Unravelling the mysteries of NECI

The n-Electron Configuration Interaction solver

Simon Smart, Nick Blunt, Kai Guther and George Booth

November 5, 2019

1 Using NECI

1.1 Getting into the game

1.1.1 Getting the code

The NECI repository is stored on bitbucket. To gain access you need to be invited. Contact one of the repository administrators [Simon Smart (simondsmart@gmail.com), George Booth (george.booth24@gmail.com) and Nick Blunt (nsb37@cam.ac.uk)] who will invite you. If you already have a bitbucket account let the repository administrators know the email address associated with your account.

You will receive an invitation email. Please accept this invitation, and create a bitbucket account as prompted if necessary.

To gain access to the NECI repository, an ssh key is required. This can be generated on any linux machine using the command[1]

```
ssh-keygen -t rsa -b 2048
```

This will create a private (`~/.ssh/id_rsa`) and a public key file (`~/.ssh/id_rsa.pub`).

The private key must be kept private. On the bitbucket homepage, go to account settings (accessible from the top-right of the main page), and navigate to "SSH keys". Click "Add key" and add the contents of the public key. This will give you access to the repository.

You can now clone the code into a new directory using the command

```
git clone git@bitbucket.org:neci_developers/neci.git [target_dir]
```

1.1.2 Required libraries

NECI requires some external software and library support to operate:

MPI
: For builds of NECI intended to be run in parallel, an implementation of MPI is required. NECI has been heavily tested with OpenMPI, and MPICH2 and its derivatives (IBM MPI, Cray MPI, and Intel MPI).

Linear algebra
: NECI makes use of the linear algebra routines normally contained in BLAS/LAPACK. There are a number of different packages which provide these routines, and are optimised for different compilers and platforms. NECI has been built and tested with either the AMD Core Math Library (ACML), the Intel Math Kernel Library

[1] `ssh-keygen` can also generate DSA keys. Some ssh clients and servers will reject DSA keys longer than 1024 bits, and 1024 bits is currently on the margin of being crackable. As such 2048 bit RSA keys are preferred. Top secret this code is. Probably. Apart from the master branch which hosted for all on github. And in molpro. And anyone that wants it obviously.

(MKL), or the more general Basic Linear Algebra Subprograms (BLAS)/Linear Algebra Package (LAPACK) combination.

HDF5 (optional)
To make use of the structured HDF5 format for reading/writing POPSFILES (files storing the population of walkers, and other information, to restart calculations). This library should be built with MPI and fortran support (`--enable-parallel --enable-fortran --enable-fortran2003`).

FFTW (optional)
A small number of options in NECI, which are not enabled by default, require Fast Fourier Transforms. If these are re-enabled then the Fastest Fourier Transform in the West (FFTW3) library is required.

If combinations of these choices are made other than those most commonly used then either the configuration files, or the resultant Makefile, will need to be modified.

On the majority of machines available to the Alavi group and department, the compilation environment is managed using the `module` command. Documentation for that command is available from the command `module help`. The most commonly used command is to load a module, using the command

```
module load <module_name>
```

Installing and configuring the module system on private machines is far beyond the scope of this document. Configuring your user account to use modules may required modifications to your `.bashrc` file, depending on the local machine configuration. Please contact the local IT administrators or Simon Smart for further advice.

A number of standard combinations of modules present themselves. Where an asterisk is presented, any version of the module can be used. Where the version is under specified, the latest module should be used. This will occur by default.

gfortran (Cambridge)
`mpi/mpich2/gnu/1.4.1p1 acml/64/gfortran/*/up`

ifort (Cambridge)
`ifort/64 mpi/openmpi/64/intel12 acml/64/ifort/*/up`

PGI (Cambridge)
`pgi/64 mpi/openmpi/64/pgi12 lapack/64/pgi`
The lapack module might not be available to all users. Please contact Simon Smart if required.

ifort (Max Planck FKF)
`ifort mpi.intel`
Note that the MKL library is included in the ifort module.

ifort (hydra)
`git intel mkl mpi.ibm hdf5-mpi cmake`

1.1.3 Building the code (using CMake)

There are two ways of building NECI. The recommended approach is to use the cmake build system. For legacy purposes, and for more explicitly customised build configura-

tions, the older Makefile system may also be used.

CMake allows building the code in a separate directory. One directory should be used per configuration that is to be built. This module can be a subdirectory of the NECI directory, or otherwise. With the `build` directory as the current working directory, execute

```
cmake [-DCMAKE_BUILD_TYPE=<type>] <path_to_neci>
```

pointing CMake at the root directory of the cloned NECI repository.

At this point CMake will automatically configure NECI according to the currently loaded modules, and available compilers and libraries. If a different set of modules or compilers are to be used a fresh directory should be initialised (or equivalently all contents of the directory deleted).

By default CMake will configure a Release build. If this is not desired an alternative build type may be specified by setting the `CMAKE_BUILD_TYPE` above. The available options are `Debug` (no optimisations, all checking enabled), `Release` (optimisations enabled), `RelWithDebInfo` (same as release, but with debugging symbols retained) or `Cluster` (inter-procedural optimisations enabled, with very long compile times).

If there is an available HDF5 library, which is compiled with support for MPI and for the Fortran compiler in use, then CMake will happily make use of it. Otherwise support for HDF5 POPSFILES will be disabled by default.

The CMake configuration for NECI contains the functionality to download, compile and use HDF5. To do this run CMake with the `-DENABLE_BUILD_HDF5=ON`.

The code is then built using the command

```
make [-j [n]] [neci|kneci|dneci|mneci]
```

The optional flag `-j` specifies that the build should be performed in parallel (with up to an optional number, `n`, threads).

The final argument specifies whether the normal code (`neci`) should be built, the complex code (`kneci`) the double run code (`dneci`) or the multi run code (`mneci`) are built. If not specified, all targets are built.

Options

Whilst every effort has been made to provide NECI with sensible default options, the user may wish to play around further. To (dis)able an option, the following should be passed as an argument to cmake:

```
-DENABLE_<option>=<(ON|OFF)>
```

The following options are available. Where an option is default "on", if the required libraries are not available, the option will be disabled and this will be noted in the build summary.

`BUILD_HDF5`
: Build the hdf5 library from source, and use that instead of one provided by the system.

HDF5
: Make use of hdf5 for popsfiles (default=on).

FFTW
: Functionality requiring FFTW (default=on).

MOLCAS
: Build with the _MOLCAS_ flag (default=off).

MPI
: Build with parallel functionality (default=off).

SHARED_MEMORY
: Use shared memory for storing integrals (default=on).

WARNINGS
: Compile with verbose compiler warnings (default=off).

1.1.4 Overriding configuration options

One of the aims of the CMake tool is to make build configuration as black-box as possible. The build system should normally detect the compilers in use automatically. The detected compilers may not, however, be the ones desired, or the build system may fail to find functionality that exists. The complier to use can be overridden by arguments passed to CMake:

```
cmake -DCMAKE_<lang>_COMPILER=XXX <neci_dir>
```

where <lang> may be Fortran, CXX or C as appropriate. This should specify the command to use which may be a compiler available in the environmental PATH, or and absolute path to the compiler to use.

Once CMake has determined the compiler to use, it determines the compilation and linker flags automatically. A number of overrides have been definied for NECI. These may be found in the cmake/compiler_flags director, and are segregated by files named according to both the compiler vendor and the language.

Compiler flags are added in a specific order. The flags defined in the NECI_<lang>_FLAGS variable are applied to all builds, whereas those in NECI_<lang>_FLAGS_<type> are only applied to builds with the appropriate build type (with the exception of when type is equal to CLUSTER when the flags are appended to those used in RELEASE mode to enable extra inter-file optimisations).

There are also flags to control how things are linked, what options are passed to the compiler to enable compiler warnings, and flags that depend on 32 or 64 bit builds. These should be self explanatory from reading the files in cmake/compiler_flags.

