HANDE Documentation

Release git

HANDE developers

April 10, 2016

Contents

CHAPTER 1

User's guide

1.1 Introduction

HANDE contains optimised, highly parallel implementations of the full configuration interaction quantum Monte Carlo (FCIQMC) *[Booth09]*, coupled cluster Monte Carlo (CCMC) *[Thom10]* and Density Matrix Quantum Monte Carlo (DMQMC) *[Blunt14]*, *[Malone15]* algorithms for a variety of systems. Development work continues to add new features and investigate the algorithms and new applications.

HANDE can perform calculations on generic systems such as molecules via an externally generated integral file. The integral file is in the FCIDUMP format *[Knowles89]*, which can be generated by several quantum chemistry codes such as MOLPRO, Q-Chem (with patches from Alex Thom) and PSI4 (with a plugin from James Spencer). HANDE can also perform calculations on model Hamiltonians, for which no additional integrals are required. The model Hamiltonians currently available are the Hubbard model, Heisenberg model and uniform electron gas.

Configuration interaction (CI) is also implemented using external libraries (lapack/scalapack and TRLan respectively) and can be performed in both serial and parallel. Lanczos diagonalisation can also be performed with or without precomputing the Hamiltonian matrix. Note that this is rather slow and intended for debugging purposes only. Most quantum chemistry codes (e.g. PSI4) contain a substantially more powerful and optimised CI implementation.

1.2 Some geography...

Files are organised in the HANDE repository as follows:

`./` Root directory of the program.

`bin/` Directory containing the compiled program, hande.x. Created during compilation.

`config/` Directory containing the configuration input files used to generate makefiles.

`dest/` Directory containing the compiled object files and dependency files. Created during compilation.

`documentation/` Directory containing documentation on the HANDE program. The documentation is written in reStructured Text and can be converted into a wide range of output formats.

`src/` Directory containing the main source files.

`lib/` Directory containing "library" source files. These are procedures which are not specific to the HANDE code but are generally useful. Some are written by the authors, some are freely available (as noted in the source files).

`tools/` Directory containing scripts and tools for compiling, running and analysing output from HANDE.

`test_suite/` Directory containing a set of tests which HANDE should agree with.

1.3 Prerequisites

HANDE builds upon several well-written, efficient libraries to aid portability, efficiency and sustainability.

1.3.1 Dependencies

Fortran and C compilers HANDE is written in (mostly) Fortran 2003 with some C code. We have tested HANDE using GCC, Intel, Cray and IBM compilers and are interested in hearing of use with other compilers.

> **Note:** HANDE is relatively aggressive in adopting new language features and hence requires a fairly modern Fortran compiler. In particular, gfortran 4.5 or earlier is unlikely to successfully compile HANDE.

LAPACK and BLAS Available from http://www.netlib.org/lapack/ and http://www.netlib.org/blas/ and vendor implementations. Typically installed on HPC systems and available from package manager. This is only required for the FCI functionality in HANDE; the performance of the QMC algorithms do not depend upon the quality of the LAPACK and BLAS libraries used.

lua 5.2 Lua (available from http://www.lua.org) is required. HANDE links to the lua library, which is used for parsing the input file. No performance critical code is written in lua.

> **Note:** The version of the AOTUS library included with HANDE is only compatible with lua 5.2. Later versions of AOTUS, which HANDE should also work with, support lua 5.3 (but not 5.2 due to API changes).

MPI (parallel compilation only) MPI 2 is required. We have used a variety of implementations (including OpenMPI and various vendor implementations).

scalapack (parallel compilation only) Available from http://www.netlib.org/scalapack/ and vendor implementations. Often already installed on HPC systems, included in Intel Maths Kernel Library and can be installed from most package managers.

python 2.7+ or python 3.2+ Almost all tools packaged with HANDE are written in python.

> **Note:** python 2.6 or earlier python 3 versions **may** be sufficient but will probably require additional work. In particular, the argparse module (included from 2.7 and 3.2 onwards) is required and installing (especially recent versions of) pandas may be problematic. Using a recent version of python is highly recommended.

pandas 0.14.1+ The HANDE data analysis tools build heavily upon the python scientific stack. In particular, pandas (available from http://pandas.pydata.org) is required for the `pyhande` module and analysis scripts, almost all of which build upon `pyhande`. pandas is not required for running HANDE but is highly recommended for data analysis (though strictly speaking is only required if `pyhande` is used, either directly or via analysis scripts).

1.3.2 Bundled dependencies

AOTUS AOTUS provides a nice Fortran wrapper to Lua's C-API. For convenience (given that module files are Fortran-specific), AOTUS is included in the HANDE source distribution.

1.3.3 Optional dependencies

The following are optional depedencies which add useful (in some cases almost critical) functionality. However, they are less likely to be compiled on HPC systems so for ease of testing the functionality which depends upon them can be disabled at compile-time.

HDF5 HDF5 is a library for storing scientific data and is used in HANDE for checkpointing (i.e. writing and reading restart files) in QMC calculations.

Highly recommended. Disabling HDF5 removes the ability to perform any checkpointing.

Note: HANDE requires the Fortran 2003 interface to HDF5, which is not compiled by default (see below), as this offers substantial advantages when working with dynamically sized arrays containing variables of arbitrary kinds/precision.

---

libuuid Provenance of a calculation, and the output file(s) produced by it, is an important topic, currently the subject of much debate in computational science. HANDE generates a universally unique identifier (UUID), which is included in all files it produces.

Highly recommended but can be disabled without impacting on performance (but perhaps not on the user's sanity).

Note: Some Linux distributions install libuuid but require an additional package (e.g. uuid-dev) to be installed in order for libuuid to exist on default search paths. Some luck may be found by looking under /lib or /lib64 instead of /usr/lib and /usr/lib64.

TRLan Required for (and only for) performing FCI calculations using the Lanczos algorithm. Available from http://crd-legacy.lbl.gov/~kewu/trlan.html.

1.3.4 Compilation and installation notes

Some notes on compiling the less common dependencies.

lua

Lua is straightforward to compile. For example:

```
$ wget -O - http://www.lua.org/ftp/lua-5.2.4.tar.gz | tar xvzf -
$ cd lua-5.2.4
$ make linux
$ make install INSTALL_TOP=$HOME/local
```

will install the lua program and library to subdirectories in $HOME/local. It is usually fine to compile lua using the GCC compiler and link HANDE against it using another compiler family (e.g. Intel).

HDF5

HDF5 uses the GNU autotools build system, so is also straightforward to compile. For example:

```
$ wget -O - http://www.hdfgroup.org/ftp/HDF5/current/src/hdf5-1.8.14.tar.gz | tar xvzf -
$ cd hdf5-1.8.14
$ ./configure --prefix=$HOME/local --enable-fortran --enable-fortran2003 --enable-cxx
```

```
$ make
$ make install
```

will compile HDF5 and install it to subdirectories in $HOME/local. By default this will use the GCC compiler suite; other compilers can be used by setting the CC, CXX and F77 environment variables. Note the use of `--enable-fortran2003`; the Fortran 2003 interface is required by HANDE.

pandas

Pandas can be installed by

```
$ pip install pandas
```

If you do not have root access, you can install the library locally with:

```
$ pip install pandas --user
```

Alternatively, where pip is not available, one can install it locally:

```
$ wget https://github.com/pydata/pandas/archive/v0.15.2.tar.gz
$ tar -xzvf v0.15.2.tar.gz
$ cd pandas-0.15.2
$ python setup.py build
$ python setup.py install
```

Again, pandas can be installed locally by replacing the final command with:

```
$ python setup.py install --user
```

1.4 Compilation

After ensuring HANDE's dependencies are installed, produce a makefile by running the `mkconfig.py` (residing in the tools subdirectory) script in the root directory:

```
$ tools/mkconfig.py config/conf
```

where `conf` is one of the platforms available and is simply the name of the relevant file residing in the `config/` directory. Various configurations are provided and it is simple to adapt one to the local environment (e.g. changing compiler or library paths).

Run

```
$ tools/mkconfig.py --help
```

to see the options available, including inspecting available configurations.

A configuration is defined using a simple ini file, consisting of three sections: main, opt and dbg. For instance:

```
[main]
fc = gfortran
ld = gfortran
libs = -llapack -lblas

[opt]
fflags = -O3
```

```
[dbg]
fflags = -g
```

Any option not specified in the 'opt' and 'dbg' sections is inherited from the 'main' section. The settings in 'opt' are used by default; the debug options can be selected by passing the -g option to mkconfig.

All options are strings unless otherwise specified. Available options are:

fc Set the fortran compiler.

fflags Set flags to be passed to the fortran compiler during compilation.

f90_module_flag Set the flag used by the compiler which is used to specify the directory where module (.mod) files are placed when created and where they should be searched for.

f90_module_flag_pad [boolean] True if a space needs to be inserted between the defined f90_module_flag and the corresponding directory argument. Default: true.

cc Set the C compiler.

cflags Set flags to be passed to the C compiler during compilation.

ccd Set the C compiler used to generate the C dependency files. Only required if cc doesn't support -MM and -MT flags. Default: use cc.

cdflags Set the flags for the c++ compiler used to generate the C++ dependency files. Default: $CFLAGS -MM -MT

cxx Set the C++ compiler.

cxxflags Set flags to be passed to the C++ compiler during compilation.

cxxd Set the C compiler used to generate the C++ dependency files. Only required if cc doesn't support -MM and -MT flags. Default: use cxx.

cxxdflags Set the flags for the c++ compiler used to generate the C++ dependency files. Default: $CXXFLAGS -MM -MT

cpp Set the C preprocessor to be used on Fortran source files. If not defined then the Fortran compiler is used to do the preprocessing.

cppflags Set flags to be used in the C preprocessing step. C preprocessing is applied to .F90, .F, .c and .cpp files (and not .f90 files).

ld Set the linker program.

ldflags Set flags to be passed to the linker during linking of the compiled objects.

libs Set libraries to be used during the linking step.

ar Set the archive program. Default: ar.

arflags Set the flags to be passed to the archive program. Default: -rcs.

To compile the code run

```
$ make
```

HANDE's build system uses the `sfmakedepend` script (http://people.arsc.edu/~kate/Perl/, supplied in `tools/`) by Kate Hedstrom to generate the list of dependencies for each Fortran source file. These are generated automatically when make is run if the dependency files do not exist.

The executable, `hande.x`, is placed in the `bin` subdirectory. Note that this is actually a symbolic link: a unique executable is produced for each platform and optimisation level and `hande.x` merely points to the most recently compiled executable for convenience. This makes testing against multiple platforms particularly easy.

There are various goals in the makefile. Run

```
$ make help
```

to see the available goals.

1.4.1 Compile-time settings

The behaviour of the program can be changed in various ways by some choices at compile-time by using C pre-processing. These choices largely influence the speed, memory usage, inclusion of parallel code and workarounds for certain compilers.

The pre-processing options which accept a value are set by:

```
-DOPTION=VAL
```

which defines the pre-processing definition OPTION to have value VAL. Similarly, the options which just need to be defined to be used are set by:

```
-DOPTION
```

These should be added to the cppflags or cppdefs lines in the configuration files or in the Makefile, as desired.

Warning: Certain options, for technical reasons, change the Markov chain of QMC calculations. Results should be in statistical agreement but the precise data produced by the calculation (even using the same random number seed) may well be changed.
This currently applies to the following options: POP_SIZE and SINGLE_PRECISION.

DET_SIZE Default: 32.

HANDE uses bit strings to store Slater determinants, where each bit corresponds to an occupied spin-orbital if the bit is set and an unoccupied spin orbital otherwise. As Fortran does not include a type for a single bit, integers are used. Note that this does lead to some wasted memory when the number of spin-orbitals is not a multiple of the size of the integer used. An array of integers is used to store the determinant bit string if a single integer is not sufficient.

This option sets the integer length to be used. Allowed values are 32 and 64, corresponding to using 32-bit and 64-bit integers respectively. As bit operations on a 64-bit integer are faster than those on two 32-bit integers, using DET_SIZE=64 is recommended for production calculations. (Note, however, that this will use more memory than DET_SIZE=32 if the number of basis functions is closer to a multiple of 32 rather than 64. This is rarely a concern in practice.)

POP_SIZE Default: 32

This option is used to specify whether 32 or 64-bit integers are used to store walker populations in HANDE. It is unlikely that 64-bit integers will be needed when using the integer code but this option is more critical when the **real_amplitudes** option is being used. When using the **real_amplitudes** option with POP_SIZE=32, the largest walker amplitude that can be stored is $2^{20} = 1048576$, while the smallest fractional part that can be represented is $2^{-11} = 0.00049$. When using this option and POP_SIZE=64 the largest amplitude is $2^{32} = 4.3 \times 10^9$ and the smallest fractional part is $2^{-31} = 4.66 \times 10^{-10}$.

DEBUG Default: not defined.

If defined then add additional information in output (e.g. stack traces) that might be useful for debugging. Recommended for developers only. The format and content of the additional debug output should not be relied upon.

DISABLE_LANCZOS Default: not defined.

If defined then Lanczos diagonalisation is disabled. This removes the dependency on the TRLan library.

DISABLE_HDF5 Default: not defined.

If defined then the QMC restart functionality is disabled and the dependency on HDF5 (which can be tricky to compile on some machines) is removed. Note that restart functionality is extremely useful in production simulations so this option should only be used during initial porting efforts.

DISABLE_UUID Default: not defined.

If defined then each calculation will not print universally unique identifier. This removes the dependency on libuuid.

DISABLE_SCALAPACK Default: not defined

If defined then FCI calculations in parallel are disabled, and the dependency on ScaLAPACK is removed.

DISABLE_BACKTRACE Default: not defined

If defined then the backtrace is disabled. The backtrace functionality is a GNU extension and not available on all POSIX architectures. No working functionality is lost.

DSFMT_MEXP Default: 19937.

HANDE uses the dSFMT random number generator (RNG). It is based on a Mersenne Twister algorithm, is extremely fast and produces high quality random numbers. See http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html for more details.

DSFMT_EXP sets the exponent of the period of the RNG. Allowed values are 521, 1279, 2203, 4253, 11213, 19937, 44497, 86243, 132049 and 216091 and lead to, for example, random numbers with a period of a Mersenne Prime such as $2^{512} - 1$.

NAGF95 Default: not defined.

If defined then code specific to, and necessary for compilation using, the NAG Fortran compiler is included.

PARALLEL Default: not defined.

Include source code required for running in parallel.

SINGLE_PRECISION Default: not defined.

Set the precision (where possible) to be single precision. The default is double precision. This is faster, but (of course) can change results significantly. Use with care.

USE_POPCNT Default: not defined.

Use the intrinsic popcnt function instead of the version implemented in HANDE.

An important procedure involves counting the number of set bits in an integer. HANDE includes a very efficient, branchless procedure to do this. However, the Fortran 2008 standard includes an intrinsic function, popcnt, for this exact operation. The performance of this intrinsic will be implementation-dependent and, with standard compilation flags, we expect the HANDE version to be competitive or more performant, based upon some simple tests. The key difference is on modern processors containing the popcnt instruction: the popcnt intrinsic can then make use of this instruction and will be much faster than the implementation in HANDE. The existence of the popcnt instruction can be found, on Unix and Linux platforms, by inspecting the flags field in `/proc/cpuinfo`: if it contains `popcnt`, then the processor contains the popcnt instruction.

Using the popcnt instruction often involves a compiler-specific flag to tell the compiler to use that instruction set; often compilers include the popcnt instruction with the flag that specifies the use of the SSE4.2 instruction set. The use of the popcnt instruction can be tested using objdump. For example:

```
$ objdump -d bin/hande.x | grep popc
0000000000400790 <__popcountdi2@plt>:
  400931:e8 5a fe ff ff         callq  400790 <__popcountdi2@plt>
```

indicates that HANDE is using a compiler-supplied function for popcnt. Exact output (especially the function name) is compiler dependent. In contrast:

```
$ objdump -d bin/hande.x | grep popc
  4008ac:f3 0f b8 c0            popcnt %eax,%eax
```

indicates HANDE is using the popcnt instruction. If the above command does not give any output, then USE_POPCNT has most likely not been defined.

1.4.2 Compilation issues

We attempt to work round any compiler and library issues we encounter but sometimes this is not possible. Issues and, where known, workarounds we have found are:

- HDF5 1.8.14 (and possibly 1.8.13) has a bug revealed by Intel compilers v15 onwards. This results in unusual error messages and/or segmentation faults when writing out restart files. Possibly workarounds:
  - use HDF5 1.8.15 (best).
  - recompile HDF5 with `-assume nostd_value`.
  - recompile HDF5 with an earlier version of the Intel compilers.
  - recompile HANDE with HDF5 support disabled.
- Compiling with GCC and linking the Intel MKL library leads to segmentation faults or incorrect answers for FCI calculation on systems with complex-valued integrals when run in parallel. Either use a different ScaLAPACK library, or use the Intel compilers.

1.5 Test suite

HANDE has an extensive test suite covering all core functionality. The tests are run using the `testcode` package (https://github.com/jsspencer/testcode). Note that the data extraction scripts for HANDE require the `pandas` python library.

testcode can be run from the test_suite subdirectory:

```
$ testcode.py
```

As the full test suite is extensive, it can take a long time to run, so the `quick` and even shorter `vquick` categories are provided that aim to test most functionality but run in a few minutes. By default the only the `quick` tests are run. The entire test suite is run every night using buildbot (http://www.cmth.ph.ic.ac.uk/buildbot/hande/).

Selected data from the HANDE output is compared to known 'good' results ('benchmarks').

testcode is quite flexible and it's easy to run subsets of tests, check against different benchmarks, compare previously run tests, run tests concurrently for speed, etc. Please see the testcode documentation for more details.

Note: For algorithmic reasons, certain compilation and runtime options (principally POP_SIZE and processor/thread count) result in different Markov chains and hence different exact results (but same results on average). The tests should therefore be run using the same compilation options and the same parallel distribution as was used for the benchmarks. The latter for MPI parallelisation is done automatically by testcode. Separate tests exist for both POP_SIZE=32 and POP_SIZE=64.

Similarly, the tests will not pass to default accuracy if using SINGLE_PRECISION. There is a single_precision category, consisting of the tests which will pass with a tolerance set to 10^{-5}.

Currently there are no QMC tests suitable for OpenMP parallelisation due to difficulties with making the scheduler behave deterministically without affecting performance of production simulations. It is advised that you make sure to set the shell variable OMP_NUM_THREADS to 1 when running the test suite - otherwise these will all be marked SKIPPED.

1.5.1 What if the tests fail?

A common cause for tests failing is that the configuration causes a different Markov Chain to be run, or part of the code has been disabled in your build. testcode should determine that some tests are inappropriate and skip them. To force testcode to skip some tests, see below.

A second cause of failure is that some floating point values have rounded differently on different architectures. The tolerances used for the tests can also be adjusted as specified below:

1.5.2 Skipping Tests

If there is a unique line printed out in the output for jobs which are to be skipped, this can be used to tell testcode this, by modifying the skip_args line in the test_suite/userconfig file. See the testcode documentation for more details

1.5.3 Adjusting Test Tolerances

The tolerance for an individual job can be modified as specified in the testcode documentation. As an example, to modify the tolerance because of the following failure:

```
dmqmc/np1/heisenberg_1d - replica.in: **FAILED**.
\sum\rho_{ij}M2{ji}
    ERROR: absolute error 1.00e-06 greater than 1.00e-10. (Test: 17.378583.  Benchmark: 17.378584.)
```

The follow section can be inserted into test_suite/jobconfig. Note the backslash-quoting of the backslashes, as the tolerance value is interpreted as a python tuple containing a python string.

```
#Job specific tolerances:
[dmqmc/np1/heisenberg_1d/]
tolerance = (1e-5,1e-5,'\\sum\\rho_{ij}M2{ji}')
```

1.6 Usage

```
$ hande.x [input_filename]
```

Output is sent to STDOUT and can be redirected as desired.

1.6.1 Parallel Usage

Using MPI only:

```
$ mpirun -np n hande.x [input_filename]
```

where `n` is the number of processors to run on in parallel. On an HPC system this may differ (for example mpirun -np n may be replaced with mpiexec), depending on how the environment has been set up.

Using OpenMP parallelism:

```
$ export OMP_NUM_THREADS=n
$ hande.x [input_filename]
```

OpenMP parallelism is currently only implemented for CCMC.

Using OpenMP and MPI parallelism:

```
$ export OMP_NUM_THREADS=n
$ mpirun -np m hande.x [input_filename]
```

where `m` is the number of MPI processes and `n` is the number of OpenMP threads per MPI process. HANDE prints this information at the top of the output, so one can easily check there environment is set up correctly.

HANDE only performs I/O operations on the root processor when run on multiple processors.

Note: Due to the implementation of efficient Monte Carlo algorithms, running the Monte Carlo algorithms in HANDE on different numbers of processors (or using OpenMP) results in different Markov chains and hence such calculations will not agree exactly but instead statistically.

1.7 Input file

HANDE is controlled via an input file which is a simple lua script. This has the advantage of creating a clean, simple interface to HANDE whilst allowing advanced users to perform complex simulations without requiring parsing complicated (and perhaps bespoke) logic in a custom input parser. Future work will include exposing more of HANDE via the lua API, thus increasing the flexibility available.

Running a simulation typically involves creating a quantum system (i.e. a collection of spins/fermions/etc acting under a specified Hamiltonian) and then performing one or more calculations on that system. Both tasks involve calling functions from the input file.

The following sections detail options available in each system and calculation function. Variables can be required (i.e. must be specified if a function is called) or optional (in which case the default is stated). The type (e.g. float, such as 1.234; integer, such as 1; boolean, either `true` or `false`) of each variable is also given.

1.7.1 Systems

All functions which create a system return a pointer to a `system` object (which currently cannot be manipulated or inspected from lua). All calculation functions take this variable as an argument.

Model lattice systems

Hubbard model (momentum-space)

```
hubbard_k {
    -- options,
}
```

Returns: a system object.

`hubbard_k` creates a system object for the Hubbard model:

$$H = -t \sum_{<r,r'>,\sigma} c^\dagger_{r,\sigma} c_{r',\sigma} + U \sum_r n_{r,\uparrow} n_{r,\downarrow}$$

using a single-particle basis of Bloch functions, ψ_k:

$$\psi_k(r) = e^{ik.r} \sum_i \phi_i(r)$$

where $\phi_i(r)$ is a single-particle basis function centred on site i in real space.

Options

`sys` type: system object produced by a previous call.

Optional.

If provided, a previously created system object is updated with the new settings supplied, otherwise a new system object is created.

`electrons` type: integer.

Required.

Number of electrons in the unit cell.

`lattice` type: N N-dimensional vectors of floats.

Required.

Unit cell on which periodic boundary conditions are placed. See below.

`ms` type: integer.

Required.

Set the spin polarisation of the system in units of electron spin (i.e. a single electron can take values 1 or -1).

`sym` type: integer.

Required for deterministic calculations and highly recommended for Monte Carlo calculations.

Set the symmetry (i.e. crystal momentum) of the system. This is the index of a specific wavevector; see the output produced by creating a system for possible values and their corresponding wavevectors. If not specified (and no reference determinant supplied for a calculation) the symmetry used is that of a determinant selected using the Aufbau principle.

`U` type: float.

Optional. Default: 1.

Specifies the U parameter in the Hamiltonian.

`t` type: float.

Optional. Default: 1.

Specifies the t parameter in the Hamiltonian.

`twist` type: N-dimensional vector

Optional. Default: 0 in each dimension.

Apply a twist to the wavevector grid. The twist is an *ndim*-dimensional vector in units of 2π. The twist angle should be within the first Brillouin zone, and hence the components should be between -0.5 and +0.5.

`verbose` type: boolean.

Optional. Default: true.

Print out the single-particle basis set.

Hubbard model (real-space)

```
hubbard_real {
    -- options,
}
```

Returns: a system object.

`hubbard_real` creates a system object for the Hubbard model:

$$H = -t \sum_{<r,r'>,\sigma} c^\dagger_{r,\sigma} c_{r',\sigma} + U \sum_r n_{r,\uparrow} n_{r,\downarrow}$$

using a single-particle basis of functions in real-space.

Options

`sys` type: system object produced by a previous call.

Optional.

If provided, a previously created system object is updated with the new settings supplied, otherwise a new system object is created.

`electrons` type: integer.

Required.

Number of electrons in the unit cell.

`lattice` type: N N-dimensional vectors of floats.

Required.

Unit cell on which periodic boundary conditions are placed. See below.

`ms` type: integer.

Required.

Set the spin polarisation of the system in units of electron spin.

`U` type: float.

Optional. Default: 1.

Specifies the U parameter in the Hamiltonian.

`t` type: float.

Optional. Default: 1.

Specifies the t parameter in the Hamiltonian.

`finite` type: boolean.

Optional. Default: false.

If false then periodic boundary conditions are applied to the unit cell, otherwise the system specified by the lattice is treated as an isolated set of sites.

`verbose` type: boolean.

Optional. Default: true.

Print out the single-particle basis set.

Heisenberg model

```
heisenberg {
    -- options,
}
```

Returns: a system object.

`heisenberg` creates a system object for the Heisenberg model, which models a set of spin 1/2 particles on a lattice:

$$\hat{H} = -J \sum_{\langle i,j \rangle} \hat{\boldsymbol{S}}_i \cdot \hat{\boldsymbol{S}}_j - h_z \sum_i \hat{S}_{iz} - h'_z \sum_i \hat{S}^{\xi}_{iz},$$

where h_z and h'_z denote the magnetic field strength and staggered magnetic field strength, respectively, and ξ is equal to +1 for sites on sublattice 1 and is equal to -1 for sites on sublattice 2.

Options

`sys` type: system object produced by a previous call.

Optional.

If provided, a previously created system object is updated with the new settings supplied, otherwise a new system object is created.

`lattice` type: N N-dimensional vectors of floats.

Required.

Unit cell on which periodic boundary conditions are placed. See below.

> **Warning:** For efficiency reasons it is assumed that the smallest dimension lattice vector is greater than 2 if periodic boundary conditions are used.

`ms` type: integer.

Required.

Set the spin polarisation of the system in units of 1/2.

`J` type: float.

Optional. Default: 1.

Set the coupling constant for the Heisenberg model.

`magnetic_field` type: float.

Optional. Default: 0.

`staggered_magnetic_field` type: float.

Optional. Default: 0.

Note: Specifying non-zero values for both `magnetic_field` and `staggered_magnetic_field` is not currently possible.

`finite` type: boolean.

Optional. Default: false.

If false then periodic boundary conditions are applied to the unit cell, otherwise the system specified by the lattice is treated as an isolated set of sites.

triangular type: boolean.

Optional. Default: false.

If true, then a triangular lattice of sites on which the spins reside is used, requiring a 2D lattice. The default is to use a N-dimensional cubic arrangement of sites.

verbose type: boolean.

Optional. Default: true.

Print out the single-particle basis set.

Chung-Landau model

```
chung_landau {
    -- options,
}
```

Returns: a system object.

`chung_landau` creates a system object for the system of spinless fermions proposed by Chung and Landau:

$$H = -t \sum_{\langle r,r' \rangle} c_r^\dagger c_{r'} + U \sum_{\langle r,r' \rangle} n_r n_{r'}$$

using a single-particle basis of functions in real-space.

Options

sys type: system object produced by a previous call.

Optional.

If provided, a previously created system object is updated with the new settings supplied, otherwise a new system object is created.

electrons type: integer.

Required.

Number of fermions in the unit cell.

lattice type: N N-dimensional vectors of floats.

Required.

Unit cell on which periodic boundary conditions are placed. See below.

U type: float.

Optional. Default: 1.

Specifies the U parameter in the Hamiltonian.

t type: float.

Optional. Default: 1.

Specifies the t parameter in the Hamiltonian.

finite type: boolean.

Optional. Default: false.

If false then periodic boundary conditions are applied to the unit cell, otherwise the system specified by the lattice is treated as an isolated set of sites.

verbose type: boolean.

Optional. Default: true.

Print out the single-particle basis set.

Specifying the lattice

The lattice is specified as a table of vectors. Sites (on which a spin or electron resides) are at unit locations on the grid. The unit cell (or, if periodic boundary conditions are not used, the geometry of the 'flake' essentially cut out of the infinite lattice) are given in this basis. The lattice variable hence requires N vectors, each of dimension N. This is specified in lua by a nested table. For example:

```
lattice = { { 10 } }
```

sets a 1D system, with the unit cell containing 10 sites;

```
lattice = { { 2, 0 }, { 0, 2 } }
```

sets a 2D system, with the unit cell containing 4 sites; and

```
lattice = { { 3, 3 }, { 3, -3 } }
```

sets a 2D system, with the (square) unit cell containing 18 sites and rotated by $45°$ relative to the primitive lattice.

HANDE supports 1-, 2- and 3-dimensional lattices. Lattice vectors must be orthogonal.

Electron gases

An electron gas contains interacting electrons in some geometry with a constant compensating positive charge.

Uniform electron gas

```
ueg {
    -- options,
}
```

Returns: a system object.

`ueg` creates a system object for the (conventional) electron gas:

$$H = -\frac{1}{2}\sum_i \nabla_i^2 + \sum_{i<j} \frac{1}{r_{ij}}$$

(including an appropriate uniform background potential to counteract the charge), using a single-particle basis of plane waves, $\psi_{\mathbf{k}} = e^{i\mathbf{k}.\mathbf{r}}$.

Options

sys type: system object produced by a previous call.

Optional.

If provided, a previously created system object is updated with the new settings supplied, otherwise a new system object is created.

electrons type: integer.

Required.

Number of electrons in the unit cell.

ms type: integer.

Required.

Set the spin polarisation of the system in units of electron spin (i.e. a single electron can take values 1 or -1).

sym type: integer.

Required for deterministic calculations and highly recommended for Monte Carlo calculations.

Set the symmetry (i.e. crystal momentum) of the system. This is the index of a specific wavevector; see the output produced by creating a system for possible values and their corresponding wavevectors. If not specified (and no reference determinant supplied for a calculation) the symmetry used is that of a determinant selected using the Aufbau principle.

rs type: float.

Optional. Default: 1.

Set the density, r_s, of the UEG.

cutoff type: float.

Optional. Default: 3.

Set the maximum kinetic energy of the orbitals included in the basis set.

Note that this is in scaled units of $(2\pi/L)^2$, where L is the dimension of simulation cell defined by *electrons* and *rs* and is compared to the kinetic energy of each plane-wave without the twist angle included. In this way the cutoff can be kept constant whilst the twist is varied and the basis set used will remain consistent.

dim type: integer.

Optional. Default: 3.

Set the dimension of the electron gas. 2- and 3-dimensional gases are implemented.

twist type: N-dimensional vector

Optional. Default: 0 in each dimension.

Apply a twist to the wavevector grid. The twist is an N-dimensional vector in units of 2π. The twist angle should be within the first Brillouin zone, and hence the components should be between -0.5 and +0.5.

verbose type: boolean.

Optional. Default: true.

Print out the single-particle basis set.

Ringium

```
ringium {
    -- options,
}
```

Returns: a system object.

