Developer's guide to NECI

Simon Smart, Nick Blunt, Oskar Weser and George Booth

March 9, 2020

1 Working in NECI

In many ways, NECI is a fairly hostile environment to code, especially for inexperienced software developers, or those who are not familiar with the ideosynchrasies of different versions of FORTRAN/Fortran.

In the following sections we aim to give some general guidance for working in the NECI codebase, and

As a general guideline, programs should be written to fail as loudly and as early as possible. This pushes the job of finding errors and debugging up the tree.

1. Just "looks wrong" to the programmer.
2. Syntax highlighting in the editor makes a mistake stand out.
3. Error detected by compiler.
4. Error detected by linker.
5. Runtime error detected by debug sanity checks through code.
6. Runtime error caused in code at a different location to the bug.
7. Runtime error only occurs when running in parallel, or in bigger calculations.
8. Simulation appears to run correctly, but gives obviously wrong output.
9. Simulation appears to run correctly, but gives subtly wrong output.

We strongly aim to be at the top of the list.

1.1 Code conventions

The code in NECI has been developed over a number of years by many different developers, and has very little standardisation of approach or code appearance. This is not an example to copy!

We are trying to (gradually) normalise sections of code, and isolate those sections which are old and generally unredemable from the rest of the code base. As such there are a number of restrictions we place on code in NECI, and a range of other guidelines.

Fortran standard

Due to the use of procedure pointers, a reasonably up to date compiler supporting (at least some of) the Fortran 2003 standard is *required* to compile NECI. The C-interoperability and procedure pointer features of Fortran 2003 should be used. Other features of this standard should be used sparingly, as Fortran 2003 support in compilers is patchy at best.

Otherwise, code should be written to the Fortran 90/95 standard. In particular, several features of FORTRAN 77 should be avoided at all costs:

- `DO` statements using `REAL` type loop variables.
- Assigned `GOTO` statements
- Cray pointers (declared with the format `pointer (ptr, pointee)`)
- Implicit variables. `implicit none` MUST appear in every module, or interface statement.
- Implicitly typed routines and subroutines. See section on modules and interfaces.
- `COMMON` blocks for sharing data between files.

All of these features do, or have appeared, in NECI at some point. It is also highly advised that `intent` arguments should be used for all argument declarations. This both improves performance, and the ability of others to quickly identify what the routine is attempting to do.

Regarding code layout the PEP8 guidelines of the python language lead to well readable code and are mostly applicable to Fortran as well. (https://www.python.org/dev/peps/pep-0008/)

CAPITAL letters

Fortran is a case insensitive programming language.

For historical reasons a large proportion of FORTRAN 77 code was written entirely in CAPITAL LETTERS (with the exception of displayable strings). This is extremely bad practice.

Humans generally read by recognising word shape. This is obliterated in fully capitalised text, making code much harder to read, and typos especially difficult to identify.

Indentation

Indentation of sections of code should use spaces (and not tabs). The Fortran 95 standard explicitly rejects the use of tabs, and tabs in source code will elicit warnings from the compiler.

All indentations should be multiples of 4 characters.

Source code in ∗.F files (old-style FORTRAN 77) has specific layout restrictions. In particular an initial indent of 7 spaces. This style should not be mimicked elsewhere.

Code line length

The Fortran 90/95 standard restricts line lengths to a (hard) maximum of 132 characters. Code with lines longer than this *may* work on *some* compilers, but this limit should be avoided.

This limit applies after preprocessing has been applied. A number of our macros in `macros.h` can create lines of considerably longer length if not used carefully. These may require using temporary variables with shorter names to control the line length.

The fixed-format FORTRAN 77 code is restricted to 72 characters per line.

On a 19m̈onitor at standard resolution, two columns of code vertically split and side by side use 79 characters each. This is a convenient soft-limit to use - although it is not trivially achievable in all code, and overall readability should be prioritised.

Variable name conventions

There are a number of competing conventions for variable and function names within NECI. That said, there are a number of existing conventions that it is *useful* to be aware of and which new code should keep in mind.

- `CamelCase` or `snake_case` should be used to provide descriptive variable names. Prefer `snake_case` when reasonable. The wider the scope of a variable, the longer and more descriptive the name should be. Trivial local variables (loop indices, etc.) can and should be trivially named. Variables should not, ever, be used entirely in capital letters.
- `tVariableName` is a logical control variable. Normally globally declared in a module for switching on (or off) an overall feature, or signalling overall calculation state.
- `TypeName_t` is a user-defined type.
- `nVariableName` is an integer containing a count of a number of a given entity.
- `Variable`, `AllVariable` are paired sets of variables tracking an extensive property of a simulation. That is properties which can be accumulated on an individual node, but that the system-relevant property needs to be collected from all nodes and amalgamated. This is done once per iteration or once per update cycle as appropriate.
- `CamelCase_t` CamelCase and a trailing t denote a derived type.

Fortran 95 restricts variable names to 31 characters. Although Fortran 2003 extends this to 63, making use of this extension can cause problems with some compilers, and this should be avoided.

Subroutine decoration (especially intent statements)

Subroutine and function declarations should be decorated to the greatest extent feasible. This should restrict the variables to only their expected role in a function.

In particular, all function arguments should be decorated with either `intent(in)`, `intent(out)`, or `intent(inout)` as appropriate. When absolutely necessary, use `value` to pass arguments by-value. The additional decorations `optional` and `target` can be used with care.

The supplied arguments should be as restrictive as possible, to maximise the likelihood of the compiler catching programming errors.

The single exception to this rule is for routines that are to be stored in procedure pointers. These routines must exactly match the definition of the relevant `abstract interface`, which may be more general than is required for the specific case.

The arguments should be sorted by non optional `in`, `inout`, `out` and then optional arguments. The declaration of dummy arguments should appear in the same order as the argument list. There should be an empty line between dummy argument declarations and local variable declarations.

If a procedure is often, but not always, called with the same argument think about making it `optional` using the `def_default` macro and introduce a placeholder local variable with appended underscore.

If possible add the `pure` attribute. This shows the human that there are no side effects and makes parallelization and encapsulation easier.

If a function is `pure` and operates on scalars, add the `elemental` attribute to automatically map it elementwise onto arrays.

The following toy function is pure, can be applied onto arrays and scalars alike. If the exponent is ommited, it defaults to squaring.

```
elemental function pow(x, n) result(res)
    integer, intent(in) :: x
    integer, intent(in), optional :: n
    integer :: n_

    integer :: i

    def_default(n_, n, 2)

    res = 1
    do i = 1, n_
        res = res * x
    end do
end function

pow([1, 3, 5]) -> [1, 9, 25]
pow([1, 3, 5], 3) -> [1, 27, 125]
```

Data types

With the exception of small integers being directly assigned to known integer variables, or used in loop counters, all constants should have their types explicitly specified. The available types are described in section 2.1.

Most specifically, the `*D*` specifier and the floating point type `double precision` should never be used. Examples such as `1.D0` should be replaced with `1.0_dp`, and the data types `real(dp)` and `complex(dp)` should be used.

As compilers have moved between 16-bit, 32-bit and 64-bit, there is ambiguity about whether `double precision` should mean a 32-bit, 64-bit or 128-bit floating point value, depending on the age of the compiler and which compiler is used. This can cause chaos and difficult to track runtime bugs that appear only on certain machines.

For Hamiltonian matrix elements (that may be real or complex depending on build configuration) the custom (preprocessor defined) data type `HElement_t` should be used, which resolves to either `real(dp)` or `complex(dp)`.

Array declarations

Fortran arrays can be declared in multiple ways. In particular, the dimensionality of an array can be declared on the variable itself, or as part of the type declaration;

```
integer, dimension(10, 20) :: arr1
integer :: arr2(10, 20)
```

In general the latter declaration is preferred for two reasons:

1. It is clear that the array property is attached to the variable, and not to the type. When scanning data declarations it is not possible to mistake a scalar for an array.

2. Multiple different arrays can be declared in the same data declaration with different bounds.

The only exception to this is in templated code, where the bounds of arrays need to be varied.

Where arrays are passed as arguments to a routine, they can be passed in three ways

```
integer, intent(inout) :: arr(*)
integer, intent(inout) :: arr(10)
integre, intent(inout) :: arr(:)
```

The first form should be avoided wherever practical, as it prevents any knowledge of the array dimensions being carried into the code. This means that whole-array manipulations will no longer work.

The second two types can be used in different circumstances. The first essentially overrides the array dimensions passed in. This can be useful for re-indexing arrays (e.g. treating a zero-based array as one-based).

The last approach allows a receiving routine to inspect the array bounds as passed in by the calling routine. This maximises the extent to which the compiler and debugging tools can assist in finding errors in the code, and should be used wherever possible.

`use` statements

Globally declared symbols can be shared between modules using `use` statements. Generally, specific symbols should be included rather than all symbols in a module using the notation `use module_name, only: symbol, ...`.

Modules containing only data that have been separated for the purposes of dependency resolution can be fully included into their related modules (e.g. `CalcData` and `Calc`).

Use statements should (where possible) be located in the module header, and not in individual subroutines - this avoids some serious issues associated with compilers resolving conflicting dependencies. If the same thing is included in multiple places in a file, the compilers dependency resolution tree can become very large, and use a lot of time and memory to resolve unambiguously.

`ASSERT` statements

`ASSERT` is a macro, defined in `macros.h`. In an optimised build these statements are entirely removed, and in a debug build they will cause execution to be aborted with an error message if the condition specified is not met.

In NECI, the error message contains the current file and line number. It also includes the current function, which must be manually supplied in a constant named `this_routine`.

An example assert statement, in a function that takes an array with the same number of elements as there are basis functions, would be:

```
subroutine foo(arr)
    integer, intent(inout) :: arr(:)
    character(*), parameter :: this_routine = 'foo'
    ASSERT(size(arr) == nBasis)
    ...
end subroutine
```

☞ Be careful not to use tests with side effects in `ASSERT` statements. In the optimised build the tests will not be called, and this can introduce bugs that appear in only one of the optimised or debug builds.

Floating point comparison and integer division

It is usually a bad idea to test floating point numbers for (in-)equality using `==` or `/=`. Equality should rather be tested with an expression like $|a - b| < \epsilon$. In the `util_mod` module there are `near_zero` and `operator(.isclose.)` which should be used for this purpose.

If one divides two integers `5 / 3 == 1` the result gets truncated to the nearest integer. Sometimes this is not wanted, and the compiler warns about it. For this reason one should use `5 .div. 3` to make it explicit that integer division is indeed wanted.

Tools for adhering to the style guide

☞ It is better to write nice code from the beginning on instead of relying on automated tools. This section is meant for existing code, that has no consistent indentation and other flaws.

One recommended program to prettify Fortran free-format code is `fprettify`. It can be installed with `pip3 install fprettify` and is automatically installed on the Alavi workstations. The `--user` option might be required for installation if you do not have `sudo`-rights.

The NECI codebase already contains the correct configuration files, so it is sufficient to just call `fprettify` on a file in `src/` or `src/lib`.

Operator Layout

The following guidelines are recommendations for formatting of code and represent the configuration of the `fprettify` tool explained in the previous paragraph. These binary operators should be surrounded with a single space on either side: assignment (`=`), comparisons (`==`, `<`, `>`, `/=`, `<=`, `>=`), Booleans (`.and.`, `.or.`, `.not.`).