If these defaults are insufficient, the compilation flags may also be overridden. Any arguments passed to cmake of the form

```
cmake -DFORCE_<lang>_FLAGS[_<type>]
```

override the corresponding NECI_<lang>_FLAGS[_<type>] flags. Essentially any NECI_* flag may be overridden on the command line with a FORCE_* flag (although it is possible that some of these have been accidentally ommitted from the implmentation).

Toolchain files

Overriding all of the CMake variables on the command line is cumbersome and error prone. Various sets of overrides can be combined into a toolchain file, which can be passed to CMake:

```
cmake -DCMAKE_TOOLCHAIN_FILE=<toolchain_file> <neci_dir>
```

These toolchain files can specify the entire chain of compilers, flags and libraries if desired. For examples see the toolchains/ directory in the NECI repository.

Additionally to the flags described above, the `CMakeForceCompiler` functionality may be used (over and above just setting the `CMAKE_<lang>_COMPILER` variable). These macros entirely disable the autodetection of compiler properties within CMake (per language), and will require all flags that are not in the `cmake/compiler_flags` directories to be specified manually. As an example:

```
include(CMakeForceCompiler)

CMAKE_FORCE_C_COMPILER       ( gcc GNU )
CMAKE_FORCE_CXX_COMPILER     ( g++ GNU )
CMAKE_FORCE_Fortran_COMPILER ( mpif90 GNU )
```

This will force the use of the commands `gcc`, `g++`, and `mpif90`, and will set the `CMAKE_<lang>_COMPILER_ID` variable to `GNU` such that the compiler flags set in the `cmake/compiler_flags/GNU_*.cmake` files are used. Because this turns off any autodetection, CMake will not autodetect that the c++ standard library needs to be linked in to combine c++ and Fortran files. This will need to be corrected manually, using:

```
set( NECI_Fortran_STATIC_LINK_LIBRARIES stdc++ )
```

Further internally required libraries may be required. In a normal build, these are output in the build summary under "Implicit C++ linker flags". A comprehensive documented example is found in `toolchains/gfortran-openmpi.cmake`.

Archer

As an example, we can consider the Archer supercomputer, located at EPCC in Edinburgh. This machine uses compiler wrapper scripts (written by Cray) to provide much of the functionality automatically, but this defeats the CMake auto-configuration system. To build on archer the cmake command

```
cmake -DCMAKE_TOOLCHAIN_FILE=<neci_dir>/toolchains/archer.cmake <neci_dir>
```

(As we already know about Archer, we autodetect that you are running on it, and CMake will fail with a message containing these instructions). used, then the build will fail.

Compilation on ADA

Begin by clearing out the build environment. At the terminal execute:

```
module purge
module load environments/addons/cmake-2.8.2
```

Next, for GNU:

```
module load environments/programming/gcc-4.8.2
```

Or Intel:

```
module load compilers/intel/15.0.0.090
module load mpi/openmpi/1.8.2/intel15.0-threads
```

then run Cmake as normal.

Overriding packages required for options

There are a number of packages that are required to build NECI (such as something providing a LAPACK-like interface), or which are required to enable certain options (librt is required to enable shared memory).

If the package searching fails, there are a number of variables that can be set on the CMake command line, or in a toolchain file as appropriate:

`NECI_FIND_<package>`
: If this is set to OFF, the package searcher will not be executed, and the associated option will be enabled without disabling the option.

`<package>_FOUND`
: For many of the package searchers, this disables the searching from inside the package rather than outside. In general, the first option is preferred.

`<package>_LIBRARIES`
: The libraries to be linked in for use of the specifide package. If package searching is disabled, then to use the package this needs to be filled in explicitly.

`<package>_DEFINITIONS`
: Any additional compiler flags that are required to use the package.

`<package>_INCLUDE_PATH`
: The location of any C or C++ header files, or fortran module files to be used during compilation

NECI has a some special package finders, called `MPI_NECI`, `LAPACK_NECI` and `HDF5_NECI`. To a large extent they are just wrappers around the underlying finders provided with CMake, but they implement some additional logic, such as automatically substituting MKL for LAPACK, checking the MPI compiler being used, and providing the capacity to build hdf5 in the source tree.

As a result, when overriding these packages, `MPI_NECI`, `LAPACK_NECI` and `HDF5_NECI` should be substituted for `<package>` above.

1.1.5 Configuring builds (Makefile system)

A specific configuration for building NECI is initialised by using the command

```
./tools/mkconfig.py config_name [-g]
```

The configuration names correspond to the configuration files contained in the config directory. If the flag `-g` is used, then a debug configuration will be created, and otherwise an optimised one.

There are a number of different configurations for differing systems, library and compiler setups. The following are some of the more important, and most likely to be used.

`gfortran_simple, ifort_simple`
: These are the basic configurations, set up for the gfortran and ifort compilers. For development these are the most likely configurations to be used. On personal machines the easiest environment to install is gfortran and openMPI, using the `gfortran\_simple` - see note regarding libraries below.

`fkf_ifort`
: The MPI and ifort installation at the FKF in Stuttgart is different to that available in most locations. Use this config file to compile there.

`PC-ifort64-MPI-TARDIS, PC-ifort64-MPI-HYDRA`
: The ifort compiler supports additional (expensive) optimisations during the link stage. For production runs on supercomputers these should be used. The -TARDIS config file is the normal one to use, with the -HYDRA for the differing environment available on HYDRA.

To specify a default configuration file to use on a particular machine, create a symbolic link called `.default` in the config directory to the appropriate configuration file. This allows `mkconfig.py` to be usedwithout specifying the configuration name.

To compile NECI, a number of linear algebra routines are required. The relevant routines are available in the AMD Core Math Library (ACML), the Intel Math Kernel Library (MKL) and in the more widely available combination of Basic Linear Algebra Subprograms (BLAS) and the Linear Algebra Package (LAPACK). The configuration files above make some assumptions about which packages are available, which are not always correct.

In particular, the elements of the linker lines

```
-lacml
-lmkl_intel_ilp64 -lmkl_core -lmkl_sequential
-lblas -llapack
```

are in principle interchangeable. On personal development machines it is easiest to install BLAS and LAPACK, but these are generally less performant so are not used by default. These lines can be substituted in the generated `Makefile` before compilation.

1.1.6 Building the code (Makefile system)

The code is built using the command

```
make [-j [n]] [neci.x|kneci.x|dneci.x|mneci.x|both|all]
```

The optional flag `-j` specifies that the build should be performed in parallel (with up to an optional number, `n`, threads).

The final argument specifies whether the normal code (`neci.x`) should be built, the complex code (`kneci.x`), the double run code (`dneci.x`), the multiple run code (`mneci.x`) or both the normal and complex codes (`both`) or all of the above and various extra utilities (`all`).

1.1.7 Git overview

It is essential if you plan to do developmental work to get familiar with the source-code management software 'git'. The code will get unusable exponentially quickly if all development and new ideas are hacked into the master branch of the code. The nature of research is that most things probably won't work, but you want to implement them and test relatively quickly, without requiring a standard of code that will remain usable in perpetuity. To avoid an inexorable increase in code 'clutter', it is essential to work in 'branches' off the main code. For a more detailed introduction to the git package,

see git-scm.com/book/en/v2/getting-started-git-basics. In short, the workflow should be:

1. Branch off a clean master version to implement something
2. Test and develop in the branch
3. Regularly merge the new code from the master branch into your personal development branch
4. Once satisfied with the development, and that it is an improvement in scope or efficiency of the existing code, ensure it is tidy, commented, documented, as bug-free as possible, and tests added to the test suite for it. This may involve reimplementing it from a clean version of master if it can be done more efficiently
5. Merge code back into master branch

A few potentially useful git commands in roughly the workflow described above:

`git branch`
: See what branch I am on. -a flag for all (inc. remote) branches.