Ringium *[Loos13]*, is a 1D system of electrons confined to a ring of radius R:

$$H = -\frac{1}{2R^2}\sum_i \frac{\partial^2}{\partial\theta_i^2} + \sum_{i<j}\frac{1}{r_{ij}}$$

where $r_{ij} = R\sqrt{2-2\cos(\theta_i-\theta_j)}$, using a single-particle basis of functions $\psi_n = e^{in\theta}$. As it is 1D, the different spin polarisations are degenerate, so without loss of generality all electrons are forced to be spin up.

Options

`sys` type: system object produced by a previous call.

Optional.

If provided, a previously created system object is updated with the new settings supplied, otherwise a new system object is created.

`electrons` type: integer

Required.

Number of electrons in the system.

`radius` type: float

Required.

The radius of the ring.

`maxlz` type: integer

Required.

The maximum angular momentum of the orbitals used in the basis set.

Note that this is in units of $\frac{\hbar}{2}$ and must have opposite parity to the number of electrons.

`verbose` type: boolean.

Optional. Default: true.

Print out the single-particle basis set.

Generic systems

```
read_in {
    -- options
}
```

Returns: a system object.

A generic system, including atoms and molecules, can be specified by providing a file containing information about the single-particle basis set and the one- and two-body integrals between these basis functions. This file is in FCIDUMP format *[Knowles89]*, which can be produced by several quantum chemistry packages including MOLPRO, Q-Chem (via additions from Alex Thom) and PSI4 (via a plugin from James Spencer). See *Generating integrals* for more details.

Options

sys type: system object produced by a previous call.

Optional.

If provided, a previously created system object is updated with the new settings supplied, otherwise a new system object is created.

electrons type: integer.

Required unless `int_file` is in HDF5 format. If specified, then `ms` **must** be specified..

Number of electrons in the unit cell.

ms type: integer.

Required unless `int_file` is in HDF5 format. If specified, then `electrons` **must** be specified..

Set the spin polarisation of the system in units of electron spin (i.e. a single electron can take values 1 or -1).

sym type: integer.

Required for deterministic calculations and highly recommended for Monte Carlo calculations.

Set the symmetry of the system. This is the index of a specific irreducible representation from the FCIDUMP file; see the output produced by creating a system for possible values. If not specified (and no reference determinant supplied for a calculation) the symmetry used is that of a determinant selected using the Aufbau principle.

Lz type: boolean.

Optional. Default: false.

If true, enable L_z symmetry. See below for details.

int_file type: string.

Optional. Default: 'FCIDUMP'.

Specify the FCIDUMP file containing the integrals and information relating to the single-particle basis. This can also be an HDF5 file previously produced by HANDE from a FCIDUMP via the `write_read_in_system` function (see *Write HDF5 system file*), which is both more compact in size and considerably faster to process.

dipole_int_file type: string.

Optional. No default.

Specify a FCIDUMP-like file containing the dipole integrals, i.e. $\langle i|x|i\rangle$, in a given direction.

Not currently used.

CAS type: 2D-vector of integers.

Optional. No default.

If specified, then the basis set is restricted to a given complete active space, whereby `CAS = {N,M}` corresponds to allowing only N electrons to be distributed among $2M$ spin orbitals. Any additional electrons are 'frozen' (i.e. forced to be in the lowest spin orbitals) and any additional high-energy spin orbitals are removed from the basis set.

Warning: This functionality is not compatible with reading from an HDF5 file; to use a CAS in combination with HDF5 initialisation, create the HDF5 file using a system with the desired CAS.

verbose type: boolean.

Optional. Default: true.

Print out the single-particle basis set.

complex type: boolean.

Optional. Default: false.

Specify if the calculation should use complex dynamics in any calculation performed, and if the FCIDUMP supplied is complex-formatted. Currently only compatible with fci calculations.

L_z symmetry For cylindrically symmetrical systems, the L_z (z-component of orbital angular momentum) operator commutes with the Hamiltonian, and this can be a convenient symmetry to conserve. L_z is measured in units of $\hbar$. Normal FCIDUMP files do not contain orbitals which are eigenfunctions of the L_z operator, so they must be transformed using post-processing. The TransLz script from the NECI project can be used for this purpose. The FCIDUMP file header format has been modified to include additional parameters: SYML, and SYMLZ which have a list of values, one for each orbital. SYML gives the magnitude of L for the orbital if known (or -20 if not) but is not used. SYMLZ give the eigenvalue of L_z (the m_l value). Orbitals with defined values of L_z are likely to be complex-valued, but luckily the integrals involving them are not, so although the FCIDUMP file must be translated, it still retains the same format (see comments in `src/read_in.F90` and `src/molecular_integrals.F90` for details if you wish to create FCIDUMP files by other means).

Warning: These transformed integral files require you to enforce L_z symmetry and will produce incorrect results if you do not.

1.7.2 Calculations

All QMC methods (FCIQMC, CCMC, DMQMC and the simple FCIQMC implementation) return a pointer to a `qmc_state` object (which cannot be directly manipulated or inspected from lua) as the first return value. They also accept such an object as an optional argument to resume a previous QMC calculation. Additional, calculation-specific, values are returned in some cases, as described in the relevant section.

Full Configuration Interaction

Calculate the ground state of a system via a full diagonalisation of the Hamiltonian matrix *[Knowles89]*.

```
fci {
    sys = system,
    fci = { ... },
    lanczos = { ... },
    reference = { ... },
}
```

Note: The FCI engine in HANDE is particularly simple (i.e. slow, dumb, memory hungry) and is designed mainly for testing. A conventional quantum chemistry package, such as MOLPRO, or PSI4, is highly recommended for production FCI calculations as these implement substantially more efficient algorithms.

Options

sys type: system object.

Required.

The system on which to perform the calculation. Must be created via a system function.

fci type: lua table.

Optional. No default.

Further FCI options. See below.

lanczos type: lua table.

Optional. No default.

Table containing Lanczos-specific options; see below. If present, the diagonalisation is performed via an iterative Lanczos algorithm. Otherwise diagonalisation is performed using LAPACK or ScaLAPACK.

reference type: lua table.

Optional. No default.

If not specified, the entire Hilbert space is used. See *reference options*.

fci options

The `fci` table can take the following options:

write_hamiltonian type: boolean.

Optional. Default: false.

Write out the diagonal and the non-zero off-diagonal elements of the Hamiltonian matrix.

hamiltonian_file type: string.

Optional. Default: 'HAMIL'.

Filename to which the Hamiltonian matrix is written.

write_determinants type: boolean.

Optional. Default: false.

Write out the enumerated list of determinants in the FCI Hilbert space.

determinant_file type: string.

Optional. Default: 'DETS'.

Filename to which the list of determinants (or, more generally, many-body basis functions) is written.

write_nwfns type: integer.

Optional. Default: 0.

Number of wavefunctions to write out (in the basis of Slater determinants). A negative value indicates all wavefunctions are to be written out.

wfn_file type: string.

Optional. Default: 'FCI_WFN'.

Filename to which the wavefunctions are written.

nanalyse type: integer.

Optional. Default: 0.

Calculate properties of the first *nwfn* FCI wavefunctions from each spin and symmetry block. If nwfn is negative (default) then all wavefunctions are analysed. This is slow, and uses a very simple algorithm. It is only designed for debugging purposes. The properties evaluated depend upon the system and are liable to change without warning.

blacs_block_size type: integer.

Optional. Default: 64.

The block size used by BLACS to distribute the Hamiltonian matrix across the processors with MPI parallelism. The Hamiltonian matrix is divided into $n \times n$ sub-matrices, where n is the block size, which are the distributed over the processors in a cyclic fashion.

rdm type: table of integers.

Optional. No default.

If present, calculate the eigenvalues for the reduced density matrix consisting of the specified list of sites, with a trace performed over all other sites.

Note: The `rdm` option is only currently available for Heisenberg systems and cannot be used with the Lanczos algorithm.

Note: The `write_wfn`, `nanalyse` and `rdm` options require the eigenvectors to be calculated in addition to the eigenvalues, which requires additional computational time.

---

lanczos options

The `lanczos` table can take the following options:

neigv type: integer.

Optional. Default: 5.

Number of lowest eigenstates to be found.

nbasis type: integer.

Optional. Default: 40.

Number of Lanczos vectors used. The size of the basis can have an impact on the performance of the Lanczos diagonalisation and which excited eigensolutions are found. See the TRLan documentation, for more details.

direct type: boolean.

Optional. Default: false.

If true, generate the Hamiltonian matrix on the fly (very slow). Otherwise generate the Hamiltonian once and store it for use at each Lanczos iteration. Not implemented with MPI parallelism.

sparse type: boolean.

Optional. Default: true.

If true store the Hamiltonian in a sparse matrix format. The generation of the Hamiltonian matrix takes longer but requires consequently *much* less memory. Not implemented with MPI parallelism.

Monte Carlo estimate of size of the Hilbert space

Whilst calculating the size of an entire Hilbert space is straightforward via combinatorics, calculating the size of a specific part of the Hilbert space meeting a given set of quantum numbers (e.g. spin and symmetry) is more challenging. Instead, the size of this subspace can be estimated via a simple Monte Carlo approach *[Booth10]*.

```
hilbert_space {
    sys = system,
    hilbert = { ... },
}
```

Options

All options should be in the hilbert table bar the sys option.

`sys` type: system object.

Required.

The system on which to perform the calculation. Must be created via a system function.

`hilbert` type: lua table.

Required.

Further options to control the Monte Carlo estimation of the Hilbert space. See below.

hilbert options

The `hilbert` table can take the following options:

`nattempts` type: integer.

Required.

Number of random attempts (i.e. the number of random determinants to generate) to perform per Monte Carlo cycle.

`ncycles` type: integer

Optional. Default: 20.

Number of Monte Carlo cycles to perform. Each cycle produces an independent estimate of the Hilbert space size. Estimates of the mean and standard error are automatically calculated from each independent value.

`rng_seed` type: integer.

Optional. Default: generate a seed based upon the time and UUID (if available).

Seed for initialising the random number generator.

`reference` type: vector of integers.

Optional. Default: attempt to make a good guess based upon the spin and symmetry quantum numbers of the system.

The reference determinant as a list of occupied spin-orbitals. The reference determinant is used in the generation of truncated Hilbert spaces only.

`ex_level` type: integer.

Optional. Default: set to the number of electrons in the system (i.e. generate the FCI space).

Maximum excitation level to consider relative to the reference determinant.

Canonical total energy

```
canonical_estimates {
    sys = system,
    canonical_estimates = { ... },
}
```

`canonical_estimates` calculates various estimates for properties of a system in the canonical ensemble at a given temperature, using knowledge of the grand canonical ensemble and the single-particle eigenvalues of the underlying non-interacting system. See *[Malone15]* for details. Currently only implemented for the UEG and read_in.

Options

`sys` type: system object.

Required.

The system on which to perform the calculation. Must be created via a system function.

`canonical_estimates` type: lua table.

Required.

Further options controlling the calculation.

kinetic options

`ncycles` type: integer.

Required.

The number of Monte Carlo iterations to perform. Each iteration produces independent estimates based upon the `nattempts` made.

`nattempts` type: integer.

Required.

Number of determinants within the canonical ensemble we attempt to generate each Monte Carlo cycle.

`beta` type: float.

Required.

The temperature of the system.

`fermi_temperature` type: boolean.

Optional. Default: false.

If true, rescale `beta` as the inverse reduced temperature: $\tilde{\beta} = 1/\Theta = T_F/T$, where T_F is the Fermi temperature. If false, `beta` is taken to be in atomic units.

`rng_seed` type: integer.

Optional. Default: generate a seed from a hash of the time and calculation UUID.

The seed used to initialise the random number generator.

Full Configuration Interaction Quantum Monte Carlo

```
fciqmc {
    sys = system,
    qmc = { ... },
    fciqmc = { ... },
    semi_stoch = { ... },
    restart = { ... },
    reference = { ... },
    load_bal = { ... },
    qmc_state = qmc_state,
}
```

Returns: a qmc_state object.

`fciqmc` performs a full configuration interaction quantum Monte Carlo (FCIQMC) calculation *[Booth09]* on a system.

Options

`sys` type: system object.

Required.

The system on which to perform the calculation. Must be created via a system function.

`qmc` type: lua table.

Required.

Further options that are common to all implemented QMC algorithms. See *qmc options*.

`fciqmc` type: lua table.

Optional.

Further options to control the FCIQMC algorithm. See *fciqmc options*.

`semi_stoch` type: lua table.

Optional.

Further options to control using a semi-stochastic projection of the Hamiltonian operator instead of a purely stochastic projection. Note that some options in the `semi_stoch` table are required to be set if the table is given. See *semi_stoch options*.

`restart` type: lua table.

Optional.

Further options to control restarting the calculation from a previous calculation. See *restart options*.

`reference` type: lua table.

Optional.

Further options to select the reference state used. See *reference options*.

`load_bal` type: lua table.

Optional.

Further options to improve the parallel load balancing of an FCIQMC simulation. If present (even if empty) an advanced load-balancing algorithm is used *[Malone16a]*. See *load_bal options* for more details.

qmc_state type: qmc_state object.

Optional.

Output of a previous calculation to resume.

> **Warning:** The qmc_state object must have been returned by a previous FCIQMC calculation. The validity of this is not checked. The system must also be unchanged.

fciqmc options

select_reference_det type: boolean or Lua table.

Optional. Default: false.

If true or if a lua table is provided, attempt to automatically set the reference state to be the state with the greatest population. A lua table can contain the following options and need only be provided in order to modify the defaults.

Note: Care should be take when analysing the projected estimator to ensure that all quantities averaged have the same reference state.

> **Warning:** Excitation levels are relative to the reference state and hence this should **not** be used with a truncated CI calculation.

update_every type: integer

Optional. Default: 20.

The number of report loops between attempts to update the reference state.

pop_factor type: float.

Optional. Default: 1.5.

The factor of the reference population another state must have in order for the reference to be changed. This helps prevent continually switching between states with similar or degenerate populations.

non_blocking_comm type: boolean.

Optional. Default: false.

Use non-blocking MPI communications instead of blocking MPI communications.

Note: This is an experimental option and may or may not improve performance. In particular, its efficiency is highly dependent upon architecture and MPI implementation. For expert use only!

---

load_balancing type: boolean.

Optional. Default: false.

Enable dynamic load balancing of determinants among processors. This will move determinants to try and keep the number of walkers on each processor roughly constant. See *load_bal options* for more details.

init_spin_inverse_reference_det type: boolean.

Optional. Default: false.

In addition to initialising the reference determinant with an initial population, initialise the spin-inversed determinant (if different) with the same population. Overridden by a restart file.

trial_function type: string.

Optional. Default: 'single_basis'.

Possible values: 'single_basis', 'neel_singlet' (Heisenberg model only).

The trial function to use in the projected energy estimator. 'single_basis' uses the single reference state as the trial function. 'neel_singlet' uses the Neel singlet state, $|NS\rangle = \sum_i a_i |D_i\rangle$, where the amplitudes a_i are defined in K. Runge, Phys. Rev. B 45, 7229 (1992).

Using a multi-reference trial function can substantially reduce stochastic noise.

guiding_function type: string.

Optional. Default: 'none'.

Possible values: 'none', 'neel_singlet' (Heisenberg model only).

The importance sampling transformation to apply to the Hamiltonian.

'neel_singlet' uses the Neel singlet state (K. Runge, Phys. Rev. B 45, 7229 (1992)) to transform the Hamiltonian such that the matrix elements, H_{ij}, are replaced with $a_i H_{ij} / a_j$. Using 'neel_singlet' automatically sets `trial_function` to 'neel_singlet'.

load_bal options

The default values are usually sufficient if load balancing is enabled. It is highly recommended to only attempt to improve load balancing for large calculations and once the population has been stabilised by the shift. It may be easiest to do this by monitoring a calculation carefully until this condition is reached, producing a restart file and then running a production calculation with load balancing enabled.

nslots type: integer.

Optional. Default: 20.

The average number of slots per processor used to distribute the list of occupied states via a hashing of the states. A large value will affect performance but could potentially result in a better distribution of walkers.

min_pop type: integer.

Optional. Default: 1000.

The minimum total population required before load balancing is attempted. This is a system dependent value and, in order to maximise performance improvements, should be set such that the population is roughly stable.

target type: float.

Optional. Default: 0.05.

Desired imbalance (as a percentage of the average population per processor) between the most and least populated processors. Note that the workload on a processor is not entirely determined by its population and that, due to the algorithms used, an arbitrary small population imbalance is not usually possible.

max_attempts type: integer.

Optional. Default: 2.

The number of attempts to make to improve load balancing. Often multiple attempts can improve the balancing but each attempt may be non-negligible and there are usually diminishing returns.

write type: boolean.

Optional. Default: false.

Write out the population of the most and least heavily populated processor before and after load balancing is carried out. Also print out the minimum slot population on the most populated processor which will indicate if load balancing is possible.

Coupled Cluster Monte Carlo

```
ccmc {
    sys = system,
    qmc = { ... },
    ccmc = { ... },
    restart = { ... },
    reference = { ... },
    qmc_state = qmc_state,
}
```

Returns: a qmc_state object.

ccmc performs a coupled cluster Monte Carlo (CCMC) calculation *[Thom10]* on a system.

Options

sys type: system object.

Required.

The system on which to perform the calculation. Must be created via a system function.

qmc type: lua table.

Required.

Further options that are common to all implemented QMC algorithms. See *qmc options*.

ccmc type: lua table.

Required.

Further options to control the CCMC algorithm. See *ccmc options*.

restart type: lua table.

Optional.

Further options to control restarting the calculation from a previous calculation. See *restart options*.

reference type: lua table.

Optional.

Further options to select the reference state used. See *reference options*.

qmc_state type: qmc_state object.

Optional.

Output of a previous calculation to resume.

Warning: The qmc_state object must have been returned by a previous CCMC calculation. The validity of this is not checked. The system must also be unchanged.

ccmc options

`move_frequency` type: integer

Optional. Default: 5.

Allow excitors to move processors every 2^x iterations, where x is the value of `move_frequency`, in order to allow all composite excitors to be correctly sampled. Relevant only when performing CCMC with MPI parallelisation. A large value may introduce a bias. Modify with caution.

`cluster_multispawn_threshold` type: float.

Optional. Default: $2^{31} - 1$.

Set the maximum value of A_C/p_C, where A_C is the cluster amplitude and p_C is the probability of selecting the cluster. A cluster with a value above this is split into multiple spawning attempts. The default value essentially disables this but a smaller option can substantially reduce population blooms, albeit potentially at a significant computational cost.

Note: This is an experimental option and feedback is most welcome. The current recommendation is to use the smallest setting such that large blooms do not occur.

`full_non_composite` type: boolean.

Optional. Default: false.

If true, allow all non-composite clusters to attempt to spawn each iteration. The original CCMC algorithm involves randomly selecting a cluster of arbitrary size consisting of any set of excitors and then making spawning attempts from it. The full non-composite algorithm is a simple modification in which all occupied non-composite clusters (i.e. those consisting of the reference or just a single excitor) are (deterministically) selected and composite clusters (involving two or more excitors) are randomly selected to make spawning attempts. This has been shown to give substantially more stable dynamics and reduce the plateau height in several systems.

`linked` type: boolean.

Optional. Default: false.

If true, sample the linked coupled cluster equations instead of the unlinked coupled cluster equations *[Franklin16]*. The original CCMC algorithm solves the equations

$$\langle D_m|\hat{H} - E|\psi_{CC}\rangle = 0.$$

It is possible to instead sample the equivalent equations

$$\langle D_m|e^{-\hat{T}}(\hat{H} - E)|\psi_{CC}\rangle = 0.$$

Using the Hausdorff expansion of the Hamiltonian and the linked cluster theorem means that the only clusters which contribute are those with at most four excitors and where the excitation sampled from the Hamiltonian has an orbital in common with each excitor in the cluster operator. Using this option can give substantial reductions in the plateau height.

Density Matrix Quantum Monte Carlo

```
dmqmc {
    sys = system,
    qmc = { ... },
    dmqmc = { ... },
    ipdmqmc = { ... },
    operators = { ... },
    rdm = { ... },
    restart = { ... },
    reference = { ... },
    qmc_state = qmc_state,
}
```

Returns: a qmc_state object. a lua table containing the sampling probabilities found if `find_weights` is set to `true`. This can be passed directly to the `weights` option of a subsequent DMQMC calculation and/or manipulated inside the lua script. If `find_weights` is `false`, only the qmc_state object is returned.

`dmqmc` performs a density matrix quantum Monte Carlo (DMQMC) calculation on a system.

Unlike *Coupled Cluster Monte Carlo* and *Full Configuration Interaction Quantum Monte Carlo*, where quantities are averaged inside each report loop, any quantities in DMQMC are evaluated at the **first** iteration of the report loop only. This is because different iterations represent different temperatures in DMQMC, and so averaging over a report loop would average over different temperatures, which is not the desired behaviour.

Note: Density Matrix Quantum Monte Carlo is currently rather experimental. In particular, it is not implemented for all systems yet and some options are only implemented for specific systems. In particular, DMQMC is only implemented for the Heisenberg model, the UEG, the real and momentum-space Hubbard model, and for molecular systems. The evaluation of operators other than the total energy, such as correlation functions and entanglement measures, is currently only possible for the Heisenberg model. The calculation of the reduced density matrices from DMQMC is also only supported for the Heisenberg model (for both temperature-dependent and ground state RDM calculations).

Options

`sys` type: system object.

Required.

The system on which to perform the calculation. Must be created via a system function.

`qmc` type: lua table.

Required.

Further options that are common to all implemented QMC algorithms. See *qmc options*.

`dmqmc` type: lua table.

Optional.

Further options to control the DMQMC algorithm. See *dmqmc options*.

`ipdmqmc` type: lua table.

Optional.

If set, even to an empty table, then interaction picture DMQMC *[Malone15]* is performed. The table can contain further options to control the IP-DMQMC algorithm. See *ipdmqmc options*.

operators type: lua table.

Optional.

Further options to select the operators for which expectation values are evaluated. See *operators options*.

rdm type: lua table.

Optional.

Further options to select which (if any) reduced density matrices and corresponding operators are to be evaluated. See *rdm options*.

restart type: lua table.

Optional.

Further options to control restarting the calculation from a previous calculation. See *restart options*.

reference type: lua table.

Optional.

Further options to select the reference state used. See *reference options*.

qmc_state type: qmc_state object.

Optional.

Output of a previous calculation to resume.

> **Warning:** The qmc_state object must have been returned by a previous DMQMC calculation. The validity of this is not checked. The system must also be unchanged.

dmqmc options

replica_tricks type: boolean.

Optional. Default: false.

Perform replica simulations (i.e. evolve two independent DMQMC simulations concurrently) if true. This allows calculation of unbiased estimators that are quadratic in the density matrix.

fermi_temperature type: boolean.

Optional. Default: false.

Rescale tau so that the simulation runs in timesteps of $\Delta\tau/T_F$ where T_F is the Fermi temperature. This is so results are at dimensionless inverse temperatures of $\Theta^{-1} = T_F/T$. This option is only valid for systems with a well defined Fermi energy.

all_sym_sectors type: boolean.

Optional. Default: false.

Sample states with all symmetries of the system instead of just those which conserve the symmetry of the reference state.

all_spin_sectors type: boolean.

Optional. Default: false.

Sample states with all spin polarisations of the system instead of just those which conserve the spin polarisation of the reference state.

beta_loops type: integer.

Optional. Default: 100.

The number of loops over the desired temperature range (each starting from $T = \infty$ and performing the desired number of iterations) to perform. Each beta loop samples the initial conditions independently.

Note: Estimators must be averaged at each temperature from different beta loops. As each beta loop is independent, this can be done in separate calculations in an embararassingly parallel fashion.

---

sampling_weights type: vector of floats.

Optional. Default: none.

Specify factors used to alter the spawning probabilities in the DMQMC importance sampling procedure. See PRB, 89, 245124 (2014) for an explanation, in particular section IV and appendix B.

The length of the vector of floats should be equal to the maximum number of excitations from any determinant in the space. For a chemical system with N electrons and more than $2N$ spin orbitals, this would be equal to N. For a Heisenberg model with N spins in the $M_s = 0$ spin sector, this should be equal to $N/2$ (each pair of opposite spins flipped is one excitation).

vary_weights type: integer.

Optional. Default: 0

The number of iterations over which to introduce the weights in the importance sampling scheme (see PRB, 89, 245124 (2014)). If not set then the full weights will be used from the first iteration. Otherwise, the weights will be increased by a factor of $(W_\gamma)^{\beta/\beta_{target}}$ each iteration, where W_γ is the final weight of excitation level γ and β_{target} is the beta value to vary the weights until (equal to the value specified by this option, multiplied by the time step size).

find_weights type: boolean.

Optional. Default: false.

Run a simulation to attempt to find appropriate weights for use in the DMQMC importance sampling procedure. This algorithm will attempt to find weights such that the population of psips is evenly distributed among the various excitation levels when the ground state is reached (at large beta values). The algorithm should be run for several beta loops until the weights settle down to a roughly constant value.

The weights are output at the end of each beta loop.

This option should be used together with the **find_weights_start** option, which is used to specify at which iteration the ground state is reached and therefore when averaging of the excitation distribution begins.

This option cannot be used together with the **excit_dist** option. The **find_weights** option averages the excitation distribution in the ground state, whereas the **excit_dist** option accumulates and prints out the excitation distribution at every report loop.

Warning: This feature is found to be unsuccessful for some larger lattices (for example, 6x6x6, for the Heisenberg model). The weights output should be checked. Increasing the number of psips used may improve the weights calculated.

find_weights_start type: integer.

Optional. Default: 0.

The iteration number at which averaging of the excitation distribution begins, when using the **find_weights** option.

symmetrize type: boolean.

Optional. Default: false.

Explicitly symmetrize the density matrix, thus only sampling one triangle of the matrix. This can yield significant improvements in stochastic error in some cases.

initiator_level type: integer.

Optional. Default: -1.

Set all density matrix elements at excitation level **initiator_level** and below to be initiator determinants. An **initiator_level** of -1 indicates that no preferential treatment is given to density matrix elements and the usual initiator approximation is imposed, 0 indicates that the diagonal elements are initiators, etc.

This is experimental and the user should identity when convergence has been reached.

ipdmqmc options

target_beta type: float.

Optional. Default: 1.0.

The inverse temperature to propagate the density matrix to. If fermi_temperature is set to True then target_beta is interpreted as the inverse reduced temperature $\tilde{\beta} = 1/\Theta = T_F/T$, where T_F is the Fermi temperature. Otherwise target_beta is taken to be in atomic units.

initial_matrix type: string.

Optional. Default: 'hartree_fock'.

Possible values: 'free_electron', 'hartree_fock'.

Initialisation of the density matrix at $\tau = 0$. 'free_electron' samples the free electron density matrix, i.e. $\hat{\rho} = \sum_i e^{-\beta \sum_j \varepsilon_j \hat{n}_j} |D_i\rangle\langle D_i|$, where ε_j is the single-particle eigenvalue and $\hat{n}_j$ the corresponding number operator. 'hartree_fock' samples a 'Hartree–Fock' density matrix defined by $\hat{\rho} = \sum e^{-\beta H_{ii}} |D_i\rangle\langle D_i|$, where $H_{ii} = \langle D_i|\hat{H}|D_i\rangle$.

It is normally best to use the hartree-fock option as this removes cloning/death on the diagonal if the shift is fixed at zero. This requires slightly more work when also using the grand_canonical_initialisation, but this is negligeable.

grand_canonical_initialisation type: boolean.

Optional. Default: false.

Use the grand canonical partition function to initialise the psip distribution. The default behaviour will randomly distribute particles among the determinants requiring a non-zero value of metropolis_attempts to be set for the correct distribution to be reached.

metropolis_attempts type: integer.

Optional. Default: 0.

Number of Metropolis moves to perform (per particle) on the initial distribution. It is up to the user to determine if the desired distribution has been reached, i.e. by checking if results are independent of metropolis_attempts.

symmetric type: boolean.

Optional. Default: false.

Use symmetric version of ip-dmqmc where now $\hat{f}(\tau) = e^{-\frac{1}{2}(\beta-\tau)\hat{H}^0} e^{-\tau\hat{H}} e^{-\frac{1}{2}(\beta-\tau)\hat{H}^0}$.

Warning: This feature is experimental and only tested for the 3D uniform electron gas.

operators options

renyi2 type: boolean.

Optional. Default: false.

Calculate the Renyi-2 entropy of the entire system. Requires `replica_tricks` to be enabled.

energy type: boolean.

Optional. Default: false.

Calculate the thermal expectation value of the Hamiltonian operator.

energy2 type: boolean.

Optional. Default: false.

Calculate the thermal expectation value of the Hamiltonian operator squared. Only available for the Heisenberg model.

staggered_magnetisation type: boolean.

Optional. Default: false.

Calculate the thermal expectation value of the staggered magnetisation operator. Only available for the Heisenberg model and with bipartite lattices.

excit_dist type: boolean.

Optional. Default: false.

Calculate the fraction of psips at each excitation level, where the excitation level is the number of excitations separating the two states labelling a given density matrix element. This fraction is then output to the data table at each report loop, and so the temperature-dependent excitation distribution is printed out.

This option should not be used with the **find_weights** option, which averages the excitation distribution within the ground state.

correlation type: 2D vector of integers.

Optional. Default: false.

Calculate the spin-spin correlation function between the two specified lattice sites, i and j, which is defined as the thermal expectation value of:

$$\hat{C}_{ij} = \hat{S}_{xi}\hat{S}_{xj} + \hat{S}_{yi}\hat{S}_{yj} + \hat{S}_{zi}\hat{S}_{zj}.$$

Only available for the Heisenberg model.

potential_energy type: boolean

Optional. Default: false

Evaluate the bare Coulomb energy. Only available for the UEG.

kinetic_energy type: boolean

Optional. Default: false

Evaluate the kinetic energy. Only available for the UEG.

H0_energy type: boolean

Optional. Default: false

Evaluate the thermal expectation value of the zeroth order Hamiltonian where $\hat{H} = \hat{H}^0 + \hat{V}$. See **initial_matrix** option. Only available when using the ip-dmqmc algorithm.

HI_energy Evaluate the expectation value of the interaction picture Hamiltonian where

$$\hat{H}_I(\frac{1}{2}(\beta-\tau)) = e^{\frac{1}{2}(\beta-\tau)\hat{H}^0}\hat{H}e^{-\frac{1}{2}(\beta-\tau)\hat{H}^0}.$$

rdm options

Note that the use of RDMs is currently only available with the Heisenberg model.

rdms type: table of 1D vectors.

Required.

Each vector corresponds to the subsystem of a reduced density matrix as a list of the basis function indices in the subsystem. For example:

```
rdms = { { 1, 2 } }
```

specifies one RDM containing basis functions with indices 1 and 2, and

```
rdms = { { 1, 2 }, { 3, 4} }
```

specifies two RDMs, with the first containing basis functions with indices 1 and 2, and the second basis functions 3 and 4.

Either `instantaneous` or `ground_state` must be enabled to set the desired mode of evaluating the RDM (but both options cannot be used together).

instantaneous type: boolean.

