/...	no	target particle group

Additional information:

- `source`: The “reference” particleset to measure distances from; actual reference points are determined together with `sgroup`.
- `sgroup`: The “reference” particle group to measure distances from.
- `source`: The “target” particleset to measure distances to.
- `sgroup`: The “target” particle group to measure distances to. For example, in *Listing 32* the distance from the up electron (“u”) to the origin of the coordinate system is recorded.
- `per_xyz`: Used to record direction-resolved distance. In *Listing 32*, the x,y,z coordinates of the up electron will be recorded separately if `per_xyz=yes`.
- `hdf5`: Used to record particle-resolved distances in the h5 file if `gdf5=yes`.

Listing 9.19: Lattice deviation estimator element.

```
<particleset name="e" random="yes">
  <group name="u" size="1" mass="1.0">
     <parameter name="charge"              >    -1                    </parameter>
     <parameter name="mass"                >    1.0                   </parameter>
  </group>
  <group name="d" size="1" mass="1.0">
     <parameter name="charge"              >    -1                    </parameter>
     <parameter name="mass"                >    1.0                   </parameter>
  </group>
</particleset>

<particleset name="wf_center">
  <group name="origin" size="1">
    <attrib name="position" datatype="posArray" condition="0">
             0.00000000        0.00000000        0.00000000
    </attrib>
  </group>
</particleset>

<estimator type="latticedeviation" name="latdev" hdf5="yes" per_xyz="yes"
  source="wf_center" sgroup="origin" target="e" tgroup="u"/>
```

9.3.9 Energy density estimator

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

$$\int dr\hat{\mathcal{E}}_r = \hat{H}, \tag{9.10}$$

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

$$\hat{\mathcal{E}}_r = \hat{\mathcal{T}}_r + \hat{\mathcal{V}}_r\ , \tag{9.11}$$

with each component given by

$$\hat{\mathcal{T}}_r = \frac{1}{2}\sum_i \delta(r - r_i)\hat{p}_i^2$$

$$\hat{\mathcal{V}}_r = \sum_{i<j} \frac{\delta(r - r_i) + \delta(r - r_j)}{2}\hat{v}^{ee}(r_i, r_j) + \sum_{i\ell} \frac{\delta(r - r_i) + \delta(r - \tilde{r}_\ell)}{2}\hat{v}^{eI}(r_i, \tilde{r}_\ell)$$
$$+ \sum_{\ell<m} \frac{\delta(r - \tilde{r}_\ell) + \delta(r - \tilde{r}_m)}{2}\hat{v}^{II}(\tilde{r}_\ell, \tilde{r}_m)\,.$$

Here, r_i and $\tilde{r}_\ell$ represent electron and ion positions, respectively; $\hat{p}_i$ is a single electron momentum operator; and $\hat{v}^{ee}(r_i, r_j)$, $\hat{v}^{eI}(r_i, \tilde{r}_\ell)$, and $\hat{v}^{II}(\tilde{r}_\ell, \tilde{r}_m)$ are the electron-electron, electron-ion, and ion-ion pair potential operators (including nonlocal pseudopotentials, if present). This form of the energy density is size consistent; that is, 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 [KYKC13].

In QMCPACK, the energy density can be accumulated on piecewise uniform 3D 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 9.20: Energy density estimator accumulated on a $20 \times 10 \times 10$ 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 9.21: 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 9.22: 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 9.3.9), 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 is available by default (see Table

9.3.9). 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 following `spacegrid` element) 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 8 Reference points available by default. 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 nonorthogonal, 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.

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`.

`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

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:` The 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 defines the (sheared) space on top of which the curved (e.g., spherical) coordinate system is built.

9.3.10 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:

$$\hat{n}_1 = \sum_n Tr_{R_n} |\Psi_T\rangle\langle\Psi_{FN}| \tag{9.12}$$

In this formula, 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 spin-down electrons, one obtains a density matrix for each spin species. The 1RDM computed by is partitioned in this way.