`git pull origin master`
: Update the master branch into the current local repository

`git checkout -b newbranchname`
: Fork off current branch to a new branch called 'newbranchname'

`git commit -a -m 'Commit message'`
: Commit a set of changes for the current branch to your local repository.

`git push origin branchname`
: Push your current local branch called branchname to a new remote branch of the same name to allow access to others and secure storage of the work

`git checkout -b newbranchname --track origin/remotebranch`
: Check out a branch stored on the remote repository, and allow pushing and pulling from the remote repository for that branch.

`git push`
: Push the current branch to the remote branch that it is tracking.

`git merge master`
: Merge the recent changes in master into your local branch (requires a pull first)

`git checkout master`
: Switch branches to the master branch

`git merge newbranch`
: Merge your code in 'newbranch' into your current branch (potentially master)

Each commit should contain one logical idea and the commit message should clearly describe *everything* that is done in that commit. It is fine for one commit to only contain a very minor change. Try and commit regularly and avoid large commits. It is also a good idea to make sure that code compiles before commiting. This helps catch errors that you may be introducing and also allows the use of debugging tools such as git bisect.

It should be noted that the 'stable' branch of the code, automatically merged into from master upon successful completion of nightly tests, is hosted on github on a public repository, and also pushed to the molpro source code. The molpro developers will quickly send us angry emails if poor code gets pushed into it from NECI, and I will be sure to forward complaints onto the relevant parties!

1.2 Calculation inputs

The NECI executable takes one input argument, which is the name of an input file containing the instructions for carrying out the calculation. The input file is organized in blocks, with each block being started and terminated by a dedicated keyword. Each block can contain a number of keywords to specify options. Here, a list of the blocks and their respective keywords is given.

The first line of the input is always `title`, the last line is always `end`.

Some keywords are mandatory, those are marked in red and are given at the beginning of the description of each paragraph. Then come recommended options, marked in blue, followed by further options given in black.

1.2.1 SYSTEM Block

The SYSTEM block specifies the properties of the physical system that is considered. The block starts with the `system` keyword and ends with the `endsys` keyword.

`system`
: Starts the SYSTEM block. Has one mandatory additional argument to specify the type of the system. The options are

  `read`
  : Read in the integrals from a FCIDUMP file, used for ab-initio calculations.

  `hubbard`
  : Uses the Hubbard model Hamiltonian.

  `ueg`
  : Uses the Hamiltonian of the uniform electron gas in a box.

`endsys`
: Terminates the SYSTEM block.

`electrons` n`, nel` n
: Sets the number of electrons to n

`spin-restrict` m
: Sets the total S_z quantum number to $\frac{m}{2}$. The argument m is optional and defaults to 0.

`hphf` s
: Uses a basis of (anti-)symmetric combinations of Slater determinants with respect to global spin-flip. $s = 0$ indicates anti-symmetric combinations, $s = 1$ symmetric

combinations. This is useful to exclude unwanted spin configurations. For example, no triplet states can occur for `hphf 0`.

`sym` k_x k_y k_z s
: Specifies the symmetry of the target state. The first three arguments set the momentum (k_x, k_y, k_z) and are only used for Hubbard and ueg-type systems, the last argument s specifies the irrep within d_{2h} and is only used for ab-initio systems.

`lztot`
: Set the total L_s quantum number. Has one mandatory additional argument, which is the value of L_s.

`useBrillouinTheorem`
: Assume that single excitations have zero matrix elements with the reference. By default, this is determined automatically.

`noBrillouinTheorem`
: Always assume that single excitations have nozero matrix elements with the reference.

`umatEpsilon` ϵ
: Defines a threshold value ϵ below which matrix elements of the Hamiltonian are rounded to 0. Defaults to 10^{-8}.

`diagonaltmat`
: Assume the kinetic operator is diagonal in the given basis set.

`noSingExcits`
: Assume there is no coupling between single excitations in the Hamiltonian.

`rohf`
: Use restricted open-shell integrals.

`read_rofcidump`
: Read the integrals from a ROFCIDUMP file.

`spinorbs`
: Uses spin orbitals instead of spatial orbtials for addressing the integrals. This can be used if the integrals depend on the spin of the orbitals.

`molproMimic`
: Use the same orbital ordering as molpro, mimicking the behaviour of calling NECI from molpro. First nelec/2 orbitals sorted by diagonal elements of the Fock matrix are considered to be occupied, and the rest to be virtual. Each sector is then sorted separately by symmetry labels.

`complexOrbs_realInts`
: The orbitals are complex, but not the integrals. This reduces the symmetry of the 4-index integrals. Only affects `kneci` and `kmneci` calculations.

`complexWalkers-realInts`
: The integrals and orbitals are real, but the wave function shall be complex. Only affects `kneci` and `kmneci` calculations.

`system-replicas` n
: Specifies the number of wave functions that shall be evolved in parallel. The argument n is the number of wave functions (replicas). Requires `mneci` or `kmneci`.

Excitation generation options

`nonUniformRandExcits`
: Use a non-uniform random excitation generator for picking the move in the FCIQMC spawn step. This can significantly speed up the calculation. Requires an additional argument, that can be chosen from the following

  `pchb`
  : Generates excitations weighted directly with the matrix elements using pre-computed alias tables. This excitation generator is extremely fast, while maintaining high acceptance rates and is generally recommended when memory is not an issue.

  `nosymgen`
  : Generate all possible excitations, regardless of symmetry. Might have a low acceptance rate.

  `4ind-weighted`
  : Generate exictations weighted by a Cauchy-Schwarz estimate of the matrix element. Has very good acceptance rates, but is comparably slow. Using the `4ind-weighted-2` or `4IND-WEIGHTED-UNBOUND` instead is recommended.

  `4ind-weighted-2`
  : Generates excitations using the same Cauchy-Schwary estimate as `4ind-weighted`, but uses an optimized algorithm to pick orbitals of different spin, being faster than the former.

  `4ind-weighted-unbound`
  : Generates excitations using the same Cauchy-Schwary estimate as `4-ind-weighted` and optimizations as `4ind-weighted-2`, but uses more accurate estimates, having higher acceptance rates. This excitation generator has high acceptance rates at negligible memory cost.

  `pcpp`
  : The pre-computed power-pitzer excitation generator [2]. Has low memory cost and scales only mildly with system size, and can thus be used for large systems.

`lattice-excitgen`
: Generates uniform excitations using momentum conservation. Requires the `kpoints` keyword.

Hubbard model and UEG options

`cell` x y z
: Sets the lattice size to $x \times y \times z$ sites.

[2] V. Neufeld, A. Thom, J. Chem. Theory Comput.2019151127-140

`kpoints`
: Use momentum conservation. Requires a momentum-space basis.

`U` U
: Sets the Hubbard interaction strengh to U. Defaults to 4.

`B` t
: Sets the Hubbard hopping strengh to t. Defaults to -1.

`twisted-bc` t_1 t_2
: Use twisted boundary conditions with a phase of $t_1 \frac{2\Pi}{x}$ applied along x-direction, where x is the lattice size. t_2 is optional and the additional phase along y-direction, in multiples of $\frac{2\Pi}{y}$.

`real`
: Use a real-space basis for the Hubbard model. Useful for large values of U.

`open-bc` *direction*
: Set the boundary condition in *direction* to open, i.e. no hopping across the cell boundary is possible. *direction* is optional and can be one of `X`, `Y` or `XY`, for open boundary conditions in x-, y- or both directions. If omitted, both directions are given open boundary conditions. Requires a real-space basis.

`ueg-offset` k_x k_y k_z
: Offset (k_x, k_y, k_z) for the momentum grid used in for the uniform electron gas.

`tilt` t_x t_y
: Replaces each site with a tilted t_x by t_y lattice. Recommended to be used only with `cell 1 1 1`.

1.2.2 CALC Block

The CALC block is used to set options concerning the simulation parameters and modes of FCIQMC. The block starts with the `calc` keyword and ends with the `endcalc` keyword.