Optional. Default: false.

Calculate the RDMs at each temperature based upon the instantaneous psip distribution.

Cannot be used with the ground_state option (either ground_state or instantaneous RDMs can be calculated, but not both concurrently).

ground_state type: boolean.

Optional. Default: false.

Accumulate the RDM once the ground state (as specified by `ground_state_start`) is reached. This has two limitations: only one RDM can be accumulated in a calculation and the subsystem should be at most half the size of the system (which is always sufficient for ground-state calculations).

Cannot be used with the instantaneous option (either ground_state or instantaneous RDMs can be calculated, but not both concurrently).

spawned_state_size type: integer.

Required if `instantaneous` is true. Ignored otherwise.

Maximum number of states (i.e. reduced density matrix elements) to store in the "spawned" list, which limits the number of unique RDM elements that each processor can set. Should be a sizeable fraction of `state_size` (see *qmc options*) and depends on the size of the subsystem compared to the full space.

Note: This is a **per processor** quantity. It is usually safe to assume that each processor has approximately the same number of states.

ground_state_start type: integer.

Optional. Default: 0.

Monte Carlo cycle from which the RDM is to be accumulated in each beta loop. Relevant only if `ground_state` is set to true and, as such, should be set to an iteration (which is a measure of temperature) such that the system has reached the ground state.

concurrence type: boolean.

Optional. Default: false.

Calculate the unnormalised concurrence and the trace of the reduced density matrix at the end of each beta loop. The normalised concurrence can be calculated from this using the `average_entropy.py` script.

Valid for `ground_state` only; temperature-dependent concurrence is not currently implemented.

renyi2 type: boolean.

Optional. Default: false.

Calculate the Renyi-2 entropy of each subsystem. More accurately, the quantity output to the data table is $S_2^n = \sum_{ij}(\rho_{ij}^n)^2$, (which differs from the Renyi-2 entropy by a minus sign and a logarithm) where ρ^n is the reduced density matrix of the n-th subsystem. The temperature-dependent estimate of the Renyi-2 entropy can then be obtained using the `finite_temp_analysis.py` script.

Valid for `instantaneous` only; ground-state Renyi-2 averaged over a single beta loop is not currently implemented. Requires `replica_tricks` to be enabled in order to obtained unbiased estimates.

von_neumann type: boolean.

Optional. Default: false.

Calculate the unnormalised von Neumann entropy and the trace of the reduced density matrix at the end of each beta loop. The normalised von Neumann entropy can be calculated from this using the `average_entropy.py` script.

Valid for `ground_state` only; temperature-dependent von Neumann entropy is not currently implemented.

write type: boolean.

Optional. Default: false.

Print out the ground-state RDM to a file at the end of each beta loop. The file contains the trace of the RDM in the first line followed by elements of the upper triangle of the RDM labelled by their index.

Valid for `ground_state` only.

Full Configuration Interaction Quantum Monte Carlo (simple)

Find the ground state of a system via FCIQMC *[Booth09]*.

```
simple_fciqmc {
    sys = system,
    sparse = true/false,
    qmc = { ... },
    restart = { ... },
    reference = { ... },
```

```
    qmc_state = qmc_state,
}
```

Returns: a qmc_state object.

`simple_fciqmc` performs a full configuration interaction quantum Monte Carlo (FCIQMC) calculation on a system using an explicitly calculated and stored Hamiltonian matrix.

> **Warning:** This is an **extremely** simple implementation of FCIQMC. In particular it makes no effort to be efficient (in time or memory), is not parallelised, and does not include any advanced features. It is, however, useful for educational purposes and (occasionally) hacking experimental ideas quickly. Do **not** use for production calculations.

Options

`sys` type: system object.

Required.

The system on which to perform the calculation. Must be created via a system function.

`sparse` type: boolean.

Optional. Default: true.

Store the Hamiltonian matrix in a sparse matrix format.

`qmc` type: lua table.

Required.

Further options that are common to all implemented QMC algorithms. Note that options relating to memory usage, excitation generation and real amplitudes are not implemented for `simple_fciqmc`. See *qmc options*.

`restart` type: lua table.

Optional.

Further options to control restarting the calculation from a previous calculation. See *restart options*.

`reference` type: lua table.

Optional.

Further options to select the reference state used. See *reference options*.

`qmc_state` type: qmc_state object.

Optional.

Output of a previous calculation to resume.

> **Warning:** The qmc_state object must have been returned by a previous simple FCIQMC calculation. The validity of this is not checked. The system must also be unchanged.

Common options

The following settings are common to multiple QMC algorithms. See the individual calculation documentation for *Full Configuration Interaction Quantum Monte Carlo*, *Coupled Cluster Monte Carlo* and *Density Matrix Quantum Monte Carlo* for details on how to perform the calculations as well as the documentation for each set of common options.

qmc options

The following options in the `qmc` table are common to the FCIQMC, CCMC and DMQMC algorithms and control the core settings in the algorithms.

`tau` type: float.

Required.

The timestep to use.

A small timestep causes the particles sampling the wavefunction/matrix to evolve very slowly. Too large a timestep, on the other hand, leads to a rapid particle growth which takes a long time to stabilise, even once the shift begins to vary, and coarse population dynamics.

`init_pop` type: float.

Required unless the calculations is initialised from a restart file or qmc_state.

Set the initial population on the reference determinant. For DMQMC calculations this option sets the number of psips which will be randomly distributed along the diagonal at the start of each beta loop.

`mc_cycles` type: integer.

Required.

Number of Monte Carlo cycles to perform per "report loop".

`nreports` type: integer.

Required.

Number of "report loops" to perform. Each report loop consists of `mc_cycles` cycles of the QMC algorithm followed by updating the shift (if appropriate) and output of information on the current state of the particle populations, including terms in the energy estimators.

`state_size` type: integer.

Required unless qmc_state is given.

Maximum number of states (i.e. determinants, excitors or density matrix elements) to store in the "main" list, which holds the number of particles on the state and related information such as the diagonal Hamiltonian matrix element. The number of elements that can be stored usually should be of the same order as the target population.

If negative, then the absolute value is used as the maximum amount of memory in MB to use for this information.

Ignored if qmc_state is given.

Note: This is a **per processor** quantity. It is usually safe to assume that each processor has approximately the same number of states.

`spawned_state_size` type: integer.

Required unless qmc_state is given.

Maximum number of states (i.e. determinants, excitors or density matrix elements) to store in the "spawned" list, i.e. the maximum number of states which can be spawned onto at a given timestep. The amount of memory required for this is usually a small fraction of that required for `state_size`, unless `real_amplitudes` is in use, in which case this should be a sizeable fraction (or potentially even greater than the memory for `state_size`, if load balancing of states across processors is poor). The amount of memory required is also dependent on the value of `tau`.

If negative, then the absolute value is used as the maximum amount of memory in MB to use for this information.

Ignored if qmc_state is given.

Note: This is a **per processor** quantity. It is recommended that a short trial calculation is run and the spawning rate for the desired timestep examined in order to estimate a reasonable value for `spawned_state_size`.

`rng_seed` type: integer.

Optional. Default: generate a seed from a hash of the time and calculation UUID.

The seed used to initialise the random number generator.

`target_population` type: float.

Optional. Default: none.

Set the target number of particles to be reached before the shift is allowed to vary. This is only checked at the end of each report loop. Once the `target_population` is reached, the shift is varied according to

$$S(t) = S(t - A\tau) - \frac{\xi}{A\tau} log\left(\frac{N_p(t)}{N_p(t - A\tau)}\right)$$

where S is the shift, t the current imaginary time, τ the timestep, A `mc_cycles`, ξ `shift_damping`, and N_p the number of particles.

`real_amplitudes` type: boolean.

Optional. Default: false.

Allow amplitudes to take non-integer weights. This will often significantly reduce the stochastic noise in the Monte Carlo estimates.

Automatically enabled if semi-stochastic is used.

Note: Real amplitudes are handled using fixed precision and so numbers which can not be exactly represented are stochastically rounded to values that can be stored.

The preprocessor option POP_SIZE=32 (default) uses 32-bit integers to store the amplitudes and stores amplitudes to within a precision/resolution of 2^{-11} and to a maximum absolute population of 2^{20}.

Consider using the preprocessor option POP_SIZE=64 to allow a greater range of amplitudes to be encoded (precision of 2^{-31} and maximum absolute population of 2^{32} at the cost of doubling the memory required to store the amplitudes.

By default uses integer weights, i.e. with the minimum resolution of 1.

`real_amplitude_force_32` type: boolean.

Optional. Default: false.

Force the precision of the real amplitudes to that used for POP_SIZE=32 irrespective of the actual POP_SIZE compile-time parameter.

Note: The main use-case for this is reproducing results produced by binaries compiled using POP_SIZE=32 with binaries compiled using POP_SIZE=64; it is not intended for use in production calculations.

`spawn_cutoff` type: float.

Optional. Default: 0.01 if `real_amplitudes` is used, 0 otherwise.

The minimum absolute value for the amplitude of a spawning event. If a spawning event with a smaller amplitude occurs then its amplitude will probabilistically be rounded up to the cutoff or down to zero in an unbiased manner. A spawning event with an amplitude above the cutoff is stochastically rounded such that it can be stored in a fixed precision value. If `real_amplitudes` is not in use, the fixed precision corresponds to unit values.

Only relevant when using `real_amplitudes`.

`excit_gen` type: string

Optional.

Possible values: 'renorm', 'no_renorm'.

System	Implemented	Default
chung_landau	renorm, no_renorm	renorm
heisenberg	renorm, no_renorm	renorm
hubbard_k	renorm, no_renorm	renorm
hubbard_real	renorm, no_renorm	renorm
ueg	no_renorm	no_renorm
ringium	no_renorm	no_renorm
read_in	renorm, no_renorm	renorm

The type of excitation generator to use. Note that not all types are implemented for all systems, usually because a specific type is not suitable for (large) production calculations or not feasible or useful.

The 'renorm' generator requires an orbitals to be selected such that a valid excitation is possible, e.g. for a double excitation $(i, j) \rightarrow (a, b)$, the combination i, j, a is only selected if there exists at least one unoccupied orbital for b which conserves any symmetry and spin quantum numbers. This is efficient in terms of generating allowed excitations but involves an expensive renormalisation step. The 'no_renorm' generator lifts this restriction at the cost of generating (and subsequently rejecting) such excitations; the excitation generation is consequently much faster. In general, 'renorm' is a good choice for small basis sets and 'no_renorm' is a good choice for large basis sets, especially with a small number of electrons (such that forbidden excitations are rarely generated).

`pattempt_single` type: float.

Optional. Default: use the fraction of symmetry-allowed excitations from the reference determinant that correspond to single excitations.

The probability of generating a single excitation.

`pattempt_double` type: float.

Optional. Default: use the fraction of symmetry-allowed excitations from the reference determinant that correspond to double excitations.

The probability of generating a double excitation.

`initial_shift` type: float.

Optional. Default: 0.0.

The initial value of the shift.

`shift_damping` type: float.

Optional. Default: 0.05.

The shift damping factor, ξ.

`vary_shift_from`

type: float or string.

Optional. Default: `initial_shift`.

Specify a value to set the shift to when `target_population` is reached. If the string 'proje' is specified then the instantaneous projected energy is used. By instantly setting the shift to a value closer to the correlation energy, the total population can be stabilised substantially faster.

There is no guarantee that the instantaneous projected energy is a good estimate of the ground state (particularly in the real-space formulation of the Hubbard model), but it is likely to be closer to it than the default shift value of 0.

`initiator` type: boolean.

Optional. Default: false.

Enable the initiator approximation (FCIQMC: *[Cleland10]*; CCMC: *[Spencer15]*; DMQMC: *[Malone16]*) in which spawned particles are only kept if they are created onto states which already have a non-zero population, or were produced by states which are already highly occupied (see `initiator_threshold`), or multiple spawning events onto a previously unoccupied state occurred in the same timestep.

Note: The initiator approximation should be considered experimental for CCMC and DMQMC (see `initiator_level` option for DMQMC).

Warning: The initiator approximation is non-variational (due to the non-variational energy estimator used) and the error should be carefully converged by performing repeated calculations with increasing `target_population` values.

`initiator_threshold` type: float.

Optional. Default: 3.0.

Set the (absolute) population above which a state is considered to be an initiator state. A value of 0 is equivalent to disabling the initiator approximation.

`tau_search` type: boolean.

Optional. Default: false. Not currently implemented in DMQMC.

Update the timestep, `tau`, automatically if by scaling it by 0.95 if a bloom event is detected. A bloom event is defined as one which spawns more than three particles in a single spawning event in FCIQMC and one which spawns more than 5% of the total current population in a single spawning event in CCMC.

Note: Experimental option. Feedback on required flexibility or alternative approaches is most welcome.

`use_mpi_barriers` type: boolean.

Optional. Default: false.

Perform MPI_Barrier calls before the main MPI communication calls (both for communication of the spawned list, and any semi-stochastic communication). These are timed, and the total time spent in these calls is reported at the end of a simulation. This is useful for assessing issues in load balancing, as it will allow you to see when certain processors take longer to perform their work than others. This is turned off by default because such calls may have an initialisation time which scales badly to many processors.

reference options

The `reference` table contains options used to control the Hilbert space used in the calculation and trial function for the projected estimator.

det type: vector of integers.

Optional. Default: a simple (but potentially not optimal) guess which satisfies the spin and, if provided, symmetry options using the Aufbau principle. In most cases the default (which for molecules typically corresponds to the Hartree–Fock determinant) is sufficient.

Specify the determinant (as a list of indices corresponding to occupied single-particle orbitals) to be used as the reference determinant, which is used in the trial function for calculating the projected energy estimator. Typically this should be the determinant expected to have the greatest overlap with the desired wavefunction.

hilbert_space_det type: vector of integers.

Optional. Default: set to `det`.

Specify the determinant (as a list of indices corresponding to occupied single-particle orbitals) used to generate the Hilbert space. Using different determinants to control the Hilbert space and the trial function allows, for example, spin-flip calculations to be performed.

Note: Only relevant if the Hilbert space is not equivalent to the FCI space, i.e. `ex_level` is smaller than the number of electrons in the system.

---

ex_level type: integer.

Optional. Default: set to the number of electrons in the system (i.e. consider all determinants in the FCI space).

Maximum excitation level to consider relative to the determinant given by `hilbert_space_det`.

restart options

The `restart` table contains options relating to checkpointing within QMC calculations.

HANDE currently uses one restart file per MPI rank with a filename of the form `HANDE.RS.X.pY.H5`, where `X` is the restart index and `Y` is the MPI rank.

read type: boolean or integer.

Optional. Default: false.

Start a QMC calculation from a previous calculation if `true` or an integer. If `true`, then the highest value of `X` is used for which a set of restart files exists, otherwise specifies the value of `X` to use.

Note: The calculation should be the same as the one that produced the ouput file, but it is possible to restart a calculation using an enlarged basis. The orbitals of the old (small) basis must correspond to the first orbitals of the new (larger) basis.

---

write type: boolean or integer.

Optional. Default: false.

Write out checkpointing files at the end of the calculation if `true` or an integer. If `true`, then the highest value of `X` is used for which a set of restart files doesn't exist, otherwise specifies the value of `X` to use.

write_shift type: boolean or integer.

Optional. Default: false.

Write out checkpointing files when the shift is allowed to vary (i.e. once `target_population` is reached) if `true` or an integer. If `true`, then the highest value of `X` is used for which a set of restart files doesn't exist, otherwise specifies the value of `X` to use.

write_frequency type: integer.

Optional. Default: $2^{31} - 1$.

Write out checkpointing files every *N* iterations, where *N* is the specified value.

Note: The index used for the restart files created with this option is the next unused index. Depending upon the frequency used, a large number of restart files may be created. As such, this option is typically only relevant for debugging or explicitly examining the evolution of the stochastic representation of the wavefunction.

---

semi_stoch options

The semi-stochastic approach *[Petruzielo12]*, *[Blunt15]* divides the Hilbert space into two regions: a small region in which the action of the Hamiltonian is applied exactly, and the remainder of the Hilbert space, in which the action is applied stochastically. This can substantially reduce the stochastic error in many cases.

space type: string.

Required.

Possible values: 'read', 'high', 'ci'.

The type of deterministic space to use. Using 'read' uses a deterministic space produced from a previous calculation and saved to file using the semi_stoch `write` option (the `write_determ_space` can be used but is now deprecated). Using 'high' sets the deterministic space to consist of the states with the highest population when the semi-stochastic projection is enabled. Using 'ci' sets the deterministic space to consist of a (small!) truncated configuration interaction space relative to a reference determinant.

size type: integer.

Required if `space` is 'high', otherwise ignored.

The number of states to include in the deterministic space.

ci_space type: reference table. See *reference options* for options.

Required if `space` is 'ci', otherwise ignored. Must contain at least `ex_level`. The reference determinant, if not supplied, is identical to that given in the calculation's `reference` option.

Defines the deterministic space to contain all determinants in a small (truncated) configuration interaction space.

start_iteration type: integer.

Optional. Default: 1.

The number of iterations to perform, during which the action of the Hamiltonian is applied entirely stochastically, before semi-stochastic projection is enabled. This allows for a period for the population to grow and the ground-state wavefunction to emerge before the deterministic space is selected if `space` is set to 'high'.

shift_start_iteration type: integer.

Optional. Default: None. Overrides `start_iteration`.

The number of iterations to perform after the shift is varied (i.e. after the `target_population` is reached) before the semi-stochastic projection is enabled.

separate_annihilation type: boolean.

Optional. Default: true.

If true, the deterministic amplitudes are communicated separately at the cost of an additional MPI call. If false, the annihilation of particles created from deterministic and stochastic projections are performed together,

which removes the need for an additional MPI call at the cost of communicating an additional $\mathcal{O}(N_p N_D)$ more amplitudes, where N_p is the number of processors and N_D the size of the deterministic space. If the deterministic space is small and communication latency high, setting `separate_annihilation` to false might improve performance. For most systems and computer architectures, the default value is faster.

`write` type: boolean or integer.

Optional. Default: false.

Write out the deterministic space to file of form `SEMI.STOCH.X.H5`, where `X` is the file id. If set to true, `X` will be the smallest non-negative id such that `SEMI.STOCH.X.H5` does not already exist, otherwise the value provided is used as the file id.

`read` type: integer.

Optional. Default: largest value of `X` such that the file `SEMI.STOCH.X.H5` exists.

Index of the file containing the deterministic space produced from a previous calculation.

1.7.3 Utilities

Utilities

Redistribution of restart files

```
redistribute {
    -- options
}
```

For speed in reading in restart files and for simplicity, HANDE produces restart files specific to the number of MPI ranks used in the calculation and hence by default calculations can only be restarted on the same number of MPI ranks the original calculation ran on. The `redistribute` function reads in a set of restart files and produces a new set to be used on a different number of processors.

Note:

- It is convenient to place this before the QMC calculation call in the input file. However, the process of redistributing particles is a somewhat serial task and hence `redistribute` may not scale well to large numbers of processors. Hence it may be more computationally efficient to do the redistribution targeting a large (ie 100s or 1000s) of processors using a much smaller number of processors in a separate run of HANDE.
- Load balancing settings are reset to their default values.

HANDE uses one restart file per MPI rank with a filename of the form `HANDE.RS.X.pY.H5`, where `X` is the restart index and `Y` is the MPI rank.

Options:

`nprocs` type: integer.

Optional. Default: number of processors the calculation is running on.

Set the number of processors that the new set of restart files are to be used on.

`read` type: integer.

Optional. Default: highest non-negative integer for which a set of restart files exists.

Set the index, `X` of the set of restart files to be read in.

write type: integer.

Optional. Default: highest non-negative integer for which a set of restart files does not yet exist.

Set the index, X of the set of restart files to be written out.

sys type: system object.

Optional.

Only used to determine the number of basis functions, if changing the value of DET_SIZE for the restart files.

Warning: Each processor must be able to access the entire set of existing restart files, which are assumed to be in the working directory.

MPI information

```
mpi_root()
```

Returns: true if the processor is the MPI root processor and false otherwise.

The input file is processed and run by each processor. It is occasionally useful to perform (for example) additional I/O from lua but only on one processor. Testing if the processor is the MPI root processor is a safe way to do this, e.g.

```
if mpi_root() then
    print('root says hello from lua!')
end
```

Memory management

Objects returned from functions (e.g. system and qmc_state objects) are deallocated by Lua's garbage collector when they are no longer required. This can either be because the variable goes out of scope or is set to `nil`. This level of memory management is sufficient in most calculations. However, there may be a substantial memory overhead when running multiple separate calculations in the same input file as the garbage collection need not take place immediately. As such, objects which are no longer required can be explicitly freed using `free` methods on all objects returned by HANDE's functions. For example, for `qmc_state` objects:

```
system = hubbard_k {
    lattice = { { 10 } },
    electrons = 6,
    ms = 0,
    sym = 1,
    U = 1,
}

qs1 = fciqmc {
    sys = system,
    qmc = {
        tau = 0.01,
        init_pop = 10,
        mc_cycles = 20,
        nreports = 100,
        target_population = 50000,
        state_size = 5000,
        spawned_state_size = 500,
    },
}
```

```
-- Deallocate all memory associated with qs1 produced by the first FCIQMC calculation.
qs1:free()

qs2 = fciqmc {
    sys = system,
    qmc = {
        tau = 0.02,
        init_pop = 10,
        mc_cycles = 10,
        nreports = 100,
        target_population = 50000,
        state_size = 5000,
        spawned_state_size = 500,
    },
}
```

and similarly for `system` objects.

Write HDF5 system file

```
write_read_in_system {
    sys = system,
    filename = filename,
}
```

Options:

`sys` type: system object.

Required.

The system on which to perform the calculation. Must be created via the read_in function.

`filename` type: string. Optional.

Filename to dump system hdf5 file to. If unset will generate a filename to dump to based on the template: int_file + CAS_information + .H5, where `int_file` and the CAS information are set in the call to `read_in` which create the `system` object.

Returns:

type: string.

name of HDF5 file created. This is currently only available on the root processor and can be passed into subsequent calls to `read_in` safely as only the root processor reads from integral and system files.

When running a calculation using a system generated from a FCIDUMP, the `system` object created by `read_in` can be dumped in HDF5 format for reuse in subsequent calculations; this speeds initialisation by a factor of ~100x and reduces the required file size by ~16x for large FCIDUMPs. When running in parallel on a large number of cores this is particularly important to utilise as it overcomes an inherent serialisation point in the calculation initialisation.

For example:

```
sys = read_in {
    int_file = "FCIDUMP",
    nel = 24,
    ms = 0,
    sym = 0,
}
```

```
hdf5_name = write_read_in_system {
    sys = sys,
}
```

produces an HDF5 file entitled "FCIDUMP.H5" and return this value to the variable `hdf5_name`. Passing this as the argument to `int_file` within `read_in` will use it in future calculations – the HDF5 format of the file is automatically detected.

If a CAS is used to produce the system object used to produce such a file it will be labelled as such and only information for basis functions within the CAS will be stored; conversion between different CAS within this functionality is not currently supported.

Important: When using a HDF5 file to initialise a system either both of nel and ms must be specified or neither; if neither are specified the values stored within the system HDF5 file will be used and otherwise the given values override those stored.

1.7.4 Appendix

A short introduction to lua

Lua is a lightweight programming language which is easy to embed and is well-suited to the task of controlling a simulation. For a quick introduction to lua, please read Learn Lua in 15 Minutes. However, for most cases the input file format can be treated as follows:

Assignment is performed by setting a variable name equal to an object, e.g.

```
pi = 3.141592654
```

Strings are created by enclosing characters in quotation marks:

```
msg = 'hello world'
```

and boolean variables can be set using the `true` and `false` keywords:

```
yes = true
yes = false
```

A key data structure in lua is the *table*, which serves both as an array and an associative array or map, and is denoted using braces. First, the following creates a table to hold a 1D vector:

```
v = { 1, 2, 3 }
```

whilst using key=value pairs creates a table as an associative array:

```
v = { x = 3, y = 4, type = 'dual' }
```

Tables can be nested.

Functions are called using:

```
x = fname(arg1, arg2, ...)
```

where *fname* is the name of the function, which returns a single value (which is stored in *x* in the above example). Keywords can be passed in by using a table. If the function takes a single table, then the parentheses need not be included, such that the following calls are identical:

```
x = fname1({ x = 3, y = 4, type = 'dual'})
x = fname1{ x = 3, y = 4, type = 'dual'}
```

All options are passed into HANDE by using a table as an associative array. Each function exposed by HANDE to the lua script takes a single (nested) table.

Lua handles multiple return values from functions in a convenient manner. If a function call returns values that are then not set to a variable, the additional values are discarded. If a function call returns fewer values than are the variables set to hold the results of the function call, the additional variables are set to `nil`. See (e.g.) http://www.lua.org/pil/5.1.html for more details.

Warning: lua, and by extension the HANDE input file, is case sensitive.

1.8 Interacting with running calculations

It is possible to interact with running calculations.

After each report loop, HANDE checks for the existence of the file HANDE.COMM in the current working directory for all processors. If HANDE.COMM exists, then the file is read and any modified parameters are then used for the rest of the calculation. HANDE.COMM is deleted after it is read in to prevent it from being detected on subsequent report loops and to enable multiple interactions with a running calculation.

HANDE.COMM is a lua script, in a similar fashion to the input file, but has a much more restricted range of options. Options which can be set or modified are:

`softexit` type: boolean.

End the calculation immediately but still perform any post-processing (e.g. dumping out a restart file). This is useful for cleanly terminating a converged calculation or cleanly stopping a calculation before the walltime is reached to allow it to be restarted.

The send_softexit.py script in the tools subdirectory is useful for running HANDE on a queueing system as it writes **softexit = true** to HANDE.COMM a certain amount of time before the walltime is reached.

`tau` type: float.

Change the timestep to be used.

`target_population` type: integer.

Change the number of particles to be reached before the calculation starts varying the shift. Meaningless if the calculation has already started varying the shift. If smaller than the current population (or negative) then the shift is immediately allowed to vary.

`shift` type: float or 1D vector of floats.

Adjust the current value of the shift. If the calculation has already entered variable shift mode then the shift will still be updated every report cycle, otherwise this is equivalent to changing the **initial_shift** value.

Passing a single value such as:

```
shift = -1
```

sets the shift in **all** spaces to the specified value. Different spaces can be modified separately by passing in a vector. For example:

```
shift = { -1, -2 }
```

sets the shift in the first space to -1, in the second space to -2 and leaves it unmodified in all other spaces.

1.9 Analysis

The following provides a brief overview for the most common analysis required for each type of Monte Carlo calculation. The guides in *Tutorials* provide a step-by-step guide to analysing HANDE calculations and explain the reasoning behind the required analysis parameters.

HANDE includes a variety of scripts and utilities in the `tools` subdirectory. However, these only provide a simple, command-line interface. A comprehensive python module, *pyhande*, drives all the analysis. *pyhande* is extremely powerful for dealing with complex analysis, data-driven investigation or bulk data analysis.

1.9.1 FCIQMC and CCMC

QMC calculations print out data from a block of iterations (a 'report loop'), the length of which is controlled by the **mc_cycles** input option. Care should be taken analysing this data and, in particular, producing accurate estimates of the errors in the means of the energy estimators. Almost all data is averaged over the report loop (see output for further details).

Note that no data is lost when quantities are summed over report loops, as the correlation length in the data is substantially longer than the length of the report loop (typically 10-20 iterations).

As the particle distribution at one iteration is not independent from the distribution at the previous iteration, estimators at each iteration are not independent. This correlation in the data needs to be taken into account when estimating standard errors. A simple and effective way of doing this is to use a blocking analysis *[Flyvbjerg89]*.

The `reblock_hande.py` script (in the `tools` subdirectory) does this. Run

```
$ reblock_hande.py --help
```

to see the available options. Estimates for the shift and projected energy are typically obtained using

```
$ reblock_hande.py --start N out
```

respectively, where `N` is the iteration from which data should be blocked (i.e. after the calculation has equilibrated) and `out` is the file to which the calculation output was saved.

Note that reblock_hande.py can accept multiple output files for the case when a calculation is restarted. More complicated analysis can be performed in python by using the `pyhande` library — `reblock_hande.py` simply provides a convenient interface for the most common analysis tasks.

1.9.2 Canonical Total Energy MC

The configurations and resulting estimates in a canonical total energy calculation are statistically independent and therefore no blocking analysis is required. The `analyse_canonical.py` script is available in `tools/dmqmc/` which performs the appropriate averaging and standard error analysis on the output file using the pyhande suite.

1.9.3 DMQMC

No blocking analysis is required for the error analysis of DMQMC calculations as estimates are averaged over statistically independent runs. The `finite_temp_analysis.py` script in `tools/dmqmc` can be used to perform a standard error analysis of the Monte Carlo data for a number of different observables.

1.10 Generating integrals

HANDE can treat *systems* other than model Hamiltonians by reading in the necessary integrals in the FCIDUMP format *[Knowles89]*. Many quantum chemistry packages can generate them following Hartree-Fock calculations, including:

HORTON https://theochem.github.io/horton/

MOLPRO https://www.molpro.net/

PSI4 http://psicode.org. Requires the fcidump plugin (https://github.com/hande-qmc/fcidump).

Q-Chem http://www.q-chem.com. FCIDUMP code contributed by Alex Thom.

We most frequently use PSI4 and Q-Chem and so these tend to be better tested. Note that the computational cost of the calculations in HANDE vastly outweighs the cost of the underlying SCF calculations and so the efficiency of the code used to generate the integrals is usually not a key factor. Please consult the documentation of the code of interest regarding how to run SCF calculations and generate the integrals in the FCIDUMP format.

The format of the FCIDUMP file expected by HANDE is documented in the comments to `src/read_in.F90`; these might be useful for users wishing to generate integrals from upon a specific Hamiltonian to feed into HANDE.

Warning: The single-particle basis is assumed to be orthonormal.

1.11 Tips

Some suggestions from the HANDE developers for using HANDE...heed our words!

1.11.1 Compilation

For optimised versions of HANDE, explore using:

- compiler-specific optimisation flags

  In general adding 'high-level' optimisation flags (`-O3`, `-Ofast`, etc.) makes a substantial impact on the calculation speed.

- interprocedural optimisation

  Many compilers can perform interprocedural optimisation, whereby optimisations are performed at link-time instead of compile-time. This allows optimisations to be performed (including inlining) on procedures specified in different source files. On some compilers (e.g. GCC, Intel) this can have a substantial benefit; on other compilers the difference is less marked.

- popcnt instruction

  If the processor being used includes it, uses the popcnt instruction rather than a software implementation to count bits set in an integer. This can have a impact of the order of a few percent for the entire calculation.

- DET_SIZE=64

  Use 64-bit integers rather than 32-bit integers to store the representation of the determinant/excitor/tensor labels. This can make certain calculations quicker (i.e. those involving more than 32 single-particle basis functions) by reducing the amount of bit operations that need to be performed.