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

$$n_1(r, r') = \langle r |\hat{n}_1| r'\rangle = \sum_n \int dR_n \Psi_T(r, R_n)\Psi^*_{FN}(r', R_n) . \tag{9.13}$$

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

$$\begin{aligned} n_1(i, j) &= \langle \phi_i |\hat{n}_1| \phi_j\rangle \\ &= \int dR\Psi^*_{FN}(R)\Psi_T(R) \sum_n \int dr'_n \frac{\Psi_T(r'_n, R_n)}{\Psi_T(r_n, R_n)} \phi_i(r'_n)^*\phi_j(r_n) . \end{aligned}$$

The integration over r' in (9.14) is inefficient when one is also interested in obtaining matrices involving energetic quantities, such as the energy density matrix of [KKR14] or the related (and more well known) generalized Fock matrix. For this reason, an approximation is introduced as follows:

$$n_1(i, j) \approx \int dR\Psi_{FN}(R)^*\Psi_T(R) \sum_n \int dr'_n \frac{\Psi_T(r'_n, R_n)^*}{\Psi_T(r_n, R_n)^*} \phi_i(r_n)^*\phi_j(r'_n) . \tag{9.14}$$

For VMC, FN-DMC, FP-DMC, and RN-DMC this formula represents an exact sampling of the 1RDM corresponding to $\hat{\rho}^\dagger_N$ (see appendix A of [KKR14] for more detail).

`estimtor 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	yes	basis comes norm'ed
volume_normed^o	boolean	yes/no	yes	basis norm is volume
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 sposet.name's. The total set of orbitals contained in all sposet's comprises the basis (subspace) onto which the one body density matrix is projected. This set of orbitals generally includes many virtual orbitals that are not occupied in a single reference Slater determinant.
- integrator: Select 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, $\sum_{b\in\texttt{basis}} |\phi_b|^2$, is 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_grid). If center is not provided, the scaled simulation cell is used as is.
- points: Number of grid points in each dimension for integrator=uniform_grid. For example, points=10 results in a uniform $10 \times 10 \times 10$ grid over the cell.
- samples: Sets the number of MC samples collected for each step (active for integrator=uniform/density).
- warmup: Number of warmup Metropolis steps at the start of the run before data collection (active for integrator=density).

- `timestep`: Drift-diffusion time step 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, 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 for each step.
- `rstats`: Print statistical information about the spatial motion of the sampled points to the log for each step.
- `normalized`: Declare whether the inputted orbitals are normalized or not. If `normalized=no`, direct Riemann integration over a $200 \times 200 \times 200$ grid will be used to compute the normalizations before use.
- `volume_normed`: Declare whether the inputted orbitals are normalized to the cell volume (default) or not (a norm of 1.0 is assumed in this case). Currently, B-spline orbitals coming from QE and HEG planewave orbitals native to QMCPACK are known to be volume normalized.
- `energy_matrix`: 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 [KKR14].

Listing 9.23: 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 9.24: 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 9.25: 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 9.26: 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 9.27: 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>
```

9.4 Forward-Walking Estimators

Forward walking is a method for sampling 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, declare a generic forward-walking estimator within a `<hamiltonian>` block, and then specify the observables to forward walk on and the 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

`Observable` element:

parent elements:	`estimator, hamiltonian, qmc`
child elements:	*None*

Name	Datatype	Values	Default	Description
`name`^{r}	text	*anything*	any	Registered name of existing estimator on which to forward walk
`max`^{r}	integer	> 0		Maximum projection time in steps (`max`$=\beta/\tau$)
`frequency`^{r}	text	≥ 1		Dump data only for every `frequency`-th to `scalar.dat` file

Additional information:

- **Cost**: Because histories of observables up to `max` time steps have to be stored, the memory cost of storing the nonforward-walked observables variables should be multiplied by `max`. Although this is not an issue for items such as potential energy, it could be prohibitive for observables such as density, forces, etc.
- **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 9.28: Forward-walking estimator element.

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

9.5 “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:

$$F_z = -Z \sum_{i=1}^{N_e} \frac{z_i}{r_i^3}[\theta(r_i - \mathcal{R}) + \theta(\mathcal{R} - r_i) \sum_{\ell=1}^{M} c_\ell r_i^\ell] \,. \tag{9.15}$$

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,

$$\chi^2 = \int_0^{\mathcal{R}} dr\, r^m \, [f_z(r) - \tilde{f}_z(r)]^2 \,. \tag{9.16}$$

Here, m is the weighting exponent, $f_z(r)$ is the unfiltered radial force density for the z force component, and $\tilde{f}_z(r)$ is the 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
`lrmethod`^{o}	text	ewald or srcoul	ewald	Select long-range potential breakup method
`type`^{r}	text	Force		Must be "Force"
`name`^{o}	text	*Anything*	Force-Base	Unique name for this estimator
`pbc`^{o}	boolean	yes/no	yes	Using periodic BCs or not
`addionion`^{o}	boolean	yes/no	no	Add the ion-ion force contribution to output force estimate

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 :math`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.
- **Long-range breakup**: With periodic boundary conditions, it is important to converge the lattice sum when calculating Coulomb contribution to the forces. As a quick test, increase the `LR_dim_cutoff` parameter until ion-ion forces are converged. The Ewald method (`lrmethod`="ewald") converges more slowly than optimized method (`lrmethod`="srcoul").
- **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.