`calc`
: Starts the CALC block

`endcalc`
: Terminates the CALC block

`time` t
: Set the maximum time t in minutes the calculation is allowed to run. After t minutes, the calculation will end.

`nmcyc` n
: Set the maximum number of iterations the calculation is allowed to do. After n iterations, the calculation will end.

`seed` s
: Sets the seed of the random number generator to s. This can be used to specifically probe for stochastic effects, but is generally not required.

`averageMcExcits` x
: Sets the average number of spawning attempts from each walker to x.

`rdmSamplingIters` n
: Set the maximum number of iterations used for sampling the RDMs to n. After n iterations of sampling RDMs, the calculation will end.

`load-balance-blocks OFF`
: Distribute the determinants blockwise in a dynamic fashion to maintain equal load for all processors. This is enabled by default and has one optional argument OFF. If given, the load-balancing is disabled.

`energy`
: Additionally calculate and print the ground state energy using an exact diagonalization technique.

`averageMcExcits` n
: The number of spawns to attempt per walker. Defaults to 1 and should not be changed without good reason.

`adjust-averageMcExcits`
: Dynamically update the number of spawns attempted per walker. Can be used if the excitation generator creates a lot of invalid excitations, but should be avoided else.

Population control options

`totalWalkers` n
: Sets the targeted number of walkers to n. This means, the shift will be varied to keep the walker number constant once it reaches n.

`diagShift` S
: Set the initial value of the shift to S. A value of $S < 0$ is not recommended, as it will decrease the population from the beginning.

`shiftDamp` ζ
: Set the damping factor used in the shift update scheme to ζ. Defaults to 10.

`stepsSft` n
: Sets the number of steps per update cycle of the shift to n. Defaults to 100.

`fixed-n0` n_0
: Instead of varying the shift to fix the total number of walkers, keep the number of walkers at the reference fixed at n_0. Automatically sets `stepsSft 1` and overwrites any `stepssft` options given.

`targetGrowRate` *grow walks*
: When the number of walkers in the calculation exceeds *walk*, the shift is iteratively adjusted to maintain a fixed grow rate *grow* until reaching the requested number of total walkers.

`jump-shift OFF`
: When entering the variable shift mode, the shift will be set to the current projected energy. This is enabled by default. There is an optional argument OFF that disables this behaviour.

`pops-jump-shift`
: Reset the shift when restarting a previous calculation to the current projected energy instead of using the shift from the previous calculation.

`trunc-nopen` n
: Restrict the Hilbert space of the calculation to those determinants with at most n unpaired electrons.

`avGrowthRate OFF`
: Average the change in walker number used to calculate the shift. This is enabled by default and has one optional argument OFF, which, when given, turns the option off.

Real walker coefficient options

`allRealCoeff`
: Allow determinants to have non-integer population. There is a minimal population below which the population of a determinant will be rounded stochastically. This defaults to 1.

`realSpawnCutoff` x
: Continuous real spawning will be performed, unless the spawn has weight less than x. In this case, the weight of the spawning will be stochastically rounded up to x or down to zero, such that the average weight of the spawning does not change. This is a method of removing very low weighted spawnings from the spawned list, which require extra memory, processing and communication. A reasonable value for x is 0.01.

`realCoeffbyExcitLevel` n
: Allow all determinants up to an excitation level of n to have non-integer population.

`setOccupiedThresh` x
: Set the value for the minimum walker weight in the main walker list. If, after all annihilation has been performed, any determinants have a total weight of less than x, then the weight will be stochastically rounded up to x or down to zero such that the average weight is unchanged. Defaults to 1, which should only be changed with good reason.

`energy-scaled-walkers` *mode* α β
: Scales the occupied threshold with the energy of a determinant. Has three optional arguments and requires `allRealCoeff`. The argument *mode* can be one of EXPONENTIAL, POWER, EXP-BOUND or NEGATIVE and defaults to POWER. Both α and β default to 1 and `realspawncutoff` β is implied.

Time-step options

tau τ SEARCH
: Sets the timestep per iteration to τ. Has one optional argument SEARCH. If given, the time-step will be iteratively updated to keep the calculation stable.

hist-tau-search *c nbins bound*
: Update the time-step based on histogramming of the ratio $\frac{H_{ij}}{p(i|j)}$. Not compatible with the tau τ SEARCH option. The three arguments c, *nbins* and *bound* are optional. $0 < c < 1$ is the fraction of the histogram used for determining the new timestep, *nbins* the number of bins in the histogram and *bound* is the maximum value of $\frac{H_{ij}}{p(i|j)}$ to be stored.

max-tau τ_{max}
: Sets the maximal value of the time-step to τ_{max}. Defaults to 1.

min-tau τ_{min}
: Sets the minimal value of the time-step to τ_{min} and enables the iterative update of the time-step. Defaults to 10^{-7}. The argument τ_{min} is optional.

keepTauFixed
: Do never update τ and the related parameter p_{singles}, p_{doubles} or p_{parallel}.

truncate-spawns n UNOCC
: Truncate spawns which are larger than a threshold value n. Both arguments are optional, n defaults to 3. If UNOCC is given the truncation is restricted to spawns onto unoccupied. Useful in combination with hist-tau-search.

maxWalkerBloom n
: The time step is scaled such that at most n walkers are spawned in a single attempt, with the scaling being guessed from previous spawning attempts.

Wave function initialization options

walkContGrow
: When reading in a wave function from a file, do not set the shift or enter variable shift mode.

defineDet *det*
: Sets the reference determinant of the calculation to *det*. If no other initialisation is specified, this will also be the initial wave function. The format can either be a comma-separated list of spin orbitals, a range of spin orbitals (like 12-24) or a combination of both.

readPops
: Read in the wave function from a file and use the read-in wave function for initialisation. In addition to the wave function, also the time-step and the shift are read in from the file. This starts the calculation in variable shift mode, maintaining a constant walker number, unless `walkContGrow` is given.

readpops-changeref
: Allow the reference determinant to be updated after reading in a wave function.

startSinglePart n
: Initialise the wave function with n walkers on the reference only unless specified differently. The argument n is optional and defaults to 1.

`proje-changeRef` *frac min*
: Allow the reference to change if a determinant obtains *frac* times the population of the current reference and the latter has a population of at least *min*. Both arguments are optional and default to 1.2 and 50, respectively. This is enabled by default.

`no-changeref`
: Never change the reference.

Initiator options

`truncInitiator`
: Use the initiator method [3].

`addToInitiator` *x*
: Sets the initiator threshold to x, so any determinant with more than x walkers will be an initiator.

`senior-initiators` *age*
: Makes any determinant that has a half-time of at least *age* iterations an initiator. *age* is optional and defaults to 1.

`superInitiator` *n*
: Create a list of n superinitiators, from which all connected determinants are set to be initiators. The superinitiators are chosen according to population. n is optional and defaults to 1.

`coherent-superInitiators` *mode*
: Apply a restriction on the sign-coherence between a determinant and any connected superinitiator to determine whether it becomes an initiator due to connection. The optional argument *mode* can be chosen from STRICT, WEAK, XI, AV and OFF. The default is WEAK and is enabled by default if the `superInitiators` keyword is given.

`dynamic-superInitiators` *n*
: Updates the list of superinitiators every n steps. This is enabled by default with $n = 100$ if the `superInitiators` keyword is given. A value of 0 indicates no update. This implies the `dynamic-core` *n* option (with the same n) unless specified otherwise.

`allow-signed-spawns` *mode*
: Never abort spawns with a given sign, regardless of initiators. *mode* can be either POS or NEG, indicating the sign to keep.

`initiator-space`
: Define all determinants within a given initiator space as initiators. The space is specified through one of the following keywords

  `doubles-initiator`
  : Use the reference determinant and all single and double excitations from it to form the initiator space.