If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies) to make the expression readable easily. Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

```
! Recommended
    i = i + 1
    x = x*2 - 1
    hypot2 = x*x + y*y
    c = (a+b) * (a-b)

! Also possible
    x = x * 2 - 1
    hypot2 = x * x + y * y
    c = (a + b) * (a - b)

! Not recommended
    i=i+1
```

Line breaks should happen before binary operators for easy association of operator and operand

```
! Not recommended: operators sit far away from their operands
    income = gross_wages + &
             taxable_interest + &
             (dividends - qualified_dividends) - &
             ira_deduction - &
             student_loan_interest

! Recommended: easy to match operators with operands
    income = gross_wages &
             + taxable_interest &
             + (dividends - qualified_dividends) &
             - ira_deduction &
             - student_loan_interest
```

Please use the new C-style relational operators.

```
! Recommended
      ==   /=   <    <=    >   >=
! Not recommended
      .EQ. .NE. .LT. .LE. .GT. .GE.
```

Whitespace in Expressions

Avoid extraneous whitespace in the following situations.

```
! No whitespace immediately inside parentheses:
    Yes: spam(ham(1), f(eggs, 2))
    No:  spam( ham( 1 ), f( eggs, 2 ) )
! No whitespace immediately before the open parenthesis that starts
! the argument list of a function call or array indexing:
    Yes: spam(1)
    No:  spam (1)

! No whitespace immediately before a comma, semicolon, or colon:
    Yes:
        use module, only: cool_function
        integer, allocatable :: A(:, :)
        integer, allocatable :: A(:,:)
    No:
        use module , only : cool_function
        integer , allocatable :: A(: , :)
! No whitespace around the = sign when used to
! call a function with a keyword argument.
    Yes: pow(2, n=3)
    No: pow(2, n = 3)
```

In a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority). In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is omitted.

```
! Recommended
    ham(1:9), ham(1:9:3), ham(:9:3), ham(1::3), ham(1:9:)
    ham(lower:upper), ham(lower:upper:), ham(lower::step)
    ham(lower+offset : upper+offset)
    ham(: upper_fn(x) : step_fn(x)), ham(:: step_fn(x))
    ham(lower + offset : upper + offset)

! Not recommended
    ham(lower + offset:upper + offset)
    ham(1: 9), ham(1 :9), ham(1:9 :3)
    ham(lower : : upper)
    ham( : upper)
```

Contained procedures

From Fortran2003 onwards it is possible to define procedures inside procedures. The inner procedure has access to the local scope of the outher procedure. (Similar to closures in other languages.) These contained procedures allow to cleanly eliminate some reasons, why one would like to use a global variable. Besides it also leads to less wrapper functions that are similar, but not the same. This can be used both to avoid code duplication and to avoid passing unnecessary arguments.

Let's assume there is a `fancy_function` with ten arguments. One of them is a `real, intent(in) :: x`. If `fancy_function` is called several times with only a varying `x` there are many lines of code doing more or less the same. In former versions of Fortran this usually lead to the introduction of a wrapper function `my_fancy_function` that depends explicitly only on `x` and gets the nine other arguments using global variables that have to be defined before calling `my_fancy_function`.

Another code that uses `fancy_function` keeps `x` constant, but varies another argument `y`. This leads to a second wrapper `my_fancy_function_2` and more global variables. In addition both wrapper function might be used only at one place.

If one instead defines the wrapper function as internal procedure only where it is used there is no need for a wrapper function. This resembles the process of *currying* in functional languages.

```fortran
function calculate_something(a) result(res)
    real, intent(in) :: a
    real :: res

    ! arg2 to arg10 are visible identifiers here

    res = exp(f(a)) + 3

contains

    function f(x) result(res)
        real, intent(in) :: x
        real :: res

        res = fancy_function(x, arg2, arg3, ..., arg10)
    end function

end function
```

Another use case for contained procedures is to extract recurrent parts of a subroutine/function without exposing them to module scope.

A contained procedure cannot make use of the `contains` statement.

Modules and interfaces

It is an aim to make the dependency between different code parts as small, as unidirectional and as explicit as possible. Module are a great tool to achieve that goal.

A good example are the different excitation generators of NECI. It is possible to seamlessly exchange different excitation generators because they all have the same public interface. On the other hand they greatly differ in their implementation details and each excitation generator uses different helper functions. It is possible to use an excitation generator without knowing about the implementation details and helper functions. It is even advised to not rely on or assume any implementation detail for a specific excitation generator.

There are some rule of thumbs to achieve similar results in the architecture of other code. If one is implementing a new module it is good to start with the `private` keyword to make all identifiers private to that module and explicitly thinking about which identifiers should be accessable from outside and declare them `public`. These `public` identifiers should only change for a good reason afterwards and should be well documented. Variables that should have read-only access from the outside (a computed energy for example) can be declared `public`, `protected`.

If other modules are imported with `use`, `only`: then it is easy to see on which code a module relies.

If there are generic functions it is possible to declare only the name of the generic interface as `public` and keep the concrete implementations `private`.

The use of `private` helper functions has the same benefit as contained procedures. It allows to write ad-hoc wrappers that have access to the module scope, but do not require `public` global variables.

If possible functions should be declared `pure` or `elemental`.

Example module layout

A sample module layout is given below:

```
#include "macros.h" ! This enables use of our precompiler macros.
module module_name

    ! To the extent possible, include statements should be at the
    ! beginning of a module, and not elsewhere.
    ! If possible they should import only actually used identifiers.
    use SystemData, only: nel, tHPHF
    use module_data
    use constants
    implicit none

    ! To the extent possible, declare all identifiers of
    !   a module as private by default
    !   and export explicitly with the public keyword.
    private
    public :: sub_name, fn_name, calculated_energy
    ! Cannot be changed from the outside.
    protected :: calculated_energy

    real(dp) :: calculated_energy


    ! Add an interface to an external (non-modularised) function
    interface external_fn
        function splat_it(in_val) result(ret_val) &
                                  bind(c, name='symbol_name')
            ! n.b. interface statements shield from modular includes
            import :: dp
            implicit none
            integer, intent(in) :: in_val
            real(dp) :: ret_val
        end function
    end interface

contains

    [pure|elemental] subroutine sub_name(in_val, out_val)

        ! This is a description of what the subroutine does
        integer, intent(in) :: in_val
        real(dp), intent(out) :: out_val

    end subroutine [sub_name]


    [pure|elemental] function fn_name(in_val) result(ret_val)

        ! This is a description of what the function does

        integer, intent(in) :: in_val
        real(dp) :: ret_val

    end function [fn_name]

end module
```

1.2 Review guidelines

Any new code which is to be added to the master branch of NECI has to undergo code review to check if it does not introduce new bugs into existing code and if the code is written in an understandable and maintanable way, and following the code conventions introduced in section 1.1.

The process of adding new code to the program contains these steps

1. Create a new git branch and add your code there. Do not forget to add tests, so the functionality can be verified.
2. Push the branch to the bitbucket repository
3. Create a pull request to master/devel (depending on where you want the code merged), selecting a set of reviewers.
4. The pull request triggers a pipeline that tries to compile the program and run a set of tests with the new code. If any of those fail, you will get a notification and you can check the reason therefore.
5. The reviewers will now check the code and can comment on it. Make sure to address these comments.
6. Once you got the approval of at least one reviewer, you can merge the code. It is now included in the program.

If you have been selected as a reviewer and decide to do the review, check the pull request. It will contain information on the ran tests and a list of all changes to the code. Go through the new code and check if it is written in a clean and well-commented way, in accordance with the code conventions. Does it have tests? You do not have to verify that the new code is bug-free nor do you need to debug it, that is not within the scope of the review. If you find something that could be improved, make a comment on that or create a task, this helps the author to increase the quality of the code.

1.2.1 Using CTAGS with VIM

It is useful, especially for new developers, to be able to easily navigate through NECI code. A simple solution for Vim users is to generate a `tags` file containing the names of all functions and global variables and their locations. Vim automatically reads this file from the current directory, if it exists, and use it to facilitate code navigation. Then you can jump to the deceleration of a variable or a function by putting the cursor over it and pressing `Ctrl+]` . To go back, press `Ctrl+t`. Other tag-related commands are explained here: https://vim.fandom.com/wiki/Browsing_programs_with_tags.

To generate the tags file, a program called `ctags` is needed. It is installed by default in many Linux distributions, but this version is most probably the one called `ctags (GNU Emacs)` and does not support the options we need. The required version is `Exuberant Ctags` or its derivative `Universal Ctags` which you can be downloaded from here: https://github.com/universal-ctags/ctags.

Once you installed the correct version and made sure it is the default one, [1] go to the source directory of NECI and run the script `gen_vim_tags.sh` which is available in the tools directory

```
 cd neci/src
../tools/gen_vim_tags.sh
```

This script does some tricks using the preprocessor to solve issues with handling macros in NECI files. Without these, `ctags` would miss many variables due to parsing issues. The script generates the necessary `tags` file in the current directory and code navigation should become available for all source files in this directory.

Note `tags` is simply a text file with the symbol's name, the file where its defined, and the line number . There is no automatic magic happening behind the scenes as one would expect from a full-fledged IDE. *Therefore, whenever the code changes, you need to explicitly re-generate the* `tags` *file.* Otherwise, Vim will simply jump to the old positions of the symbols.

Tip Another useful tool is `Tagbar` plugin for Vim which lists all functions/variables in the current file in a side window. Using this plugin does not require the `tags` file, because it generates its own tags on-the-fly. However, you will need to install an updated version of `ctags` executable. All necessary details are explained on the plugin's homepage: https://github.com/majutsushi/tagbar

EMACS Usage Like Vim, Emacs accepts `TAGS` file (notice the capital letters). This file can similarly be generated using `gen_emacs_tags.sh`, then you can use `Meta+.` to jump to definition and `Meta+*` to go back. Other tag-related commands are explained here: https://www.emacswiki.org/emacs/EmacsTags.

1.3 Don't Repeat Yourself (DRY)

When information becomes duplication in software, eventually the people that knew about the duplication will forget. And then the information will be changed — but at least one of the duplicates won't be. This introduces bugs that are extremely difficult to track down.

[1] Executing `ctags --version` should print either `Universal Ctags` or `Exuberant Ctags` but not `ctags (GNU Emacs)`

The Don't Repeat Yourself (DRY) principle is one that say a developer should systematically, and always, avoid duplication of information.

NECI is a terrible example of this, but it has been improved over time with a lot of effort.

Information is a very broad term. There are many types of duplication that can occur. A non exhaustive list of some types of duplication (and what can be done about them) follows.

Algorithm duplication across data types

There are many algorithms that are either the same, or similar, across many different data types. The logic involved in these should be written once.

Major examples are the sorting routines in `sort_mod`, general utilities in `util_mod`, shared memory in `shared_alloc` and MPI routines in `Parallel_neci`. Prior to implementing a generalised quicksort there were 37 different sort routines in NECI, using different sort methods, and containing different bugs.

This duplication should be controlled using templating, as described in section 1.7.

Logic duplication across source files

If the same chain of decision making is recurring in different regions of the code, these should be abstracted into their own subroutine which is called from each location. This prevents the logic being duplicated, and then diverging.

Numerous bad examples of this still persist in NECI.

Duplication of data

Compile time constants should only be specified in one place. A large proportion of these are found in the `lib/cons_neci.F90` source file. Other examples include the layout of the bit representations (`BitReps.F90`). A significant proportion of these vary depending on the compile configuration, and prior to collecting them here the code was extremely fragile.

Ongoing cases which are problematic include the use of the literal constant 6 to specify output to stdout in statements such as `write(6,*)`, which doesn't interact well with `molpro`.

Duplication of representations in memory

It is important to have a well defined canonical representation of data in memory. The same data should not be allowed to become duplicated in multiple places.

Temporary arrays, with working data copied into them, should be clearly temporary and discarded as soon as not necessary. If the primary data shifts to a new location, the old storage should be deallocated (if possible), or damaged so that attempts to use it fail loudly (such as putting a value of -1 into a variable that would normally hold an index).

Avoid situations where code might work by accident.

An ongoing situation of this type is the array variable `nBasisMax`. It shadowed a large number of global control variables, and there are still locations in the code where its value is used in preference to the global control value as these values diverge, and its value is the one that works in some obsolete code.

There is one, major, exception to this rule. Code that only exists for testing purposes (such as the contents of `ASSERT` statements, or unit tests) may be as explicitly duplicated as desired. Their *purpose* is to explicitly flag up when anything elsewhere changes - so the risk of them getting out of sync with the code base is their purpose.

1.3.1 Procedure pointers (function pointers)

One issue that becomes immediately obvious when repeated logic has been abstracted into specific functions is that conditional logic is executed *every* time certain actions are taken.

In many cases this is not very important, as it occurs high up the call hierarchy, and the controlled code consumes the vast majority of the execution time. However, the closer we get to the inner most tight loops, the more expensive repeated conditional logic becomes. This is particularly frustrating if the decision making is based on global control parameters, and thus always results in the same code path being taken in a simulation. If the decision lies against the branch prediction metrics, then this is especially bad.

The canonical example of this is accessing the 4-index integrals, which is performed very frequently.

In these case, it is a good idea to separate the decision making logic from the execution, so that the conditional logic is only executed at runtime. This can be done using *procedure pointers* (called function pointers, or similarly functors in other programming languages.