```
<simulationcell>
  ...
  <parameter name="LR\_dim\_cutoff">  20  </parameter>
</simulationcell>
<estimator name="myforce" type="Force" mode="cep" addionion="yes" lrmethod="srcoul">
    <parameter name="rcut">0.1</parameter>
    <parameter name="nbasis">4</parameter>
    <parameter name="weightexp">2</parameter>
</estimator>
```

CHAPTER

TEN

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	???
`checkpoint`	integer	-1, 0, n	-1	Checkpoint frequency
`record`	integer	n	0	Save configuration ever n steps
`target`	text			???
`completed`	text			???
`append`	text	yes/no	no	???

Additional information:

- `move`: There are two ways to move electrons. The more used method is the particle-by-particle move. In this method, only one electron is moved for acceptance or rejection. The other method is the all-electron move; namely, all the electrons are moved once for testing acceptance 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.
- `checkpoint`: This enables and disables checkpointing and specifying the frequency of output. Possible values are:
  - **[-1]** No checkpoint (default setting).
  - **[0]** Dump after the completion of a QMC section.
  - **[n]** Dump after every n blocks. Also dump at the end of the run.

The particle configurations are written to a `.config.h5` file.

Listing 10.1: The following is an example of running a simulation that can be restarted.

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

The checkpoint flag instructs QMCPACK to output walker configurations. This also works in VMC. This outputs an h5 file with the name `projectid.run-number.config.h5`. Check that this file exists before attempting a restart.

To continue a run, specify the `mcwalkerset` element before your VMC/DMC block:

Listing 10.2: Restart (read walkers from previous run).