1.11.2 Plotting calculation output using gnuplot

The first section of the output file contains information about the basis functions used in the calculations. This gives spurious data points when the contents of the file is plotted using gnuplot. They can be removed by creating an executable file called `gphande` in the user's $PATH, containing:

```
#!/bin/sed -nf
1,/iterations/d
/^ *[0-9]/p
```

When plotting in gnuplot, using the command

```
plot '<gphande file'
```

instead of

```
plot 'file'
```

will then remove the extra points.

1.12 Old (removed) functionality

Unused and **not useful** functionality is occasionally removed from HANDE, in order to remove the maintenance burden for code that really has no benefit. In general, keeping failed experiments in the codebase is not helpful to developers (more work) and users (not obvious if an option should or should not be used). When it transpires that something falls into the category, we may hence remove it and detail it below. If you are interested in resurrecting this functionality, please dig through the git history and/or speak to a developer.

folded-spectrum FCIQMC The folded-spectrum approach allows, in principle, access to excited states in FCIQMC via using the Hamiltonian $(H - \epsilon)^2$, where ϵ is an energy offset. It emerged in practice to be very painful/impossible to converge to excited states for systems beyond the reach of conventional FCI.

defining an initiator determinant via a complete active space Originally the initiator space was defined by a population threshold and a complete active space (CAS). It turns out that it is simpler to allow the initiator space to emerge naturally just through the population threshold (as used in later studies), whereas defining a CAS that is small but effective is not easy in large systems. Furthermore, using just a population threshold makes the initiator approximation easier to extend to other algorithms (i.e. CCMC and DMQMC).

1.13 Tutorials

The tutorials below demonstrate how to set up and run Monte Carlo calculations in HANDE. The input files in the test suite also demonstrate how calculations can be performed. The aim here is to provide an introduction to setting up, running and analysing calculations and only basic input options are considered; for advanced options please consult the appropriate section of the manual.

The tutorials assume that HANDE has been successfully compiled and the test suite has been sucessfully run. Any reference to `hande.x` should be replaced with the full path to the HANDE executable and similarly for the `reblock_hande.py`, `finite_temperature_analysis.py` and `analyse_canonical.py` scripts.

Note: The exact command to launch HANDE with MPI depends upon the exact configuration of MPI. The command may be different (e.g. `mpirun` instead of `mpiexec`) and might require the number of processors to be passed as an argument. The tutorials show the exact command we used, which varies depending upon the machine used to run the

tutorials. We regularly use the OpenMPI implementation (`mpiexec -np <# processors>` or `mpirun -np <# processors>`), Intel MPI (`mpiexec`) and Cray-MPICH (`aprun`).

The input and output files from the calculations performed in the tutorials can be found under the `documentation/manual/tutorials/calcs/` directory. The example calculations are deliberately not trivial and may require up to a few hundred core hours to run as shown. Smaller calculations can be performed by reducing the system size (e.g. using fewer electrons or orbitals) or running for fewer iterations.

Note: None of the tutorials fix a random number seed (as this is the best approach for running multiple production calculations on the same system) so results will not be exactly identical (but should agree statistically) from those in the above dataset unless the same seeds (which can be found in the output files) are used.

We recommend working through the FCIQMC tutorial before the iFCIQMC, CCMC or DMQMC tutorials.

1.13.1 Full Configuration Interaction Quantum Monte Carlo

In this tutorial we will run FCIQMC on the 18-site 2D Hubbard model at half filling with $U/t = 1.3$. The input and output files can be found under the `documentation/manual/tutorials/calcs/fciqmc` subdirectory of the source distribution. Knowledge of the terminology and theory given in *[Booth09]* and *[Spencer12]* is assumed.

First, we will set up the system and estimate the number of determinants in Hilbert space with the desired symmetry using a Monte Carlo approach.

We are interested in the state with zero crystal momentum, as there is theoretical work showing this will be the symmetry of the overall ground state. HANDE uses an indexing scheme for the symmetry label. The easiest way to find this out is to run an input file which only contains the system definition:

```
hubbard = hubbard_k {
    lattice = {
        { 3,  3 },
        { 3, -3 },
    },
    electrons = 18,
    ms = 0,
    U = 1.3,
    t = 1,
}
```

This file can be run using:

```
$ hande.x hubbard_sym.lua > hubbard_sym.out
```

The output file, `hubbard_sym.out`, contains a symmetry table which informs us that the wavevector $(0, 0)$ corresponds to the index 1; this value should be specified in subsequent calculations to ensure that the calculation is performed in the desired symmetry subspace.

It is useful to know the size of the FCI Hilbert space, i.e. the number of Slater determinants that can be formed from the single-particle basis given the number of electrons and total spin. Whilst the full space can be determined from simple combinatorics, the size of the subspace containing only determinants of the desired symmetry is less straightforward and it is the latter number that is of interest as it only includes determinants that are connected via non-zero Hamiltonian matrix elements and hence can be accessed in a Monte Carlo calculation. A fast way to determine the size of the accessible subspace is to use Monte Carlo sampling *[Booth10]* with an input file containing:

```
hubbard = hubbard_k {
    lattice = {
```

```
        { 3,  3 },
        { 3, -3 },
    },
    electrons = 18,
    ms = 0,
    U = 1.3,
    t = 1,
    sym = 1,
}

hilbert_space {
    sys = hubbard,
    hilbert = {
        nattempts = 100000,
        ncycles = 30,
    }
}
```

The Monte Carlo algorithm produces `nattempts` random determinants per cycle, from which it estimates the size of the Hilbert space. The independent cycles are used to provide an estimate of the mean and standard error of the data; the running estimates of these are printed every cycle and the final estimate at the end.

This calculation can be run in a similar fashion to before:

```
$ hande.x hubbard_hilbert.lua > hubbard_hilbert.out
```

Inspecting the `output`, we find that the Hilbert space contains 1.3×10^8 determinants with the desired symmetry.

FCIQMC requires a critical population to be exceeded in order to converge to the correct answer. This system-specific population is determined by the plateau. A calculation initially uses a constant energy offset ('shift') and a small starting population and hence the population grows exponentially. A plateau in the population growth spontaneously appears, during which the correct sign structure of the ground state wavefunction emerges. The plateau is equally spontaneously exited and the population grows at an exponential rate (albeit slower than the initial growth).

The simplest way to find the plateau is to run an FCIQMC calculation with a small initial population and allow the population to grow until a large size; this can be accomplished by setting `target_population`, which is the population at which the shift is allowed to vary, to a large value (i.e. effectively infinite) such that the plateau should occur before it. This is done using an input file like [1]:

```
hubbard = hubbard_k {
    lattice = {
        { 3,  3 },
        { 3, -3 },
    },
    electrons = 18,
    ms = 0,
    U = 1.3,
    t = 1,
    sym = 1,
}

fciqmc {
    sys = hubbard,
    qmc = {
        tau = 0.002,
        mc_cycles = 20,
        nreports = 500,
```

[1] With some scripting it is possible to automatically detect the plateau and interact with the calculation at this point.

```
        init_pop = 100,
        target_population = 10^10,
        state_size = -1000,
        spawned_state_size = -100,
    },
}
```

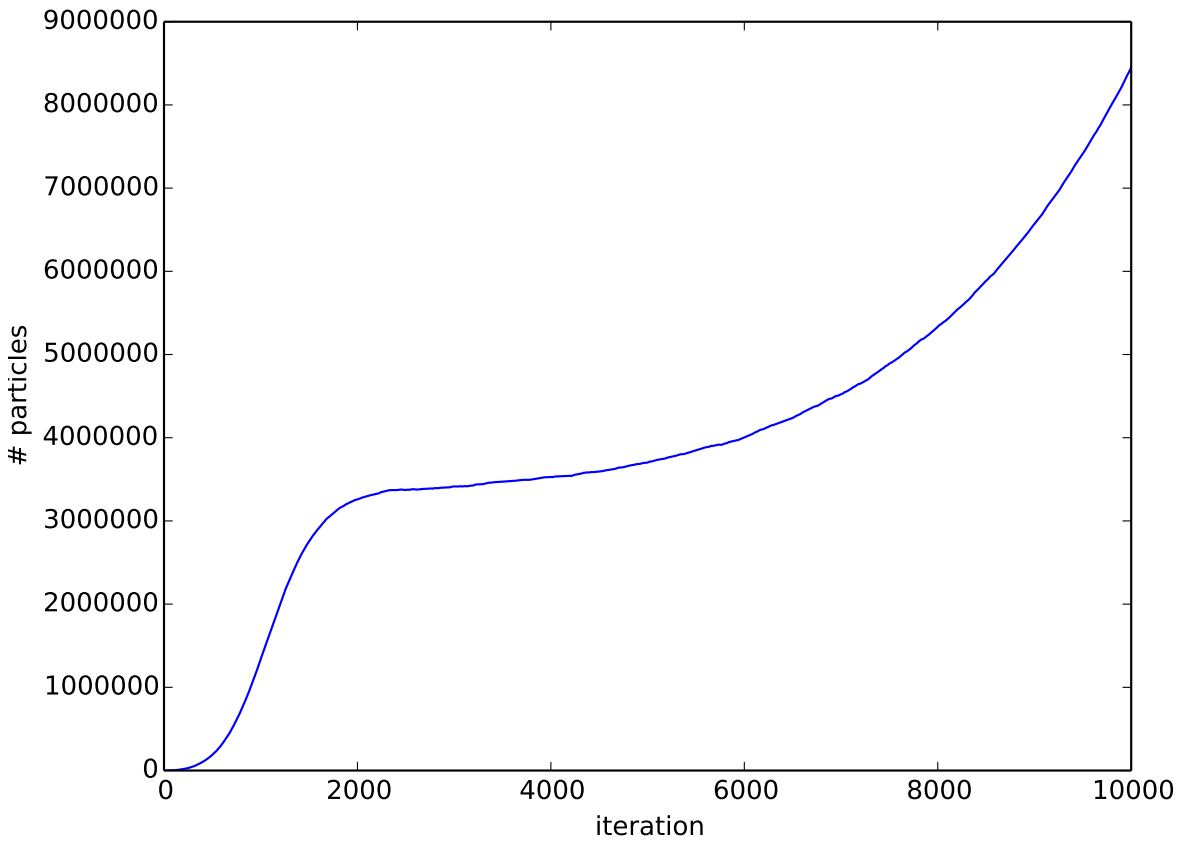
As the input file is a lua script, we can use lua expressions (e.g. `10^10` for 1×10^{10}) at any point.

The choice of timestep is beyond the scope of a simple tutorial; broadly it is chosen such that the population is stable and there are no 'blooms' (spawning events which create a large number of particles). HANDE will print out a warning and a summary at the end of the calculation if blooms occur. The other key values are how many iterations to run for and the amount of memory to use for the main and spawned particle data objects. These were chosen such that enough states could be stored and the plateau occurs within the iterations used. Choosing these for a new system typically requires some trial and error. Given the large population, we will run this calculation in parallel using MPI:

```
$ mpiexec hande.x hubbard_plateau.lua > hubbard_plateau.out
```

The parallel scaling of HANDE depends upon the system being studied and quality of the hardware being used. Typically using a minimum population per core of $\sim 10^5$ (assuming perfect load balancing, which can rarely be achieved) results in an acceptable performance.

The `output file` is (hopefully!) fairly intuitive. The QMC output table contains one entry per 'report loop' (a set of Monte Carlo cycles). *pyhande* can be used to extract this information so that the population growth can be easily plotted:



We hence see that the plateau occurs at around 3.5×10^6 ($\sim 2.8\%$ of the entire Hilbert space) and hence FCIQMC is

very successful for this system.

Note: In some cases the plateau may not be present (e.g. in sign-problem free systems) or not easily visible (e.g. in systems with a small Hilbert space) or appear as a shoulder (common in CCMC calculations). *pyhande* contains two algorithms for determining the plateau, which are helpful in such cases or for automatically analysing large numbers of calculations.

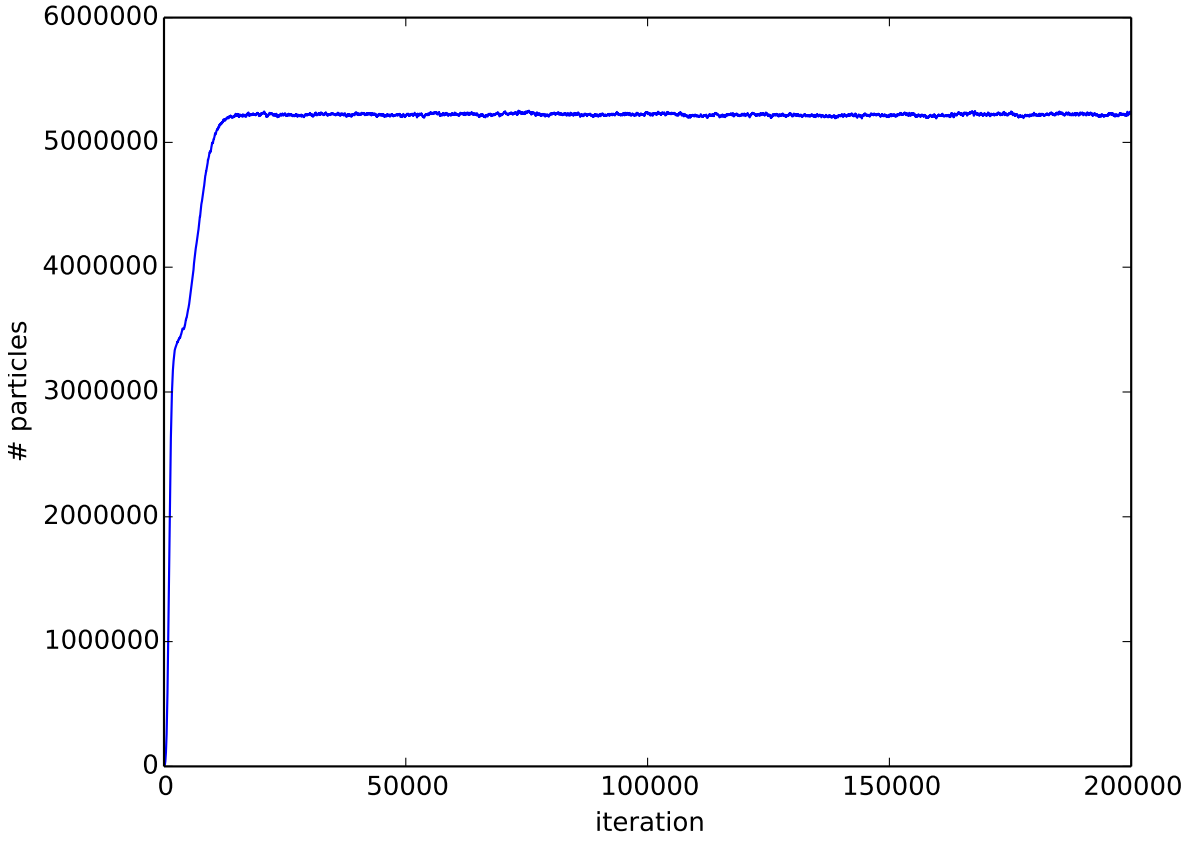
We can now run a production calculation to find the ground state energy of this system. To do so, we make two changes to the input used to find the plateau: `target_population` is set to a value above the plateau (but not so large that the computational cost is overwhelming) and the simulation is run for more iterations, i.e.:

```
hubbard = hubbard_k {
    lattice = {
        { 3,  3 },
        { 3, -3 },
    },
    electrons = 18,
    ms = 0,
    U = 1.3,
    t = 1,
    sym = 1,
}

fciqmc {
    sys = hubbard,
    qmc = {
        tau = 0.002,
        mc_cycles = 20,
        nreports = 10000,
        init_pop = 100,
        target_population = 4*10^6,
        state_size = -1000,
        spawned_state_size = -100,
    },
}
```

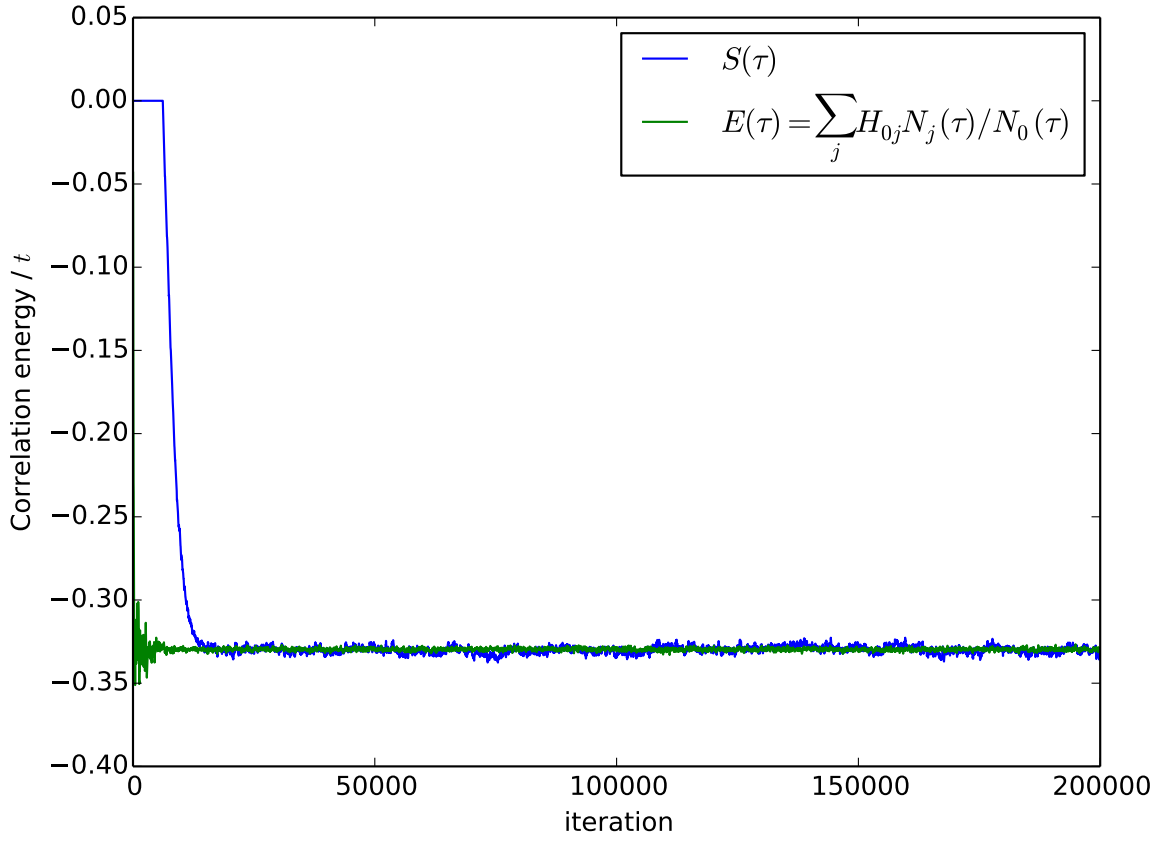
and can again be run using:

```
$ mpiexec hande.x hubbard_fciqmc_real.lua > hubbard_fciqmc.out
```

This time, the population starts to be controlled after it reaches the desired `target_population`:



Note that it takes some time for the population to stabilise as the shift gradually decays towards the ground state correlation energy. Once the population is stable, both the shift and the **instantaneous** projected energy vary about a fixed value, namely the ground state energy:



Care must be taken in evaluating the mean and standard error of these quantities, however. The state of a simulation at one iteration depends heavily upon the state at the previous iteration and hence each data point is not independent. Further, in the case of the projected energy estimator, the correlation between the numerator and denominator must be taken into account. The former issue is dealt with using a blocking analysis *[Flyvbjerg89]*; the latter by taking the covariance into account. Both of these are implemented in *pyhande* and the `reblock_hande.py` script provides a convenient command line interface to this functionality. See *Analysis* for more information. The above graphs show that the popualation, shift and instantaneous projected energy estimator have all stabilised by iteration 30000, so we will accumulate statistics from that point onwards. `reblock_hande.py` can produce a lot of useful output but for now we'll only concern ourselves with the best guess of the standard error *[Lee11]*, hence the use of the `--quiet` flag:

```
$ reblock_hande.py --quiet --start 30000 hubbard_fciqmc.out
```

which gives

```
Recommended statistics from optimal block size:

                                # H psips \sum H_0j N_j        N_0        Shift Proj. Energy
fciqmc/hubbard_fciqmc.out   5222000(1000)      -7791(7)  23630(20)  -0.3299(3)  -0.32969(2)
```

The stochastic error can be reduced by running with more particles and/or running for longer. Another very effective method is to allow determinants to have fractional numbers of particles on determinants rather than just using a strictly integer representation of the wavefunction. This is done using the `real_amplitudes` keyword:

```
hubbard = hubbard_k {
    lattice = {
        { 3,  3 },
        { 3, -3 },
    },
    electrons = 18,
    ms = 0,
    U = 1.3,
    t = 1,
    sym = 1,
}

fciqmc {
    sys = hubbard,
    qmc = {
        tau = 0.002,
        mc_cycles = 20,
        nreports = 10000,
        init_pop = 100,
        target_population = 4*10^6,
        state_size = -1000,
        spawned_state_size = -100,
        real_amplitudes = true,
    },
}
```

The calculation can be run and analysed in the same manner:

```
$ mpiexec hande.x hubbard_fciqmc_real.lua > hubbard_fciqmc_real.out
$ reblock_hande.py --quiet --start 30000 hubbard_fciqmc_real.out
```

which results in:

```
Recommended statistics from optimal block size:

                                  # H psips \sum H_0j N_j       N_0        Shift  Proj. Energy
fciqmc/hubbard_fciqmc_real.out  5230600(100)     -15460(2)  46890(6)  -0.32976(3)  -0.329694(4)
```

Whilst using real amplitudes is substantially slower, the reduction in stochastic error more than compensates; it is much more efficient than simply running for longer. Real ampltiudes also reduce the plateau height in some cases (as is the case here) though this has not been investigated carefully in a wide variety of systems.

One reason that the calculation with real ampltiudes took so much longer than that with integer ampltiudes is due to the nature of the Hubbard model: all non-zero off-diagonal Hamiltonian matrix elements are identical in magnitude. Carefully inspecting the output in `hubbard_fciqmc_real.out` reveals that there is almost one spawning event for every particle [2]. This results in a costly communication overhead every timestep. We can improve this by changing the `spawn_cutoff` parameter, which is the minimum absolute value of a spawning event *[Overy2014]*. A spawning event with a smaller cutoff is probabilistically rounded to zero or the cutoff value [3]. The default cutoff value, 0.01, need only be changed in cases such as this and is set using the `spawn_cutoff` parameter in the `qmc` table:

```
hubbard = hubbard_k {
    lattice = {
        { 3,  3 },
        { 3, -3 },
```

[2] This can be confirmed analytically using knowledge of the internal excitation generators and the associated probabilities, the value of U/t and the calculation timestep.

[3] One can hence view the integer amplitudes algorithm, ignoring death and spawning events which produce multiple particles, as having a `spawn_cutoff` of 1.

```
    },
    electrons = 18,
    ms = 0,
    U = 1.3,
    t = 1,
    sym = 1,
}

fciqmc {
    sys = hubbard,
    qmc = {
        tau = 0.002,
        mc_cycles = 20,
        nreports = 10000,
        init_pop = 100,
        target_population = 4*10^6,
        state_size = -1000,
        spawned_state_size = -100,
        real_amplitudes = true,
        spawn_cutoff = 0.1,
    },
}
```

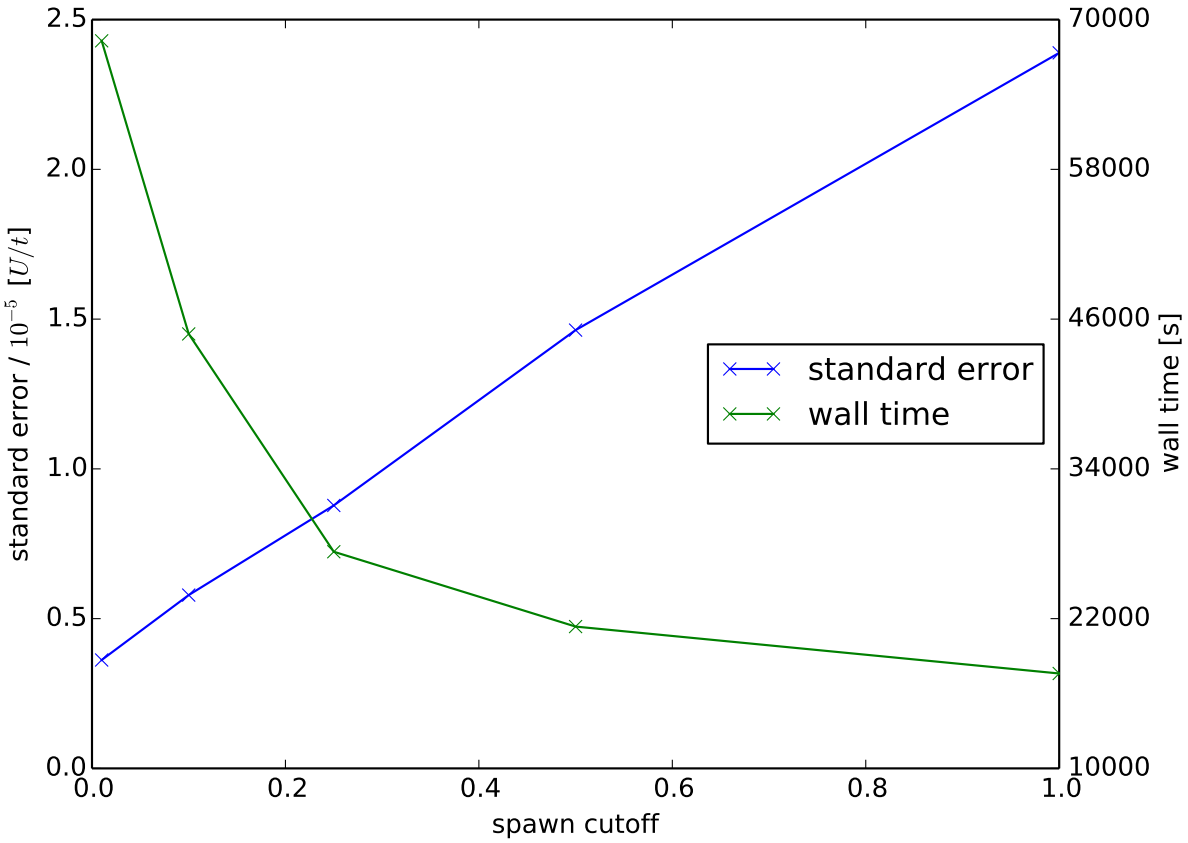
Note that a value of 1 is comparable to using integer ampltiudes except for the death step, which acts without stochastic rounding if `real_amplitudes` is enabled.

We can run calculations with different values of `spawn_cutoff` as before; here we set use values of 0.1, 0.25 and 0.5. `reblock_hande.py` can analyse multiple calculations at once and so we can easily see the impact of changing `spawn_cutoff` compared to the default value and the original FCIQMC calculation using integer ampltiudes:

```
$ reblock_hande.py --quiet --start 30000 hubbard_fciqmc*out
```

```
Recommended statistics from optimal block size:

                                          # H psips \sum H_0j N_j        N_0         Shift  Proj. Ene
fciqmc/hubbard_fciqmc.out               5222000(1000)      -7791(7)  23630(20)   -0.3299(3)   -0.32969
fciqmc/hubbard_fciqmc_real.out          5230600(100)      -15460(2)   46890(6)  -0.32976(3)  -0.329694
fciqmc/hubbard_fciqmc_real_sc0.1.out    5225400(200)      -13172(1)   39952(4)  -0.32968(5)  -0.329704
fciqmc/hubbard_fciqmc_real_sc0.25.out   5219300(400)      -10627(4)  32230(10)   -0.3298(1)  -0.329696
fciqmc/hubbard_fciqmc_real_sc0.5.out    5220500(600)       -9595(4)  29100(10)   -0.3296(2)   -0.32970
```

As expected, increasing the `spawn_cutoff` results in an increase in the stochastic error (linear, in this case, due to the identical magnitude of non-zero off-diagonal Hamiltonian matrix elements). Finally, we can compare the change in stochastic error to the wall time of the calculation:



For convenience, the integer amplitude calculation is shown as having a `spawn_cutoff` of 1. Clearly there is a playoff between the computational cost and the desired stochastic error; choosing a value of 0.25 for `spawn_cutoff` in this case seems sensible as it is around the point where the rate of change in the wall time begins to slow [4].

1.13.2 Initiator Approximation to FCIQMC

We shall again calculate the ground state energy of the 18-site 2D Hubbard model at half-filling and with $U/t = 1.3$, as in the *FCIQMC tutorial*. The initiator approximation *[Cleland10]* greatly reduces the number of particles required to sample the wavefunction. The drawback, however, is that the approximation must be carefully controlled to obtain an accurate estimate of the FCI energy by running multiple calculations with increasing populations.