These require defining the "shape" of a function call (i.e. its arguments, and return values) in an `abstract interface`. A variable can then be set to point at which of a range of functions with this signature should be executed.

In NECI the global controlling procedure pointers are located in the module `procedure_pointer`, which contains both the abstract interface definitions, and the actual pointer variables. These variables can then be used as functions throughout the code.

These procedure pointers are largely initialised in the routine `init_fcimc_fn_pointers`, where decisions are made between types of excitation generator, matrix element evaluation, etc. The procedure pointers involved in integral evaluation are set in `init_getumatel_fn_pointers`.

The use of procedure pointers generates a strict Fortran 2003 dependency for NECI. We used to make use of a hacky abuse of the linker and templating system to implement function pointers without language support, but this was deprecated once compiler support for procedure pointers was reasonably widespread.

1.4 Let the compiler help you

If real NECI is compiled in debug mode, warnings will be treated as errors. This means that code with real numbers may not produce any warnings to be merged into main development branches. The complex version of the code however, does not treat warnings as errors.

Unfortunately the warnings for unused variables had to be deactivated, because there are too many incidents.

☞ There are too many conversion warnings in the complex NECI code that could not be cleaned up yet and probably lead to serious bugs. It is necessary to clean them first before complex NECI can be used reliably.

Sometimes a warning is a false-positive. To work around such problems there is a `__WARNING_WORKAROUND` compile flag that gets activated, if warnings are activated.

A common false-positive warning is an unused variable that has to stay in the code, because it is e.g. in the interface of a function that is the target of a function pointer. For this case the `unused` macro exists. It is not necessary to put this macro behind the `__WARNING_WORKAROUND` compile flag. It is recommended to mark unused variables directly after declaration to make it explicit to the human reader.

```
#include "macros.h"
integer, intent(in) :: arr1(:), n
real(dp), inten(in) :: arr2(:, :)
integer :: ierr

unused(arr2); unused(n)
```

1.5 Don't optimise prematurely

Obviously, good algorithm design is important. If an algorithm scales badly, then no implementation will be able to salvage it.

However, there are many tricks and optimisation that can be made to eek out small and large performance gains in the implementation of a particular algorithm. It is important not to optimise too early for a number of reasons.

- Good optimisation is extremely time intensive. On the whole time is better spent getting the code to work, and making the algorithm efficient. Once an implementation works, then code can be profiled and performance improved.
- Optimisations are often highly non-obvious, involving storing information in unexpected ways and places, leading to code that is hard to write, harder to read and extremely bug prone.
- The compiler is very good. The obvious 'tricks' that you see will be done by the compiler anyway.
- One place where performance gains can legitimately be made is in avoiding conditional switching. In many cases this would involve duplicating code paths, and horrifically breaking the DRY principle above. There are occasions that this is worthwhile, but this should be actively justified by profiling data rather than just a hunch.

1.6 Tracking memory usage

NECI contains automated tracking of memory usage. This enables output statistics to indicate which memory uses are dominating during a calculation.

The MemoryManager module keeps track of all units of memory, and assigns a tag value to each of them. It is the responsibility of the developer to store this tag, and pass it when the memory is deallocated. This tag is an integer.

Memory should always be allocated using error checking. That is, an allocate statement should always be passed an error value as follows

```
integer, allocatable :: arr1(:)
real(dp), allocatable :: arr2(:,:)
integer :: ierr
allocate(arr1(10), arr2(20, 30), stat=ierr)
```

This value will be zero if the allocation was successful, and non-zero otherwise. The memory logging routines check this value, and report an error if the memory allocation failed.

Memory is logged using the functions LogMemAlloc and LogMemDealloc.

1.7 Code templating

NECI supports two ways of templating code, the python-based Fortran preprocessor fypp (, https://github.com/aradi/fypp) and the custom script tools/f90_template.py. It is strongly suggested for new code to make use of fypp, which allows for handling preprocessor flags using python syntax, and provides an easy route to templating code with little effort. All files named *.fpp have the fypp preprocessor applied. An in-depth documentation can be found at https://fypp.readthedocs.io/en/stable/.

As fypp support is a recent addition, most of NECI's templates are contained in *.F90.template files, which are automatically converted by tools/f90_template.py into files with the corresponding names *.F90 prior to running the C preprocessor.

This mechanism exists to allow general code to be written, and reused for different types, and combinations of types. While this is largely a combinatorial pattern-matching and substitution problem, the templater contains specific additional features to facilitate dealing with array types in Fortran. There are also a number of specific considerations that need to be made.

The templated code in NECI has been written largely by Simon Smart and Alex Thom, who should be able to help with any particularly nasty issues arising.

1.7.1 How it works

Fortran permits multiple routines to be referenced by the same name through the use of interface blocks such as

```
interface sub_name
    module procedure actual_name_1
    module procedure actual_name_2
end interface
```

which allows either of the routines actual_name_1 or actual_name_2 to be called using the Fortran symbol sub_name. Note that these procedure constructs can be used directly in the code for a hard-coded set of routines which can be called from one interface name if desired (see section 1.7.7).

It is perfectly acceptable to have multiple interface blocks for a specific routine name, so long as all of the referenced routines have different calling signatures. That is, they must accept differently typed arguments so that it is possible for the compiler to determine *at compile time* which of the routines should be called. In principle the actual routine names can always be used.

The Fortran templater creates one module per specified configuration, each with a unique module name. It then performs substitutions on a specified template model to create routines for all of the specified combinations of input types. These routines have their names adjusted to make them unique for each configuration, and an interface block is created to make them accessible under their original name. Finally all of these newly created modules are collected, with `use` statements, into a macroscopic module which can be used from elsewhere.

1.7.2 Overall structure

A sample templated module structure is given here for reference. The different sections are explained below.

```
# This is the configuration block. Note that it has *.ini syntax,
# and that comments are preceeded by hashes.
[int]
type1=integer(int32)

[float]
type1=real(dp)

===================
#include "macros.h"

module module_name

    ! This is the module which is templated to generate the ensemble
    ! of routines with differing types
    use constants
    implicit none

contains

    elemental function test_fn(arg) result(ret)

        %(type1)s, intent(in) :: arg
        %(type2)s :: ret

    end function

end module


supermodule module_name
    !
    ! Here we include code that should be included in the module but
    ! does not need to be templated.
    !
end supermodule
```

1.7.3 Configuration names and substitution

The top section of the file defines the templated configurations. It has the structure of an INI file, and is processed by the standard python ini file parser.

Configurations are defined by a name, contained in square brackets, and then by a series of key-value pairs. All values are treated as strings for the purpose of substitution in the main body of the routine.

Configurations are *inherited*. That is to say that all key-value pairs (with the exception of `conditional_enable`) are carried forward to the next configuration in the file unless they are overridden. This permits quite sparse configuration files, at the expense of being a bit more tricky to modify.

The length of configuration names must be considered. They will be appended to module names and the subroutine and function names contained therein. The templater does not have a magic means to circumvent the Fortran 95 limit of 31 characters in any symbol. Therefore it makes sense to use highly abbreviated configuration names.

The templater modifies the names of subroutines and functions as it processes the module. As such, it is a little pick about syntax. Normal routine decoration specifiers such as `pure` and `elemental` are supported, but functions *must* be declared using the `result()` specifier to define the return type.

If the special key `conditional_enable` is present, this is used to wrap the generated module in `#if` `#endif` elements. See `lib/quicksort.F90.template` for examples.

Values from the key-value pairs are directly substituted into the templated module below, where they replace the element `%(key)s`. (This specifier is the standard python named-string specifier). Keys named beginning with `type` are treated specially, as described in the following section.

The templater cannot circumvent Fortran line length limits. If necessary a value to substitute can be extended over multiple lines by ensuring the first character on a new line is a space, and then just continuing. Make sure you remember the Fortran line continuation characters, as in this example from the MPI wrapper code:

```
mpilen=((ubound(v,1)-lbound(v,1)+1)*(ubound(v,2)-lbound(v,2)+1)*&
 (ubound(v,3)-lbound(v,3)+1))
```

There is a special variable, which can be accessed using `%(name)s`. This contains the name of the current configuration.

1.7.4 Variable substitution

The templater has extremely powerful mechanisms to manipulate the types of variables. Variable manipulation is enabled by using a key in the key-value pair section that begins with `type`. In particular, the code is able to manipulate the number of dimensions that different arrays have.

As an example, take a routine which is passed an array, and a value that could be an element of that array (such as is necessary for a binary search), such that

```
    subroutine example(arr, elem)
        %(type1)s :: arr(:)
        %(type1)s :: elem()
        ...
    end subroutine
```

If `type1` is a scalar value, this does a substitution exactly as would be expected:

```
                                  subroutine example_int(arr, elem)
[int]                                 integer(int32) :: arr(:)
type1=integer(int32)        ⟹        integer(int32) :: elem
                                      ...
                                  end subroutine
```

However, it may be necessary for the value which is being considered in the array to itself be an array. An example of this would be the bit representations used in NECI — a list of of these is a two dimensional array, and any intermediate values would be arrays themselves.

In this case, an array type should be specified using the `dimension` keyword, and the code will be automatically adjusted as follows:

```
                                         subroutine example_int(arr, elem)
[arr_int64]                                  integer(int64) :: arr(:,:)
type1=integer(int64), dimension(:)   ⟹      integer(int64) :: elem()
                                             ...
                                         end subroutine
```

Essentially the number of : delimiters appearing in the variable definition is combined with the number of dimensions specified in the type.

As a special case, temporary variables can be created of an appropriate size which are either scalars, or have one dimension. For the definition

```
%(type1)s :: arr(:)
%(type1)s :: tmp(size(arr(1)))
```

then adjustment occurs as follows

```
[int]                                ⟹   integer(int32) :: arr(:)
type1=integer(int32)                     integer(int32) :: tmp

[arr_int64]                          ⟹   integer(int64) :: arr(:,:)
type1=integer(int64), dimension(:)       integer(int64) :: tmp(size(arr(1)))
```

In a similar way, the references made to these variables within the routines must be adjusted. This is to ensure that correct sized array slices are used at all times. For the original templated code

```
arr1(j) = arr2(i)
```

the following will result in the the templated output if the variable is of an adjustable type and declared at the top of the function:

```
[int]                                ⟹   arr1(j) = arr2(i)
type1=integer(int32)

[arr_int64]                          ⟹   arr1(:,j) = arr2(:,i)
type1=integer(int64), dimension(:)
```

1.7.5 The supermodule

In many modules, there are routines that do not need to be templated for different variable types. As an example, within the MPI wrapper routines, the code to initialise MPI is not variable type specific.

Code which is placed in the supermodule is not templated, and is included directly in the final generated module.

1.7.6 Optional parameters and lines of code

It is good practice to write templated routines as generally as possible. This likely involves adding more functionality than is needed in all cases, and switching this functionality on and off in some way.

For example, the sorting routine can sort multiple arrays in parallel, according to the order in the first array (such as sorting a list of determinants into energy order, where the energies are stored in a separate array). It also needs to have comparison functions defined for scalars as well as arrays.

The extent to which interesting features can be developed is limited only by the developers imagination in using the template substition. But two tricks are generally useful.

Additional optional arguments

Subroutines can easily be given flexible numbers of arguments. This is useful for adding additional functionality (and allows multiple templated routines to use the same `type` values). The templated subroutine definition

```
subroutine example(arg%(extra_args)s)
```

will generate the following code

```
[simple]
extra_args=                  ⟹   subroutine example(arg)

[extended]
extra_args=, arg2, arg3      ⟹   subroutine example(arg, arg2, arg3)
```

The next trick is useful for adding the type definitions of these additional arguments, and enabling the code which uses them.

Switching off lines of code

Lines of code in Fortran are trivially disabled when they are commented out. Prefixing lines with a switch-value allows it to be disabled. For example

```
%(use_type2)%(type2) :: val()
```

will allow an additional type to be used in a routine depending on the configuration:

```
[unused]
type2=                            ⟹   ! :: val()
use_type2=!

[arr_real]
type2=real(dp), dimension(:)      ⟹   real(dp) :: val(:)
use_type2=
```

1.7.7 Manual renaming of routines

The user can additionally manually create interface blocks for the templated routines. This is useful where there is more than one possible function to call for each of the variable types.