[3] D. Cleland, G.H. Booth, A. Alavi, J. Chem. Phys. 132, 041103 (2010)

`cas-initiator cas1 cas2`
: Use a CAS space to form the initiator space. The parameter cas1 specifies the number of electrons in the cas space and cas2 specifies the number of virtual spin orbitals (the cas2 highest energy orbitals will be virtuals).

`ras-initiator ras1 ras2 ras3 ras4 ras5`
: Use a RAS space to form the initiator space. Suppoose the list of spatial orbitals are split into three sets, RAS1, RAS2 and RAS 3, ordered by their energy. ras1, ras2 and ras3 then specify the number of spatial orbitals in RAS1, RAS2 and RAS3. ras4 specifies the minimum number of electrons in RAS1 orbitals. ras5 specifies the maximum number of electrons in RAS3 orbitals. These together define the RAS space used to form the initiator space.

`optimised-initiator`
: Use the iterative approach of Petruzielo *et al.* (see PRL, 109, 230201). One also needs to use either optimised-initiator-cutoff-amp or optimised-initiator-cutoff-num with this option.

`optimised-initiator-cutoff-amp` $x1$`,` $x2$`,` $x3$`...`
: Perform the optimised initiator option, and in iteration i, choose which determinants to keep by choosing all determinants with an amplitude greater than xi in the ground state of the space (see PRL 109, 230201). The number of iterations is determined by the number of parameters provided.

`optimised-initiator-cutoff-num` $n1$`,` $n2$`,` $n3$`...`
: Perform the optimised initiator option, and in iteration i, choose which determinants to keep by choosing the ni most significant determinants in the ground state of the space (see PRL 109, 230201). The number of iterations is determined by the number of parameters provided.

`fci-initiator`
: Use all determinants to form the initiator space. A fully deterministic projection is therefore performed with this option.

`pops-initiator` n
: When starting from a POPSFILE, this option will use the n most populated determinants from the popsfile to form the initiator space.

`read-initiator`
: Use the determinants in the INITIATORSPACE file to form the initiator space. A INITIATORSPACE file can be created by using the write-initiator option in the LOGGING block.

Adaptive shift options

`auto-adpative-shift` t α c
: Scale the shift per determinant based on the acceptance rate on a determinant. Has three optional arguments. The first is the threshold value t which is the minimal number of spawning attempts from a determinant over the full calculation required before the shift is scaled, with a default of 10. The second is the scaling exponent

α with a default of 1 and the last is the minimal scaling factor, which uses $\frac{1}{\text{HF conn.}}$ as default.

`linear-adaptive-shift` σ f_1 f_2
: Scale the shift per determinant linearly with the population of a determinant. All arguments are optional and define the function used for scaling. σ gives the minimal walker number required to have a shift and defaults to 1, f_1 the shift fraction to be applied at σ with a default of 0 and f_2 is the shift fraction to be applied at the initiator threshold, defaults to 1. Every initiator is applied the full shift.

`exp-adaptive-shift` α
: Scales the shift expoentially with the population of a determinant. The optional argument α is the exponent of scaling, the default is 2.

`core-adaptive-shift`
: By default, determinants in the corespace are always applied the full shift. Using this option also scales the shift in the corespace.

`aas-matele2`
: Uses the matrix elements for determining the scaling factor in the `auto-adaptive-shift`. The recommended option to scale the shift.

Multi-replica options

`multiple-initial-refs`
: Define a reference determinant for each replica. The following n lines give the reference determinants as comma-separated lists of orbitals, where n is the number of replicas.

`orthogonalise-replicas`
: Orthogonalise the replicas after each iteration using Gram Schmidt orthogonalisation. This will converge each replica to another state in a set of orthogonal eigenstates. Can be used for excited state search.

`orthogonalise-replicas-symmetric`
: Use the symmetric Löwdin orthonaliser instead of Gram Schmidt for orthogonalising the replicas.

`replica-single-det-start`
: Starts each replica from a different excited determinant.

Semi-stochastic options

`semi-stochastic`
: Turn on the semi-stochastic adaptation.

`pops-core` n
: When starting from a POPSFILE, this option will use the n most populated determinants from the popsfile to form the core space.

`doubles-core`
: Use the reference determinant and all single and double excitations from it to form the core space.

`cas-core cas1 cas2`
: Use a CAS space to form the core space. The parameter cas1 specifies the number of electrons in the cas space and cas2 specifies the number of virtual spin orbitals (the cas2 highest energy orbitals will be virtuals).

`ras-core ras1 ras2 ras3 ras4 ras5`
: Use a RAS space to form the core space. Suppoose the list of spatial orbitals are split into three sets, RAS1, RAS2 and RAS 3, ordered by their energy. ras1, ras2 and ras3 then specify the number of spatial orbitals in RAS1, RAS2 and RAS3. ras4 specifies the minimum number of electrons in RAS1 orbitals. ras5 specifies the maximum number of electrons in RAS3 orbitals. These together define the RAS space used to form the core space.

`optimised-core`
: Use the iterative approach of Petruzielo *et al.* (see PRL, 109, 230201). One also needs to use either optimised-core-cutoff-amp or optimised-core-cutoff-num with this option.

`optimised-core-cutoff-amp` $x1$`,` $x2$`,` $x3$`...`
: Perform the optimised core option, and in iteration i, choose which determinants to keep by choosing all determinants with an amplitude greater than xi in the ground state of the space (see PRL 109, 230201). The number of iterations is determined by the number of parameters provided.

`optimised-core-cutoff-num` $n1$`,` $n2$`,` $n3$`...`
: Perform the optimised core option, and in iteration i, choose which determinants to keep by choosing the ni most significant determinants in the ground state of the space (see PRL 109, 230201). The number of iterations is determined by the number of parameters provided.

`fci-core`
: Use all determinants to form the core space. A fully deterministic projection is therefore performed with this option.

`read-core`
: Use the determinants in the CORESPACE file to form the core space. A CORESPACE file can be created by using the write-core option in the LOGGING block.

`dynamic-core` n
: Update the core space every n iterations, where n is optional and defaults to 400. This is enabled by default if the `superinitiators` option is given.

Trial wave function options

`trial-wavefunction` n
: Use a trial wave function to obtain an estimate for the energy, as described in 1.4. The argument n is optional, when given, the trial wave function will be initialised n iterations after the variable shift mode started, else, at the start of the calculation. The trial wave function is defined through one of the following keywords

pops-trial n
: When starting from a POPSFILE, this option will use the n most populated determinants from the popsfile to form the trial space.

doubles-trial
: Use the reference determinant and all single and double excitations from it to form the trial space.

cas-trial cas1 cas2
: Use a CAS space to form the trial space. The parameter cas1 specifies the number of electrons in the cas space and cas2 specifies the number of virtual spin orbitals (the cas2 highest energy orbitals will be virtuals).

ras-trial ras1 ras2 ras3 ras4 ras5
: Use a RAS space to form the trial space. Suppoose the list of spatial orbitals are split into three sets, RAS1, RAS2 and RAS 3, ordered by their energy. ras1, ras2 and ras3 then specify the number of spatial orbitals in RAS1, RAS2 and RAS3. ras4 specifies the minimum number of electrons in RAS1 orbitals. ras5 specifies the maximum number of electrons in RAS3 orbitals. These together define the RAS space used to form the trial space.

optimised-trial
: Use the iterative approach of Petruzielo *et al.* (see PRL, 109, 230201). One also needs to use either optimised-trial-cutoff-amp or optimised-trial-cutoff-num with this option.

optimised-trial-cutoff-amp $x1$, $x2$, $x3$...
: Perform the optimised trial option, and in iteration i, choose which determinants to keep by choosing all determinants with an amplitude greater than xi in the ground state of the space (see PRL 109, 230201). The number of iterations is determined by the number of parameters provided.

optimised-trial-cutoff-num $n1$, $n2$, $n3$...
: Perform the optimised trial option, and in iteration i, choose which determinants to keep by choosing the ni most significant determinants in the ground state of the space (see PRL 109, 230201). The number of iterations is determined by the number of parameters provided.

fci-trial
: Use all determinants to form the trial space. A fully deterministic projection is therefore performed with this option.