It is efficient (both computationally and in terms of elapsed time) to treat each calculation separately. For compactness, we shall simply run multiple calculations with different `target_population` values one after the other in the same HANDE calculation. This is trivial to do by using a lua loop as `fciqmc` is simply a function call:

```
hubbard = hubbard_k {
    lattice = {
        { 3,  3 },
        { 3, -3 },
    },
    electrons = 18,
    ms = 0,
    U = 1.3,
```

[4] A more comprehensive approach to assessing the efficiency of the calculations can be found in *[Vigor16]*.

```
    t = 1,
    sym = 1,
}

targets = {2.5*10^3, 5*10^3, 7.5*10^3, 10^4, 2.5*10^4, 5*10^4, 1*10^5, 2.5*10^5, 5*10^5, 1*10^6}
for i,target in ipairs(targets) do
    qmc_state = fciqmc {
        sys = hubbard,
        qmc = {
            tau = 0.002,
            mc_cycles = 20,
            nreports = 10000,
            init_pop = 100,
            target_population = target,
            state_size = -1000,
            spawned_state_size = -100,
            initiator = true,
        },
    }
    -- For memory efficiency, explicitly free qmc_state after each calculation.
    qmc_state:free()
end
```

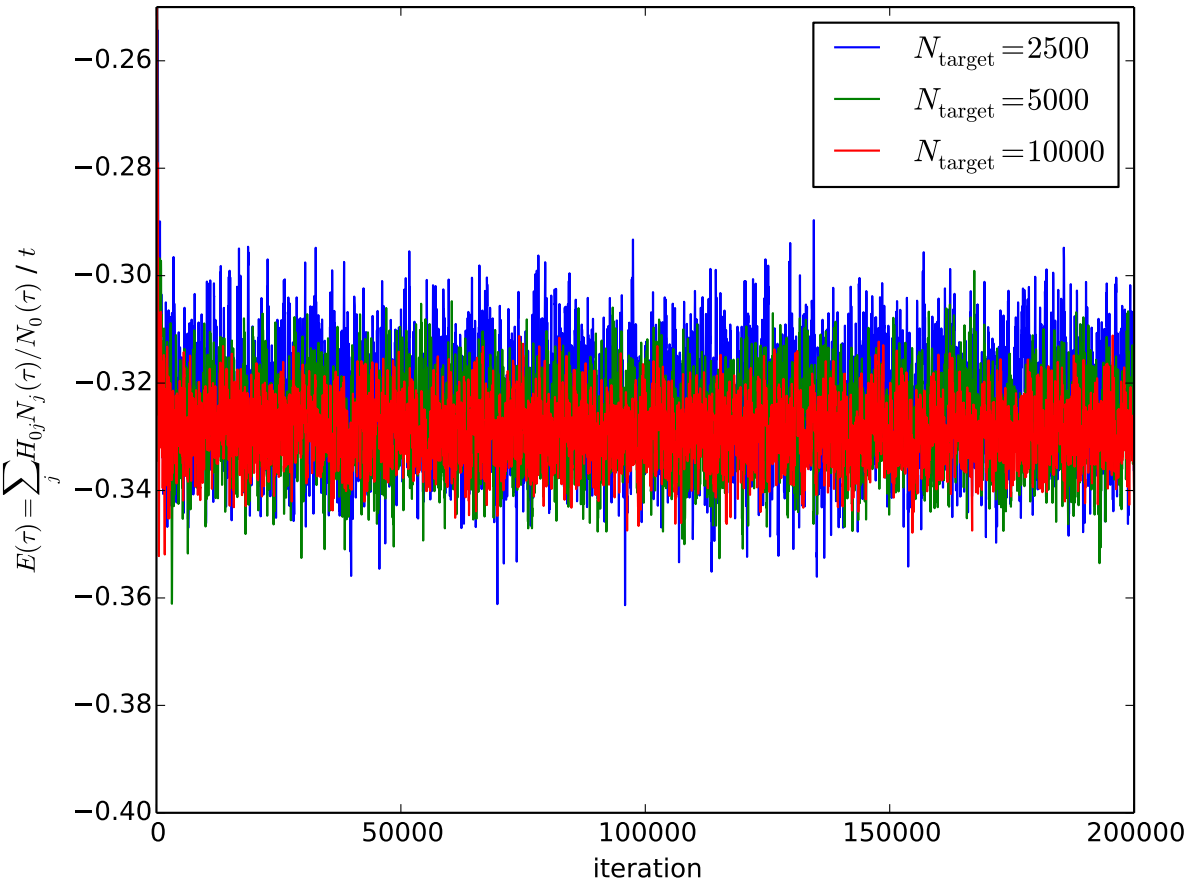
The only difference between the above input and an FCIQMC calculation is the setting `initiator = true`. As in the examples in the *FCIQMC tutorial*, this can be run using:

```
$ mpiexec hande.x hubbard_ifciqmc.lua >  hubbard_ifciqmc.out
```

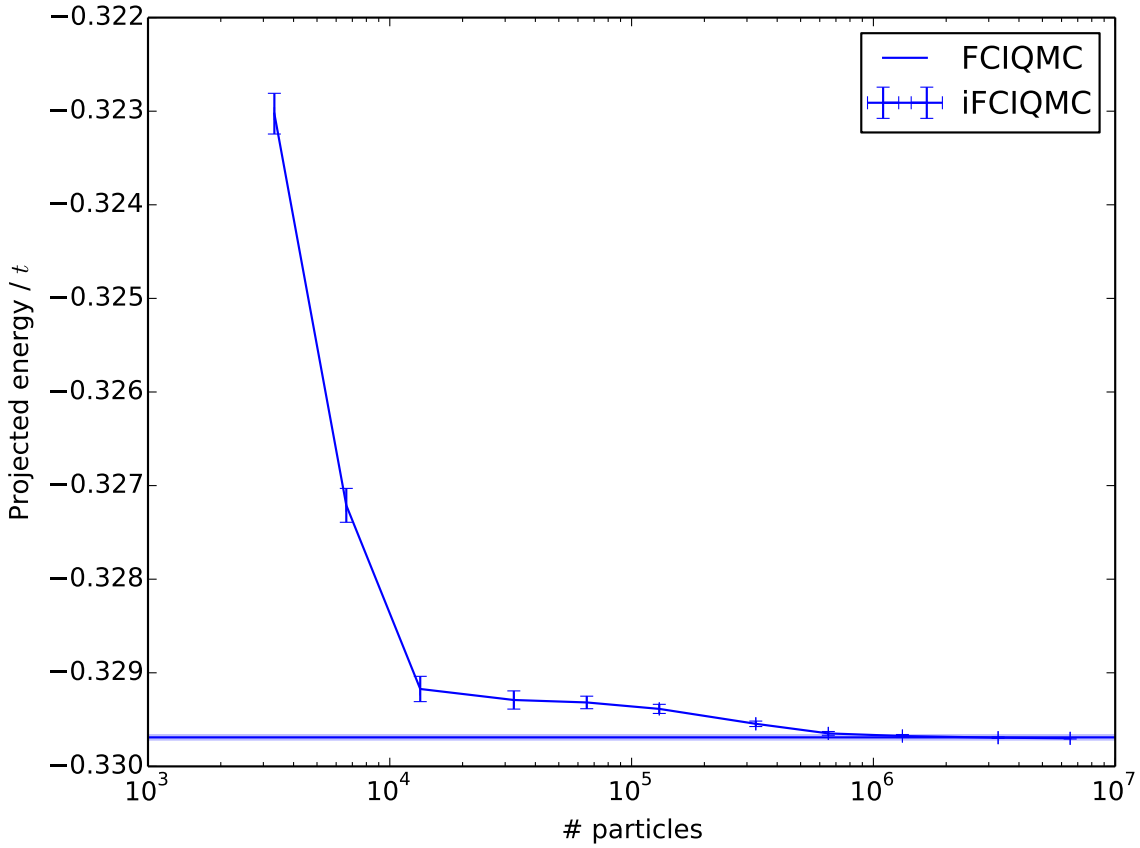
Again, the exact command to launch MPI will vary with MPI implementation and local configurations.

Inspecting the `output`, we see one iFCIQMC calculation was run for each call to the *fciqmc* function. *pyhande* (and, by extension, `reblock_hande.py`) can handle such cases, so we easily extract and inspect the data for each calculation.

Let's start by inspecting instantaneous projected energy estimator for the three smallest populations:



Whilst the difference is small on this scale, it is evident that the calculation with the smallest population has a slightly higher mean than calculations with larger populations. To confirm this, we will plot the energy as a function of population. As `target_population` is the population at which the population **starts** to be controlled, we should consider the average population (which is somewhat higher). We can also compare directly to the FCIQMC energy in this case, as the population required for the FCIQMC calculation is sufficiently small:



The light blue region indicates the extent of the FCIQMC stochastic error, as calculated in the *FCIQMC tutorial*. In this case, the initiator approximation reduces the population required by a factor of ~ 2. However, many studies (including on the electron gas and molecular systems) have demonstrated the initiator approximation can reduce the population required by many orders of magnitude.

The estimates for each calculation can be found directly by using `reblock_hande.py`:

```
$ reblock_hande.py --quiet --start 30000 hubbard_ifciqmc.out
```

where again we chose the start point from inspecting the population growth. This gives:

```
Recommended statistics from optimal block size:

                                  # H psips \sum H_0j N_j         N_0         Shift  Proj. Energy
ifciqmc/hubbard_ifciqmc.out 0        3332(6)      -91.6(1)    283.6(4)    -0.325(2)    -0.3230(2)
                            1        6609(9)     -153.6(2)    469.4(8)    -0.328(2)    -0.3272(2)
                            2      13360(10)     -293.3(3)    890.9(7)    -0.329(1)    -0.3292(1)
                            3      32580(20)     -718.7(4)     2182(2)   -0.3293(6)    -0.3293(1)
                            4      65230(30)    -1403.2(6)     4261(2)   -0.3291(5)   -0.32932(7)
                            5     130170(30)    -2586.4(5)     7853(2)   -0.3289(3)   -0.32939(5)
                            6     326320(60)    -5520.8(8)    16753(3)   -0.3298(2)   -0.32955(3)
                            7     651010(80)      -9937(1)    30146(4)   -0.3296(1)   -0.32965(2)
                            8   1318400(100)     -19354(2)    58707(6)  -0.32969(9)   -0.32968(2)
                            9   3279800(200)     -48971(4)   148530(20)  -0.32962(7)  -0.329693(8)
                            10  6512700(200)     -97670(4)   296240(10)  -0.32971(5)  -0.329700(7)
```

`reblock_hande.py` can also handle the case where each calculation is run separately and each separate file is

passed in as a separate argument on the command line.

Note: We highly recommend a visual inspection of the plot of the initiator error as a function of population as the convergence can be non-monotonic and, as a result, at least two calculations at different populations with statistically equivalent results are required in order to confirm the error due to the initiator approximation is smaller than the stochastic error.

Finally, using real populations can, as with the *FCIQMC tutorial*, have a significant impact on the stochastic error. Again, this is done by setting `real_amplitudes = true` in the input file (see `hubbard_ifciqmc_real.lua`). We also choose to set `spawn_cutoff` to 0.25 following the investigation in *FCIQMC tutorial*; this results in a small increase in the stochastic error but results in the calculation taking roughly half the time. Again, note this is somewhat unique to the Hubbard model. Running:

```
$ mpiexec hande.x hubbard_ifciqmc_real.lua >  hubbard_ifciqmc_real.out
```

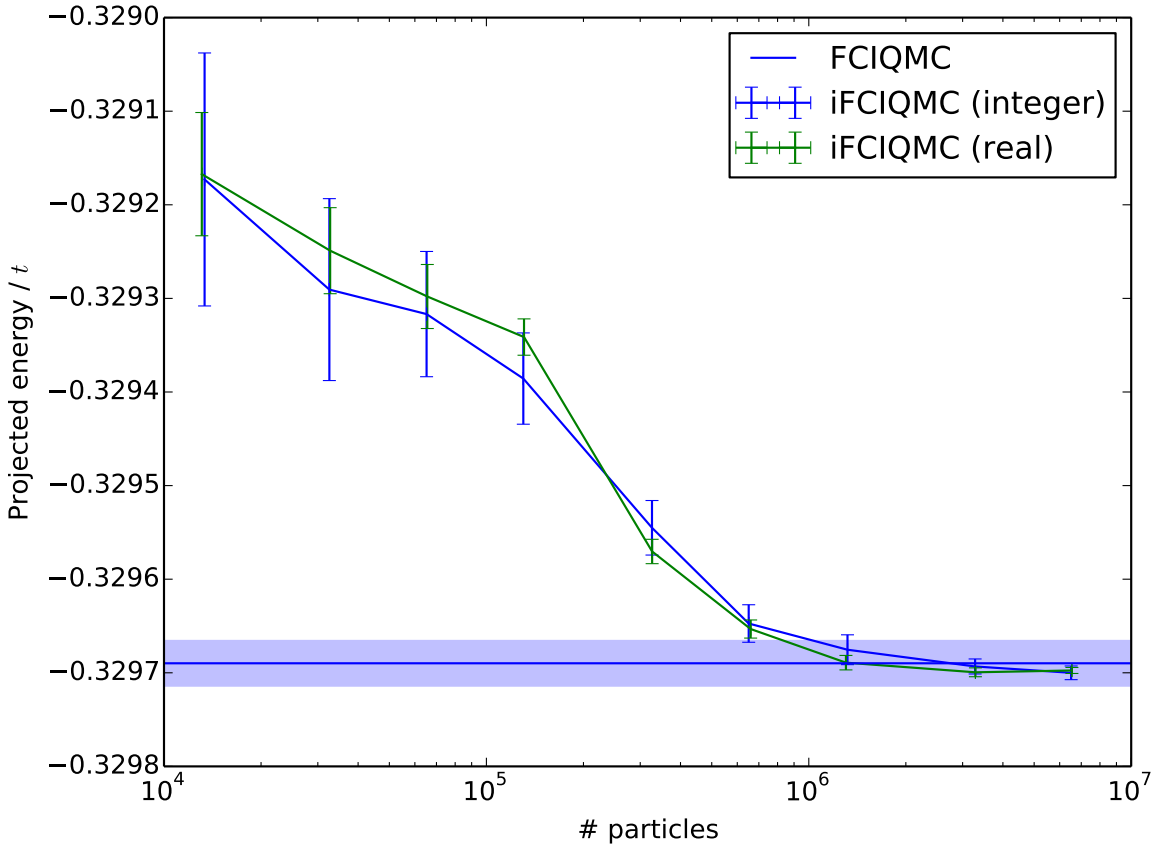
followed by the blocking analysis on the `output`:

```
$ reblock_hande.py --quiet --start 30000 hubbard_ifciqmc_real.out
```

results in

```
Recommended statistics from optimal block size:

                                        # H psips \sum H_0j N_j          N_0         Shift  Proj. Energ
ifciqmc/hubbard_ifciqmc_real.out 0        3271(2)     -91.03(5)    280.7(2)   -0.3234(9)    -0.3243(1
                                 1        6643(4)    -148.28(8)    451.5(3)   -0.3298(7)   -0.32844(7
                                 2       13100(7)     -288.8(2)    877.4(7)   -0.3285(7)   -0.32917(7
                                 3       32762(7)     -744.6(2)   2261.7(7)   -0.3290(3)   -0.32925(5
                                 4      65570(20)    -1510.1(4)     4586(1)   -0.3300(3)   -0.32930(3
                                 5     130830(20)    -2855.0(3)     8669(1)   -0.3295(1)   -0.32934(2
                                 6     326640(30)    -5821.8(6)    17665(2)   -0.3296(1)   -0.32957(1
                                 7     660180(40)   -10118.3(5)    30694(2)  -0.32981(7)   -0.32965(1
                                 8    1303850(60)     -19413(1)    58882(4)  -0.32958(5)  -0.329689(8
                                 9    3287160(90)     -51020(2)   154749(5)  -0.32969(3)  -0.329699(5
                                10   6548200(100)    -103229(2)  313100(10)  -0.32967(2)  -0.329698(3
```

Again, there is a general trend (though not entirely smooth) for the energy estimators to converge to the same energy as a function of total population. It is interesting to take a close look at the convergence of the projected energy estimator:



The cluster of results around populations of 5×10^5 shows that it is vital to reduce the stochastic error before deciding the remaining initiator error is negligible. Further, it is interesting to note that the initiator approximation results in a much more efficient sampling of the Hilbert space: for a similar population ($\sim 10^6$), the iFCIQMC calculations have a **much** smaller stochastic error for a similar computational cost.

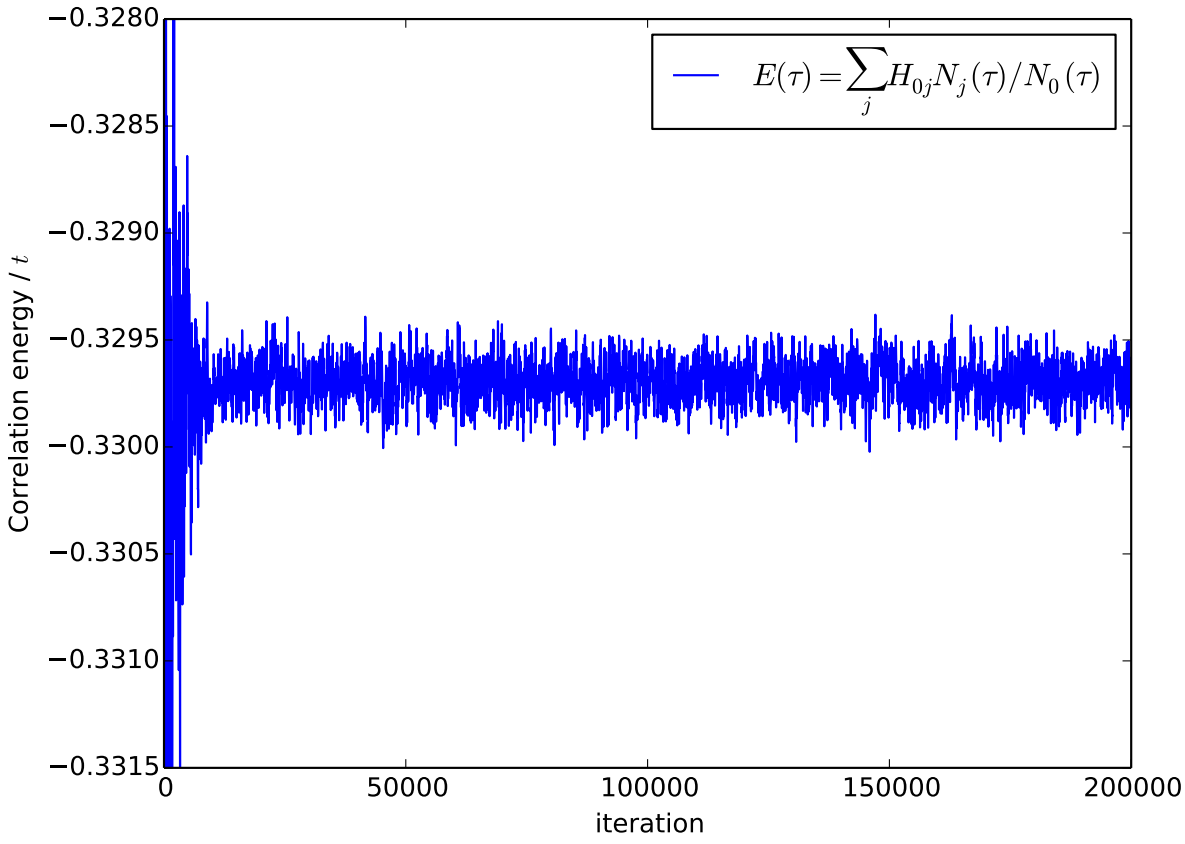
1.13.3 Semi-Stochastic FCIQMC

In this tutorial we will explain how to run FCIQMC calculations using the semi-stochastic adaptation to reduce stochastic errors *[Petruzielo12]*. We will consider the half-filled 18-site 2D Hubbard model at $U/t = 1.3$, as previously considered in the basic *Full Configuration Interaction Quantum Monte Carlo* tutorial. In particular, we shall begin from the input file presented at the end of the FCIQMC tutorial, which introduces the use of non-integer psip amplitudes through the `real_amplitudes` keyword:

```
hubbard = hubbard_k {
    lattice = {
        { 3,  3 },
        { 3, -3 },
    },
    electrons = 18,
    ms = 0,
    U = 1.3,
    t = 1,
    sym = 1,
}
```

```
fciqmc {
    sys = hubbard,
    qmc = {
        tau = 0.002,
        mc_cycles = 20,
        nreports = 10000,
        init_pop = 100,
        target_population = 4*10^6,
        state_size = -1000,
        spawned_state_size = -100,
        real_amplitudes = true,
    },
}
```

and which results in the following simulation:



The semi-stochastic adaptation provides a way to reduce the stochastic noise in such simulations. It does so by choosing a certain subspace (called the deterministic subspace), which is deemed to be most important (in that most of the wave function amplitude resides in this subspace), and performing projection exactly within it. Projection outside the subspace is performed by the usual FCIQMC stochastic spawning.

Thus, we simply need to specify what subspace to use for the exact projection. One way of doing this is by using the scheme of *[Blunt15]*, where the subspace is formed from the determinants on which the largest number of psips reside. We therefore simply need to tell HANDE what iteration to start using the semi-stochastic adaptation, and how many determinants to form the deterministic subspace from.

There are a couple of things to consider when choosing the size of the deterministic space. Firstly, within the deterministic subspace, the Hamiltonian is stored exactly in a sparse form. Therefore, using semi-stochastic does increase memory requirements. The deterministic Hamiltonian storage (and multiplication) is distributed across processing cores, which allows larger subspaces to be used. The other consideration is that, for very large deterministic spaces, and for certain systems (particularly strongly correlated systems), semi-stochastic can slow simulations down slightly. Through investigation (for example, see *[Blunt15]*), it has been found that a deterministic space of size 10^4 allows a very large reduction in stochastic error for most simulations, while not increasing simulation time. We therefore suggest this as a black box subspace size. This is also small enough that the deterministic Hamiltonian can always be stored, even if using only a typical desktop computer.

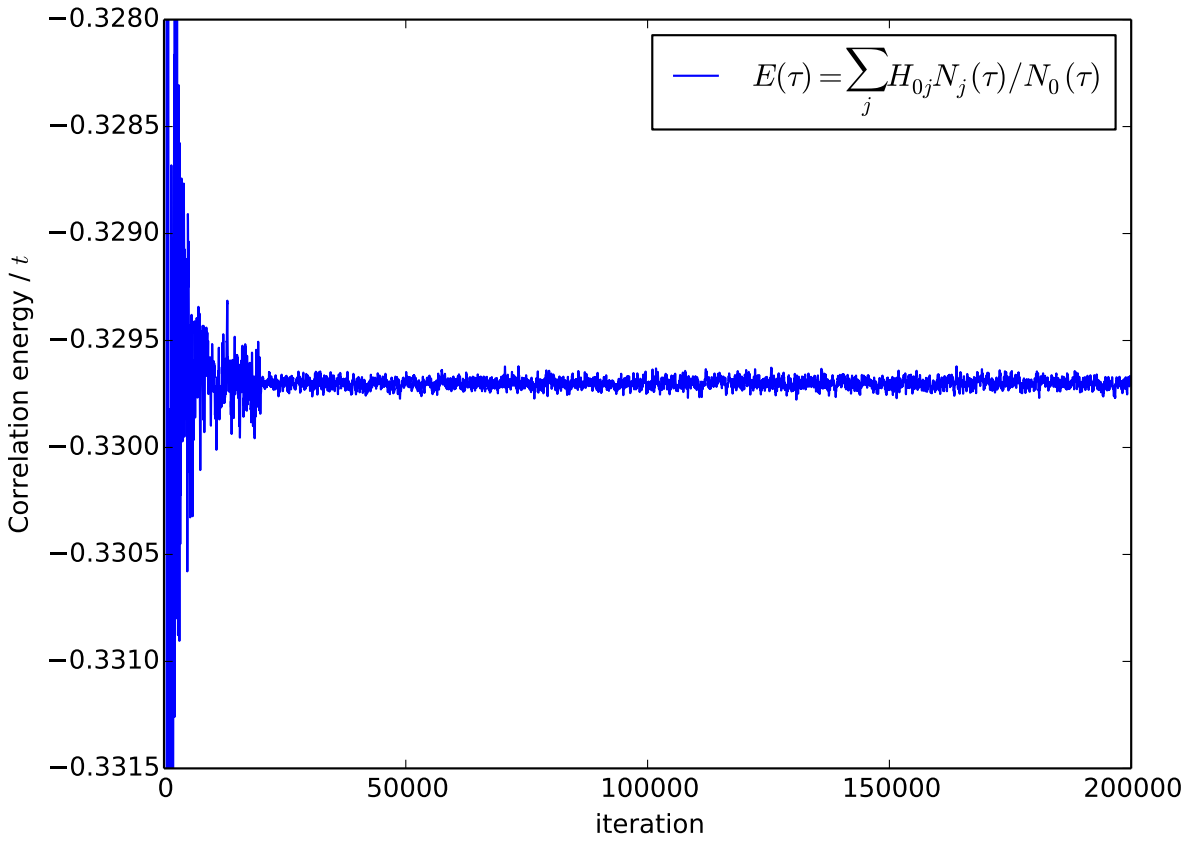
Note that because semi-stochastic does not usually reduce iteration time much (and sometimes increases it), one should not worry that we do not use semi-stochastic from the first iteration. We are only concerned with reduction in stochastic error from the point where data will be averaged later.

Looking at the above simulation, it appears that the energy has converged by iteration 2×10^4. This is not a guarantee that the wave function is also fully converged, but full convergence is not critical – so long as the most important determinants are in the deterministic subspace, then a large reduction in stochastic error will occur. As discussed above, a reasonable deterministic space size is 10^4. So, to start using a deterministic space of size 10^4 at iteration 2×10^4, we modify the above input to the following:

```
hubbard = hubbard_k {
    lattice = {
        { 3,  3 },
        { 3, -3 },
    },
    electrons = 18,
    ms = 0,
    U = 1.3,
    t = 1,
    sym = 1,
}

fciqmc {
    sys = hubbard,
    qmc = {
        tau = 0.002,
        mc_cycles = 20,
        nreports = 10000,
        init_pop = 100,
        target_population = 4*10^6,
        state_size = -1000,
        spawned_state_size = -100,
        real_amplitudes = true,
    },
    semi_stoch = {
        size = 10000,
        start_iteration = 20000,
        space = "high",
    },
}
```

Here, the `semi-stoch` table contains three keywords. The use of `size` and `start_iteration` keywords is hopefully clear. The `space` keyword determines which method is used to generate the deterministic space - in this case by choosing the determiniants with the highest weights.

This results in the following simulation:



As can be seen, at iteration 2×10^4 there is a large reduction in stochastic error.

When performing a blocking analysis, the user should not begin averaging data until after the semi-stochastic adaptation has been turned on, since there is a significant change in the probability distributions of data beyond this point. This is particularly true in initiator FCIQMC simulations, where the use of semi-stochastic can alter the initiator error (although we typically find that semi-stochastic does not alter the magnitude of initiator error significantly, it can in some cases, see *[Petruzielo12]*). We can therefore analyse the above simulation using

```
$ reblock_hande.py --quiet --start 30000 hubbard_semi_stoch_high.out
```

which results in:

```
Recommended statistics from optimal block size:

                                # H psips \sum H_0j N_j       N_0         Shift    Proj. Energy
hubbard_semi_stoch_high.out   5216460(80)     -15356(2)   46576(5)   -0.32970(2)   -0.3296994(8)
```

Compared to the equivalent non-semi-stochastic simulation performed in the *FCIQMC tutorial* tutorial, the error bars on the shift and projected energy estimators have reduced from 4×10^{-5} and 3×10^{-6} to 2×10^{-5} and 8×10^{-7}, respectively.

Note that if you do not specify a `start_iteration` value in the `semi_stoch` table of the input file, then the semi-stochastic adaptation will be turned on from the first iteration. This should not be done when starting a new simulation, because wave functions in HANDE are initialised as single determinants. However, if restarting a simulation from an HDF5 file then this is a sensible approach - the simulation will begin from the wave function stored in the HDF5 file, and the deterministic space will be chosen from the most populated determinants in this wave function. An input file

for such a restarted simulation would contain the following `semi_stoch` and `restart` tables within the `fciqmc` table:

```
fciqmc {
    sys = hubbard_k{...},
    qmc = {...},
    semi_stoch = {
        size = 10000,
        space = "high",
    },
    restart = {
        read = 0,
    },
}
```

(see the *restart options* entry in the documentation for more options relating to restarting simulations).

Finally, when restarting simulations which were already using the semi-stochastic adaptation, it is important to use exactly the same deterministic space to ensure that estimators are statistically consistent before and after restarting. However, the approach in HANDE uses the instantaneous FCIQMC wave function to generate the deterministic space, which changes during the simulation. Using the above approach would therefore lead to a slightly different space being generated after restarting. One can get around this by outputting the deterministic space in use to a file, and reading it back in for the restarted calculation. For example, to generate a deterministic space from the 10^4 most populated determinants at iteration 2×10^4, and to then print this space to a file, one should use the `write` keyword in the `semi-stoch` table:

```
fciqmc {
    sys = hubbard_k{...},
    qmc = {...},
    semi_stoch = {
        size = 10000,
        start_iteration = 20000,
        space = "high",
        write = 0,
    },
}
```

Here, the value of the `write` keyword, 0, is an index used in the name of the resulting file. Note that `write` can also be a boolean, in which case HANDE will find and use the smallest unused id available in the directory.

When restarting the simulation, one can then specify the `space` option to read a semi-stochastic HDF5 file, using:

```
fciqmc {
    sys = hubbard_k{...},
    qmc = {...},
    semi_stoch = {
        space = "read",
    },
}
```

The deterministic space file is an HDF5 file. As such, both writing and reading of such files requires *compilation* of HANDE with HDF5 enabled, which is the default compilation behaviour.

1.13.4 Coupled Cluster Monte Carlo

In this tutorial we will run CCMC on the carbon monoxide molecule in a cc-pVDZ basis. For details of the theory see *[Thom10]* and *[Spencer15]*.

This tutorial only presents the basic options available in a CCMC calculation; for the full range of options see the main *documentation*.

To perform calculations on a molecular system in HANDE, we need the one- and two- electron *integrals* in some appropriate basis from an external source. For the calculations in this tutorial, the integrals were calculated using Psi4; input and output files can be found with the files from the calculations herein in the `documentation/manual/tutorials/calcs/ccmc` subdirectory.

The system definition is exactly the same as for FCIQMC:

```
sys = read_in {
    int_file = "CO.CCPVDZ.FCIDUMP",
    nel = 14,
    ms = 0,
}
```

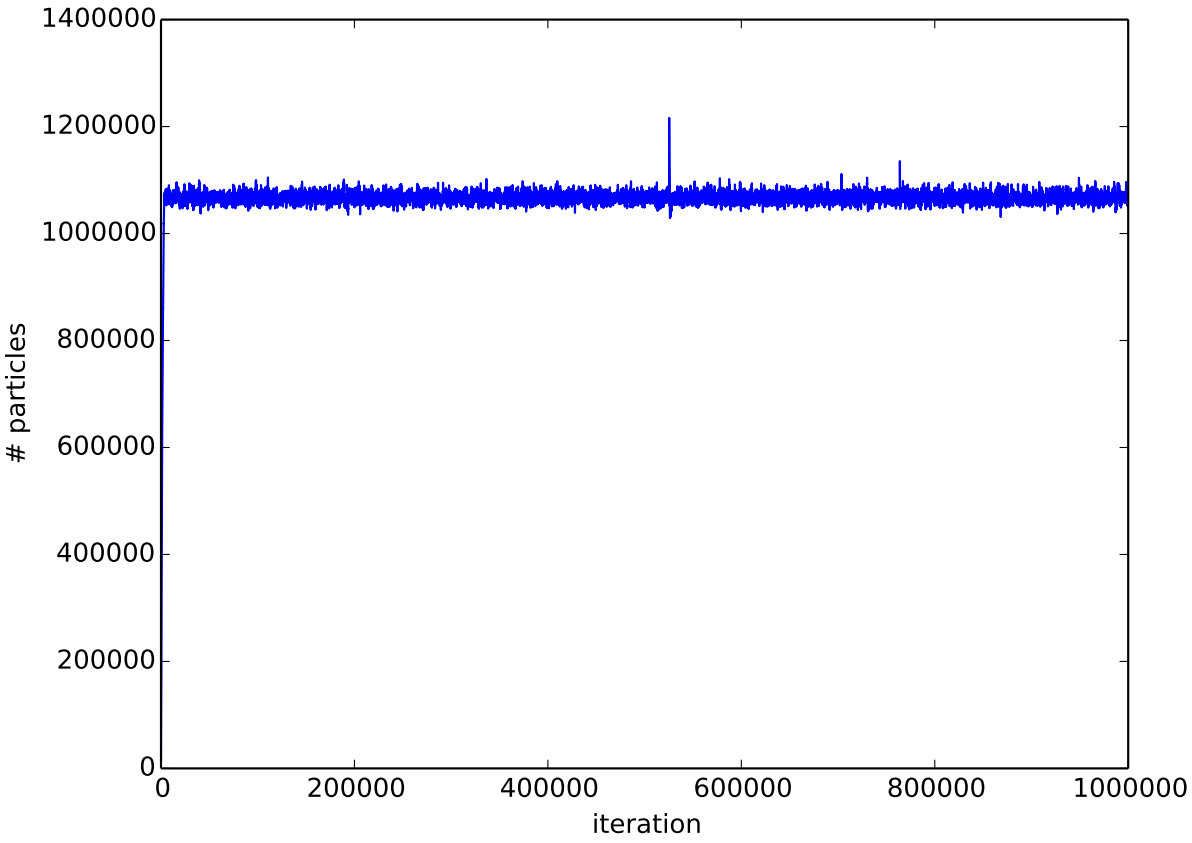
Note that we have not specified an overall symmetry. In this case HANDE uses the Aufbau principle to select a reference determinant.