An example of this is given in the MPI wrapper functions, where there are versions of routines that require manually specifying the lengths of various parameters, and automatic versions which take the lengths from the sizes of the arrays passed in. At the top of the templated module definiton lie interfaces blocks such as

```
interface MPIReduce
    module procedure MPIReduce_len_%(name)s
    module procedure MPIReduce_auto_%(name)s
end interface
```

which makes use of the special `%(name)s` element to reference the generated routines after templating.

In this case, the templated routines `MPIReduce_len` and `MPIReduce_auto` will be available to the user as usual, but the routines can both be called by the more generic name of `MPIReduce` with the appropriate arguments supplied.

1.7.8 Examples

All of the features of the templating code have been heavily used the Shared Memory code, in `lib/allocate_shared.F90.template`, the sorting code in `lib/quicksort.F90.template` and the MPI wrapper code in `lib/Parallel.F90.template`. Other less aggressively used case can be found elsewhere.

1.8 Testing

NECI comes with a set of tests in the test_suite directory. Each of these tests have a benchmark file. When you run the tests, the results of your calculation will be compared against those from the benchmark files. If the values of certain results agree to within a predefined tolerance, the test will pass.

The test suite is run using a python program called testcode2. This program will call the desired tests, compare the results against benchmarks, and let the user know the outcome of each test.

You can clone testcode2 from github with the following command:

```
$ git clone https://github.com/jsspencer/testcode ~/testcode2}
```

There are tests for each of the three executables: neci, mneci and kneci. These are stored in the three directories with the corresponding names. These directories are then divided into further directories for the different types of tests. For example, mneci has subdirectories called rdm, excited_state, kpfciqmc and so on, with tests for each of these corresponding features of NECI.

To run the entire test suite, just do

```
$ ~/testcode2/bin/testcode.py
```

in the test_suite directory (assuming you cloned testcode2 to your home directory). To run this, you will have to have all of neci, mneci and kneci compiled. However, you can also run a subset of tests. For example, to run all mneci tests do

```
$ ~/testcode2/bin/testcode.py -c mneci
```

or to run a particular single test do

```
$ ~/testcode2/bin/testcode.py -c mneci/rdm/HeHe_int
```

or to run two particular tests do

```
$ ~/testcode2/bin/testcode.py -c mneci/rdm/HeHe_int -c mneci/rdm/HeHe_real
```

By default, testcode will just tell you whether or not the test passed. If the test failed, you can get further information by increasing the verbosity of the output. For example,

```
$ ~/testcode2/bin/testcode.py -c mneci/rdm/HeHe_int -v
```

or

```
$ ~/testcode2/bin/testcode.py -c mneci/rdm/HeHe_int -vv
```

which will tell you why the individual test did not pass. You can also use the verbosity flags when running the entire set of all tests.

1.8.1 Adding a new test

To add a new test, first create a directory in the appropriate subdirectory (for a parallel neci job, inside ./neci/parallel). In this directory add the test's input file (with a name ending in '.inp') and any necessary additional files, such as integral files or POPSFILEs.

You should then add these files to git, for example:

```
$ git add neci/parallel/new_test
```

If the test is added to a new subdirectory then you may need to add it to ./jobconfig. If you added it to an already-exisiting directory, such as neci/parallel, then it should be automatically found by testcode using the globbing in jobconfig.

You must then create a benchmark file by running the test suite. To create a new benchmark for *only* the new test then run, for example,

```
$ testcode2/bin/testcode.py make-benchmarks -ic neci/parallel/new_test
```

This should run just the new test. You will be told that the test has failed, and asked if you would like to set the new benchmark. If you believe that the test has run correctly then do so with 'y'.

'-i' tells testcode to 'insert' the new benchmark ID at the start of the old list of benchmarks (located in ./userconfig). When testcode is run later, it will use the benchmark files with these IDs to compare against.

If you don't include '-i' then testcode will remove all previous benchmarks. This is useful if you want to reset benchmarks for the whole test suite, which can be done with:

```
$ testcode2/bin/testcode.py make-benchmarks
```

Finally, once this is done you need to add the new benchmark file to git. If the new benchmark ID is, for example,

mneci-c3462e0.kneci-c3462e0.neci-c3462e0

then you can do:

```
$ git add neci/parallel/new_test/*dneci-c3462e0.kneci-c3462e0.neci-c3462e0*
```

You should then commit the newly added files. You might like to try running testcode on the new test, and making sure it runs and passes as expected, confirming the that the test was added correctly:

```
$ ~/testcode2/bin/testcode.py -c neci/parallel/new_test
```

1.8.2 Unit tests

Unit tests should test discrete, small, elements of functionality. Ideally these should be the smallest elements such that functionality is then composed from "units" that have all been tested. By considering each of the elements explicitly, it is possible to writ tests for edge-case behaviour, where it is difficult to ensure that this behaviour will be tested in a full integration test case.

Unit tests are found in the `unit_tests` directory. They are arranged in subdirectories, each of which corresponds to one of the *files* of NECI source code. There are a number of technical steps to integrating new unit tests.

1. A passing test is an executable that returns 0, and any other return value indicates failure. The developer has an entirely free choice to determine how they wish to write test executables.
2. The library FRUIT is provided to assist in writing unit tests. Within a given test (i.e. testing a given function, or unit), there should be a *suite* of tests to cover all possbile cases. FRUIT provides helper functionality to keep track of which element of a suite is currently running (using the `TEST()` macros), check values (using `call assert_equals`, `call assert_true` and so forth), keep track of where errors occurred and report them in an easily readible form. The module can be imported with `use fruit`. See existing tests for examples.
3. If the current directory of tests does not contain a `CMakeLists.txt` file, create it. Ensure that it contains a `foreach()` loop over the available `\${PROJECT_NAME}_CONFIGURATIONS` so that all the build configurations of neci get tested. The directory should be added to the main `CMakeLists.txt` in the `unit_tests` directory using the `add_subdirectory()` command.
4. Add the test to the `CMakeLists.txt` file in the directory in which it resides using the `neci_add_test` command. This will require you to specify a name for the test, and the appropriate .F90 file.
5. To test the appropriate configuration of neci, add `lib{k,m,d,}neci` to the LIBS line. To use the FRUIT helpers, add `fruit` to this line.

The easiest way to get these details correct is to copy existing examples.

Unit tests can be run using the command

```
ctest [-R <regex>]
```

which is a built in part of the CMake toolkit. By default this will execute all avaialable tests and print a report on successes and failures. The optional `-R` flag specifies a regular expression, and only tests matching this will be executed. There are a number of further options available.

A test can also be run directly by running its executable manually. This can be more straightforward for capturing the output of a failing test whilst debugging. The executables are found in the same position in the build directory as the source files are in the main repository (i.e. `unit_tests/det_bit_ops/test_countbits.F90` gives `<build_dir>/unit_tests/det_bit_ops/test_countbits`).

1.9 Interfacing C (and C++) code

Developers have always mixed Fortran code with external routines implemented in other languages, especially C. This has generally been done on an ad-hoc basis by exploiting the naivety of the linker — in particular that the linker will resolve dependencies with any symbol of the specified name. This is useful, but introduces a number of potential problems:

Name clashes

Different compilers follow different naming conventions. In particular Fortran compilers often (but not always) append or prepend one or two underscores to symbols in the object files. This is fine when resolving against other Fortran generated symbols, but requires coordination with the symbol names produced in C.

A solution with an array of compile flags controlling the naming in the Fortran compiler, and underscores liberally scattered through C files is fragile and unreliable. It also makes it difficult to call library routines written in C.

No checking of parameters

The linker is extremely stupid - it only matches by parameter name. If this method is used, absolutely no checking is done on the parameters passed to the C routine from Fortran. This is a recipe for disaster, and will generate only runtime errors.

Calling

By default C passes arguments by value, whereas Fortran passes them by pointer. This requires writing wrappers for almost any non-trivial C library routine to access it from Fortran. Some constructs simply cannot be emulated.

Variable types

As an extension of the lack of checking of parameters, there is no checking of argument types across the Fortran/C interface. This relies on the Fortran and C code using the same types - in particular the same size of floating point and integer variables. This is extremely hard to guarantee, and these can fluctuate according to compiler flags. This can result in compiler and computer specific runtime errors that are extremely difficult to track.

All of these problems can be solved using structured interfacing, at the cost of introducing a dependency on the Fortran 2003 standard.

All access to C routines in NECI must be through a declared interface. This should be declared only once, in a module. An example is given here:

```fortran
interface
    ! Note that we can define the name used in fortran code, and the C
    ! symbol that is linked to independently.
    subroutine fortran_symbol(arg1, arg2) bind(c, name="c_symbol")
        ! The module iso_c_hack is a wrapper for iso_c_binding. This
        ! contains some workaround for incomplete Fortran 2003 support
        ! across compilers.
        use iso_c_hack
        integer(c_int), intent(inout) :: arg1       ! Passed by pointer
        integer(c_bool), intent(in), value :: arg2 ! Passed by value
    end subroutine
end interface
```

A good summary of the rules and procedures for using this interoperability are given in this Stack Overflow answer: http://stackoverflow.com/tags/fortran-iso-c-

`binding/info`

C++ routines can be made suitable for access from Fortran by prepending symbol declarations in the C++ code with `extern "C"`.

1.10 Debugging tips

Code breaks. Sometimes it appears to rot with time. Finding bugs takes the majority of most programmers time, and practices which make this quicker are invaluable!

Obviously, the techniques you will use will depend on the nature of the problem (tracking down small numerical changes in output is generally much harder than finding what causes a segfault), but there are number of tricks which can help!

1.10.1 Build configurations

As soon as you have a problem, build a debug rather than an optimised version of the code. This cause a large array of changes to the compiled code:

Array bounds checking
All Fortran arrays have well defined bounds on all of their dimensions. In debug mode the compiler will insert code to check that all memory accesses are within these bounds. If not, execution will be terminated with a message indicating at what line of what file the error occurred. If running in a debugger (see later) execution will be interrupted at this point.

Disable optimisations
Hopefully this will not make the bug go away! If it does, you are almost certainly looking at either an uninitialised variable, or access beyond the end of an array.

The primary purpose of disabling optimisations is to make the mapping between the source code and the executable more linear. This results in any error messages, and the output of any tools, being easier to interpret.

Adds debugging symbols
When your code crashes it is really useful to know what routine was running, and what the stack trace (list of routines that have been called to get to this point in the code) is. Adding debugging symbols provides the information to convert the memory addresses into files and lines of source code. This makes error messages useful.

Enables the `ASSERT` macro
There are many consistency checks internally in NECI that can be turned on in debug mode. Particularly for difficult-to-find bugs, these are likely to fail substantially earlier in a run than it is possible to view the problems in the normal output.

Defines `__DEBUG`
Any blocks contained inside `#ifdef __DEBUG` sections are only enabled in debug mode. Most of these contain either additional output specifically targetted to make debugging easier, or additional consistency checks.

1.10.2 `ASSERT` statements are your friend

Generally the time to add `ASSERT` statements to your code is when you are writing it. Debugging will make you acutely aware of their benefit. If you have a suspicion in which bits of code a bug might lie, liberally sprinkling it with `ASSERT` statements can help to find bugs.

More usefully, when you find the bug, add `ASSERT` statements to the code to catch similar errors in the future.

1.10.3 Learn to use your tools

Most of the programming tools in existence are for the purposes of debugging. Learn to use them! Practice using them. The really work.

Debuggers

The debugger is the most powerful tool you have. Essentially you run your code in a harness, with the debugger hooked into everything important. On most Linux systems, the most readily available debugger is `gdb`.

The debugger can trap any execution errors, and will interrupt (break) execution of your program at this point — while preserving all of the memory and execution state. You can examine the call stack (the nested list of functions that have been called), and the status of all of the memory. You can also change the contents of any relevant memory and continue execution to see the effects.

Break points can be added to the code at any line of any function, and execution interrupted at that point. If your bug is only showing late in execution, you can interrupt on the n-th time a function is called. You can step through the execution line by line in the source code, examining all variables, and watch precisely what goes wrong.

The debugger is the swiss-army sledgehammer of tools.

Valgrind

Valgrind is a tool for memory debugging. In essence it replaces most of the memory manipulation primitives provided by the operating system with instrumented versions. It will track what happens to memory, where it is created, where it is destroyed, and what code (in)correctly accesses it.