```
<mcwalkerset fileroot="BH.s002" version="0 6" collected="yes"/>
 <qmc method="dmc" move="pbyp"  checkpoint="0">
   <parameter name="timestep">         0.004  </parameter>
   <parameter name="blocks">           100   </parameter>
   <parameter name="steps">            400    </parameter>
 </qmc>
```

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

In the project id section, make sure that the series number is different from any existing ones to avoid overwriting them.

10.1 Variational Monte Carlo

`vmc` method:

parameters:

Name	Datatype	Values	Default	Description
`walkers`	integer	> 0	dep.	Number of walkers per MPI task
`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	yes	Use the algorithm with drift
`timestep`	real	> 0	0.1	Time step for each electron move
`samples`	integer	≥ 0	0	Number of walker samples for DMC/optimization
`stepsbetweensamples`	integer	> 0	1	Period of sample accumulation
`samplesperthread`	integer	≥ 0	0	Number of samples per thread
`storeconfigs`	integer	all values	0	Show configurations o
`blocks_between_recompute`	integer	≥ 0	dep.	Wavefunction recompute frequency

Additional information:

- `walkers` The number of walkers per MPI task. The initial default number of ixml{walkers} is one per OpenMP thread or per MPI task if threading is disabled. The number is rounded down to a multiple of the number of threads with a minimum of one per thread to ensure perfect load balancing. One walker per thread is created in the event fewer `walkers` than threads are requested.
- `blocks` This parameter is universal for all the QMC methods. The MC processes are divided into a number of `blocks`, each containing a number of steps. At the end of each block, the statistics accumulated in the block are dumped into files, e.g., `scalar.dat`. Typically, each block should have a sufficient number of steps that the I/O at the end of each block is negligible compared with the computational cost. Each block should not take so long that monitoring its progress is difficult. There should be a sufficient number of `blocks` to perform statistical analysis.
- `warmupsteps` - `warmupsteps` are used only for equilibration. Property measurements are not performed during warm-up steps.
- `steps` - `steps` are the number of energy and other property measurements to perform per block.
- `substeps` For each substep, an attempt is made to move each of the electrons once only by either particle-by-particle or an all-electron move. Because the local energy is evaluated only at each full step and not each substep, `substeps` are computationally cheaper and can be used to reduce the correlation between property measurements at a lower 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 time step input. In the drift algorithm, electrons are moved by Langevin dynamics.
- `timestep` The meaning of time step depends on whether or not the drift is used. In general, larger time steps reduce the time correlation but might also reduce the acceptance ratio, reducing overall statistical efficiency. For VMC, typically the acceptance ratio should be close to 50% for an efficient simulation.
- `samples` Seperate from conventional energy and other property measurements, samples refers to storing whole electron configurations in memory ("walker samples") as would be needed by subsequent wavefunction optimization or DMC steps. *A standard VMC run to measure the energy does not need samples to be set.*

$$\texttt{samples} = \frac{\texttt{blocks} \cdot \texttt{steps} \cdot \texttt{walkers}}{\texttt{stepsbetweensamples}} \cdot \texttt{number of MPI tasks}$$

- `samplesperthread` This is an alternative way to set the target amount of samples and can be useful when preparing a stored population for a subsequent DMC calculation.

$$\texttt{samplesperthread} = \frac{\texttt{blocks} \cdot \texttt{steps}}{\texttt{stepsbetweensamples}}$$

- `stepsbetweensamples` Because samples generated by consecutive steps are correlated, having `stepsbetweensamples` larger than 1 can be used to reduces that correlation. In practice, using larger substeps is cheaper than using `stepsbetweensamples` to decorrelate samples.
- `storeconfigs` If `storeconfigs` is set to a nonzero value, then electron configurations during the VMC run are saved to files.
- `blocks_between_recompute` Recompute the accuracy critical determinant part of the wavefunction from scratch: =1 by default when using mixed precision. =0 (no recompute) by default when not using mixed precision. Recomputing introduces a performance penalty dependent on system size.

An example VMC section for a simple VMC run:

```
<qmc method="vmc" move="pbyp">
  <estimator name="LocalEnergy" hdf5="no"/>
  <parameter name="walkers">    256 </parameter>
```

(continues on next page)

(continued from previous page)

```
  <parameter name="warmupSteps">  100 </parameter>
  <parameter name="substeps">  5 </parameter>
  <parameter name="blocks">  20 </parameter>
  <parameter name="steps">  100 </parameter>
  <parameter name="timestep">  1.0 </parameter>
  <parameter name="usedrift">   yes </parameter>
</qmc>
```

Here we set 256 `walkers` per MPI, have a brief initial equilibration of 100 `steps`, and then have 20 `blocks` of 100 `steps` with 5 `substeps` each.

The following is an example of VMC section storing configurations (walker samples) for optimization.

```
<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>
```

10.2 Wavefunction optimization

Optimizing wavefunction is critical in all kinds of real-space QMC calculations because it significantly improves both the accuracy and efficiency of computation. However, it is very difficult to directly adopt deterministic minimization approaches because of the stochastic nature of evaluating quantities with MC. Thanks to the algorithmic breakthrough during the first decade of this century and the tremendous computer power available, it is now feasible to optimize tens of thousands of parameters in a wavefunction for a solid or molecule. QMCPACK has multiple optimizers implemented based on the state-of-the-art linear method. We are continually improving our optimizers for robustness and friendliness and are trying to provide a single solution. Because of the large variation of wavefunction types carrying distinct characteristics, using several optimizers might be needed in some cases. We strongly suggested reading recommendations from the experts who maintain these optimizers.

A typical optimization block looks like the following. It starts with method="linear" and contains three blocks of parameters.