Memory options

memoryFacPart x
: Sets the factor between the allocated space for the wave function and the required memory for the specified number of walkers to x. Defaults to 10.

memoryFacSpawn x
: Sets the factor between the allocated space for new spawns and the estimate of required memory for the spawns of the specified number of walkers on a single processor to x. The memory required for spawns increases, the more processors

are used, so when running with few walkers on relatively many processors, a large factor might be needed. Defaults to 3.

`prone-walkers`
: Instead of terminating when running out of memory, randomly delete determinants with low population and few spawns.

`store-dets`
: Employ extra memory to store additional information on the determinants that had to be computed on the fly else. Trades in memory for faster iterations.

Reduced density matrix (RDM) options

`rdmSamplingIters` n
: Set the number of iterations for sampling the RDMs to n. After n iterations of sampling, the calculation ends.

`inits-rdm`
: Only take into account initiators when calculating RDMs.

`non-variational-rdms`
: Only take into account initiators for the right vector used in RDM calculation. This makes the RDMs non-variational, and the resulting energy is the projected energy on the initiator space.

`no-lagrangian-rdms`
: This option disables the correction used for RDM calculation for the adaptive shift. Use this only for debugging purposes, as the resulting RDMs are flawed.

METHODS Block

The METHODS block is a subblock of CALC, i.e. it is specified inside the CALC block. It sets the main algorithm to be used in the calculation. The subblock is started with the `methods` keyword and terminated with the `endmethods` keyword.

`methods`
: Starts the METHODS block

`endmethods`
: Terminates the METHODS block.

`method` *mode*
: Sets the algorithm to be executed. The relevant choice for *mode* is VERTEX FCIMC to run an FCIQMC calculation. Alternative choices are DETERM-PROJ to run a deterministic calculation and SPECTRAL-LANCZOS to calculate a spectrum using the lanczos algorithm.

1.2.3 INTEGRAL Block

The INTEGRAL block can be used to freeze orbitals and set properties of the integrals. The block is started with the `integral` keyword and terminated with the `endint` keyword.

`integral`
: Starts the INTEGRAL block.

`endint`
: Terminates the INTEGRAL block.

`freeze` n m
: Freeze n core and m virtual orbitals which are not to be considered active in this calculation. The orbitals are selected according to orbital energy, the n lowest and m highest orbitals in energy are frozen.

`freezeInner` n m
: Freeze n core and m virtual orbitals which are not to be considered active in this calculation. The orbitals are selected according to orbital energy, the n highest and m lowest orbitals in energy are frozen.

`partiallyFreeze` $n_{\texttt{orb}}$ $n_{\texttt{holes}}$
: Freeze n_{orb} core orbitals partially. This means at most n_{holes} holes are now allowed in these orbitals.

`partiallyFreezeVirt` $n_{\texttt{orb}}$ $n_{\texttt{els}}$
: Freeze n_{orb} virtual orbitals partially. This means at most n_{els} electrons are now allowed in these orbitals.

1.2.4 KP-FCIQMC Block

This block enables the Krylov-projected FCIQMC (KPFCIQMC) method [4] which is fully implemented in NECI. It requires `dneci` or `mneci` to be run. When specifying the KP-FCIQMC block, the METHODS block should be omitted. This block is started with the `kp-fciqmc` keyword and terminated with the `end-kp-fciqmc` keyword.

`kp-fciqmc`
: Starts the KP-FCIQMC block

`end-kp-fciqmc`
: Terminates the KP-FCIQMC block.

`num-krylov-vecs` N
: N specifies the total number of Krylov vectors to sample.

`num-iters-between-vecs` N
: N specifies the (constant) number of iterations between each Krylov vector sampled. The first Krylov vector is always the starting wave function.

`num-iters-between-vecs-vary` i_{12}, i_{23}, i_{34}...
: $i_{n,n+1}$ specifies the number of iterations between the nth and (n+1)th Krylov vectors. The number of parameters input should be the number of Krylov vectors asked for minus one. The first Krylov vector is always the starting wave function.

`num-repeats-per-init-config` N
: N specifies the number repeats to perform for each initial configuration, i.e. the number of repeats of the whole evolution, from the first sampled Krylov vector to the last. The projected Hamiltonian and overlap matrix estimates will be output

[4] N. S. Blunt, Ali Alavi, George H. Booth, Phys. Rev. Lett. 115, 050603

for each repeat, and the averaged values of these matrices used to compute the final results.

`averagemcexcits-hamil` N
: When calculating the projected Hamiltonian estimate, an FCIQMC-like spawning is used, rather than calculating the elements exactly, which would be too computationally expensive. Here, N specifies the number of spawnings to perform from each walker from each Krylov vector when calculating this estimate. Thus, increasing N should improve the quality of the Hamiltonian estimate.

`finite-temperature`
: If this option is included then a finite-temperature calculation is performed. This involves starting from several different random configurations, whereby walkers are distributed on random determinants. The number of initial configurations should be specified with the num-init-configs option.

`num-init-configs` N
: N specifies the number of initial configurations to perform the sampling over. An entire FCIQMC calculation will be performed, and an entire subspace generated, for each of these configurations. This option should be used with the finite-temperature option, but is not necessary for spectral calculations where one always starts from the same initial vector.

`memory-factor` x
: This option is used to specify the size of the array allocated for storing the Krylov vectors. The number of slots allocated to store unique determinants in the array holding all Krylov vectors will be equal to ABx, where here A is the length of the main walker list, B is the number of Krylov vectors, and x is the value input with this option.

`num-walker-per-site-init` x
: For finite-temperature jobs, x specifies the number of walkers to place on a determinant when it is chosen to be occupied.

`exact-hamil`
: If this option is specified then the projected Hamiltonian will be calculated exactly for each set of Krylov vectors sampled, rather than randomly sampling the elements via an FCIQMC-like spawning dynamic.

`fully-stochastic-hamil`
: If this option is specified then the projected Hamiltonian will be estimated without using the semi-stochastic adaptation. This will decrease the quality of the estimate, but may be useful for debugging or analysis of the method.

`init-correct-walker-pop`
: For finite-temperature calculations on multiple cores, the initial population may not be quite as requested. This is because the quickest (and default) method involves generating determinants randomly and sending them to the correct processor at the end. It is possible in this process that walkers will die in annihilation. However, if this option is specified then each processor will throw away spawns to other processors, thus allowing the correct total number of walkers to be spawned.

`init-config-seeds seed1, seed2...`
: If this option is used then, for finite-temperature calculations, at the start of each calculation over an initial configuration, the random number generator will be re-initialised with the corresponding input seed. The number of seeds provided should be equal to the number of initial configurations.

`all-sym-sectors`
: If this option is specified then the FCIQMC calculation will be run in all symmetry sectors simultaneously. This is an option relevant for finite-temperature calculations.

`scale-population`
: If this option is specified then the initial population will be scaled to the populaton specified with the 'totalwalkers' option in the Calc block. This is relevant for spectral calculations when starting from a perturbed `POPSFILE` wave function, where the initial population is not easily controlled.

In spectral calculations, one also typically wants to consider a particular perturbation operator acting on the ground state wave functions. Therefore, you must first perform an FCIQMC calculation to evolve to the ground state and output a `POPSFILE`. You should then start the KP-FCIQMC calculation from that `POPSFILE`. To apply a perturbation operator to the `POPSFILE` wave function as it is read in, use the pops-creation and pops-annihilate options. These allow operators such as

$$\hat{V} = \hat{c}_i \hat{c}_j + \hat{c}_k \hat{c}_l \hat{c}_m \tag{1.1}$$

to be applied to the `POPSFILE` wave function. The general form is `pops-annihilate n_sum orb1 orb2... ...` where n_{sum} is the number of terms in the sum for $\hat{V}$ (2 in the above example), and *orbi* specify the spin orbital labels to apply. The number of lines of such orbitals provided should be equal to n_{sum}. The first line provides the orbital labels for the first term in the sum, the second line for the second term, etc...