Intel Inspector XE

If you have access to Intel tools, the Intel Inspector is an extremely powerful debugger, memory analysis tool, performance enhancement and problem tracking tool. An top of a good interface to normal debugging tools, it can apply a lot of analysis to your code's execution, and then filter through it to find where things go wrong.

1.10.4 Machete Debugging

When bugs are particularly non-obvious, a good first step is to reduce the complexity of the problem. Try and remove as much code as possible. A good rule of thumb is to remove half of the code, and retest. If the bug has gone away, then it required interacting with the other half of the code.

This technique can very quickly isolate a test case (that still fails) into something of manageable size. In doing so, the bug normally becomes obvious. You can then revert to the original code, and fix the problem there.

In a complex code, this process can be tricky. Lots of the sections of code are coupled to each other. This can require some creative thinking. Choose the domain of your problem carefully (it may only be necessary to apply the machete to one file), and aggressively decouple sections by commenting out function calls.

Remember — it doesn't matter if you break functionality doing this (you obviously will) so long as you retain the buggy behaviour in the code that is left.

This is really good for finding strange memory interactions — left with the two routines that are trampling on each others toes.

1.10.5 `fcimcdebug 5`

For bugs that change the trajectory of simulations, turning on the maximum debug output level in NECI (by adding `fcimcdebug 5` to the LOGGING block of the input file in a debug build) can often give a quick insight into where the problem is located.

Many problems will exhibit with obviously pathological behaviour. For example trying to spawn `NaN` particles, or an extremely large number, generally indicates a problem in either Hamiltonian matrix element generation or calculating the generation probabilities. Similarly walkers being spawned with repeated or zeroed orbitals are a give away.

Similarly, if the bug is recently introduced and a prior version functioned correctly (such as a test code failure)

1.10.6 Look at the git logs

If you are chasing down a regression (such as a testcode failure), then there is certainly a version that used to work, and a version which now does not. Frequently there are not many commits between these two versions — have a look at what they are. Often the failure is really obvious!

If there are a large number of commits between the last known good commit and the first known bad commit, then using `git bisect` is a very efficient way to locate the first bad commit and the last good commit.

2 Guide to specific code in NECI

2.1 Important modules and common (global) variable names

2.1.1 Inexplicable names and anachronisms

A number of the variable names in NECI are largely inexplicable, or confusing, outside of their origin in historical accident. This section tries to clarify some of these.

ARR, BRR

`ARR(:,1)` contains a list of spin orbital (Fock) energies in order of increasing energy.

`ARR(:,1)` contains a list of spin orbital (Fock) energies indexed by the spin-orbital indices used in the calculation.

`BRR` contains a list of spin orbital indices in increasing (Fock) energy order. Where multiple degenerate orbitals have the same symmetry, they are clustered so that spin orbitals with the same symmetry are adjacent to each other, and within that ordered by m_s value.

TotWalkers and TotParts

`TotWalkers` refers to the number of determinants or sites that must be looped over in the main list. This may include a number of blank slots if the hashed storage is being used.

The number of particles (walkers) on a core is stored in `TotParts`. The total number of particles in the system (including all nodes) is stored in `AllTotParts`.

InitWalkers

The number of particles *per processor* before a simulation will enter variable shift mode.

G1

The variable name `G1` is a historical anachronism. It is an array containing symmetry information about given basis functions. In particular `G1(orb)` contains information about the one electron orbital `orb`.

The data is of type `BasisFN`:

```
type symmetry
    sequence
    integer(ints64) :: S
end type
type BasisFn
    type(symmetry) :: sym ! Spatial symmetry
    integer :: k(3)       ! K-vector
    integer :: Ms         ! Spin of electron. Represented as +/- 1
    integer :: Ml         ! Magnetic quantum number (Lz)
    integer :: dummy      ! Padding for alignment
end type
```

Not all of these values are required for all simulations. The `sym` element points at a padded 64-bit integer. This is done for memory alignment reasons that are no longer important.

2.2 Parallelism

FCIQMC is a highly parallelisable algorithm. Implementationally this is achieved through the use of independent MPI processes that communicate using MPI.

The raw `MPI_*` routines, provided by MPI should not be used directly inside NECI for a couple of reasons:

- Depending on the build configuration and compiler, many variables may change their size or alignment, leading to code which only works on some compilers, and
- Naming conventions for linking vary between compilers.

The `Parallel_neci` module abstracts these implementational details away, presenting a consistent interface.

2.2.1 Usage

The relevant MPI functions are accessible through wrapper functions with the underscore in the name removed. For example, the MPI routine `MPI_AllReduce` is available as `MPIAllReduce`.

The arguments to the wrapper routines are a subset of those specified in the MPI standard, and the explanations of those values may be found there.

Many of the routines have versions which require specifying the array dimensions manually (`*_len`) or automatically (`*_auto`). The latter obtain the dimensions of the arrays to communicate by analysis of the array bounds and are preferred, as they are less prone to user error.

If communication is desired between subsets of the available processors (as is used in the CCMC code), the MPI communicator may be specified through the `Node` parameter. No further detail is provided here about how to construct these objects.

If the necessary MPI routine is not present in the `Parallel_neci` module it will be necessary to add it. This can be awkward as discussed below.

☞ The `inplace` MPI functionality is present (although disabled) in NECI. For the time being it should not be used due to serious bugs in the interaction between ifort and OpenMPI.

☞ Different implementations of MPI are not interchangeable at runtime, even if they initially appear to be. For instance, code compiled using OpenMPI on the ifort compiler will run on MPICH built with the gnu toolset, but no communication will occur, and `nProcessors` independent (identical) simulations will run.

☞ In keeping with the notion of failing early, and failing loudly, outside of the initialisation and cleanup code, most of the templated MPI routines do not quietly report errors in status variables, but will kill the entire simulation with a runtime error.

2.2.2 Important variables

A number of important control variables are available for use within NECI.

`nProcessors`
: The total number of processors initialised in the MPI calculation.

`iProcIndex`
: The (zero based) index of the current processor. This will be between 0 and `nProcessors-1`.

`root`
: The (zero based) index of the root (head) processor. This should be tested in code using: `if (iProcIndex == root)`.

`nNodes`
: The number of nodes initialised in the MPI calculation. In most cases this will equal `nProcessors`. These values will only differ if the MPI space is being subdivided into smaller nodes.

`iNodeIndex`
: This (zero based) index gives the position of the processor in the current node. For most calculations this will be zero on all processors.

`bNodeRoot`
: This specifies if the current processor is the root processor of a node. I.e. that this processor is responsible for macroscopic communication. For most simulations this is `.true.` on all processors.

2.2.3 Implementation

The current implementation of the `Parallel_neci` module works extremely well. It may, however, be necessary to modify it, either to deal with changes in the available compilers, or (more likely) to add support for additional MPI functions.

Fortunately, the latter is much more straightforward than the former! A modified version of a currently implemented routine is likely to be the best approach.

Much of the complication introduces is the requirement that NECI interoperate with MOLPRO, which uses C based MPI libraries. In order to function correctly in this environment, it is necessary to wrap the MPI routines in a C-wrapper, and interface this with our templated library.

As such, the MPI wrapper implementation has a number of layers and non-obvious facets. This results in the behaviour being relatively opaque, and the code scattered with a large number of `#ifdef` statements.

C-wrapper initialisation

The constants required for operation of the C MPI libraries are not necessarily of a form compatible with Fortran. The C wrapper code (in `lib/parallel_helper.cpp`) contains lookup tables of the values defined in the C MPI headers. The module `ParallelHelper`, which is included in `Parallel_neci` contains Fortran definitions of these values which are the indices to the lookup tables.

The C types required for MPI communicatiors and groups `MPI_Comm` and `MPI_Group` are not Fortran compatible. Fortunately the MPI standard provides inbuilt converters to integer types (`MPI_comm_2cf`, etc.). The c wrapper code mimcs the normal Fortran MPI code, returing Fortran compatible values, but internally converting them to the required types.

Determination of parameter sizes

The code to determine the lengths of the input (`v`) and output (`ret`) arrays is passed to the code a template parameters `mpilen` and `mpilen2`. These make use of the `lbound` and `ubound` intrinsics to determine the number of elements in each array.

Conversion to pointers

Fortran, by default, passes values by pointer. In the case arrays, this passes a pointer to the array structure, rather than the data. This structure includes a pointer to the data and information about the data type and the array bounds.

If the code is to be passed to the C wrapper, the actual data pointer needs to be extracted.

When the C wrapper layer is being used, a value of type `c_ptr_t` is declared. If compiler support is present this is equal to `type(c_ptr)`, but otherwise is a raw 32 or 64 bit integer as appropriate.

The memory address of the array (strictly of its first element) is obtained using either the standardised `c_loc`, or the non-standard `loc` intrinsics.

> ☞ Due to a bug in some versions of gfortran an extra level of indirection is used which hides the type of the array being passed using a routine called `g_loc`. This code circumvents normal interfacing.

Calls to MPI

In the normal case (when the C-wrapper is not being used), the `MPI_*` routines are then called directly. After the return values are checked for errors, the routine returns the output data as normal.

C wrapper layer

Interfaces to each of the C mpi wrappers are defined in the `ParallelHelper` module. These accept the pointers discussed earlier. The Fortran symbols for these wrappers are named so that they coincide with the normal `MPI_*` routines, so that no change to the calling code is required.

2.3 Shared memory

In principle, when using MPI all of the processes are entirely independent except for the explicit communication made through the MPI library. Within NECI, as all of the particles are entirely independent between annihilation steps this is a good thing.

However, we have a large amount of read-only data. The integrals which are read in from the FCIDUMP files can consume a non-trivial proportion of the available system memory, and are duplicated on each of the processes. On modern multi-processor and multi-core systems this is an outrageous waste of system memory — which is made worse by the fact that if the same regions of this memory are requested on multiple different processors the computer will not know these are the same, and will not be able to make use of the L1, L2 and L3 cache to speed memory access.

The shared memory provisions in NECI substantially abuse the MPI specification by mixing the available memory address space between the different processes. This is extremely useful when done careful, but there are some caveats to be aware of.

2.3.1 How to use shared memory provisions

The shared memory provisions have been designed to be as interchangeable with the normal Fortran memory allocation provisions. Essentially memory is allocated largely as usual, but all of the processes on the same physical node will end up with pointers to the same block of memory.

Memory is allocated with a call to

```
subroutine shared_allocate(name, ptr, dims)
```

This is a templated routine, and so will work for a wide variety of data types and array dimensions. The `name` parameter specifies a unique name for this array. This name is used to identify the particular block of shared memory between the processes (there may be an arbitrary number). The `ptr` parameter is a Fortran `pointer` with the relevant data type and the correct array shape. `dims` is an array of integers indicating the size of each dimension of the array to be allocated.

As an example,

```
real(dp), pointer :: real_arr(:,:)
...
call shared_allocate("example", real_arr, (/6, 13/))
```

will allocate a 2-dimensional array named "example" with the array bounds `(1:6, 1:13)`, equivalent to `allocate(real_arr(6,13))`.

The read only data should now be written. In principle, the data only needs to be written from one of the processors per node — in practice it is easiest to get all of the nodes to write the read only data. It is important that data is only written at this stage — any processing that needs to be done on this data must not happen in the shared memory, as it could be disrupted by the other processes. This scheme only works because all of the processes write exactly the same data.

After data initialisation, the processes must be synchronised with a call to `MPIBarrier`. This will ensure that the read-only data is in the same state in all of the threads.

☞ The POSIX memory model makes few guarantees about *when* memory that is shared between processes gets updated (as opposed to between threads). This shared memory ***must not*** be used for communicating between the MPI processes. That is what MPI is for.

☞ There are no synchronisation primitives provided for use with these shared memory blocks. No assumptions can be made about the order of access between the processes. All actions that are carried out ***must*** be inherently thread-safe, or errors will eventually (and unexpectedly) occur.

The data should be deallocated using the routine `shared_deallocate`.

2.3.2 Quirks and limitations

Customising data types

The shared memory routines are templated so that they can be used with a wide variety of plain-old-data types and array sizes. For any data types outside of those already templated, new configuration options will be needed in `lib/allocate_shared.F90.template`. The information required is the number of bytes per element of the array.

If custom `types` are desired, this may be quite tricky. It is difficult to guarantee the size, memory alignment and packing of the data in a custom type — the compiler is free to choose how it wishes to do this, and as such it varies from compiler to compiler. It is necessary to use the `sequence` keyword must be used to force the compiler to store data contiguously. If there are `pointer` or `allocatable` elements in the custom type, it cannot be used with shared memory, as the size of these elements is highly variable between compilers.