```
<loop max="10">
 <qmc method="linear" move="pbyp" gpu="yes">
   <!-- Specify the VMC options -->
   <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>
   <estimator name="LocalEnergy" hdf5="no"/>
   ...
   <!-- Specify the correlated sampling options and define the cost function -->
```

(continues on next page)

(continued from previous page)

```
   <parameter name="minwalkers">            0.3 </parameter>
        <cost name="energy">                0.95 </cost>
        <cost name="unreweightedvariance"> 0.00 </cost>
        <cost name="reweightedvariance">   0.05 </cost>
   ...
   <!-- Specify the optimizer options -->
   <parameter name="MinMethod">    OneShiftOnly </parameter>
   ...
 </qmc>
</loop>

-  Loop is helpful to repeatedly execute identical optimization blocks.

-  The first part is highly identical to a regular VMC block.

-  The second part is to specify the correlated sampling options and
   define the cost function.

-  The last part is used to specify the options of different optimizers,
   which can be very distinct from one to another.
```

10.2.1 VMC run for the optimization

The VMC calculation for the wavefunction optimization has a strict requirement that `samples` or `samplesperthread` must be specified because of the optimizer needs for the stored `samples`. The input parameters of this part are identical to the VMC method.

Recommendations:

- Run the inclusive VMC calculation correctly and efficiently because this takes a significant amount of time during optimization. For example, make sure the derived `steps` per block is 1 and use larger `substeps` to control the correlation between `samples`.
- A reasonable starting wavefunction is necessary. A lot of optimization fails because of a bad wavefunction starting point. The sign of a bad initial wavefunction includes but is not limited to a very long equilibration time, low acceptance ratio, and huge variance. The first thing to do after a failed optimization is to check the information provided by the VMC calculation via `*.scalar.dat files`.

10.2.2 Correlated sampling and cost function

After generating the samples with VMC, the derivatives of the wavefunction with respect to the parameters are computed for proposing a new set of parameters by optimizers. And later, a correlated sampling calculation is performed to quickly evaluate values of the cost function on the old set of parameters and the new set for further decisions. The input parameters are listed in the following table.

`linear` method:

parameters:

Name	**Datatype**	**Values**	**Default**	**Description**
`nonlocalpp`	text	yes, no	no	include non-local PP energy in the cost function
`minwalkers`	real	0–1	0.3	Lower bound of the effective weight
`maxWeight`	real	> 1	1e6	Maximum weight allowed in reweighting

Additional information:

- maxWeight The default should be good.
- nonlocalpp The nonlocalpp contribution to the local energy depends on the wavefunction. When a new set of parameters is proposed, this contribution needs to be updated if the cost function consists of local energy. Fortunately, nonlocal contribution is chosen small when making a PP for small locality error. We can ignore its change and avoid the expensive computational cost. An implementation issue with GPU code is that a large amount of memory is consumed with this option.
- minwalkers This is a critical parameter. When the ratio of effective samples to actual number of samples in a reweighting step goes lower than minwalkers, the proposed set of parameters is invalid.

The cost function consists of three components: energy, unreweighted variance, and reweighted variance.

```
<cost name="energy">                   0.95 </cost>
<cost name="unreweightedvariance">     0.00 </cost>
<cost name="reweightedvariance">       0.05 </cost>
```

10.2.3 Optimizers

QMCPACK implements a number of different optimizers each with different priorities for accuracy, convergence, memory usage, and stability. The optimizers can be switched among “OneShiftOnly” (default), “adaptive,” “descent,” “hybrid,” and “quartic” (old) using the following line in the optimization block:

```
<parameter name="MinMethod"> THE METHOD YOU LIKE </parameter>
```

10.2.4 OneShiftOnly Optimizer

The OneShiftOnly optimizer targets a fast optimization by moving parameters more aggressively. It works with OpenMP and GPU and can be considered for large systems. This method relies on the effective weight of correlated sampling rather than the cost function value to justify a new set of parameters. If the effective weight is larger than minwalkers, the new set is taken whether or not the cost function value decreases. If a proposed set is rejected, the standard output prints the measured ratio of effective samples to the total number of samples and adjustment on minwalkers can be made if needed.

linear method:

parameters:

Name	Datatype	Values	Default	Description
shift_i	real	> 0	0.01	Direct stabilizer added to the Hamiltonian matrix
shift_s	real	> 0	1.00	Initial stabilizer based on the overlap matrix

Additional information:

- shift_i This is the direct term added to the diagonal of the Hamiltonian matrix. It provides more stable but slower optimization with a large value.
- shift_s This is the initial value of the stabilizer based on the overlap matrix added to the Hamiltonian matrix. It provides more stable but slower optimization with a large value. The used value is auto-adjusted by the optimizer.

Recommendations:

- Default shift_i, shift_s should be fine.

- For hard cases, increasing shift_i (by a factor of 5 or 10) can significantly stabilize the optimization by reducing the pace towards the optimal parameter set.
- If the VMC energy of the last optimization iterations grows significantly, increase minwalkers closer to 1 and make the optimization stable.
- If the first iterations of optimization are rejected on a reasonable initial wavefunction, lower the minwalkers value based on the measured value printed in the standard output to accept the move.

We recommended using this optimizer in two sections with a very small minwalkers in the first and a large value in the second, such as the following. In the very beginning, parameters are far away from optimal values and large changes are proposed by the optimizer. Having a small minwalkers makes it much easier to accept these changes. When the energy gradually converges, we can have a large minwalkers to avoid risky parameter sets.

```
<loop max="6">
 <qmc method="linear" move="pbyp" gpu="yes">
   <!-- Specify the VMC options -->
   <parameter name="walkers">                1 </parameter>
   <parameter name="samples">            10000 </parameter>
   <parameter name="stepsbetweensamples">    1 </parameter>
   <parameter name="substeps">               5 </parameter>
   <parameter name="warmupSteps">            5 </parameter>
   <parameter name="blocks">                25 </parameter>
   <parameter name="timestep">             1.0 </parameter>
   <parameter name="usedrift">              no </parameter>
   <estimator name="LocalEnergy" hdf5="no"/>
   <!-- Specify the optimizer options -->
   <parameter name="MinMethod">    OneShiftOnly </parameter>
   <parameter name="minwalkers">           1e-4 </parameter>
 </qmc>
</loop>
<loop max="12">
 <qmc method="linear" move="pbyp" gpu="yes">
   <!-- Specify the VMC options -->
   <parameter name="walkers">                1 </parameter>
   <parameter name="samples">            20000 </parameter>
   <parameter name="stepsbetweensamples">    1 </parameter>
   <parameter name="substeps">               5 </parameter>
   <parameter name="warmupSteps">            2 </parameter>
   <parameter name="blocks">                50 </parameter>
   <parameter name="timestep">             1.0 </parameter>
   <parameter name="usedrift">              no </parameter>
   <estimator name="LocalEnergy" hdf5="no"/>
   <!-- Specify the optimizer options -->
   <parameter name="MinMethod">    OneShiftOnly </parameter>
   <parameter name="minwalkers">            0.5 </parameter>
 </qmc>
</loop>
```

For each optimization step, you will see

```
The new set of parameters is valid. Updating the trial wave function!
```

or