A CCMC calculation can be run in a very similar way to FCIQMC. As for FCIQMC we can substantially reduce stochastic error by using real amplitudes, which we do for all calculations presented here. The most significant difference from an FCIQMC input is that it is standard to use truncation with CCMC, specified by the `ex_level` option, (i.e. 2 for CCSD, 3 for CCSDT, etc.). The determination of a plateau and hence a suitable value for `target_population` is exactly analogous to *FCIQMC*, as the sign problem is similar between the two methods; we will not discuss it further here. The CCSDTMC calculation can be run using an input file such as:

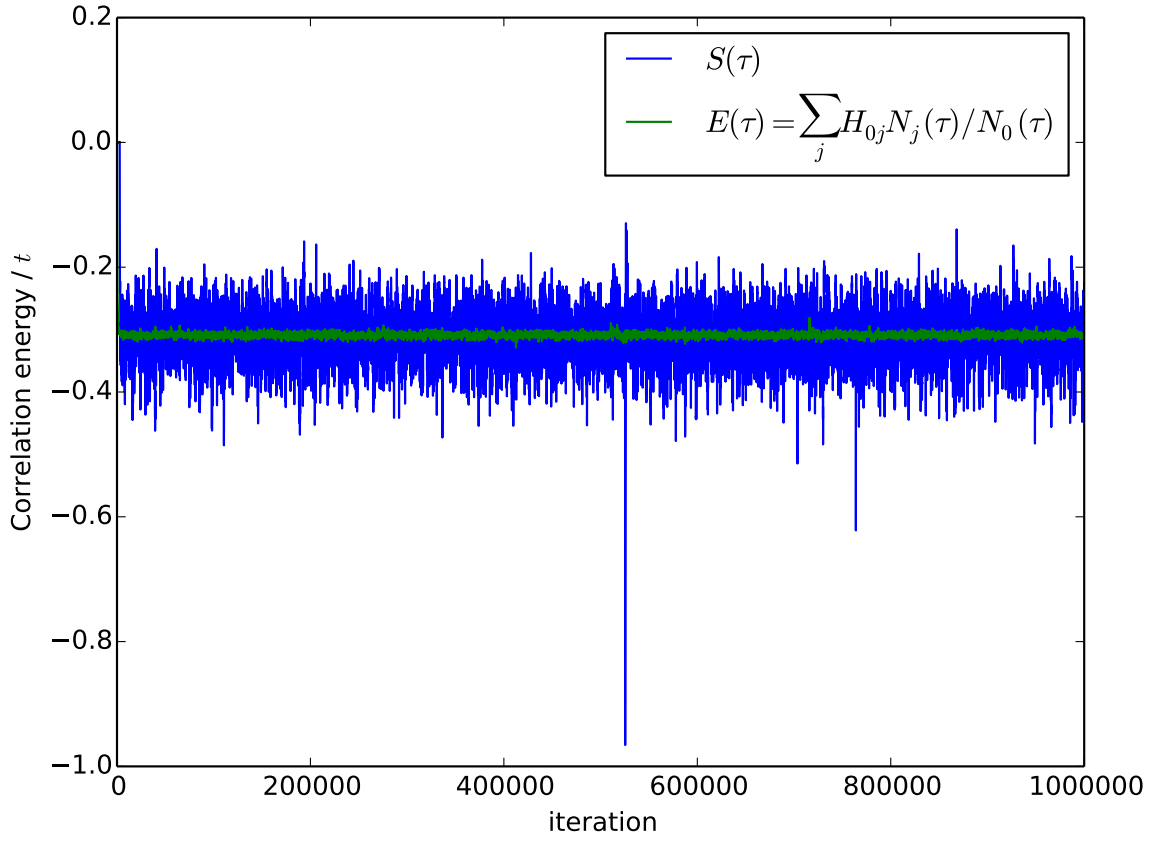
```
sys = read_in {
    int_file = "CO.CCPVDZ.FCIDUMP",
    nel = 14,
    ms = 0,
}

ccmc {
    sys = sys,
    qmc = {
        tau = 1e-3,
        mc_cycles = 10,
        nreports = 1e5,
        state_size = -500,
        spawned_state_size = -200,
        init_pop = 1e4,
        target_population = 1e6,
        real_amplitudes = true,
    },
    reference = {
        ex_level = 3,
    },
}
```

Note the much larger initial population compared to an FCIQMC calculation; if this is too low the correct wavefunction will not be obtained.

Looking at the `output`, we see the evolution of the population has a similar form to FCIQMC:



and the shift and projected energy vary about the correlation energy:



The output of the calculation can be analysed in exactly the same way as for FCIQMC:

```
$ reblock_hande.py --quiet --start 100000 co_ccmc.out
```

giving

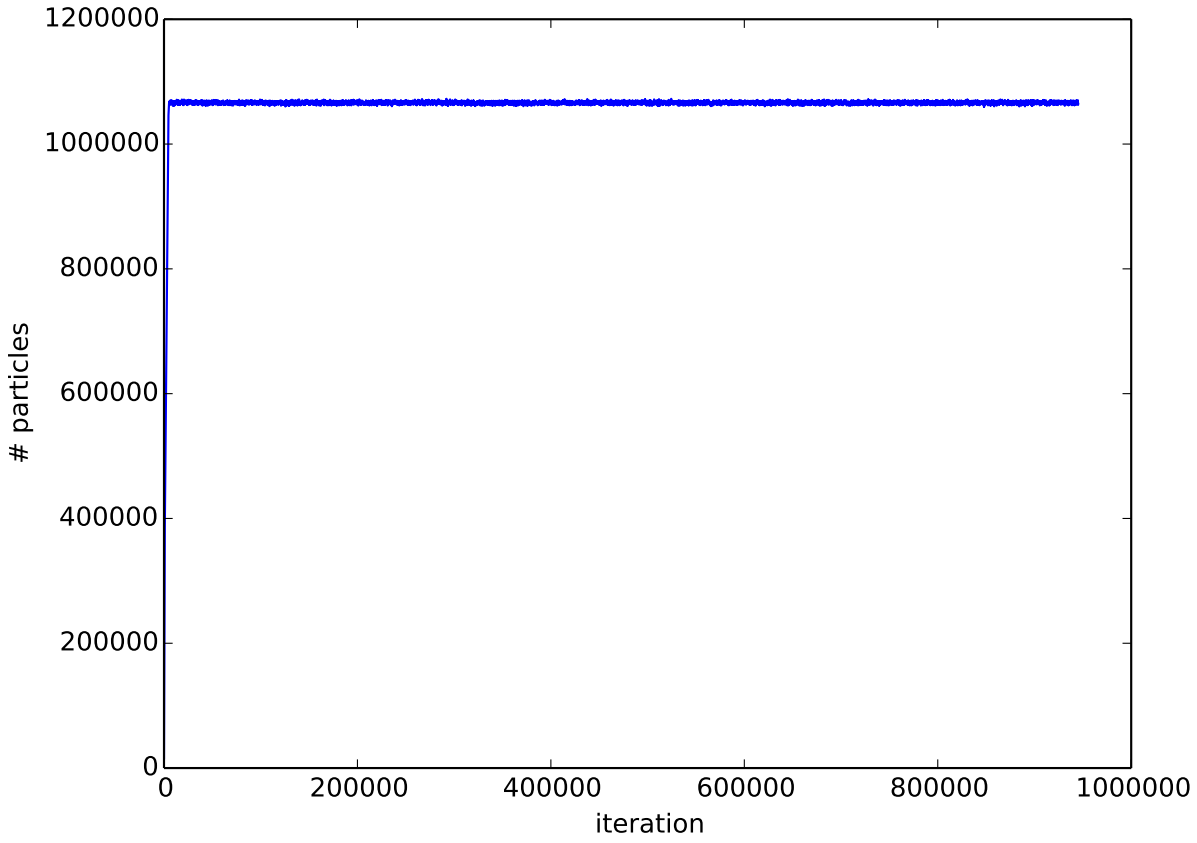
```
Recommended statistics from optimal block size:

                      # H psips \sum H_0j N_j        N_0        Shift Proj. Energy
ccmc/co_ccmc.out   1066600(100)     -6510(10)  21040(40)  -0.3092(6)  -0.30925(7)
```

Due to the sampling of the wavefunction in CCMC, it is more prone to "blooming" events where many particles are created in a single spawning event than is FCIQMC. Details of blooming during a calculation are reported at the end of the output. It can be seen that significant blooming occurred. This substantially increases the stochastic error, and in particularly severe cases can cause the calculation to not give a correct result due to the instability. These events can be avoided by reducing the timestep, but the timestep required to eliminate them entirely is often prohibitively small. Another way of reducing them is the use of the `cluster_multispawn_threshold` keyword, whereby large spawning attempts are divided into a number of smaller spawns:

```
sys = read_in {
    int_file = "CO.CCPVDZ.FCIDUMP",
    nel = 14,
    ms = 0,
}

ccmc {
```

```
    sys = sys,
    qmc = {
        tau = 1e-3,
        mc_cycles = 10,
        nreports = 1e5,
        state_size = -500,
        spawned_state_size = -200,
        init_pop = 1e4,
        target_population = 1e6,
        real_amplitudes = true,
    },
    ccmc = {
        cluster_multispawn_threshold = 10,
    },
    reference = {
        ex_level = 3,
    },
}
```

Running as before, and inspecting the `output`, it can be seen that there are now no blooms. Additionally, plotting the population growth and comparing to the previous plot we see that there are now no spikes in the population:



This substantially reduces the stochastic error:

```
Recommended statistics from optimal block size:

                                # H psips \sum H_0j N_j       N_0       Shift Proj. Energy
```

```
ccmc/co_ccmc_multispawn.out  1066160(30)     -11191(1)  36178(7)  -0.3091(1)  -0.30933(3)
```

The extra spawning causes the calculation to run more slowly, but the reduction in error bars can often more than make up for this.

1.13.5 Density Matrix Quantum Monte Carlo

In this tutorial we will run DMQMC on the 2D Heisenberg model and the uniform electron gas. The input and output files can be found under the `documentation/manual/tutorials/calcs/dmqmc` subdirectory of the source distribution. Knowledge of the terminology and theory given in *[Booth09]*, *[Blunt14]* and *[Malone15]* is assumed.

To begin we will focus on the 6x6 antiferromagnetic Heisenberg model on a square lattice with periodic boundary conditions. The input file for this system is given as

```
sys = heisenberg {
    lattice = {
        {6, 0},
        {0, 6},
    },
    J = -1.0,
    ms = 0,
}

dmqmc {
    sys = sys,
    qmc = {
        tau = 0.001,
        init_pop = 5e6,
        rng_seed = 19838,
        mc_cycles = 10,
        nreports = 500,
        shift_damping = 0.5,
        target_population = 5e6,
        state_size = -400,
        spawned_state_size = -400,
    },
    dmqmc = {
        beta_loops = 1,
    },
    operators = {
        energy = true,
        excit_dist = true,
    },
}
```

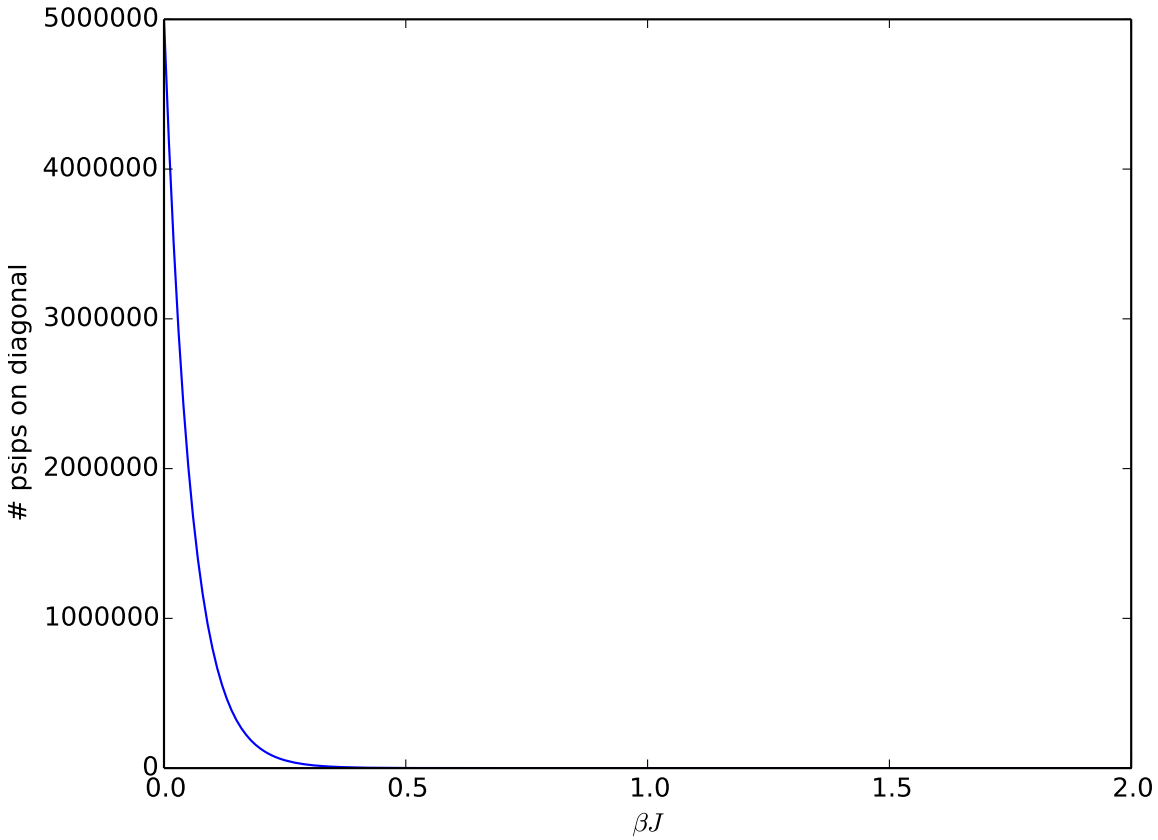
and is largely analogous to that found in the *FCIQMC tutorial*. We refer the reader to the discussion there and the manual for system specific input options. Note that `init_pop` here controls the population with which the density matrix at $\beta = 0$ is sampled. Typically the shift is allowed to vary from the beginning of a simulation by setting `target_pop` equal to `init_pop`. Here we will attempt to run to a final temperature of $\beta = 5/J$. The `beta_loops` option determines the number of independent simulations over which observables are averaged, see *dmqmc options* for more options. The operators table specifies which observables are to be evaluated in a given simulation. Here only the total energy is considered, a full list is available in *operators options*.

An issue encountered when applying DMQMC to larger systems is that the population on the diagonal (denoted Trace in the output file) decays with increasing β which results in poor estimates for observables. The seriousness of this problem needs to be assessed on a system by system basis and should be tested for as a first step, which we'll do now.

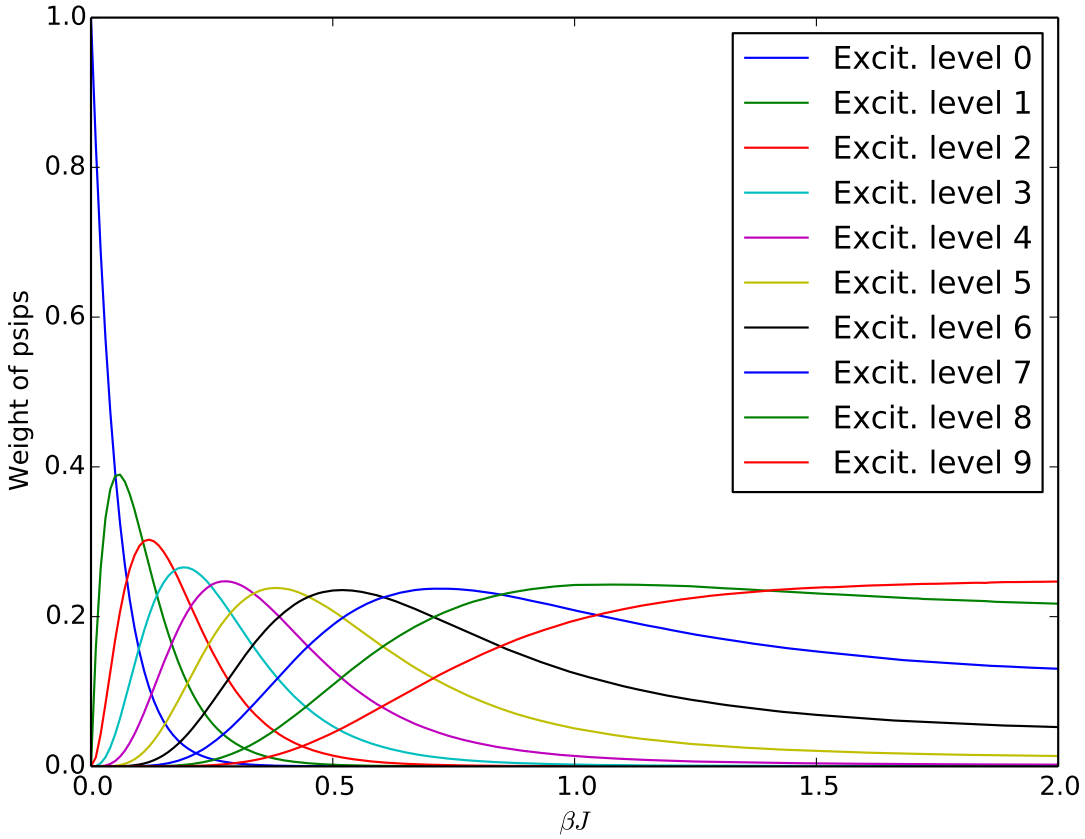
To do this we set `beta_loops` to 1 in the input file and run the code as:

```
$ aprun -B hande.x heisenberg_dmqmc.lua > heisenberg_dmqmc.out
```

We find that for this system the population on the diagonal does indeed decay to zero rapidly:



The source of this problem can be investigated by analysing the distribution of psips on different excitation levels of the density matrix, which was calculated in anticipation of this result using the `excit_dist` option in the operators table. Here the excitation level is defined as the difference between the bra and ket of a density matrix element i.e., number of spin flips or number of particle-hole pairs for electronic systems. We see the majority of the total weight is redistributed from the diagonal to highly excited determinants.



To overcome this *[Blunt14]* invented an unbiased importance sampling scheme to encourage psips to stay on or near the diagonal by penalising spawning moves away from excitation levels. This is sensible as typically the majority of the weight contributing to most physically significant observables originates from the determinants at lower excitation levels which we wish to sample more regularly.

Practically this amounts to first running a calculation with the `find_weights` option. This will output importance sampling weights which are appropriate as input for the production calculation. It is worthwhile to run the calculation for a few `beta_loops` to ensure the weights are not fluctuating too much, and also check they don't fluctuate too much with the `target_population`. The algorithm currently tries to ensure that the number of walkers on each excitation level is roughly constant once the ground state is thought to have been to be reached. The iteration number where this is deemed to have been reached is controlled by the `find_weights_start` option.

For this system we do

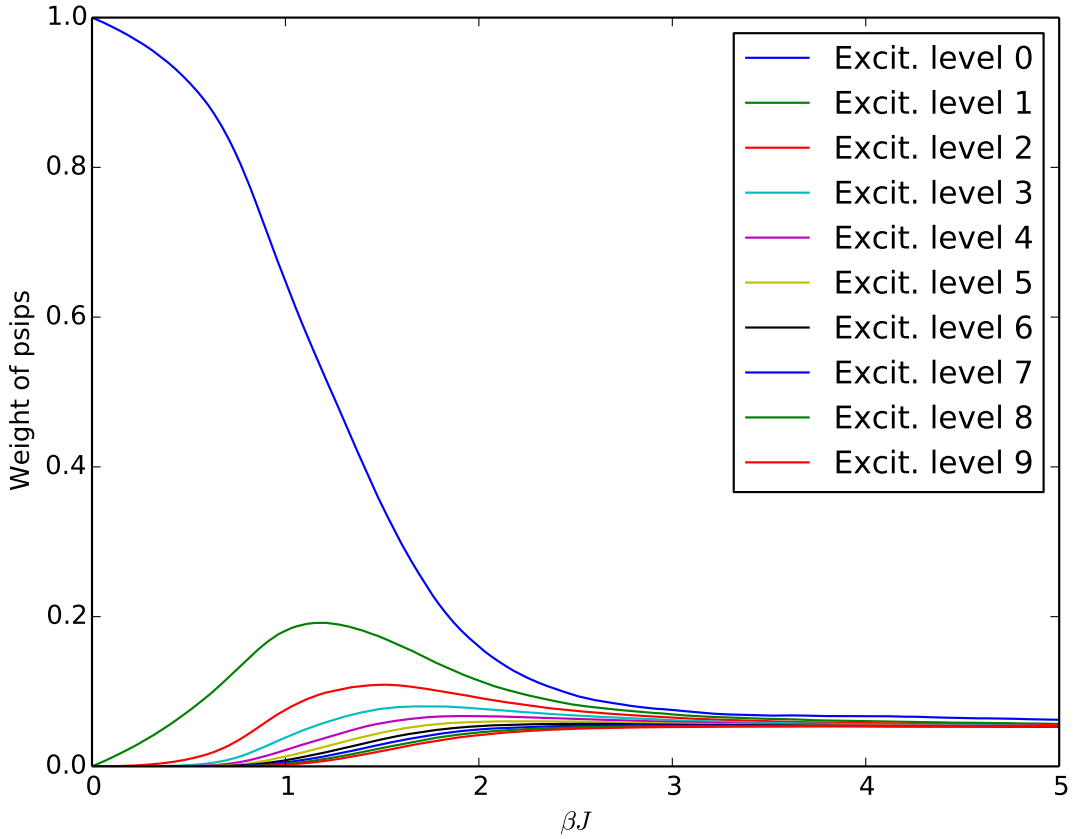
```
$ aprun -B hande.x heisenberg_find_weights.lua > heisenberg_reweighted.out
```

Here we first run a simulation for 10 beta loops to find the weights and then use the last iteration's weights as input to the production calculation. This procedure can simplified using lua as seen in the input file.

To see what is going on we can copy the weights from the output file and run for a single iteration and again examine the excitation distribution

```
$ aprun -B hande.x heisenberg_reweight_single.lua > heisenberg_reweight_single.out
```

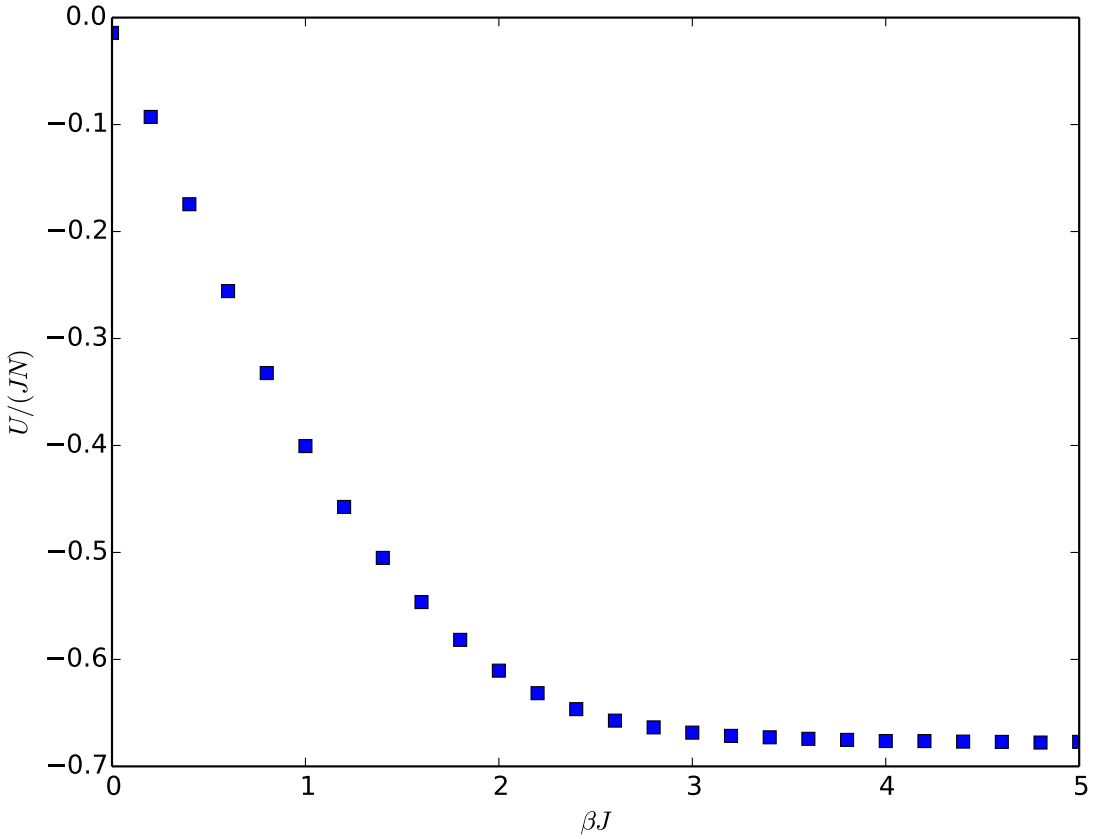
and we find that the psips are now more equally distributed among excitation levels:



The results of the full reweighted calculation can be analysed using the `finite_temperature_analysis.py` script provided in the `tools/dmqmc` subdirectory:

```
$ finite_temp_analysis.py heisenberg_reweighted.out  > heisenberg_reweighted_block.out
```

Finally, we can plot the results of the internal energy, U, as a function of temperature:



1.13.6 Interaction Picture Density Matrix Quantum Monte Carlo

It turns out that the original formulation of DMQMC can run into problems for moderately weakly interacting systems which are relatively well described by Hartree–Fock theory. An extreme example of this is the uniform electron gas (UEG) especially at higher densities (low r_s). This issue is largely overcome by switching to the interaction picture which enables us to start from a (temperature dependent) mean-field distribution at $\tau = 0$ ensuring low energy determinants are initially sampled. See *[Malone15]* for details. For systems with a good mean-field ground state the user should consider using IP-DMQMC.

Most of the running details for IP-DMQMC are the same as for DMQMC, however there are some additional considerations. This is best demonstrated by running a simulation. We will focus on a 7-electron, spin polarised system in 319 plane waves at $r_s = 1$.

Looking at the input file

```
sys = ueg {
    nel = 7,
    ms = 7,
    sym = 1,
    dim = 3,
    cutoff = 10,
    rs = 1,
}

dmqmc {
```

```
    sys = sys,
    qmc = {
        tau = 0.001,
        rng_seed = 7,
        init_pop = 10000,
        mc_cycles = 10,
        nreports = 100,
        target_population = 10000,
        state_size = -200,
        spawned_state_size = -100,
    },
    dmqmc = {
        fermi_temperature = true,
        all_sym_sectors = true,
        beta_loops = 100,
    },
    ipdmqmc = {
        target_beta = 1.0,
        initial_matrix = 'free_electron',
        grand_canonical_initialisation = true,
        symmetric = false,
    },
    operators = {
        energy = true,
    },
}
```

we see most of the same options are present as for dmqmc. Note that unlike DMQMC where estimates for the whole temperature range are gathered in a single simulation, in IP-DMQMC only one temperature value is (directly) accessible, specified by the `target_beta` option. We've also set the energy scale to be determined by the Fermi energy of the corresponding (thermodynamic limit) free electron gas so that the temperatures are interpreted as fractions of the Fermi temperature (here $\Theta = 0.5$. `all_sym_sectors` ensures all momentum symmetry sectors are averaged over. To average over spin polarisation the `all_spin_sectors` option must be specified.

Moving on through the ipdmqmc table we've set the `initial_matrix` to be the free electron density matrix, i.e., Fermi-Dirac like. Additionally we're using the `grand_canonical_initialisation` option to initialise this density matrix (see *[Malone15]*). This is the recommended method to initialise the density matrix; the Metropolis algorithm should only be used for testing.

Finally we will use the asymmetric form of the original IP-DMQMC algorithm by specifying `symmetric` to be false. The symmetric algorithm is somewhat experimental but can lead to better estimates for quantities other that the internal energy especially at lower temperatures. This is thought to be due to sampling issues at low temperatures where the initial mean field guess becomes significantly different (in terms of energy scales) to the fully interacting theory. Symmetrising the equations allows psips to move along rows and which improves sampling. See *[Malone16]*.

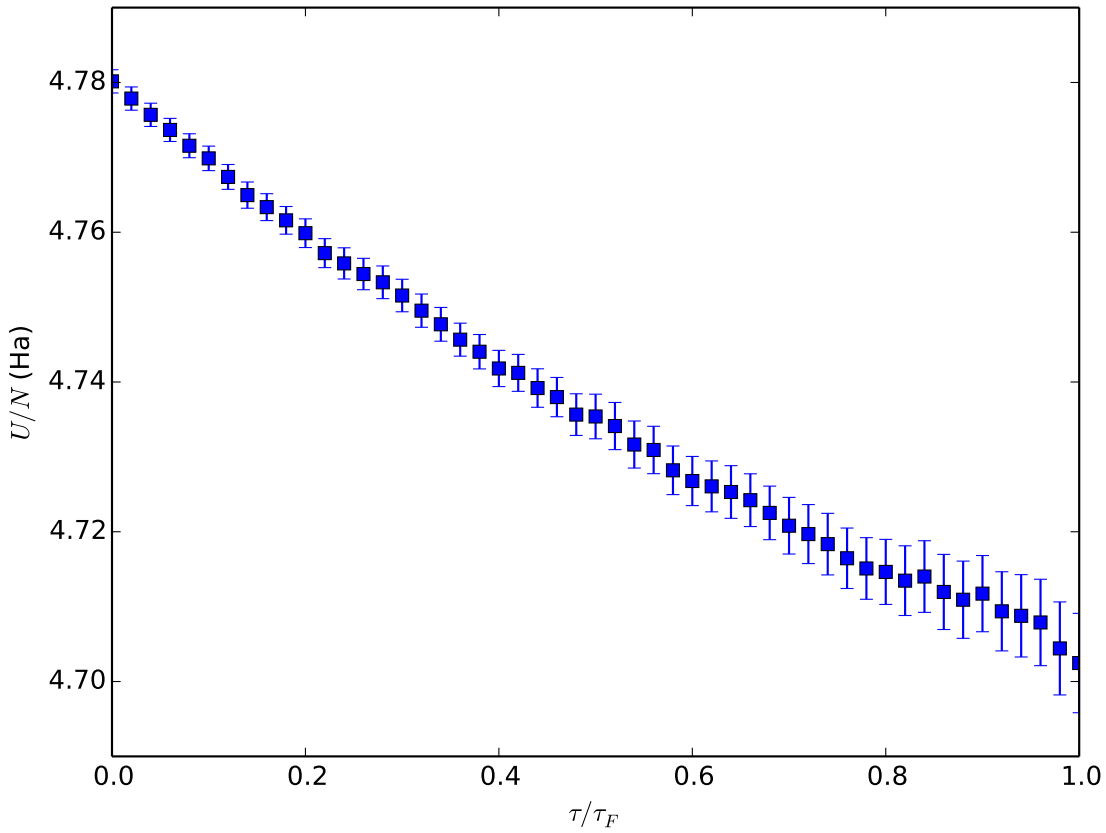
Running the code

```
$ hande.x ipdmqmc_ueg.lua > ipdmqmc_ueg.out
```

and analysing the `output`:

```
$ finite_temp_analysis.py ipdmqmc_ueg.out > ipdmqmc_ueg_block.out
```

we find



where again only estimates at the final iteration are physical, i.e., when $\tau = \beta$. Note that the estimates do not contain a Madelung constant.