Storing bit representations

There is a special allocate routine `shared_allocate_iluts` which can be used for storing bit representations — these differ in that the lower bound of the first array index must be 0, rather than 1.

Non-uniform memory architectures

Modern chip architectures (in particular the intel i3, i5 and i7 series of processors) move the memory controller onto the processor. This dramatically improves memory access speeds.

The cost is that on multi-processor (as opposed to multi-core) architectures the memory is segmented into regions that are controlled by the different processors. Memory access within the region controlled by the current processor is faster than that between them.

A performance increase could be obtained by only sharing memory between processes which are hosted on the same physical processor. This will require using the processor affinity options of MPI, which is not routinely done currently with NECI. Further, it will require subdividing the processes on each physical node into sub-nodes — support for this exists in the code in NECI, but the processor specific information required to correctly subdivide the processes is not included, and this facility is currently switched off.

For processor topology, see the Intel documentation at `https://software.intel.com/en-us/articles/intel-64-architecture-processor-topology-enumeration`.

Disabling shared memory

Shared memory is enabled by the pre-processor define `__SHARED_MEM`. This can be found in the config files for platforms that support it. If necessary, this can be removed from relevant config files and Makefiles, which will disable inter process memory sharing, and fall back on the normal Fortran `allocate` mechanism.

2.3.3 How it works

The way that inter-process memory sharing works is system specific, and depends on operating system primitives being used directly.

The code templating functionality in NECI is used to make the shared memory wrapper work for a wide range of data types, with differing numbers of dimensions in the arguments. From this the size of memory required in bytes is calculated, and this is passed to a C helper function that allocates the memory. The returned raw pointer is converted to a Fortran one through the Fortran 2003 intrinsic `c_f_pointer`.

The C helper library contains a number of utility functions to help it interface with Fortran. In particular wrapper so it can print to the same output, and a mapping to store additional data about the allocated memory to assist in deallocating without the Fortan code requiring knowledge of how the operating system intrinsics work.

Beyond that, there are three main code paths:

POSIX

A unique file name is created, based on the current working directory. The function call `shm_open` with the control parameter `O_CREAT` will open, or create, a POSIX shared memory object. As a result, one of the processes on a node will create the mapping, and the others will join it — it is unknown in advance which will do the creating.

The returned descriptor acts as a file, and so is set to the desired length with `fdtruncate`, and then mapped into memory as if it were a memory-mapped file using `mmap`. The file descriptor is then closed as it is no longer needed.

Once all of the processes have mapped the memory, the shared memory object is unlinked, ensuring that the memory will be deallocated by the operating system when all of the processes stop (preventing a memory leak if the processes crash).

The memory can then be manually deallocated using `munmap`.

System V

A unique file name is generated, based on the current working directory, and the file is created. A System V Inter Process Communication Key is created and obtained from this file using the routine `ftok`.

Using this key, a shared memory object of the correct size is created using `shmget`, and this region is mapped into memory using `shmat`.

Once all of the processes have reached this point, the shared memory control object is destroyed using `shmctl` to ensure the operating system will deallocate the memory in case of a crash.

The memory can be manually deallocated using `shmdt`.

Windows

The memory mapped file provisions in Windows are used to generate a shared memory region. A unique file name is created, based on the current working directory, and then a non-filesystem backed file is created with `CreateFileMappingW`. This can be opened by all of the other processors on the same node using `OpenFileMappingW`, and this "file" is mapped into memory on all of the processes using `MapViewOfFile`.

The memory can be manually deallocated using `UnmapViewOfFile` followed by `CloseHandle`.

The subroutine `iluts_pointer_jig`, which is used when allocating bit representations which start from an index of 0 rather than 1, is an interesting demonstration of how to manipulate the bounds of arrays declared in Fortran in compilers that do not support the array reshaping assignments described in Fortran 2003 (most compilers).

2.4 Integral retrieval

Integral retrieval is found in the tightest of the tightest loops within NECI. Calculating each Hamiltonian matrix element may require a number of different two- and four-index integrals, and at least one Hamiltonian matrix element is required for each generated excitation.

As a result, the normal approach taken to prevent code repetition introduces a bottleneck. If there is a `get_umat_el` function, this will have to contain the logic as to which type of integral is being obtained, and where this should be located. This conditional logic will be executed for every required integral.

As such, access to the integrals is through a procedure pointer, with the interface

```
abstract interface
    function get_umat_el(i, j, k, l) result(hel)
        use constants
        implicit none
        integer, intent(in) :: i, j, k, l
        HElement_t :: hel
    end function
end interface
```

This function pointer can then be called from anywhere in the code (after it is initialised) and this will call the correct routine.

2.4.1 FCIDUMP files

The most commonly usedintegrals routines store their integrals in memory after reading them in from a `FCIDUMP` file.

These routines support FCIDUMP files produced by various codes, including MOLPRO, PSI3 and VASP (hacked versions of Dalton and QChem also support this - I believe MOLCAS as well now). Certain (generally legacy) FCIDUMP files have a strictly fixed format, which can result in adjacent columns of indices merging. As such, they are read via an explicit format. To use MOLPRO or other FCIDUMP files the `MOLPROMIMIC` option should be supplied in the `SYSTEM` block of the input file. Other sources of FCIDUMP file should use the `FREEFORMAT` option. In general, this should *always* be used.

The FCIDUMP files can be briefly summarised by

Header

The header is specified as a Fortran namelist, with the following possible elements:

`NORB`
: Specifies the number of spatial orbitals in the system if RHF, or the number of spin-orbitals if UHF.

`NELEC`
: Specifies the number of electrons the Hartree–Fock determinant should have.

`MS2`
: Specifies twice the total projected spin of the system (such that it is always an integer)

`ORBSYM`
: A list of spatial symmetries of the orbitals specified in (??? Check name with Giovanni) format in orbital order. Note that the structure of the reference determinant will be visible here if the `MOLPROMIMIC` option is used, and the reference is to be determined by the order of the orbitals.

`ISYM`
: The symmetry of the reference determinant specified in the same format.

`UHF`
: T if the file contains a UHF basis, otherwise F.

`SYML, SYMLZ`
: The total and projected orbital angular momentum for systems with an axis of rotation. Currently FCIDUMP files which use this option can only be generated using QChem, and the resultant file must be pre-processed using the TransLz utility.

`PROPBITLEN, NPROP`
: These parameters are used to describe the behaviour of k-point symmetries. For more details contact George Booth.

4-index integrals

Following the header, all of the integral and energy lines may follow in arbitrary order.

The 4-index integrals are specified with the format `Z i j k l`, where `Z` is the `real` or `complex` element, and the indices are integers. All indices are one-based. Indices are in **chemical notation**!

2-index integrals

The 2-index integrals are specified in the same way with the final two indices equal to zero.

Fock energies

The Fock energies are specified in the same way, with the final three indices equal to zero.

These values are used for determining the initial (Hartree–Fock) determinant, which is constructed from the lowest energy spin-orbitals. They are strictly optional. If the `MOLPROMIMIC` option is supplied then the order of the orbitals determines the reference determinant.

Core energy

The core energy is specified in the same way, but with all four indices set to zero.

2.4.2 Generation on the fly

A number of schemes exist for generating integrals on the fly. In particular the Uniform Electron Gas and Hubbard model have integrals that can be trivially calculated, and so FCIDUMP files are not used.

2.4.3 Nested schemes

There exist two function pointers with the same abstract interface, `get_umat_el` and `get_umat_el_secondary`. Once the primary method of determining integrals has been selected, then this be moved to the secondary pointer, and another wrapper routine used instead. This allows additional filtering logic to be applied.

The wrapper routine should then call `get_umat_el_secondary` internally.

2.4.4 Fixed Lz and complex orbitals

If either projected angular momentum (Lz) symmetry, or complex orbitals are being used, the pattern of restricted zeros in the integrals changes. Wrappers around the normal `get_umat_el` routine are used to supply these zeros.

These are examples of the nested schemes above.

2.4.5 Further schemes

It is likely that further schemes will be required in the future. In particular, approximate, and interpolating schemes are likely to be required to reduce the $\mathcal{O}(M^4)$ memory dependence on the number of orbitals. These should be implemented as new routines to be pointed at using the function pointers.

2.5 Hamiltonian matrix element evaluation

The two- and four-index integrals are the primary input to a FCIQMC calculation. However, they are used via Hamiltonian matrix elements. There are a number of considerations for the way that Hamiltonian matrix elements are used in NECI.

2.5.1 Different ways to obtain Matrix elements

Certain pieces of information are required in order to calculate Hamiltonian matrix elements. The Excitation level (the number of differing orbitals between the determinants), the corresponding excitation matrix and the parity of the excitation are necessary. Depending on the excitation level, the decoded list of occupied orbitals is required.

Depending on where in the execution path the Hamiltonian matrix elements are required, different information is readily availabile. As some of this is relatively expensive to calculate (in particular the parity) it is important that as much information as is available is used.

As part of the spawning and particle generation process, there is a function pointer called `get_spawn_helement`. This is passed the natural integer and bit representations of the determinant, the excitation level, excitation matrix, parity and (potentially) precalculated matrix element. Depending on the initialisation options, the routine which is selected will make use of the correct subset of these. It is important that the excitation generator and the choice of Hamiltonian matrix element generation here are tightly coupled.

Elsewhere in the code, the routine `get_helement` is used to obtain matrix elements.[1] This routine comes in a number of flavours. The available options are

`nI, nJ`
: This routine will return the matrix element between any two, arbitrary, decoded determinants.

`nI, nJ, ic`
: This version is provided the excitation level of the determinant in addition.

`nI, nJ, ic, ilutI, ilutJ`
: The bit representations of the two determinants are provided, which greatly enhances calculating the parity if needed.

`nI, nJ, ilutI, ilutJ`
: If both the bit representations and the decoded versions are present but no further information is known then this form should be used.

`nI, nJ, ilutI, ilutJ, ic_ret`
: This version is the same as the above, but returns the excitation level of the pair of determinants in addition to the matrix element.

`nI, nJ, ic, excitMat, tParity`
: When everything about the relationship between the two determinants is fully known, this form should be used. It is used implicitly after excitation generation when the excitation level, matrix and parity are known. The second decoded determinant is not used in determinental calculations, but is provided to support systems such as CSFs.

For calculating diagonal Hamiltonian matrix elements, the routine `get_diag_helement`

[1] As an exception, some old code makes use of `gethelement` or `gethelement2`. These should not be used in new code. Some of the initialisation code also uses `gethelement` as the prerequisites for `get_helement` have not yet been met.

should be used[2]

The cost of the parity

The overall sign of the returned Hamiltonian matrix element is modulated by the parity of the excitation — that is, the whether the number of pairwise swaps of orbitals required to maximimally align the two determinants according to the standard order of the first is odd or even.[3]

The parity may be obtained by actively aligning the decoded representations, or by examination of the bit representations. The latter is much faster than the former, but is still the rate limiting factor for calculating Hamiltonian matrix elements.

Where this factor is known (i.e. after excitation generation) it should be used. It is worth noting that the weighted excitation generation scheme is only viable because only the absolute value of the matrix elements is modelled, so the parity generation step can be entirely ignored.

Slater–Condon rules implementations

The Slator–Condon rules are implemented in the file `sltcnd.F90`, in the routines `sltcnd` and then more specifically `sltcnd_0`, `sltcnd_1` and `sltcnd_2`. There is a certain amount of duplication both of code paths and of conditional testing in these routines — which is a clear violation of the DRY principle.

Access to matrix elements is inside the smallest of the tight loops in NECI. The provision of duplicated logical pathways, rather than conditional switching, has a measurable performance benefit in this case.

HPHF matrix elements

In most cases the HPHF matrix elements are trivially obtained from the normal determinental matrix elements through multiplication by a factor of $\sqrt{2}$ in all cases where the determinant is not closed shell.

If the excitation level is less than or equal to 2, then the elements are a little more complicated. It is possible that both the target determinant and its spin pair are connected to the source determinant, and so the resultant matrix elements will require two calls to the Slater–Condon routines.

Spin eigenfunctions

Spin eigenfunctions may be expressed as linear combinations of all determinants with a given spatial structure. The Hamiltonian matrix elements may be expressed as a two

[2]This general routine is not implemented at the time of writing, but will added shortly.