```
The new set of parameters is not valid. Revert to the old set!
```

Occasional rejection is fine. Frequent rejection indicates potential problems, and users should inspect the VMC calculation or change optimization strategy. To track the progress of optimization, use the command qmca -q ev

`*.scalar.dat` to look at the VMC energy and variance for each optimization step.

10.2.5 Adaptive Organizer

The default setting of the adaptive optimizer is to construct the linear method Hamiltonian and overlap matrices explicitly and add different shifts to the Hamiltonian matrix as "stabilizers." The generalized eigenvalue problem is solved for each shift to obtain updates to the wavefunction parameters. Then a correlated sampling is performed for each shift's updated wavefunction and the initial trial wavefunction using the middle shift's updated wavefunction as the guiding function. The cost function for these wavefunctions is compared, and the update corresponding to the best cost function is selected. In the next iteration, the median magnitude of the stabilizers is set to the magnitude that generated the best update in the current iteration, thus adapting the magnitude of the stabilizers automatically.

When the trial wavefunction contains more than 10,000 parameters, constructing and storing the linear method matrices could become a memory bottleneck. To avoid explicit construction of these matrices, the adaptive optimizer implements the block linear method (BLM) approach. [Zhao:2017:blocked_lm] The BLM tries to find an approximate solution $\vec{c}_{opt}$ to the standard LM generalized eigenvalue problem by dividing the variable space into a number of blocks and making intelligent estimates for which directions within those blocks will be most important for constructing $\vec{c}_{opt}$, which is then obtained by solving a smaller, more memory-efficient eigenproblem in the basis of these supposedly important block-wise directions.