1.2.5 LOGGING Block

The LOGGING block specifies the output of the calculation and which status information of the calculation shall be collected. This block is started with the `logging` keyword and terminated with the `endlog` keyword.

`logging`
: Starts the LOGGING block.

`endlog`
: Terminates the LOGGING block.

`hdf5-popsfile`
: Sets the format to read and write the wave function to HDF5. Requires building with the `ENABLE-HDF5` cmake option.

`popsfile` n
: Save the current wave function on disk at the end of the calculation. Can be used to initialize subsequent calculations and continue the run. This is enabled by default. n is optional and, when given, specifies that every n iteration, the wave function shall be saved. Setting $n = -1$ disables this option.

`popsFileTimer` n
: Write out a the wave function to disk every n minutes, each time overwriting the last output.

`hdf5-pops-write`
: Sets the format to write the wave function to HDF5. Requires building with the `ENABLE-HDF5` cmake option.

`hdf5-pops-read`
: Sets the format to read the wave function to HDF5. Requires building with the `ENABLE-HDF5` cmake option.

`highlyPopWrite` n
: Print out the n most populated determinants at the end of the calculation. Is enabled by default with $n = 15$.

`inits-exlvl-write` n
: Sets the excitation level up to which the number of initiators is logged to n. Defaults to $n = 8$.

`binarypops`
: Sets the format to write the wave function to binary.

`nomcoutput`
: Suppress the printing of iteration information to stdout. This data is still written to disk.

Semi-stochastic output options

`write-core`
: When performing a semi-stochastic calculation, adding this option to the Logging block will cause the core space determinants to be written to a file called CORESPACE. These can then be further read in and used in subsequent semi-stochastic options using the `read-core` option in the CALC block.

`write-most-pop-core-end` n
: At the end of a calculation, output the n most populated determinants to a file called CORESPACE. This can further be read in and used as the core space in subsequent calculations using the `read-core` option.

RDM output options

These options control how the RDMs are printed. For a description of how the RDMs are calculated and the content of the files, please see section 1.7.

`calcRdmOnfly` i *step* *start*
: Calculate RDMs stochastically over the course of the calculation. Starts sampling RDMs after *start* iterations, and outputs an average every *step* iterations. i indicates whether only 1-RDMs (1), only 2-RDMs (2) or both are produced.

`rdmLinSpace` *start n step*
: A more user friendly version of `calcrdmonfly` and `rdmsamplingiters`, this samples both 1- and 2-RDMs starting at iteration *start*, outputting an average every *step* iterations n times, then ending the calculation.

`diagFlyOneRdm`
: Diagonalise the 1-RDMs, yielding the occupation numbers of the natural orbitals.

`printOneRdm`
: Always output the 1-RDMs to a file, regardless of which RDMs are calculated. May compute the 1-RDMs from the 2-RDMs.

`writeRdmsToRead off`
: The presence of this keyword overrides the default. If the OFF word is present, the unnormalised `TwoRDM_POPS_a***` files will definitely not be printed, otherwise they definitely will be, regardless of the state of the `popsfile/binarypops` keywords.

`readRdms`
: This keyword tells the calculation to read in the `TwoRDM_POPS_a***` files from a previous calculation. The restarted calc then continues to fill these RDMs from the very first iteration regardless of the value put with the `calcRdmOnFly` keyword. The calculation will crash if one of the `TwoRDM_POPS_a***` files are missing. If the `readRdms` keyword is present, but the calc is doing a `StartSinglePart` run, the `TwoRDM_POPS_a***` files will be ignored.

`noNormRdms`
: This will prevent the final, normalised `TwoRDM_a***` matrices from being printed. These files can be quite large, so if the calculation is definitely not going to be converged, this keyword may be useful.

`writeRdmsEvery` *iter*
: This will write the normalised `TwoRDM_a***` matrices every *iter* iterations while the RDMs are being filled. At the moment, this must be a multiple of the frequency with which the energy is calculated. The files will be labelled with incrementing values - `TwoRDM_a***.1` is the first, and then next `TwoRDM_a***.2` etc.

`write-spin-free-rdm`
: Output the spin-free 2-RDMs to disk at the end of the calculation.

`printRoDump`
: Output the integrals of the natural orbitals to a file.

1.3 Useful References Containing Technical Details

Original FCIQMC method:

- Fermion Monte Carlo without fixed nodes: a game of life, death, and annihilation in Slater determinant space.
  GH Booth, AJ Thom, A Alavi The Journal of chemical physics (2009) 131, 054106

Quite a bit of symmetries some stuff on the initiator method that is actually implemented:

- Breaking the carbon dimer: the challenges of multiple bond dissociation with full configuration interaction quantum Monte Carlo methods. GH Booth, D Cleland, AJ Thom, A Alavi The Journal of chemical physics (2011) 135, 084104

Linear scaling algorithm, (uniform) excitation generation and overall algorithm of FCIQMC:

- Linear-scaling and parallelisable algorithms for stochastic quantum chemistry GH Booth, SD Smart, A Alavi Molecular Physics (2014) 112, 1855

Density matrices, real walker weights and sampling bias:

- Unbiased Reduced Density Matrices and Electronic Properties from Full Configuration Interaction Quantum Monte Carlo. Catherine Overy, George H. Booth, N. S. Blunt, James Shepherd, Deidre Cleland, Ali Alavi, http://arxiv.org/abs/1410.6047

KP-FCIQMC:

- Krylov-projected quantum Monte Carlo N. S. Blunt, Ali Alavi, George H. Booth, Phys. Rev. Lett. 115, 050603

1.4 Trial wave functions

By default, NECI uses a single reference determinant, $|D_0\rangle$, in the projected energy estimator, or potentially a linear combination of two determinants if the the HPHF code is being used.

$$E_0 = \frac{\langle D_0|\hat{H}|\Psi\rangle}{\langle D_0|\Psi\rangle}. \tag{1.2}$$

This estimator can be improved by using a more accurate estimate of the true ground state, a trial wave function, $|\Psi^T\rangle$,

$$E_0 = \frac{\langle \Psi^T|\hat{H}|\Psi\rangle}{\langle \Psi^T|\Psi\rangle}. \tag{1.3}$$

Such a trial wave function can be used in NECI using by adding the `trial-wavefunction` option to the Calc block. You must also specify a trial space. The trial wave function used will be the ground state of the Hamiltonian projected into this trial space.

The trial spaces available are the same as the core spaces available for the semi-stochastic option. However, you must replace `core` with `trial`. For example, to use all single and double excitations of the reference determinant, one should use the 'doubles-trial' option.

1.5 Sampling excited states with FCIQMC

As well as sampling the ground state, NECI can be used to estimate excited-state properties using an orthogonalisation procedure. Specifically, by performing m FCIQMC simulations simultaneously, the lowest m energy states can be sampled.

To do this, one must use the mneci compilation. Using dneci is not sufficient.

To specify how many states are to be sampled, one should use the `system-replicas` option in the System block of the input file.

Then, the `orthogonalise-replicas` option should be included in the Calc section of the input file. This will tell NECI to orthogonalise the FCIQMC wave functions against each other. States representing high-energy wave functions will be orthogonalised against those representing low-energy wave functions. This prevents higher-energy states being projected to the ground state, and instead allows excited states to be converged upon.

Also, one must tell NECI how to initialise each FCIQMC wave function. It is a bad idea to start from single determinants, as many of these will be poor estimates to the desired excited states, and so convergence will be slow. Instead, one should start from trial estimates to the desired excited states. These trial states are generated by calculating the lowest-energy states within a subspace. Thus, one must simply tell NECI what subspace to use. The options available are the same as for semi-stochastic and trial spaces (see above).