[3]Note that for double excitations, in principle there are two alignments that work. The two new orbitals could be either way around and the parity of these versions are inverted relative to each other. A convention for the ordering of the new orbitals (in NECI they are required to be numerically increasing) is required but arbitrary. The overall simulation will give the same results either way.

index sum over all the determinants with each of the involved spatial configurations.

Given that the number of configurations increases permutationally with the number of unpaired electrons, this scheme scales extremely badly. NECI has some aggressive optimisation to slightly improve this scaling (see Simon's thesis), but ultimately it is a dead end for big calculations.

See the SPINS branch for Hamiltonian matrix element calculation between other types of spin eigenfunctions. These can scale extremely well, but introduce other problems.

2.6 Excitation generation

Excitation generation is the most complicated and intricate part of NECI. Not only must random moves be made, but they must be made in a manner which is efficient (taking account of symmetry, and in terms of implementation). They must also correctly compute the generation probabilities, taking into account multiple possible selection routes to the same determinant.

A failure to generate all of the connected determinants, or mistakes made in calculating the generation probabilites will lead to silent errors that manifest themselves only through (potentially subtly) incorrect energies being produced by the simulation.

There are a large number of factors that must be considered when writing an excitation generator. This section aims to give an overview of some of them, and a brief outline of the two most commonly used excitation generators.

The Alavi group collectively has a large amount of experience writing excitation generators, and anybody intending to write or modify them would be advised to speak to Simon Smart or George booth first.

2.6.1 Interface

Excitation generation is accessed through a procedure pointer. As such, all excitation generators *must* have the same signature. It is described by

```
subroutine generate_excitation_t (nI, ilutI, nJ, ilutJ, exFlag, ic, &
                                  ex, tParity, pGen, hel, store)

    use SystemData, only: nel
    use bit_rep_data, only: NIfTot
    use FciMCData, only: excit_gen_store_type
    use constants
    implicit none

    integer, intent(in) :: nI(nel), exFlag
    integer(n_int), intent(in) :: ilutI(0:NIfTot)
    integer, intent(out) :: nJ(nel), ic, ex(2,2)
    integer(n_int), intent(out) :: ilutJ(0:NifTot)
    real(dp), intent(out) :: pGen
    logical, intent(out) :: tParity
    HElement_t, intent(out) :: hel
    type(excit_gen_store_type), intent(inout), target :: store

end subroutine
```

`nI` and `ilutI` describe the natural integer and bit representations of the source determinant for the excitation. `nJ` and `ilutJ` will store the generated excitation. `exFlag` specifies the type of excitation — this is generally unused, but in some excitiaton generators permits selecting between single and double excitations for the purposes of testing.

`store` provides a location for the excitation to store information that is specific to the source determinant. This information is available for further excitations from the same site.

`ex` will contain the excitation matrix; `ex(1,:)` contains the *spin-orbitals* of the source electrons in `nI`, and `ex(2,:)` contains the *spin-orbitals* that have replaced these values.

`ic` is used to return the excitation level of the resultant site relative to the source site, `pGen` returns the generation probability for the generated determinant, `tParity` returns the parity of the excitation (is the number of pairwise swaps required to align `nI` and `nJ` even or odd), and `hel` can return the Hamiltonian matrix element for this excitation so that it need not be calculated later (this is only really of interest for HPHF calculations, where it can be easier to calculate this matrix element in the excitation generator for technical reasons).

☞ Note that the determinant returned in `nJ` *must* be sorted in the same fashion as the source determinant `nI`. This normally means in order of ascending spin-orbital number. This can result in substantial reshuffling of the internal order of the detrminant.

☞ It is possible that to implement a sensibly efficient excitation generator, the simulation may make choices which leave no possible excitations remaining. An excitation may be aborted by setting the first element of `nJ` to zero, so long as the calculated generation probabilites for successful excitations are correct.

2.6.2 Symmetry handling

The excitation generator must always preserve the symmetry of the determinant. The way in which this is done depends on the excitation generator, but will involve measuring the symmetry of the source orbitals and selecting the target orbitals appropriately.

The symmetry of a specific orbital, `orb`, may be found in the array `G1`. The spatial symmetry is found at `G1(Orb)%Sym%S`, the k-vector (if appropriate) at `G1(orb)%k`, and the projected spin and orbital angular momentum values at `G1(orb)%Ms` and `G1(orb)%Ml`.

There are also a number of useful macros for measuring and manipulating spin values. `is_beta` and `is_alpha` test if orbitals have beta and alpha spin respectively, whilst `is_one_alpha_beta` test that two orbitals have differing spins. The `get_spin` macro gets the spin in the unusual format for 1 for alpha and 2 for beta as used in some other NECI routines, and `get_spin_pn` obtains the more standard values of ± 1. The `is_in_pair` macro tests if two orbitals belong to the same spatial orbital (including the case where they are the same orbital).

The macros get_alpha and get_beta obtain the alpha or beta orbital corresponding to the same spatial orbital as the current spin orbital (which may or may not be the same orbital). ab_pair obtains the spin paired orbital to the current one.

It is normally necessary to combine and manipulate symmetries, as for a double excitation $\Gamma_i \otimes \Gamma_j = \Gamma_a \otimes \Gamma_b$, but there is no reason for any of $Gamma_i, Gamma_j, Gamma_a, Gamma_b$ to be the same. The products of symmetry labels should be taken using RandExcitSymLabelProd, rather than directly using ieor commands (which will normally obtain the same result), to protect against the consequences of using different types of symmetry. When combining Ml values it is important to bear in mind the maximum permitted value of Ml, and abort the excitation if necessary.

The class count arrays, described above, are indexed by a a combined symmetry index, that combines spin, k-point, momentum and spatial symmetries. The index to this array is provided by the function ClassCountInd. Given a total spin, momentum and spatial symmetry, the paired class count index may be obtained using get_paired_cc_ind. The number of occupied, and unoccupied spin orbitals are stored in the data store as decribed above. The total number of orbitals corresponding with a given class count is found at the corresponding location in the array OrbClassCount.

The array SymLabelList2 contains all orbitals sorted by symmetry class, with offsets given in SymLabelCounts2, such that all of the orbitals with the same symmetry as a given orbital, orb, may be found as follows

```
integer :: sym, spn, ml, norb, cc_ind, offset

! Obtain the symmetry labels associated with this orbital
sym = G1(orb)%Sym%s
spn = get_spin(orb)
Ml = G1(orb)%Ml

! Obtain the combined symmetry index
! cc_ind = ClassCountInd(orb) ! ... alternatively
cc_ind = ClassCountInd(spn, sym, ml)

! Get position and count of orbitals
norb = OrbClassCount(cc_ind)
offset = SymLabelCounts2(1, cc_ind)

! And this slice contains the appropriate orbitals (including orb)
SymLabelList2(offset:offset+norb-1)
```

2.6.3 Manipulating determinants, and bit representations

The primary output of the excitation generator is the newly generated determinant. It is much more computationally efficient to copy and modify the source determinant and bit representation than to start from scratch. As such there are two processes to consider.

Manipulating bit representations

Orbitals can be cleared from the bit representation using the macro clr_orb, and set using the macro set_orb. For an excitation from orbital a to orbital i this would look like

```
clr_orb(ilut, a)
set_orb(ilut, i)
```

`make_single` and `make_double`

Manipulating the natural integer representation of determinants is substantially more complicated, as there is a *requirement* that they remain sorted by spin-orbital number. Although it would be straightforward to, essentially, replace the excited orbital with a new one, and sort it, this is inefficient as the parity of the excitation is going to be required. If the manipulation can be carried out in such a way that the parity is naturally returned this will substantially improve the efficiency.

The routines `make_single` are passed the source determinant and respectively one or two source electron positions and target orbitals. They perform the substitution and measure how much the newly placed orbitals must be moved by to restore sorting (accounting for edge cases).

These functions return the sorted determinant, the excitation matrix and the parity of the excitation. They should be used by all determinental excitation generators.

2.6.4 Data storage

Each site in the main particle list may be occupied by multiple particles. There is a reasonable amount of information which the excitation generator must generate for each site it excites from — it makes sense to store this information and reuse it for each of the particles on the site. A data structure of type `excit_gen_store_type` is passed to the excitation generator that it can use for this purpose:

```
type excit_gen_store_type
    integer, pointer :: ClassCountOcc(:) => null()
    integer, pointer :: ClassCountUnocc(:) => null()
    integer, pointer :: scratch3(:) => null()
    integer, pointer :: occ_list(:,:) => null()
    integer, pointer :: virt_list(:,:) => null()
    logical :: tFilled
    integer, pointer :: dorder_i (:) => null()
    integer, pointer :: dorder_j (:) => null()
    integer :: nopen
end type
```

The arrays that are required for the current excitation generator are allocated during calculation initialisation by the routine `init_excit_gen_store`.

When the excitation generator is being called on a new determinant, the `tFilled` variable will be set to `.false.` by the main loop code. Once the generator has filled in the required information this should be set to `.true.` to indicate that the data may be reused.

The arrays `ClassCountOcc` and `ClassCountUnocc` are initialised by calling `construct_class_counts` at the start of the excitation generator. They contain a count of the number of occupied, and the number of unoccupied, orbitals associated with each spin-symmetry combination. These are used for calculating the probabilities in essentially all excitation generators.

The `occ_list` and `virt_list` variables store only information relevant to the excitation generator in `symrandexcit3.F90`.

The `dorder_*`, `nopen` and `scratch3` variables are used in the CSF excitation generation code.

As there is only one instantiation of this data structure, in the main loop, adding additional elements adds only negligible overhead. If future excitation generators need a place to store information between calls, it should be added here.

The excitation generation routines should be as tightly coupled to the Hamiltonian matrix element generation routines as possible. In particular, information such as the excitation level and the excitation matrix are trivially available in the excitation generator and expensive to calculate otherwise. As it stands, the interface for the excitation generator permits passing the excitation level, parity and excitation matrix to the matrix element generation routines — this is sufficient for determinental systems.

If NECI is to be extended to efficiently consider other basis functions, then other data will be required. A structure should be created to pass this information around, so that each element which is added does not require adjusting the interface of each and every excitation generator. This has been done in the SPINS branch, with a type named `spawned_info_t`, which is unlikely to be merged into master for independent reasons, but may provide a template for doing this if required in the future.

2.6.5 Re-use and rescaling of random numbers

In determinental calculations, even with large basis sets, each site is connected to at most a few thousand others. A single 64-bit random number contains vastly more random information than is required to make a good choice between all of these options.

However, most of the potential algorithms for excitation generation involve a hierarchy of choices. The natural implementation of these choices is to generate a new random number for each new choice that is required. This is highly wasteful, especially as random number generation is relatively expensive.

More bang-for-your-buck can be extracted from random numbers by partitioning them, and rescaling them as you move through the hierarchy of choices. A simple example demonstrates this, considering the macroscopic choice between single excitations and double excitations:

```fortran
real(dp) :: r
r = genrand_real2_dSFMT()
if (r < pDoubles) then
    ! Double excitation selected. Now rescale r
    r = r / pDoubles
    ...
else
    ! Single excitation selected. Now rescale r
    r = (r - pDoubles) / (1.0_dp - pDoubles)
    ...
end if
```

This is not always done in the current implementations of excitation generators, but should always be considered for efficiency.

☞ This technique should not be used where selections may need to be redrawn. Repeated repartitioning and scaling of the random number will rapidly deplete the available randomness if it is repeated a number of times due to redrawing.

2.6.6 Timestep selection and other control parameters

The spawning process is normally the limiting factor for the selection of the imaginary timestep. The magnitude of the spawn, n_s, is given by

$$n_s = \delta\tau \left| \frac{H_{ij}}{p_{\text{gen}}(j|i)} \right| .$$

As such, a limit on the value of $\delta\tau$ may be obtained from the maximum value of that ratio, combined with a (chosen) maximum size of spawn;

$$\delta\tau_{max} = n_{s,max} \times \left(\max \left| \frac{H_{ij}}{p_{\text{gen}}(j|i)} \right| \right)^{-1} .$$

The ratio should be accumulated through a calculation, and can be used to determine the optimum time step on the fly.

This approach may be generalised to a larger number of parameters. Additional control parameters that influence decisions in the excitation generator may be introduces (such as the choice between single and double excitations, or the extent to which a bias is made towards or against double excitations which are spin aligned).