```
QMCPACK website: http://www.qmcpack.org

Releases & source code: https://github.com/QMCPACK

Google Group: https://groups.google.com/forum/#!forum/qmcpack
```

BIBLIOGRAPHY

[MLC+17] Amrita Mathuriya, Ye Luo, Raymond C. Clay, III, Anouar Benali, Luke Shulenburger, and Jeongnim Kim. Embracing a new era of highly efficient and productive quantum monte carlo simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, 38:1–38:12. New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3126908.3126952, doi:10.1145/3126908.3126952.

[EKCS12] Kenneth P. Esler, Jeongnim Kim, David M. Ceperley, and Luke Shulenburger. Accelerating quantum monte carlo simulations of real materials on gpu clusters. *Computing in Science and Engineering*, 14(1):40–51, 2012. doi:http://doi.ieeecomputersociety.org/10.1109/MCSE.2010.122.

[NC95] Vincent Natoli and David M. Ceperley. An optimized method for treating long-range potentials. *Journal of Computational Physics*, 117(1):171 – 178, 1995. URL: http://www.sciencedirect.com/science/article/pii/S0021999185710546, doi:http://dx.doi.org/10.1006/jcph.1995.1054.

[AlfeG04] D. Alfè and M. J. Gillan. An efficient localized basis set for quantum Monte Carlo calculations on condensed matter. *Physical Review B*, 70(16):161101, 2004.

[BN17] N. S. Blunt and Eric Neuscamman. Charge-transfer excited states: Seeking a balanced and efficient wave function ansatz in variational Monte Carlo. *The Journal of Chemical Physics*, 147(19):194101, November 2017. doi:10.1063/1.4998197.

[Cep78] D. Ceperley. Ground state of the fermion one-component plasma: a monte carlo study in two and three dimensions. *Phys. Rev. B*, 18:3126–3138, October 1978. URL: https://link.aps.org/doi/10.1103/PhysRevB.18.3126, doi:10.1103/PhysRevB.18.3126.

[DTN04] N. D. Drummond, M. D. Towler, and R. J. Needs. Jastrow correlation factor for atoms, molecules, and solids. *Physical Review B - Condensed Matter and Materials Physics*, 70(23):1–11, 2004. doi:10.1103/PhysRevB.70.235119.

[EG13] M. Caffarel E. Giner, A. Scemama. Using perturbatively selected configuration interaction in quantum monte carlo calculations. *Canadian Journal of Chemistry*, 91:9, 2013.

[EKCS12] Kenneth P. Esler, Jeongnim Kim, David M. Ceperley, and Luke Shulenburger. Accelerating quantum monte carlo simulations of real materials on gpu clusters. *Computing in Science and Engineering*, 14(1):40–51, 2012. doi:http://doi.ieeecomputersociety.org/10.1109/MCSE.2010.122.

[FWL90] S. Fahy, X. W. Wang, and Steven G. Louie. Variational quantum Monte Carlo nonlocal pseudopotential approach to solids: Formulation and application to diamond, graphite, and silicon. *Physical Review B*, 42(6):3503–3522, 1990. doi:10.1103/PhysRevB.42.3503.

[Gas61] T Gaskell. The collective treatment of a fermi gas: ii. *Proceedings of the Physical Society*, 77(6):1182, 1961. URL: http://stacks.iop.org/0370-1328/77/i=6/a=312.

[Gas62] T Gaskell. The collective treatment of many-body systems: iii. *Proceedings of the Physical Society*, 80(5):1091, 1962. URL: http://stacks.iop.org/0370-1328/80/i=5/a=307.

[Kat51] T Kato. Fundamental properties of hamiltonian operators of the schrodinger type. *Transactions of the American Mathematical Society*, 70:195–211, 1951.

[LEKS18] Ye Luo, Kenneth P. Esler, Paul R. C. Kent, and Luke Shulenburger. An efficient hybrid orbital representation for quantum monte carlo calculations. *The Journal of Chemical Physics*, 149(8):084107, 2018.

[LK18] Ye Luo and Jeongnim Kim. An highly efficient delayed update algorithm for evaluating slater determinants in quantum monte carlo. *in preparation*, ():, 2018.

[MDAzevedoL+17] T. McDaniel, E. F. D'Azevedo, Y. W. Li, K. Wong, and P. R. C. Kent. Delayed slater determinant update algorithms for high efficiency quantum monte carlo. *The Journal of Chemical Physics*, 147(17):174107, November 2017. doi:10.1063/1.4998616.

[NC95] Vincent Natoli and David M. Ceperley. An optimized method for treating long-range potentials. *Journal of Computational Physics*, 117(1):171 – 178, 1995. URL: http://www.sciencedirect.com/science/article/pii/S0021999185710546, doi:http://dx.doi.org/10.1006/jcph.1995.1054.

[Sce17] A. Scemamma. Quantum package. https://github.com/LCPQ/quantum_package, 2013–2017.

[SBB+93] Michael W. Schmidt, Kim K. Baldridge, Jerry A. Boatz, Steven T. Elbert, Mark S. Gordon, Jan H. Jensen, Shiro Koseki, Nikita Matsunaga, Kiet A. Nguyen, Shujun Su, Theresa L. Windus, Michel Dupuis, and John A. Montgomery. General atomic and molecular electronic structure system. *Journal of Computational Chemistry*, 14(11):1347–1363, 1993. URL: http://dx.doi.org/10.1002/jcc.540141112, doi:10.1002/jcc.540141112.

[CCMH06] Simone Chiesa, David M. Ceperley, Richard M. Martin, and Markus Holzmann. Finite-size error in many-body simulations with long-range interactions. *Phys. Rev. Lett.*, 97:076404, August 2006. doi:10.1103/PhysRevLett.97.076404.

[KKR14] Jaron T. Krogel, Jeongnim Kim, and Fernando A. Reboredo. Energy density matrix formalism for interacting quantum systems: quantum monte carlo study. *Phys. Rev. B*, 90:035125, July 2014. doi:10.1103/PhysRevB.90.035125.

[KYKC13] Jaron T. Krogel, Min Yu, Jeongnim Kim, and David M. Ceperley. Quantum energy density: improved efficiency for quantum monte carlo calculations. *Phys. Rev. B*, 88:035137, July 2013. doi:10.1103/PhysRevB.88.035137.

[MSC91] Lubos Mitas, Eric L. Shirley, and David M. Ceperley. Nonlocal pseudopotentials and diffusion monte carlo. *The Journal of Chemical Physics*, 95(5):3467–3475, 1991. doi:10.1063/1.460849.

[NC95] Vincent Natoli and David M. Ceperley. An optimized method for treating long-range potentials. *Journal of Computational Physics*, 117(1):171–178, 1995. URL: http://www.sciencedirect.com/science/article/pii/S0021999185710546, doi:10.1006/jcph.1995.1054.

[ZBMAlfe19] Andrea Zen, Jan Gerit Brandenburg, Angelos Michaelides, and Dario Alfè. A new scheme for fixed node diffusion quantum monte carlo with pseudopotentials: improving reproducibility and reducing the trial-wave-function bias. *The Journal of Chemical Physics*, 151(13):134105, October 2019. doi:10.1063/1.5119729.