The initiator approximation can significantly extend the range of applicability of DMQMC but is somewhat experimental. See the options, in particular `initiator_level` in the manual for more discussion. The user should ensure results are meaningful by comparing answers at various walker populations. See *[Malone16]* for further discussion.

1.13.7 Canonical Estimates

In this tutorial we will discuss how estimates for various mean-field properties of a system can be evaluated in the canonical ensemble at finite temperatures. These estimates are useful for basis set extrapolation as well as comparison to the fully interacting results and are non-trivial to evaluate analytically. See *[Malone15]* for details.

The input file is fairly simple:

```
sys = ueg {
    nel = 7,
    ms = 7,
    dim = 3,
    cutoff = 10,
    rs = 1,
}

canonical_estimates {
    sys = sys,
```

```
    canonical_estimates = {
        beta = 1,
        nattempts = 10000,
        ncycles = 1000,
        fermi_temperature = true,
    },
}
```

Here we attempt to generate N particle states making `nattempts` attempts and then run the simulation for `ncycles*nattempts` iterations in total. The only other options available are the inverse temperature desired, which can be scaled by the Fermi temperature (where appropriate). Here we restrict ourself to the fully spin polarised UEG in M=389 plane waves, which can be compared to the IP-DMQMC simulation in the *DMQMC tutorial*.

Running the input file we find

```
$ hande.x canonical_estimates.lua >  canonical_estimates.out
```

Inspecting the `output`, we see a number of columns for various estimates including the kinetic, potential, internal, free energy and entropy - precise definitions of everything can be found in the output file. The data can be analysed to find the mean and standard error using the `analyse_canonical.py` script in the `tools/dmqmc` subdirectory:

```
$ analyse_canonical.py canonical_estimates.out
```

which gives

```
          Beta             U_0      U_0_error             T_0      T_0_error             V_0      V_0_e
1.00000000e+00 3.34489604e+01 7.12207413e-03 3.42505858e+01 7.00598613e-03 -8.01625332e-01 1.4328202
```

In particular, we can compare the values of U_0 and U_{HF} to the value of 32.91(4) Ha from the IP-DMQMC tutorial.

1.13.8 Shoulder Plots

This tutorial looks further into finding the optimal target particle population in more detail. It is advisable to have read the *FCIQMC* and *CCMC* tutorials before this one. More information and details on shoulder plots can be found in *[Spencer15]*.

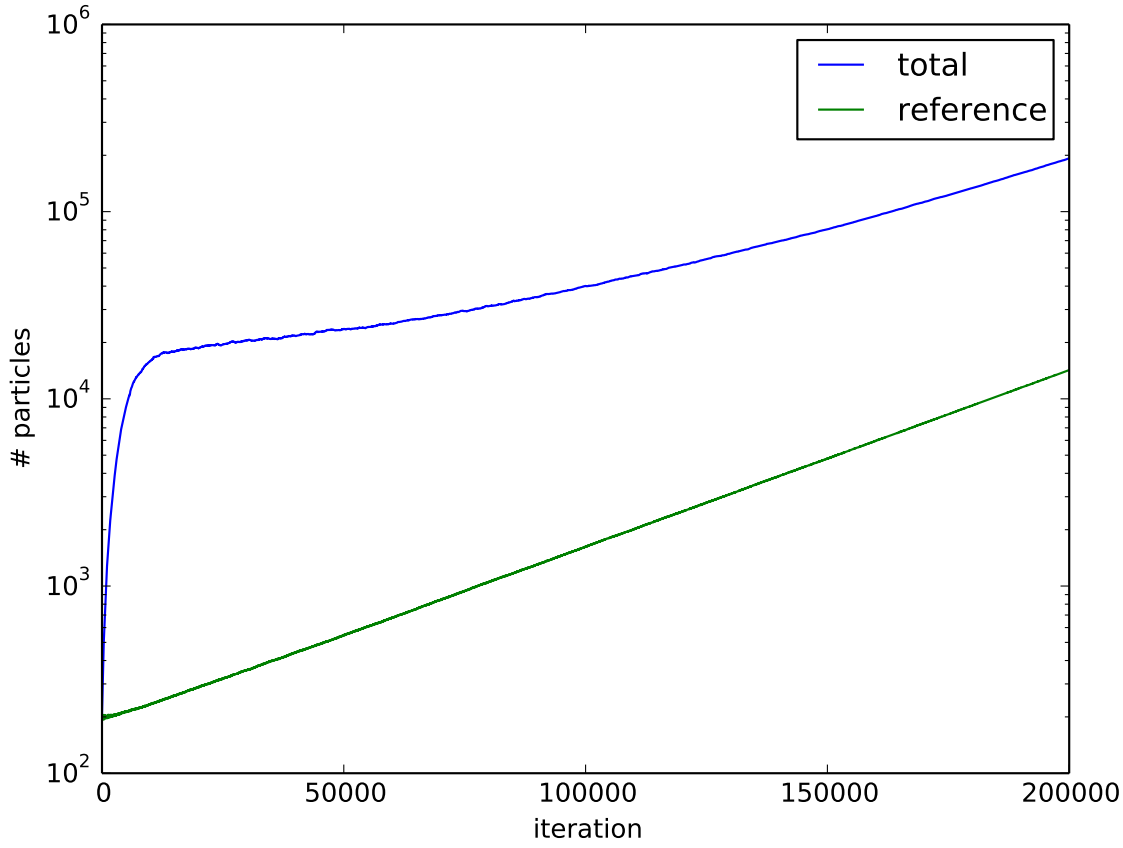
The example used here is a CCSDT Monte Carlo calculation on water in a cc-pVDZ basis *[Dunning89]*. As in the *CCMC tutorial*, the integrals were calculated with PSI4 (see *Generating integrals* for details). Input and output files are in `documentation/manual/tutorials/calcs/shoulder/`.

The first calculation was run using

```
sys = read_in {
    int_file = "H2O_INTDUMP",
    nel = 10,
    ms = 0,
}

ccmc {
    sys = sys,
    qmc = {
        tau = 1e-4,
        mc_cycles = 10,
        nreports = 2e4,
        state_size = -500,
        spawned_state_size = -200,
        init_pop = 200,
        real_amplitudes = true,
```

```
        target_population = 3e5,
    },
    reference = {
        ex_level = 3,
    },
}
```

Just like *FCIQMC*, a plateau can be seen in a total population vs iteration plot, which indicates roughly the minimum particle number to make the calculation stable:

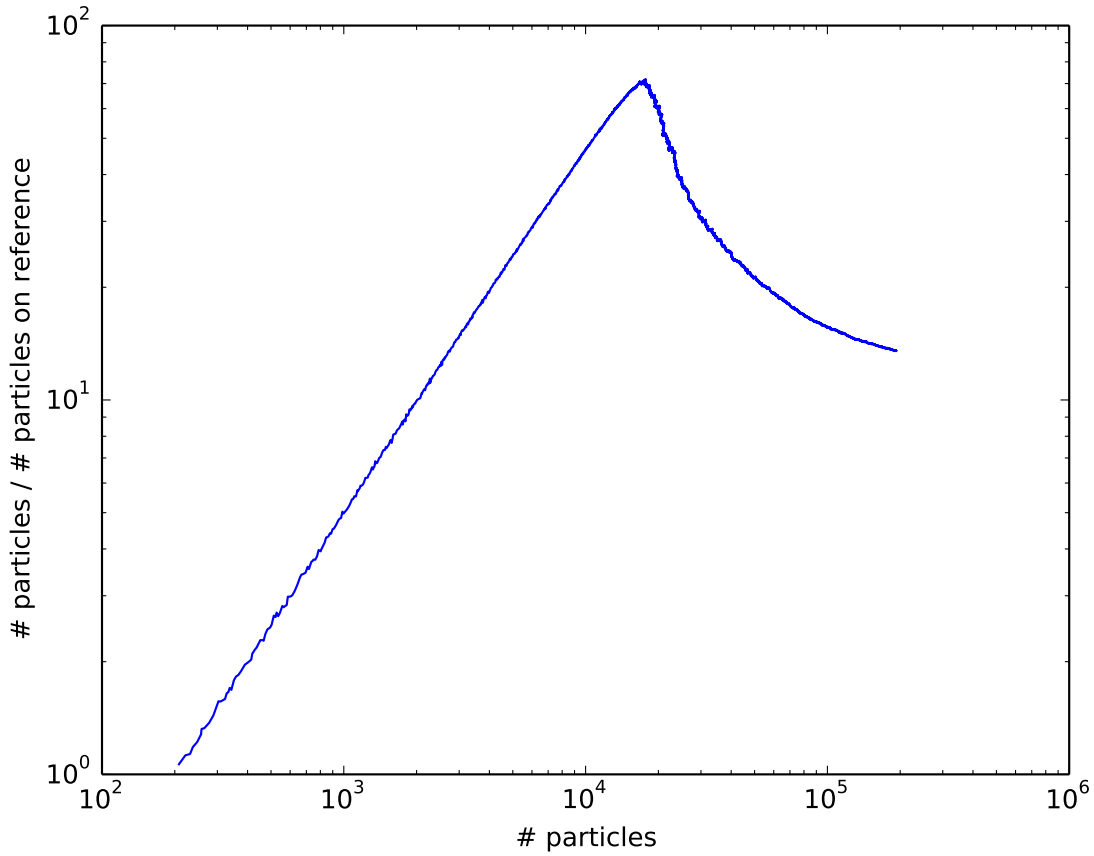


The plateau is clearly visible at around 20000 particles. This is one technique but the plateau is frequently not so easy to observe by visual inspection, especially for CCMC and being able to estimate it computationally is useful for analysing large numbers of calculations.

In the beginning of a typical simulation, only the reference is occupied. Its particles then spawn to occupy parts of the remaining Hilbert space, making the total population grow at a greater pace than the population on the reference does. At the plateau point, annihilation, spawning and death balance each other which temporarily leads to a constant total population while the reference population keeps growing. After a bit, the total population grows again and leaves the plateau. It then grows at a smaller or the same rate as the reference population because the system is now converged and the distribution of particles stochastically represents the ground state wavefunction of the system. See *[Spencer12]* and *[Spencer15]* for details.

The ratio of total population to population on the reference therefore peaks at roughly the plateau with respect to the total population. A good way to find the position of the plateau is therefore to look at the ratio of total population to population on the reference vs total population plots and find the position of the peak. We call this "shoulder" plot and the peak, or "shoulder height", is an upper limit for the position of the plateau, see *[Spencer15]*. The shoulder plot for

our example from above is:



The position of the shoulder is at about 20000 which corresponds to the position of the plateau.

Note: *pyhande* contains two functions to estimate the position of the plateau/shoulder: `pyhande.analysis.plateau_estimator()`, which looks for the peak in the shoulder plot *[Spencer15]*, and `pyhande.analysis.plateau_estimator_hist()`, which uses a histogram approach to identifying the plateau *[Shepherd14]*. As a result of the difference in approaches, the former tends to pick up the population at the start of the plateau whilst the latter favours the end of the plateau and is less well suited to cases without a clear plateau.

In this case, `plateau_estimator` gave 18481 with an estimated standard error of 38 for the shoulder height and `plateau_estimator_hist` gave 20155 (rounded to 0 d.p.). The difference is not important as the plateau is not exactly constant; its value to a few significant values is the important quantity.

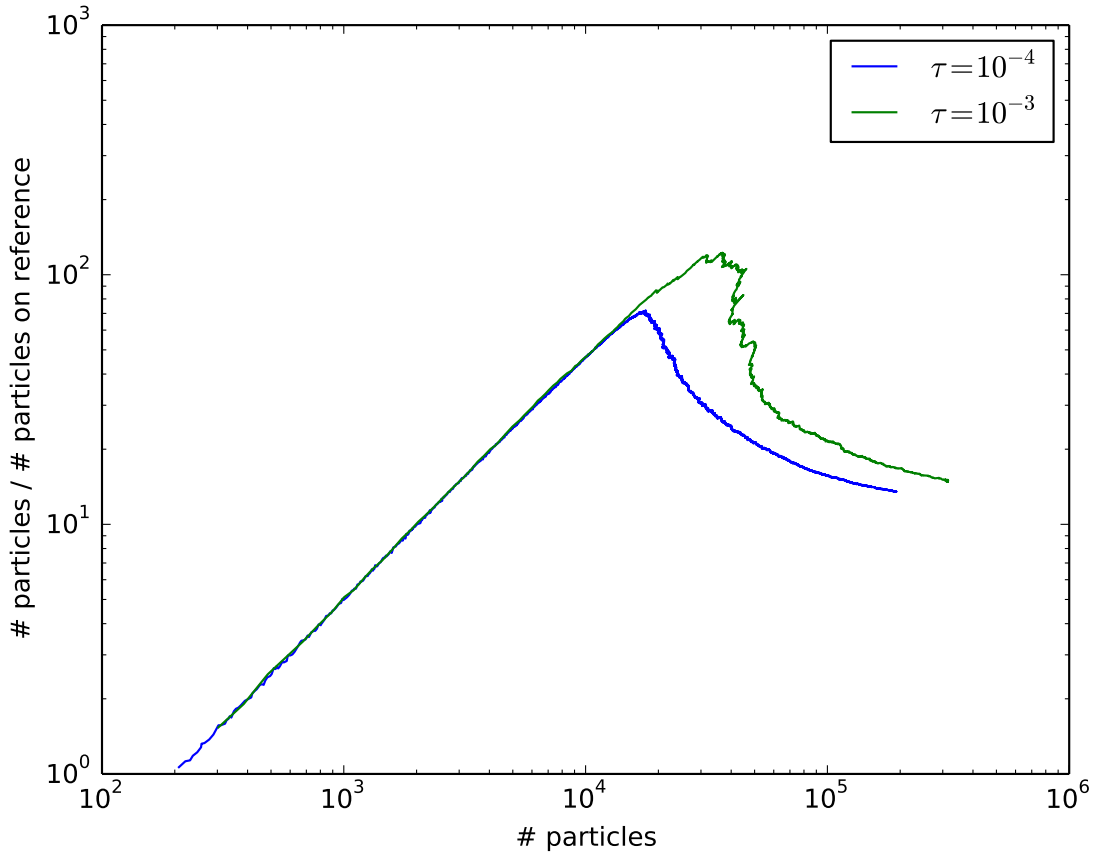
The position of the plateau/shoulder is somewhat sensitive to input parameters and can be varied with changing the time step `tau` or the `cluster_multispawn_threshold` (if applicable) for example, more details below. A large initial population `init_pop` can also lead to overshooting of the shoulder.

Effects of the Time Step

Now we will run another calculation with a higher time step, see input file below:

```
sys = read_in {
    int_file = "H2O_INTDUMP",
    nel = 10,
    ms = 0,
}

ccmc {
    sys = sys,
    qmc = {
        tau = 1e-3,
        mc_cycles = 10,
        nreports = 3.6e3,
        state_size = -500,
        spawned_state_size = -200,
        init_pop = 200,
        real_amplitudes = true,
        target_population = 3e5,
    },
    reference = {
        ex_level = 3,
    },
}
```

The two resulting shoulders are shown in the following graph:



A smaller time step can lead to fewer particles at the shoulder position, as described in *[Booth09]*, *[Vigor16]*.

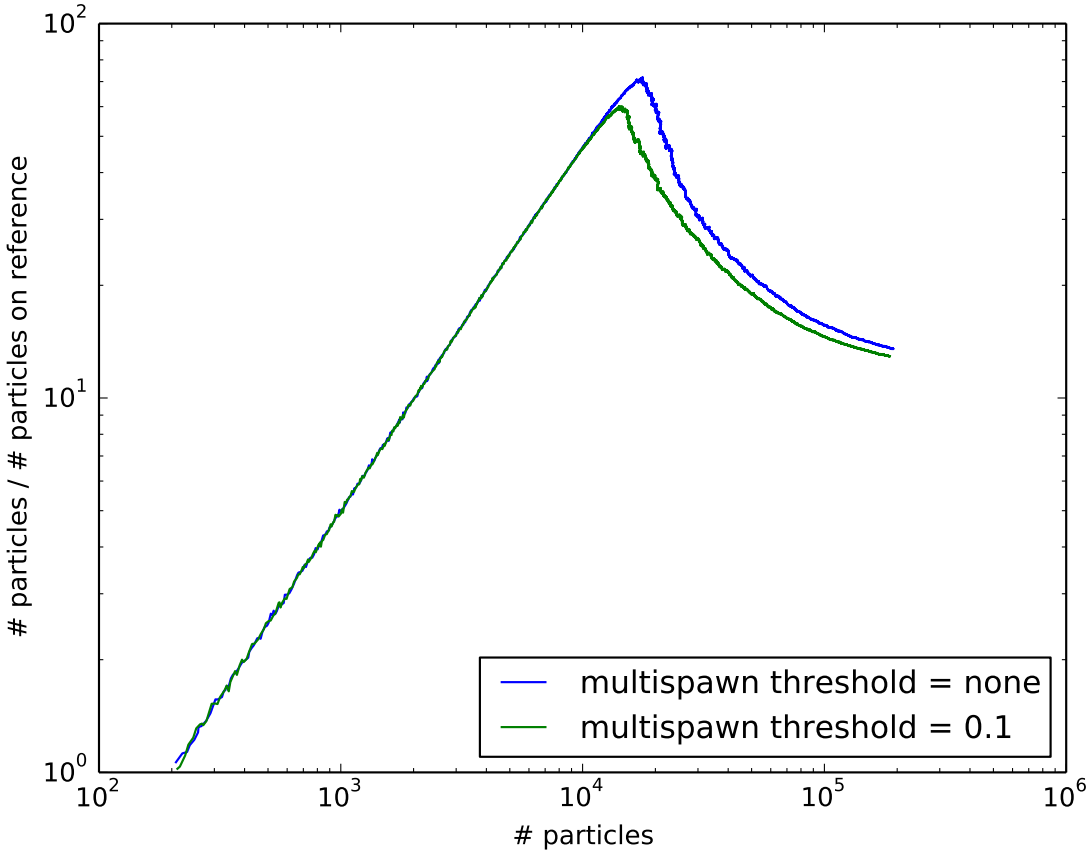
Effects of Cluster Multispawn Threshold

This part looks at changing the multispawn threshold. This is another feature which can change the number of particles at the shoulder. Positive effects of that have already been shown in *CCMC*. Note that while changing the time step changes the position of the plateau for FCIQMC for example as well, cluster multispawn threshold is specific to CCMC. The lower the multispawn threshold, the lower will be the number of "blooming" events which spawn multiple particles at the same spawning attempt. "Blooming" events can lead to greater uncertainty as the wavefunction is then sampled in a more coarse and less fine manner. It is therefore not surprising that less particles are needed to converge to the correct wavefunction for a lower multispawn threshold.

To demonstrate the effects of decreasing the multispawn threshold, we will run the following calculation with a low multispawn threshold:

```
sys = read_in {
    int_file = "H2O_INTDUMP",
    nel = 10,
    ms = 0,
}

ccmc {
    sys = sys,
    qmc = {
        tau = 1e-4,
        mc_cycles = 10,
        nreports = 2e4,
        state_size = -500,
        spawned_state_size = -200,
        init_pop = 200,
        real_amplitudes = true,
        target_population = 3e5,
    },

    ccmc = {
        cluster_multispawn_threshold = 0.1,
    },

    reference = {
        ex_level = 3,
    },
}
```

The plot below compares the shoulder plot of this and the first calculation on top of this tutorial:



Note that "multispawn threshold = none" means that there is no threshold within computer number representation limits.

Clearly, setting a low multispawn threshold lowers the total number of particles at the shoulder. This is, like with a smaller timestep, due to more efficient sampling: an excitor with a large amplitude is allowed to explore more of the space (via multiple spawning attempts) than an excitor with a smaller amplitude.

Effects of Initial Population

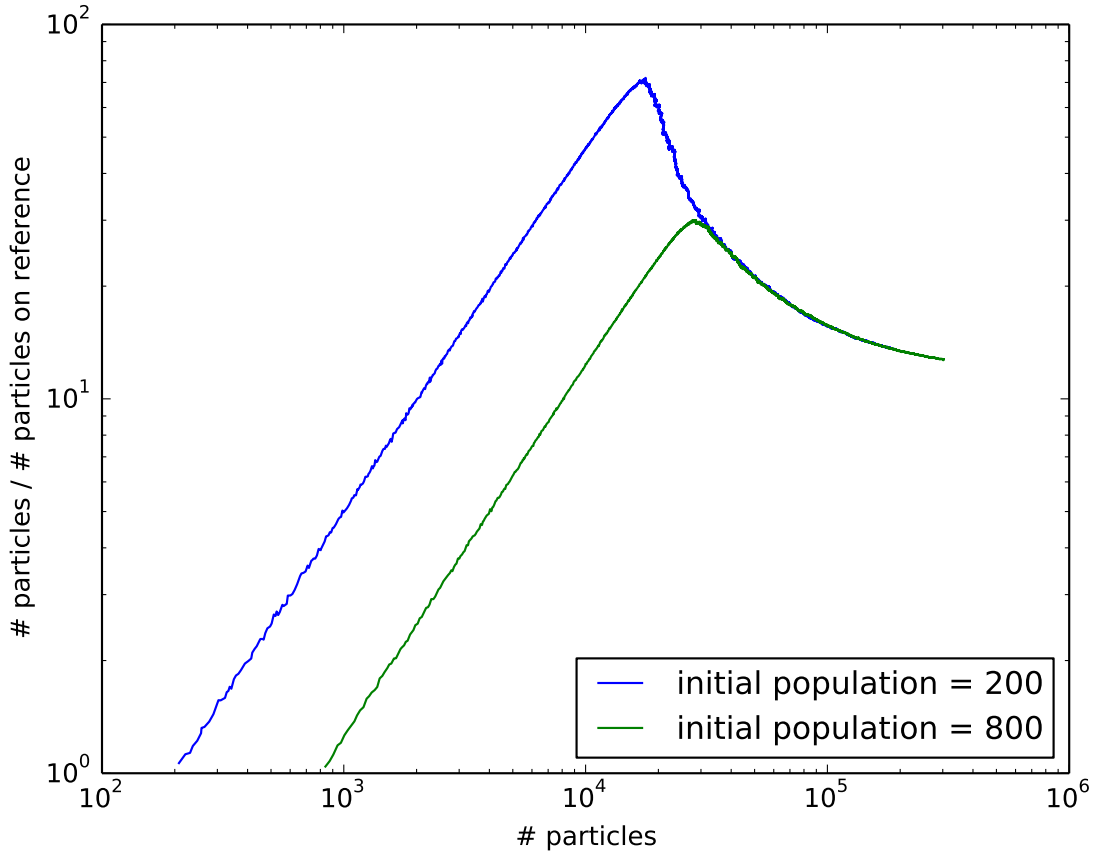
In this part of the tutorial we will see that a large initial population can lead to overshooting the shoulder.

As a demonstration, we look at almost the same calculation as the first one but with a larger initial population.

```
sys = read_in {
    int_file = "H2O_INTDUMP",
    nel = 10,
    ms = 0,
}

ccmc {
    sys = sys,
    qmc = {
        tau = 1e-4,
        mc_cycles = 10,
        nreports = 2e4,
```

```
        state_size = -500,
        spawned_state_size = -200,
        init_pop = 800,
        real_amplitudes = true,
        target_population = 3e5,
    },
    reference = {
        ex_level = 3,
    },
}
```

The following plot compares the original with the calculation starting with a large initial population:



We see that the calculation with a larger initial population has a shoulder at a larger number of particles, effectively overshooting the shoulder. At yet larger numbers of particles than this, we expect the calculation to be stable once population control is enabled (i.e. the shift is allowed to vary).

The overshooting can be explained by considering that the only significant difference between the two curves above is that they start with a different population at the reference. Before they reach a shoulder, each calculation has a very fast growth in total population without changing the reference population. This results in an initial linear growth on the shoulder plots, which lasts until the reference populations begin to grow.

The calculation with the greater initial population will require a greater total population to reach this point, and it occurs when this calculation's curve hits that which begins with a smaller population.

Once a calculation has passed its shoulder, the location on the shoulder plot can generally be used to describe its 'state'. Two calculations with different initial populations, but otherwise identical, will end up on the same curve once

equilibrated, and will follow the curve if total particle numbers are allowed to grow. Modifying the algorithm (e.g. with multispawn_threshold) or changing the timestep will cause the equilibrium curve to shift position, and therefore affect the position of the shoulder.

All calculations were analysed using *pyhande* and all graphs were plotted using matplotlib. Parts of the plot generation code were adapted from the matplotlib tutorials.

CHAPTER 2

pyhande

pyhande provides powerful abstractions for analysing HANDE calculations.

HANDE includes many scripts for common analysis tasks which are (typically) thin wrappers around pyhande. More complicated data analysis or examining large numbers of output files can be easily performed by using pyhande directly from a python interpreter or a custom script.

pyhande can extract data from output produced by HANDE and perform a variety of data analysis tasks on the data obtained. See the documentation for each submodule for more details.

2.1 pyhande.analysis

Analysis of data from FCIQMC and CCMC calculations.

pyhande.analysis.**projected_energy**(*reblock_data*, *covariance*, *data_length*, *sum_key='\\sum H_0j N_j'*, *ref_key='N_0'*, *col_name='Proj. Energy'*)

Calculate the projected energy estimator and associated error.

The projected energy estimator is given by

$$E = \frac{\sum H_0 j N_j}{N_0}$$

The numerator and denominator are correlated and so their covariance must be taken into account.

Parameters

- **reblock_data** (pandas.DataFrame) – reblock data for (at least) the numerator and denominator in the projected energy estimator.
- **covariance** (pandas.DataFrame) – covariance at each reblock iteration between (at least) the numerator and denominator in the projected energy estimator.
- **data_length** (pandas.DataFrame) – number of data points in each reblock iteration.
- **sum_key** (*string*) – column name in reblock_data containing $\sum H_0 j N_j$, i.e. the sum of the population weighted by the Hamiltonian matrix element with the trial wavefunction.
- **ref_key** (*string*) – column name in reblock_data containing N_0, i.e. the population of the trial wavefunction (often/originally just a single determinant).

Returns **proje** – The projected energy estimator at each reblock iteration.

Return type pandas.DataFrame

See also:

pyblock.pd_utils.reblock()

pyhande.analysis.**qmc_summary**(*data*, *keys=('\\sum H_0j N_j', 'N_0', 'Shift', 'Proj. Energy')*, *summary_tuple=None*)

Summarise a reblocked data set by the optimal block.

Parameters

- **data** (pandas.DataFrame) – reblocked data (i.e. data with the reblock iteration as the index).
- **keys** (*list of strings*) – columns (by top-level index) of the data table to inspect. Each top-level column must contain an optimal block column.
- **summary_tuple** ((pandas.DataFrame, list of strings)) – Optionally append summary data to this tuple. Allows repeated calling of this function.

Returns

- **opt_data** (pandas.DataFrame) – Data for each column from the optimal block size of that column.
- **no_opt** (*list of strings*) – list of columns for which no optimal block size was found.

pyhande.analysis.**extract_pop_growth**(*data*, *ref_key='N_0'*, *shift_key='Shift'*, *min_ref_pop=10*)

Select QMC data during which the population was allowed to grow.

We define the region of population growth as the period in which the shift is held constant.

Parameters

- **data** (pandas.DataFrame) – HANDE QMC data. *pyhande.extract.extract_data_sets()* can be used to extract this from a HANDE output file.
- **ref_key** (*string*) – column name in reblock_data containing the number of psips on the reference determinant.
- **shift_key** (*string*) – column name in reblock_data containing the shift.
- **min_pop** (*int*) – discard data entries with fewer than min_pop on the reference.

Returns **pop_data** – The subset of data prior to the shift being varied.

Return type pandas.DataFrame

pyhande.analysis.**plateau_estimator**(*data*, *total_key='# H psips'*, *ref_key='N_0'*, *shift_key='Shift'*, *min_ref_pop=10*, *pop_data=None*)

Estimate the (plateau) shoulder from a FCIQMC/CCMC calculation.

The population on the reference starts to grow exponentially during the plateau, whilst the total population grows exponentially from the start of the calculation before stabilising (perhaps only briefly) during the plateau phase. As a result, the ratio of the total population to the population on the reference is at a maximum at the start of the plateau.

The shoulder estimator is defined to be mean of the ten points with the smallest proportion of the population on the reference (excluding points when the population drops below min_pop excips (psips). The shoulder height is the total population at this point.

Credit to Alex Thom for original implementation.

Parameters

- **data** (pandas.DataFrame) – HANDE QMC data. *pyhande.extract.extract_data_sets()* can be used to extract this from a HANDE output file.
- **total_key** (*string*) – column name in reblock_data containing the total number of psips.
- **ref_key** (*string*) – column name in reblock_data containing the number of psips on the reference determinant.
- **shift_key** (*string*) – column name in reblock_data containing the shift.
- **min_ref_pop** (*int*) – exclude points with less than min_ref_pop on the reference.
- **pop_data** (pandas.DataFrame) – The subset of data prior to the shift being varied. Calculated if not supplied from extract_pop_growth.

Returns **plateau_data** – An estimate of the shoulder (plateau) from a FCIQMC (CCMC) calculation, along with the associated standard error.

Return type pandas.DataFrame

pyhande.analysis.**plateau_estimator_hist**(*data*, *total_key='# H psips'*, *shift_key='Shift'*, *pop_data=None*, *bin_width_fn=None*)

Estimate the plateau height via a histogram of the population.

The population (approximately) stabilises during the plateau phase. By taking a histogram of the population, the plateau can be estimated from the histogram bin with greatest frequency. Due to the exponential population growth outside of the plateau, we histogram the logarithm of the population.

This tends to give similar numbers to shoulder_estimator, though may be less useful for shoulder-like plateaus. Detecting a plateau automatically is tricky so having multiple approaches for comparison helps with corner cases.