Considering the different choices that are made in the excitation generator, this splits the excitations into a number of different categories. For optimal timestep values the worst case for each of these categories should be optimised to give the maximum spawn size.

The impact of each of the parameters to be optimised must be removed from the generation probability values, so that an unaffected value can be maximised. For example considering a split between single and double excitations;

$$n_{s,max} = \delta\tau \frac{\delta\tau}{p_{single}} \left| \frac{H_{ij} p_{single,iter}}{p_{\text{gen}}(j|i)} \right|_{single} = \delta\tau \frac{\delta\tau}{1.0 - p_{single}} \left| \frac{H_{ij}(1.0 - p_{single,iter})}{p_{\text{gen}}(j|i)} \right|_{double} .$$

In this case the current value of $p_{single,iter}$ (i.e. the value on the particular iteration the generation occurred) is removed from the ration which can then be maximised. Closed form expressions for the optimal values of $\delta\tau$ and p_{single} can then be found straightforwardly.

This approach can be extended, although the means to isolate the impact of parameters on the generation probability, and the structure of the final closed form expressions will depend on the form of the excitation generator.

2.6.7 Testing excitation generators

It is critical that excitation generators are *correct*. That is that all of the connected determinants are generated, and that the generated probability is correct.

There are a number of metrics that can be used to test this. A testing function should be written that takes a given source determinant as a parameter, and runs the excitation generator a very large number of times on this determinant (for example, 10 million times). This test function should contain a number of tests:

Are the correct determinants generated

A list of all single and double excitations of the correct symmetry should be enumerated in a brute force manner (see `GenExcitations3`). It should then be checked that *all* of the determinants in this list are generated by the excitation generator, and no determinants outside this list are generated.

Normalisation of generation probabilities

An accumulator value should be kept for each possible determinant that can be generated. Each time the determinant is generated, the value p_{gen}^{-1} should be added to that determinants accumulator.

On average this will add a value of 1.0 to the accumulator for every excitation generation attempt. Thus, when all of the accumulated values are divided by the number of generation attempts made by the test function, all of the values should give roughly 1.0.

There may be a reasonable amount of variation, but repeated across a few different runs with different random number seeds it should be clear that all of these values give roughly 1.0.

Note that if some connections are *extremely* strongly weighted against (as can happen with the weighted excitation generator) they will sum in a very large term very rarely, and so their averaged value can jump around quite dramatically.

Overall probability normalisation

As an extension to the above, the sum of all of the accumulators should stocastically tend towards the number of connections multiplied by the number of spawning attempts.

If this total accumulated value is divided by the number of connections multiplied by the number of spawning attempts it should average very strongly to 1.0, normally to four or five decimal places.

For an example test function, see `test_excit_gen_4ind` in `symrandexcit4.F90`. An excitation generator that passes these tests is not guaranteed to be correct, but it is quite likely.

2.6.8 How this interacts with HPHF functions

HPHF functions present a complexity for excitation generation. Excitation generation occurs on determinants, but the HPHF function includes the spin flipped pair. When there are only a small number of unpaired electrons, the possibility of the excitation generator having generated the spin flipped version also needs to be available, although this is not readily accessible in the excitation generator.

As a result, a routine must be provided to calculate the probability of generating $|D_j\rangle$ from $|D_i\rangle$ without actually generating an excitation.

☞ Once the excitation generator has been planned out, it makes sense to write this calculation function first (it tends to be simpler than the excitation generator). A call to it can then be included at the end of the excitation generator inside `#ifdef __DEBUG` flags, which provides a powerful correctness check, and catches any cases where these functions diverge.

2.6.9 The uniform selection excitation generator

A full discussion of this excitation generator is found in the paper (...). The operation of this excitation generator is also the basis for a number of other more specialised excitation generators (such as the CSF excitation generator). This excitation generator is implemented in `symrandexcit2.F90`.

Although the generation probabilities associated with this excitation generator are highly non-uniform, it fundamentally makes choices at each *stage* of the excitation generator uniformly between the options presented at this point.

For single excitations, an electron is chosen uniformly at random. This fully specifies a symmetry index, and from the appropriate list a vacant orbital is chosen at random. This selection is generally made by picking an orbital with the appropriate symmetry at random, and re-drawing if it is already occupied (unless there are very few available choices, in which case the n-th available orbital is chosen from an enumeration of available orbitals).

For double excitations, a pair of electrons are chosen at uniform (using a triangular mapping). This specifies the total symmetry of the excitation. A vacant orbital, a, is then chosen at random from the entire array of orbitals (in the same way as above, redrawing if an occupied one is chosen, or if an orbital is selected that has no available orbital b that could produce the correct total symmetry). This in turn specifies the symmetry of orbital b, which is chosen in the same way as the orbital for a single excitation.

The possibility of having picked the orbital b first, and then a must be accounted for in calculating the generation probabilities.

2.6.10 The weighted excitation generator

This excitation generation scheme is discussed in a great deal more detail in a paper which is currently pre-publication. This excitation generator is implemented in `symrandexcit4.F90` as `gen_excit_4ind_weighted`.

Fundamentally, a weighting value is used to bias the choice of orbitals towards those which are likely to correspond to large Hamiltonian matrix elements. This is done using a Cauchy–Schwarz decomposition of the Hamiltonian matrix elements. The specific elements used depend on the spin choices made, and are discussed in the referred paper.

For speed, the entire choice of orbitals of a given symmetry are considered, and the weightings corresponding to orbitals which are occupied in the source determinant are set to have a zero weighting.

For single excitations, an electron is chosen uniformly at random. This determines the target symmetry. A cumulative list of the weighted terms for each of the available orbitals is generated, and a random number on the range of the largest element in this list is generated. A specific index into this list is selected by binary searching for the first element in the cumulative list which is greater than or equal to this random number, and this index specifies which of the orbitals of the given symmetry are selected as the target orbital.

For double excitations, a choice is made of whether the two orbitals should have the same spin or different spin (the probability of either is optimised on the fly at run time). If the two electrons are to have the same spin, they are chosen uniformly using a triangular mapping, and otherwise uniformly using a rectangular mapping. This entirely determines the target combined symmetry index.

A list is constructed, with a length equal to the total number of available target symmetry indices. This is populated with a cumulative count of the product of the number of available orbitals of a given symmetry with the number of available orbitals of the paired combined symmetry required to result in the determined total symmetry. To prevent double counting, there are considered to be no combinations where the first index is larger than the second index. A pair of symmetries are chosen using binary searching as described before, weighted towards the symmetries with the most available choices.

Two orbitals, a, b, with the corresponding symmetries, are then picked in the same way as the orbital was selected for single excitations (although the weighting terms must now consider the effects of both source orbitals). When choosing orbital b, if it comes from the same symmetry category as orbital a the relevant elements must be adjusted in the cumulative list.

For calculating the probabilities, the only duplication of generating the same determinant is for the case when both orbitals have the same spin and symmetry. This must be accounted for in the generation probability.

2.7 Determinant data storage

There are two main locations where blocks of information concerning particles associated with determinants are stored.

2.7.1 Transmitted data

This concerns data that will be (or could be) transmitted between different processors. Essentially this includes persistent information that is generated by the excitation generator. Combined, this information forms the *bit representation* of a determinant.

All data access should be through the getter and setter functions in the `bit_reps` module (`BitReps.F90`), with the exception of the orbital representation which may be manipulated directly as it will always be first. There are general `encode_bit_rep` and `extract_bit_rep` routines which package and extract all of the relevant information, as well as specific accessors.

In most of the code the encoded values for specific determinants are referred to as `ilut`, standing for "Integer Look-Up Table" which relates to the orbital representation.

The packaging of this data varies substantially between different compiler and runtime configurations. For example, integer runs on 64-bit machines co-opt some of the additional bits in the storage of the signed particle counts to store the flags. It is essential that no attempt is made to access the data in this array direcly.

The representation of each determinant is of dimensions `0:nIfTot`, that is of length `nIfTot+1`. Unusually for an array in Fortran it is a 0-based index. The component of the overall representation required to uniquely determine one site is `0:nIfDBO` (DBO stands for DetBitOps), which includes the orbital description and any CSF descriptiors.

Each of the components of the representation has a `nIf*` (Number of Integers For) and `nOff*` (offset of) value. These should not be used directly unless adding data to this representation.

Orbital description

Determinants are stored by a bit-representation of the choice of orbitals to construct their Slater Determinants from. This representation is of length $2M$, i.e. the same as the number of available spin-orbitals. An occupied orbital is represented with a 1, and a vacant orbital by a 0.

This representation can be accessed directly. To simplify things, various macros are available. The macros `IsOcc` and `IsNotOcc` test the occupancy of particular orbitals, and the macros `set_orb` and `clr_orb` modify it without requiring the developer to worry about the data representation.

To decode the bit representation of the orbitals into the natural orbital representation use the `decode_bit_det` routine. Equivalently, the `EncodeBitDet` generates the orbital bit representation from the natural integer version.

CSF descriptors

If CSF descriptor labels are required, they are stored here.

Signed particle count

The representation of the coefficient, or "sign" of the determinant is the primary value which is evolved during an FCIQMC simulation.

These values are represented by an array of `real(dp)` coefficients of length `lenof_sign`. In a standard simulation this length is unity, but for complex walkers two values are used and the double-run code uses extras.

These values can be used via the routines `extract_sign` and `encode_sign`. The specific elements of the sign array can be accessed by `extract_part_sign` and `encode_part_sign`. If all particles are to be removed the routines `nullify_ilut` and `nullify_ilut_part` can be used.

Flags

Any number of flags may be associated with a particular site. A current full list may be found in the file `bit_rep_data.F90`. The most commonly used flags are the `flag_is_initiator` and `flag_parent_initiator` flags — be aware that these are different names for the same thing (depends if considering particles in the main or spawned lists).

There is a pseudo-flag, `flag_negative_sign`, that only exists to help combining the representation of the coefficients and flags. It should never be used directly outside of the representation manipulation routines.0

These flags may be tested using the `test_flag` and `set_flag` routines referencing the specific flag desired from the above list.

Further, then entire set of flags may be extracted or stored using the `extract_flags` and `encode_flags` routines. Once they have been extracted, they may be examined and manipulated with the `btest`, `ibset` and `ibclr` in the same way as the set and test functions above. A special routine clear_all_flags exists for resetting the flag status.

It is extremely important that the flags element of the bit representation is not accessed directly.

2.7.2 Locally stored data

This concerns information that will *never* be transmitted. All of this information is either used in tracking the status of an occupied determinant, or for values that could in principle be regenerated when required but are stored for optimisation.

All data access should be through the get_* and set_* routines found in the module global_det_data.

This data is stored in the global array global_determinant_data, but this array should not be accessed directly! The location of data within this array is determined by the initialisation routine in the module global_det_data, and depends on the calculation being performed.

The data currently stored in this array are

- The diagonal Hamiltonian matrix element for the determinant (diagH),
- The average signed occupancy of the determinant (av_sgn, only for RDMs).
- The iteration on which determinants became occupied, (iter_occ, only for RDMs) and
- The moment in imaginary time on which the (contiguous) occupation of this determinant began (tm_occ, only with experimental initiators).

2.8 Transcorrelated integrals

The usage of the transcorrelated approach for ab-initio systems requires us to include 3-body interactions, that come with 6-index integrals to be handled by NECI, as well as triple excitations to be generated. The triples excitation generator is uniform and is located in tc_three_body_excitgen.F90, while the handling of the 6-index integrals is done by the LMat_class.F90 (for the storage and reading of integrals) and the LMat_mod.F90 (for getting matrix elements).

When reading 6-index integrals, they can either be read from an ASCII formatted file, or an HDF5 file. When reading them from an ASCII file, the file has to be formatted in a general FCIDUMP fashion, containing all non-zero integrals L_{ijk}^{abc} in the format

```
<integral> i j k a b c
```

where i, j, k are the indices of the orbitals to excite from and a, b, c those of the orbitals to excite to. No further information is required in the file. The default filename is TCDUMP.

When reading the 6-index integrals from an HDF5 file, all data is required to be contained in a group called tcdump, containing an attribute named nInts containing the number of non-zero integrals, and two datasets called values and indices, containing the values of the non-zero 6-index integrals and their indices. The indices dataset has to be of 6 times the size than the `values` dataset, each group of 6 indices is attributed to one value (in storage order).