For example, if one wants to initialise from the lowest-energy CISD states, one should use the `doubles-init` option in the Calc block. If one wants to start from the lowest-energy states in a $(10, 10)$ CAS space, you should put `cas-init 10 10` in the Calc block.

Also, single-determinant energy estimators can be give poor results for excited states. Instead, trial wave function-based estimators should be used. This should be done exactly as for the ground state – see above for more details on this. For example, to use CISD wave functions in the trial energy estimators, include both the `trial-wavefunction` and `doubles-trial` options in the Calc block of the input file.

For some example excited-state calculations with NECI, see the test_suite/mneci/excited_state directory and the tests therein.

1.6 Davidson RAS code

NECI has an option to find the ground state of a RAS space using a direct CI davidson approach, which does not require the Hamiltonian to be stored. This code is particularly efficient for FCI and CAS spaces, but is less efficient for CI spaces.

To perform a davidson calculation, put

```
davidson ras1 ras2 ras3 ras4 ras5
```

in the Methods block, inside the Calc block. The parameters ras1-ras5 define the RAS space that will be used. These are defined as follows. First, split all of the spatial orbitals into theree sets, RAS1, RAS2 and RAS3, so that RAS1 contains the lowest energy orbitals, and RAS3 the highest. Then, ras1, ras2 and ras3 define the the number of spatial orbitals in RAS1, RAS2 and RAS3. ras4 defines the minimum number of electrons in RAS1. ras4 defines the maximum number of electrons in RAS3. These 5 parameters define the ras space.

This method will allocate space for up to 25 Krylov vectors. It will iterate until the norm of the residual vector is less than 10^{-7}. If this is not achieved in 25 iterations, the calculation will simply stop and output whatever the current best estimate at the ground state is.

This code should be able to perform FCI or CAS calculations for spaces up to around 5×10^6 or so, but will probably struggle for spaces much larger than this.

The method has only been implemented with RHF calculations and with $M_s = 0$.

1.7 RDM generation

Currently the 2-RDMs can only be calculated for closed shell systems. However, calculation and diagonalisation of only the 1-RDM is set up for either open shell or closed shell systems.

The original theory behind the calculation of the RDMs (including details of parallelisation) can be found in the paper: http://arxiv.org/abs/1410.6047. The most accurate RDM method (which is also unbiased) is the double-run approach, which requires the code to be compiled with the `-D__DOUBLERUN` flag in `CPPFLAGS` in the Makefile. This propagates two completely independent populations of walkers, and calculates an unbiased RDM by taking cross-terms between the two populations.

The calculation of the diagonal elements is done by keeping track of the average walker populations of each occupied determinant, and how long it has been occupied. The diagonal element from Di is then calculated as ¡Ni¿(pop1) x ¡Ni¿(pop2) x [No. of iterations in this block], and this is included every time we start a new averaging block, which can occur when a determinant becomes unoccupied in either population, or when we require the calculation of the RDM energy during the simulation. As such, the exact RDM accumulated is dependent on the interval of RDM energy calculations, but in an unbiased way.

The off diagonal elements are sampled through spawning events, and use instantaneous walker populations. Subtle details in the code are:

1. RDMs take contributions to diagonal elements and HF connections whenever the RDM energy is calculated. As the averaging blocks are now reset at this point too, this change is unbiased. 2. The off-diagonal contributions (both HF connections and other contributions sampled through spawning events) contain contributions from both cross-terms. I.e $N_i(pop1) * N_j(pop2)$ as well as $N_i(pop2) * N_j(pop1)$.

There is also the facility to do a single-run calculation of the RDM. This method is BIASED, so should not be used for high accuracy calculations. However, it is cheaper than the double run method, both in memory and in simulation time, so may be useful for rough-and-ready calculations. Reducing the effect of the bias in the SR method can be done by applying a cutoff to the diagonal contributions, such that contributions are only added in if the average sign of the determinant at the time of adding in the contribution exceeds some preset parameter. When calculating RDMs, the RDM energy will be printed at the end of the calculation, which is one measure of the accuracy of the RDMs. Also printed by default are the maximum error in the hermiticity (2-RDM(i,j;a,b) - 2-RDM(a,b;i,j)) and the sum of the absolute errors.

1.7.1 Reading in / Writing out the RDMs for restarting calculations

Two types of 2-RDMs can be printed out. The final normalised hermitian 2-RDMs of the form `TwoRDM_a***`, or the binary files `TwoRDM_POPS_a***`, which are the unnormalised RDMs, before hermiticity has been enforced. The first are the `2-RDM(i,j;a,b)` matrices, which are printed in spatial orbitals with $i < j$, $a < b$ and $i, j < a, b$. The second are the ones to read back in if a calculation is restarted (they are also printed in spatial orbitals with $i < j$ and $a < b$, but for both i, j, a, b and a, b, i, j because they are not yet hermitian). These are the matrices exactly as they are at that point in the calculation.

By default the final normalised 2-RDMs will always be printed, and the `TwoRDM_POPS_a***` files are connected to the `popsfile/binarypops` keywords - i.e. if a wavefunction popsfile is being printed and the RDMs are being filled, a RDM `POPSFILE` will be also. If only the 1-RDM is being calculated, `OneRDM_POPS/OneRDM` files will be printed in the same way.

1.8 Performing error analysis

Data from an FCIQMC calculation is usually correlated. As a result, standard error analysis for uncorrelated data cannot be used. Instead we perform a so-called blocking analysis (JCP 91, 461). In this, data is grouped into blocks of increasing size until the data in subsequent blocks becomes uncorrelated, to a good approximation.

A blocking analysis can be performed in NECI in one of two ways. Firstly, a rough blocking analysis is performed automatically after a job is finished. The final result is output to standard output and further information about the blocking analysis at various block sizes is output to separate files, such as `Blocks_num` and `Blocks_denom`. This should only be used as a rough and quick estimate as there are issues with this approach. For example, the analysis starts as soon as the shift is turned on. This is before the population has stabilised, and so unusual results can occur in the analysis of the denominator and numerator. Also, data is not taken from the optimal block size.

A better approach for a more careful analysis is to use the blocking script in the utils directory, called blocking.py. The key command is

```
./blocking.py -f start_iter -d24 -d23 -o/ FCIMCStats
```

This will perform a blocking analysis starting from iteration `start_iter`. The analysis should be started only once the energy estimate, (column 11 in `FCIMCStats`) and the numerator and denominator (columns 24 and 25) have stabilised and are fluctuating about some final value. Just because the energy looks stable, it does not mean that the populations is not still growing!

`-d24 -d23'` tells the script to perform the blocking on columns 25 and 24 of the `FCIMCStats` file, which correspond to the numerator and denominator of the energy estimator, respectively. `-o/` tells the script to also provide data for the results of dividing columns 25 and 24, which gives the energy estimate that we want.

Running this will produce a graph of the errors for both the numerator and denominator as a function of the number of blocks (and therefore of the block size). As the block size increases, the error estimates should increase, tending towards the true values. Eventually the estimates will plateau. This indicates that, at this block length, the data in the blocks are uncorrelated to a good approximation, and the error estimate calculated is accurate. The data from this block length should therefore be used.

Each estimate of the error will also have an error on it. As the block length increases this 'error on the error' will increase. One should therefore use the *first* block length where the plateau is reached, so as to minimise the error on the final error estimate.

If no plateau is seen in the plot then the simulation has not been run for long enough, and needs to be continued by restarting from the `POPSFILE`. It can take on the order of $10^5 - 10^6$ iterations to perform an accurate blocking analysis.

The `blocking.py` script will also output the final estimates on the energy at the different block lengths. You should find the blocking length where the errors plateau and read of the final estimates (the rightmost columns) from here.

More information (including example plots, similar to those that `blocking.py` produces) is available at JCP 91, 461.