Used in *[Shepherd14]*.

Credit to James Shepherd for the idea and original (perl) implementation.

Parameters

- **data** (pandas.DataFrame) – HANDE QMC data. *pyhande.extract.extract_data_sets()* can be used to extract this from a HANDE output file.
- **total_key** (*string*) – column name in reblock_data containing the total number of psips.
- **shift_key** (*string*) – column name in reblock_data containing the shift.
- **pop_data** (pandas.DataFrame) – The subset of data prior to the shift being varied. Calculated if not supplied from extract_pop_growth.
- **bin_width_fn** (*function*) – A function which calculates the bin width in the histogram based upon pop_data. 12500/len(data)^2 (obtained empirically) is used if not supplied.

Returns **plateau** – An estimate of the population at the plateau.

Return type float

References

Shepherd14 J.J. Shepherd et al., Phys. Rev. B 90, 155130 (2014).

2.2 pyhande.canonical

Analysis of data from canonical thermodynamic calculations.

pyhande.canonical.**analyse_hf_observables**(*means*, *covariances*, *nsamples*)
Perform Error analysis for Hartree-Fock estimates which are the ratio of two quantities.

Parameters

- **means** (pandas.DataFrame) – Data frame containing means of verious observables.
- **covariances** (pandas.DataFrame) – Data frame containing covariances between various observables.
- **nsamples** (*int*) – Number of samples contributing to estimates and standard errors

Returns **results** – Averaged Hartree-Fock estimates along with error estimates.

Return type pandas.DataFrame

pyhande.canonical.**estimates**(*metadata*, *data*)
Perform error analysis for canonical thermodynamic estimates.

Parameters

- **metadata** (*dict*) – metadata (i.e. calculation information, parameters and settings) extracted from output files.
- **data** (pandas.DataFrame) – HANDE QMC data.

Returns **results** – Averaged estimates.

Return type pandas.DataFrame

2.3 pyhande.extract

Extract data from the output of a HANDE calculation.

Note: All *pyhande* analysis procedures assume data is in the format produced by *extract_data()* and *extract_data_sets()*.

---

pyhande.extract.**extract_data_sets**(*filenames*)
Extract QMC data tables from multiple HANDE calculations.

Parameters **filenames** (*list of strings*) – names of files containing HANDE QMC calculation output.

Note: Files compressed with gzip, bzip2 or xz (python 3 only) are automatically decompressed.

Returns **data** – Calculation output represented by a tuple for each calculation, consisting of metadata (dict) and a pandas.DataFrame (MC calculations) or pandas.Series (other calculations) containing the calculation output/results.

Return type list of (dict, pandas.DataFrame or pandas.Series)

See also:

extract_data() underlying data extraction implementation.

pyhande.extract.**extract_data**(*filename*)
Extract QMC data table from a HANDE calculation.

Parameters **filename** (*string*) – name of file containing the HANDE QMC calculation output.

Note: Files compressed with gzip, bzip2 or xz (python 3 only) are automatically decompressed.

Returns **data_pairs** – Calculation output represented by a tuple for each calculation, consisting of metadata (dict) and a pandas.DataFrame (MC calculations) or pandas.Series (other calculations) containing the calculation output/results.

Return type list of (dict, pandas.DataFrame or pandas.Series)

2.4 pyhande.lazy

Tools for the lazy amongst us: automation of common HANDE analysis tasks.

pyhande.lazy.**std_analysis**(*datafiles*, *start=0*, *select_function=None*, *extract_psips=False*, *reweight_history=0*, *mean_shift=0.0*, *arith_mean=False*)
Perform a 'standard' analysis of HANDE output files.

Parameters

- **datafiles** (*list of strings*) – names of files containing HANDE QMC calculation output.
- **start** (*int*) – iteration from which the blocking analysis is performed.
- **select_function** (*function*) – function which returns a boolean mask for the iterations to include in the analysis. Not used if set to None (default). Overrides start. See below for examples.
- **extract_psips** (*bool*) – also extract the mean number of psips from the calculation.
- **reweight_history** (*integer*) – reweight in an attempt to remove population control bias. According to *[Umrigar93]* this should be set to be a few correlation times.
- **mean_shift** (*float*) – prevent the weights from beoming to large.

Returns

info – raw and analysed data, consisting of:

metadata, data from *pyhande.extract.extract_data_sets()*. If data consists of several concatenated calculations, then the only metadata object is from the first calculation.

data_len, reblock, covariance from pyblock.pd_utils.reblock(). The projected energy estimator (evaluated by *pyhande.analysis.projected_energy()*) is included in reblock.

opt_block, no_opt_block from *pyhande.analysis.qmc_summary()*. A 'pretty-printed' estimate string is included in opt_block.

Return type list of collections.namedtuple()

Examples

The following are equivalent and will extract the data from the file called hande.fciqmc.out, perform a blocking analysis from the 10000th iteration onwards, calculated the projected energy estimator and find the optimal block size from the blocking analysis:

```
>>> std_analysis(['hande.fciqmc.out'], 10000)
>>> std_analysis(['hande.fciqmc.out'],
...              select_function=lambda d: d['iterations'] > 10000)
```

References

Umrigar93 Umrigar et al., J. Chem. Phys. 99, 2865 (1993).

2.5 pyhande.utils

Utility procedures for manipulating HANDE data.

pyhande.utils.**groupby_beta_loops**(*data*)
Group a HANDE DMQMC data table by beta loop.

Parameters **data** (pandas.DataFrame) – DMQMC data table (e.g. obtained by *pyhande.extract.extract_data()*.

Returns **grouped** – GroupBy object with data table grouped by beta loop.

Return type pandas.DataFrameGroupBy

pyhande.utils.**groupby_iterations**(*data*)
Group a HANDE QMC data table by blocks of iterations.

Parameters **data** (pandas.DataFrame) – QMC data table (e.g. obtained by *pyhande.extract.extract_data()*.

Returns **grouped** – GroupBy object with data table grouped into blocks within which the iteration count increases monotonically.

Return type pandas.DataFrameGroupBy

2.6 pyhande.weight

Attempt to remove the population control bias by reweighting estimates.

pyhande.weight.**reweight**(*data*, *mc_cycles*, *tstep*, *weight_history*, *mean_shift*, *weight_key='Shift'*, *arith_mean=False*)
Reweight using population control to reduce population control bias.

Reweight estimators linear in the number of psips by the factor:

$$W(\tau, N) = \Pi_{m=0}^{N-1} e^{-A\delta\tau S(\tau - m\delta\tau)}$$

where A is the number of steps per shift update cycle, $\delta\tau$ is the time step and $S(\tau - m\delta\tau)$ is the shift at time $\tau - m\delta\tau$, and m is the number of iterations to reweight over.

See *[Umrigar93]* Eqs. 14-20 for details and *[Vigor15]* for use in FCIQMC.

Parameters

- **data** (pandas.DataFrame) – HANDE QMC data.
- **tstep** (*float*) – The time step used in the weight factor.
- **mc_cycles** (*int*) – The number of monte carlo cycles per update step.
- **weight_history** (*integer*) – The number of iterations to reweight over.
- **mean_shift** (*float*) – The mean shift. Used to prevent weights becoming too big.
- **weight_key** (*string*) – Column to generate the reweighting data.
- **geom_mean** (*bool*) – Reweight using the geometric mean

Returns **data** – HANDE QMC data with weights appended

Return type pandas.DataFrame

References

Umrigar93 C.J. Umirigar et al., J. Chem. Phys. 99, 2865 (1993)

Vigor15 W.A. Vigor, et al., J. Chem. Phys. 142, 104101 (2015).

CHAPTER 3

Developers' Guide

Compiled from various email threads on (and before) the nascent hande-dev list. The HANDE project's paper *[Spencer14]* for the 2nd Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE2) at SC14 contains a summary of working practices and our approach for developing software in an academic environment.

3.1 Git

3.1.1 git repository

HANDE can be downloaded by cloning the repository from github:

```
$ git clone https://github.com/hande-qmc/hande
```

We periodically tag releases

A private git repository, where much of the day-to-day development work takes place, is currently located at hande@tycpc15.cmth.ph.ic.ac.uk:hande.git and can be cloned using:

```
$ git clone hande@tycpc15.cmth.ph.ic.ac.uk:hande.git
```

If you would like access, please speak to one of the developers. The rest of this guide assumes you used the default remote name during the clone (i.e. `origin`). If this is not the case, we assume you are capable of appropriately adjusting the commands given in the rest of the guide.

Note: Bug fixes and similar work are applied to both public and private repositories. New features are often developed in the private repository (which hooks into our buildbot server for regression testing), whilst we iron them out. Once we are happy that new features are ready for production use, they will also be migrated to the public repository.

3.1.2 Precepts

- All development happens in branches.
- Branches belong to a relevant namespace (feature/XXX indicates XXX is a branch (name) for a new feature, he/XXX for a HANDE enhancement (he), bug_fix/XXX for a bug fix, config/XXX for a new config file, etc).
- Branches are merged into master after review. Merging between development branches should be avoided.
- Branches should be reviewed by one other person (at least) before merging into master.

- To review, send a pull request email (see git request-pull) to all developers (perhaps including a summary of work in the branch, which is not generated by request-pull!). This should be viewed as starting a conversation on the work.
- Make changes prompted by the review and resend the pull request. (This might take a few iterations.)
- After a happy conclusion to the review, merge into master.

Notes:

- We would like each commit to at least compile but don't expect each commit to be perfect in its own right! This is extremely useful for using git-bisect when investigating regression errors.
- New functionality should be incorporated by new tests. I intend to spend a day soon creating new tests and checking the code coverage (lcov is a wonderful tool) of the test suite.

See http://nvie.com/posts/a-successful-git-branching-model/ for a popular variant on this approach.

The hope is that this approach will lead to better code and also (with a little work) everyone will be more familiar/comfortable with the code that they're not directly working on themselves.

3.1.3 Branch namespaces

A (non-exhaustive!) list of namespaces we use for branches:

he/XXX for an enhancement to HANDE (usually a modification to existing algorithms).

bug_fix/XXX for a bug fix to a specific area of the codebase.

opt/XXX for optimisation work (please include performance details in the commit message!).

feature/XXX for a new feature (generally bigger than an enhancement).

doc/XXX for fixes/enhancements solely to the documentation. (Often this kind of work is coupled to feature/enhancement development work and the documentation is updated directly in the relevant branches consisting mainly of changes to the source code.)

config/XXX for new configuration file(s)/updates to existing configurations.

Obviously there is some overlap between the he, feature and (to a lesser extent) opt namespaces. Broadly speaking, new algorithms or changes to existing algothims which require a new input options are best suited to the feature namespace, speed/memory improvements to opt/ and other improvements (code tidying, logging, etc.) to the he namespace.

3.1.4 How to generate a pull request

First push your work to the relevant branch on the git sever and then generate template text for the pull request:

```
$ git request-pull startref origin [endref]
```

where startref (endref) is the commit you want to be reviewed from (to) and origin is the name of remote configured to the git sever. startref and endref can be any way of referring to a specific commit and endref defaults to HEAD if not given. Usually the branch would have been created from master, in which case you can simply do (even if master has been committed to since the branch was created):

```
$ git request-pull master origin
```

which generates (for example):

```
$ git request-pull master origin
The following changes since commit 7a58a8d1a8f2e8af15df1c9946e7596078649d79:

  Updated the config files for cx2. (2013-12-09 11:07:52 +0000)

are available in the git repository at:

  git@tyc-svn.cmth.ph.ic.ac.uk:hubbard_fciqmc config/cx2

for you to fetch changes up to 1a5522648378f406d3e5fbd87e22e3768da490bc:

  Fixed typo cx2 config comment (2013-12-13 14:35:42 +0000)

----------------------------------------------------------------
William Vigor (1):
      Fixed typo cx2 config comment

 config/cx2 |    2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Copy and paste this text into your email client and send the pull request to hande-dev@imperial.ac.uk (possibly with some additional text describing motivation/benchmark results/etc). If sendmail/exim4/other MTA is set up properly (naturally the CMTH ones are) then

```
$ git request-pull master origin | mail -s "Pull request" hande-dev@imperial.ac.uk
```

works as one would expect.

3.1.5 Merging to master

Here's a workflow to make merging to master simple. Remember that with git it's extremely difficult to make permanently destructive changes so if it goes wrong it can be fixed.

Before you start make sure your code compiles and passes the test suite. Do not merge broken code into master.

Now make sure your master branch is up to date. Here I do this in a fetch then a pull just to see what else has changed. I do a diff to be sure I'm the same as the origin master.

```
[master]$ git fetch
    remote: Counting objects: 340, done.
    remote: Compressing objects: 100% (182/182), done.
    remote: Total 200 (delta 137), reused 47 (delta 16)
    Receiving objects: 100% (200/200), 96.89 KiB, done.
    Resolving deltas: 100% (137/137), completed with 58 local objects.
    From tyc-svn.cmth.ph.ic.ac.uk:hubbard_fciqmc
       c17ef9e..2d8e130  master     -> origin/master
        ...

[master]$ git pull
    Updating c17ef9e..2d8e130
    Fast-forward
     lib/local/parallel.F90       |    9 ++-------
     src/full_diagonalisation.F90 |   30 ++++++++++++++++++------------
     2 files changed, 14 insertions(+), 25 deletions(-)

[master]$ git diff origin/master
```

The blank output from this indicates we're at origin/master.

I'm going to merge the branch bug_fix/rdm_init. Crucially we use the –no-ff flag to ensure that the merge creates a commit on master; this keeps the history clean (by keeping development work in logical chunks after merging) and also makes it very easy to roll-back and revert an entire feature if problems are encounted.

```
[master]$ git merge --no-ff bug_fix/rdm_init
    Merge made by the 'recursive' strategy.
     src/fciqmc_data.f90 |    2 +-
     1 file changed, 1 insertion(+), 1 deletion(-)

[master]$ git log --graph --oneline --decorate | head
    *   647b7dd (HEAD, master) Merge branch 'bug_fix/rdm_init'
    |\
    | * 3c67d81 (bug_fix/rdm_init) Fix uninitialised doing_exact_rdm_eigv breaking fci
    * |   2d8e130 (origin/master, origin/HEAD) Merge branch 'bug_fix/small_fci_mpi'
    |\ \
```

This shows that a new commit has been created on master.

At this point it's possible that the merge needed some manual intervention. It's fine to make these changes directly and commit them in the merge to your local master. If the merge is starting to get messy it might be best to rebase first to make it easier.

Very importantly, you should now compile the code and run the tests, even if the merge completed without any problems — there might be unintented effects. Only continue if the code compiles and the tests pass. If you need to make changes at this point, you can modify your local existing merge commit with

```
[master]$ git commit --amend
```

Now we've made sure that the code works, all we do is push to the main repo

```
[master]$ git push origin master
    Counting objects: 12, done.
    Delta compression using up to 12 threads.
    Compressing objects: 100% (7/7), done.
    Writing objects: 100% (7/7), 705 bytes, done.
    Total 7 (delta 5), reused 0 (delta 0)
    To git@tyc-svn.cmth.ph.ic.ac.uk:hubbard_fciqmc.git
       2d8e130..647b7dd  master -> master

[master]$ git log --graph --oneline --decorate | head
    *   647b7dd (HEAD, origin/master, origin/HEAD, master) Merge branch 'bug_fix/rdm_init'
    |\
    | * 3c67d81 (bug_fix/rdm_init) Fix uninitialised doing_exact_rdm_eigv breaking fci
    * |   2d8e130 Merge branch 'bug_fix/small_fci_mpi'
    |\ \
```

Almost there. We now ought to clean up the namespace to avoid old branch names hanging around (the code of course will always stay).

```
[master]$ git branch --delete bug_fix/rdm_init
[master]$ git push origin --delete bug_fix/rdm_init
```

The list of branches merged into HEAD can be found by doing

```
[master]$ git branch --all --merged
```

All done!

3.1.6 Unwanted experimental branches

Occasionally (frequently?!) we have tried something which didn't work out. If we don't want to keep any of the history, we can simply delete the local (and if necessary) remote branches:

```
$ git branch --delete unwanted_branch
$ git push origin --delete unwanted_branch
```

But what about branches that we don't intend to continue working on in the near future, would like to keep around but without cluttering up the main repository, making it unclear which branches need some TLC before merging? We have a separate repository where such branches can be sent, to be resurrected if desired later. The repository is at hande@tycpc15.cmth.ph.ic.ac.uk:hande_graveyard.git. To push a local branch there:

```
$ git remote add graveyard hande@tycpc15:hande_graveyard.git
$ git push remote graveyard unwanted_branch
```

and then delete the branch (both local and remote) from the main repository using the same commands as before. If the branch is not local, then you can either check it out and then do the push and delete (easier) or use a refspec:

```
$ git push graveyard refs/remotes/origin/unwanted_branch:refs/head/unwanted_branch
```

where origin/unwanted_branch is the remote branch to be moved to the graveyard repository. The branch on origin can then be deleted as before.

3.2 Adding a new test

1. Ensure the test suite passes with the master on your system.
2. Now checkout the branch you're working on where you'd like to add the test.
3. Rebuild HANDE so that the HANDE binary prints out the SHA1 hash of the current commit. Make sure that there are no uncommitted changes to the source directory so that the benchmarks can be reproduced at a later date using the same binary.
4. Inside test_suite find the appropriate directory in which to add your test, or create a new directory, appropriately named, if necessary.
5. Inside this directory create a new directory with a sensible name describing your test, and change to it.
6. Place the input files for your test in the directory. You can have multiple input files in a single directory.
7. git add your directory (this avoids having to separate out files generated during the tests).
8. If you created a directory for a new category of tests then you will probably need to add the directory name in [ ] to the jobconfig file. If not, then the test should already be included through the globbing in jobconfig.
9. If required, pick some appropriate categories to add your test to in jobconfig.
10. Run testcode.py make-benchmarks to create new benchmarks e.g.

```
    $ ../../testcode2/bin/testcode.py make-benchmarks
    Using executable: /home/Alex/code/HANDE/master/test_suite/../bin/hande.x.
    Test id: 09042014-2.
    Benchmark: 288ad50.

    ...

    Failed tests in:
        /home/Alex/code/HANDE/master/test_suite/H2-RHF-cc-pVTZ-Lz
    Not all tests passed.
```

```
Create new benchmarks? [y/n] y
Setting new benchmark in userconfig to be 6d161d0.
```

Hopefully the only failed tests are your new tests (which you've checked).

Alternatively, a better method is to make a benchmark for the new test only:

```
$ ../../testcode2/bin/testcode.py make-benchmarks -ic fciqmc/H2-RHF-cc-pVTZ-Lz

...

Setting new benchmark in userconfig to be: 6d161d0 288ad50.
```

The use of the 'i' flag tells testcode2 to insert the new benchmark at the start of the existing list of benchmarks, as can be seen in this example.

If you leave the 'i' flag out then it will remove all old benchmarks, which we do not want.

11. Now remember to add the benchmark files and the jobconfig and userconfig files to the repository.

```
$ git add userconfig jobconfig fciqmc/*/benchmark.out.6d161d0.inp*
```

where 6d161d0 is the hash of the newly-created benchmark.

12. Do a quick git status to make sure you haven't missed anything important out, and then you're ready to commit the tests:

```
$ git commit -m "Added new test H2-RHF-cc-pVTZ-Lz and benchmark 6d161d0."
```

Remember you're committing to a branch not the master.

13. Push this to the main repository and send round a pull request for review before its to be merged with master.

3.3 Debugging options

There are a couple of compilation options to help with debugging HANDE.

- The `-g` option to `tools/mkconfig.py` enables compiler options for warnings and run-time checking.
- The `-DDEBUG` preprocessor flag enables additional debugging output. Currently this is stack traces when `stop_all` is called to terminate with an error - the addresses given can be converted to `file:line number` information with `addr2line`.

3.4 FAQ

- Is it ever ok to commit directly to master?

  Yes, but only under very restricted circumstances! If in doubt make a branch and let someone else do the merge.

  - I've got a quick bugfix which I've tested - can I commit it to master?

    Well done on the testing. A bugfix should go in a bugfix/XXX branch. It's a single command to create this. Another few commands and you'll have an pull request email to the hande-dev list for review.

  - But it's a really quick fix! Surely it won't hurt?

    If it will affect functionality (and potentially someone else's jobs) then it probably ought to be reviewed! If it's a very minor corner case of which you're certain, then commit to a bugfix branch and then do the

merge yourself. Always do this via a branch - don't commit directly to master. It's sensible to ask the original author if you're fixing their code however.

- But I need to use this fix to make my runs work.

  You can always run from a bugfix branch. Because you've committed it to the central git repository, you'll have access to it everywhere.

- What if I need this bugfix to develop a new feature?

  I don't know. Ask James! One option is to base your subsequent feature branch off the bug fix branch before it's merged into master (git handles merges very well!) or to cherry-pick the bug fix into your feature branch or make enough noise to get the bug fix merged quickly.

- I've added some comments to clear up something.

  This might be ok to commit to master. If you designed the feature/documentation then you're effectively reviewing yourself. If it's somebody else's code it's polite to have consulted someone on this (either by email, or a review branch).

- But I've modified a feature that only I'm using...

  It sounds like this should be in an enhancement branch he/XXX. If only you're using it it's even more important than someone else review it.

- I've accidentally committed some changes to my local master. What do I do?

  Remember that you can always push to a different branch on the main server.

```
$ git push origin master:he/XXX
```

  would push your changes to the he/XXX branch. It's probably better, however to checkout your changes locally to a branch, and then roll back your master, and then commit the branch:

```
$ git checkout -b he/XXX
$ git push --set-upstream origin he/XXX
$ git checkout master
$ git reset --hard origin/master
```

  Note the last command resets your local master to the same state as that on origin. You should adapt the reset command to set your master to point to the desired commit (ie the first commit shared with the new branch he/XXX).

- Ok - I've gone through the review process and I'd like to try to merge to master myself. Is it easy?

  Easy as pie. There's a workflow in the section Merging to master

- I've got a local branch which I've been working on for some time, but I don't want the pain of a large merge at the end.

  This sounds like a workflow problem. Some comments on this:

  - We need to lose the idea of personal branches (note the branch namespace is organised by topic rather than person), even though a branch might be written entirely/mostly by one person. In that sense, long-running development work should be split into small, logical chunks, each of which is attached one-at-a-time in its own branch. We have always regretted having (multiple) long-running branches.
  - When wrenched away from a WIP with only a distant prospect of future free time, a commit and push with light notes is a very worthwhile thing. It's probably even worthwhile committing a plan before committing any actual code. If these are fast and flexible enough they will hopefully not discourage, but actually encourage organization. It might also encourage (*gasp*) collaboration. Perhaps you could create a directory in documentation as a place for such notes/roadmaps, somewhere between Python's PEP system and informal topic-based TODO lists?

  - We are pretty happy for development branches to be regularly rebased against master (*note*: not merged in either direction), to lessen the pain of one final merge between two very disparate branches.

- This is all very well (and I enjoy the Socratic method), but I'm stuck with a huge branch I don't have time to merge. What do I do?

  Commit it as a feature/XXX or he/XXX and ask for help from the hande-dev list.

- How do I review code?

  We're working on a workflow for this. One method is to make a branch (if you're not already in one) and just add comments to the source. It's helpful if the review is part of the git history (even if the comments never actually make it to the master). We currently are using watson-style tags in comments for code review and discussion, for example:

```
! [review] - JSS: How about doing it this way?
! [reply] - AJWT: I thought about it but that causes problems due to X.
```

  where JSS and AJWT are the initials of the reviewer and code author respectively.

- Will *my* code actually get reviewed?

  We're all usually terribly busy and have very little time, but in a group effort a little from each person goes a long way. If you review others' code then they're more likely to review yours. Make it easy to review, by keeping it clean and the features short. Remember, this kind of review is far more lightweight than peer review of publications, and should be able to slot into people's 'free' time. (Each branch is far more lightweight than a paper.) A simple pull-request should be enough to get people to review. This is rather intricately tied in with the idea of project management. Prodding/cajoling/bullying emails are all possible to aid the review

- What happens if no-one replies to the pull request?

  Here are some opinions:

  - I suggest that after an agreed upon time (X working days?) without even a "I'll review but am too busy until next week" reply, the author is free to merge it into master (but should be open to fixes/improvements to that work that others subsequently suggest).
  - Having been burdened with years-long old dirty branches from other projects, merging is certainly vital. I don't think lack of review should stop merging, but it should prompt someone to ask why.
  - I would view it as a sign that the work is stable and relatively complete (for the time being) and is ready to be used by others/in production calculations.

- What about major (long-term) development work? Perhaps anyone engaged in major projects should send out 'pull-requests' to request review of ongoing work periodically?

  Yes.

- Why are we bothering with review? Surely it makes life more difficult?

  In an attempt to avoid heaps of

  1. completely redundant code
  2. untested code
  3. buggy code

  all ending up in master. The main reason is to encourage something resembling a coherent design and prevent someone going off in a (technical) direction others don't agree with/can see major problems with. A big plus is that it helps everyone become familiar with code that they didn't write (which is why doing code review is good for newcomers).

- PhD students are going to be working on this. How do you see the work they produce on a single project over the course of 3 years going? How often should their code be subject to review?

  PhD projects are never one single monolithicproject (or at least shouldn't be!). The amount and frequency of review is probably a function of how experienced a developer is (in general and with HANDE). Remember a pull request can simply be an indication that the developer would like to start a conversation rather than presenting the final result. Developers should also be encouraged to consider how a development task can be broken down into smaller projects, which might well aid design and testing, as well as reducing horrible merge conflicts from attempting to merge long-standing branches.

- How do I signify a 'fine - no need to comment' commit?

  We suggest a pull request to the email list followed immediately by an email announcing that the requester had also merged into master (or perhaps just the latter email).

CHAPTER 4

Bibliography

- search

Bibliography

[Blunt14] N.S. Blunt, T.W. Rogers, J.S. Spencer, W.M.C. Foulkes, Density-matrix quantum Monte Carlo method, Phys. Rev. B 89, 245124 (2014).

[Blunt15] N.S. Blunt, S.D. Smart, J.A.F. Kersten, J.S. Spencer, G.H. Booth, A. Alavi, Semi-stochastic full configuration interaction quantum Monte Carlo: Developments and application, J. Chem. Phys. 142, 184107 (2015).

[Booth09] G.H. Booth, A.J.W. Thom, A. Alavi, Fermion Monte Carlo without fixed nodes: a game of life, death, and annihilation in Slater determinant space, J. Chem. Phys. 131, 054106 (2009).

[Booth10] G.H. Booth, A novel Quantum Monte Carlo method for molecular systems, PhD thesis, University of Cambridge (2010).

[Cleland10] 4. Cleland, G.H. Booth, A. Alavi, Communications: Survival of the fittest: accelerating convergence in full configuration-interaction quantum Monte Carlo. J. Chem. Phys. 132, 041103 (2010).

[Dunning89] Dunning, Thom H., Gaussian basis sets for use in correlated molecular calculations. I. The atoms boron through neon and hydrogen. J. Chem. Phys. 90, 1007 (1989)

[Flyvbjerg89] 8. Flyvbjerg, H. G. Petersen, Error estimates on averages of correlated data, J. Chem. Phys. 91, 461 (1989).

[Franklin16] R.S.T. Franklin, J.S. Spencer, A. Zoccante and A.J.W Thom, Linked coupled cluster Monte Carlo, J Chem. Phys. 144, 044111 (2016).

[Knowles89] P.J. Knowles, N.C. Handy, A determinant based full configuration interaction program, Comput. Phys. Comm. 54, 75 (1989).

[Lee11] 18. (a) Lee, G. J. Conduit, N. Nemec, P. Lopez Rios, and N. D. Drummond, Strategies for improving the efficiency of quantum Monte Carlo calculations", Phys. Rev. E. 83, 066706 (2011).

[Loos13] P.F. Loos, P.M.W. Gill, Uniform electron gases. I. Electrons on a ring. J. Chem. Phys. 138, 164124 (2013).

[Malone15] F.D. Malone, N.S. Blunt, J.J. Shepherd, D.K.K. Lee, J.S. Spencer, W.M.C. Foulkes, Interaction picture density matrix quantum Monte Carlo, J. Chem. Phys. 143, 044116 (2015).

[Malone16] F.D. Malone, N.S. Blunt, E.W. Brown, D.K.K. Lee, J.S. Spencer, W.M.C. Foulkes, J.J. Shepherd, Accurate exchange-correlation energies for the warm dense electron gas arXiv:1602.05104 [cond-mat.str-el].

[Malone16a] F.D. Malone, W.M.C. Foulkes, J.S. Spencer, Improved parallel algorithms for full configuration interaction quantum Monte Carlo, in preparation.

[Overy2014] 3. Overy, G.H. Booth, N.S. Blunt, J.J. Shepherd, D. Cleland, A. Alavi, Unbiased reduced density matrices and electronic properties from full configuration interaction quantum Monte Carlo, J. Chem. Phys. 141, 244117 (2014).

[Petruzielo12] F.R. Petruzielo, A.A. Holmes, H.J. Changlani, M.P. Nightingale, C. J. Umrigar, Semistochastic projector monte carlo method, Phys. Rev. Lett. 109, 230201 (2012).

[Shepherd14] J.J. Shepherd, G.E. Scuseria, J.S. Spencer, Sign problem in full configuration interaction quantum Monte Carlo: Linear and sublinear representation regimes for the exact wave function, Phys. Rev. B 90, 155130 (2014).

[Spencer12] J.S. Spencer, N.S. Blunt, W.M.C. Foulkes, The sign problem and population dynamics in the full configuration interaction quantum Monte Carlo method, J. Chem. Phys. 136, 054110 (2012).

[Spencer14] J.S. Spencer, N.S. Blunt, W.A. Vigor, F.D. Malone, W.M.C. Foulkes, J.J. Shepherd, and A.J.W. Thom, The Highly Accurate N-DEterminant (HANDE) quantum Monte Carlo project: Open-source stochastic diagonalisation for quantum chemistry, arXiv:1407.5407 (2014).

[Spencer15] J.S. Spencer, A.J.W. Thom, Developments in Stochastic Coupled Cluster Theory: The initiator approximation and application to the Uniform Electron Gas, J. Chem. Phys. 144, 084108 (2016), arXiv:1511.05752 [physics.chem-ph].

[Thom10] A.J.W. Thom, Stochastic Coupled Cluster Theory, Phys. Rev. Lett. 105, 236004 (2010).

[Umrigar93] C.J. Umrigar, M.P. Nightingale, K.J. Runge, A diffusion Monte Carlo algorithm with very small time-step errors, J. Chem. Phys. 99, 2865 (1993).

[Vigor15] W.A. Vigor, J.S. Spencer, M.J. Bearpark, A.J.W. Thom, Minimising biases in full configuration interaction quantum Monte Carlo, J. Chem. Phys. 142, 104101 (2015).

[Vigor16] W.A. Vigor, J.S. Spencer, M.J. Bearpark, A.J.W. Thom, Understanding and improving the efficiency of full configuration interaction quantum Monte Carlo, J. Chem. Phys. 144, 094110 (2016).

p

A

E

G

P

Q

R

S