Coding standards for the R-MADAM project [*]

Martin Plummer <martin.plummer@stfc.ac.uk>,
Jimena Gorfinkiel <j.gorfinkiel@open.ac.uk>

Version 1.0
November 2017

Contents

---

[*]Adapted from the coding standards of the UKRmol-in project version 1.0 from January 2010, written by M. Plummer, Joanne Carr and J. Gorfinkiel.

1 Introduction

This document refers to the RMT and UKRmol+ suites being developed under the R-MADAM project (the UKRmol suite of programs, UK polyatomic R-matrix inner-region, was re-engineered

(2009 – 2015) to produce a modern, efficient and maintainable code base). All suites are under version control currently either via the collaborative software development environment tool CCPForge or GitLab:

`http://ccpforge.cse.rl.ac.uk/projects/ukrmol-in/` (but likely to be moved to GitLab soon with access via CCPForge still possible) and

`https://gitlab.com/Masin/Molecular_RMT` (the latter is also visible in CCPForge: `https://ccpforge.cse.rl.ac.uk/project/rmt/`). All developments should use the appropriate version control repository in their day-to-day workflows. The repository and other facilities provided by GitLab/CCPForge constitute a community resource that will make code development, *and therefore the science that can be done with the programs*, more efficient and less error-prone, as long as the community uses it responsibly.

The aim of this document – a combination of coding standards and recommended guidelines – is to promote good programming practice within the community. Good programming practice will enable the writing of correct, efficient, portable code that is straightforward to maintain in the future.

Documenting and testing are an integral part of the code development workflow, and it is of paramount importance that sufficient time should be allocated to them within a project. Programmers should also consider, where possible, the effects of their coding decisions on run-time performance (including memory and disk usage). Sections on these three issues (documentation, testing and performance optimization) can be found in this coding guide.

We recommend that this guide is read as a whole before starting development. Please note, though, that we expect it to be upgraded over the course of the R-MADAM project to deal with any queries sent in by readers and to improve readability. Notification of changes will be given via the project mailing lists. However, the basic coding standards set out here will be maintained over the course of the project and beyond. If there is any doubt about something covered in this document, or the application of a feature not discussed here, then please consult before coding. Code that does not satisfy the parts of this document that are strict requirements (rather than guidelines) simply will not be incorporated into the release versions of the programs. Furthermore, if such code is committed to the main development stream of the version control repository then it runs the risk of being removed from future revisions.

The authors originally took guidance and inspiration from the coding guide for the materials science code CASTEP (`http://www.castep.org`): we acknowledge the CASTEP Developers' Group for this.

2   Fortran 95 and Fortran 2003: Standards Compliance

RMT and the 2009–2015 UKRmol code are written in Fortran 95 with some legacy Fortran 77 coding, while the new UKRmol+ inner region suite has been completely rewritten in Fortran 2003 with new functionality. From this starting point, we have devised the following rules and recommendations.

Fortran 2003 should be used only for new code in clearly defined groups of modules or for full rewrites, by people who are confident in using it. Fortran 95 should be used by people who are not (yet) happy writing Fortran 2003, or who are making minor adaptations to existing code or writing with the aim of translating physics into code without wanting to think first or simultaneously about the optimized HPC implications (we understand but don't recommend the latter approach). If people are moving to Fortran from scripting languages (or are used to C/C++ programming) then they should be encouraged to learn Fortran 2003 as the basic Fortran language, if courses are available, but be aware that of a lot of its object oriented features are only just becoming 'standard' on some compilers. The page
`http://fortranwiki.org/fortran/show/Fortran+2003+status`
helps here, and validated tests of the Fortran 2003 code should be provided by the developer.

Fortran 95 code should comply with the Fortran 95 standard (apart from the two extensions discussed below). This can be tested using either a compiler set to flag up non-standard coding (note that the NAG compiler is known to be strict on standard compliance even without setting a flag) or a tool for static analysis such as Forcheck (which is available from the STFC-SCD Software Engineering support Centre, `http://www.softeng-support.ac.uk`, for example). This document assumes the reader knows and understands Fortran95. Additional explanations will be added in updates according to reader feedback.

Overall, as far as possible, we would like to emphasize from the beginning the importance of localized coding that can be debugged, further developed and otherwise adapted and optimized with minimal resulting changes to the rest of the bulk code, but which on the other hand doesn't involve code (and data assignment) duplication or adversely affect performance.

2.1 Extensions to Fortran 95

There are two extensions to Fortran 95 that we will consider as 'standard', even though they strictly belong to Fortran 2003:

- Use of allocatable arrays in derived types (from Technical Report 15581: see, for example, `http://www.nag.co.uk/nagware/np/doc/TR.asp`).

- Calls to C routines via 'iso_c_binding' constructs. The incorporation of C code should be avoided unless it is necessary (usually for direct interactions with the operating system) or a significant performance gain would result. The C code must only be accessed via an interfacing module (see section 4.1).

The above features are accepted by most/all modern Fortran95 compilers. Other concepts from Fortran 2003, or from High-Performance Fortran (e.g. PURE), should not be used within Fortran 95 code without consulting the authors of this document. If such code is considered essential for performance on a particular machine but does not compile on other machines, the code should be isolated within a module and an alternative provided. The choice of routine may

be made at the time of the invocation, either by an input parameter to the wrapper routine (see Section 4.1) or via an explicit interface (see Section 4.3), or by preprocessing commands (see Section 2.3) in the isolated module, whichever is most practical.

2.2 Guide notes on Fortran 2003 (and possible extension to Fortran 2008)

We are assuming that people confident in coding in Fortran 2003 will be expert enough to conform to the standard. Fortran 2003 has lots of advantages in being able to define properties of types and have them inherited along with a more built in expectation of 'contained' localization so that changes to one part of the code require zero or minimal changes to the rest. However, one of the original advantages of Fortran was that once the code has been fully debugged, it is then relatively easy for people primarily interested in science to read and work out what's going on (Formula translation) while still producing efficient performance. Ultra-heavy use of Fortran 2003 can obscure this clarity if people are not careful or expert enough.

Where a piece of Fortran 2003 coding is extremely helpful for performance, memory, structure etc but parts of it don't work with commonly used compilers, the developer should provide a workaround (either Fortran 95 or alternative Fortran 2003 for particular circumstances) for that part which keeps it reasonably portable.

Similarly, if an expert developer particularly wishes to use features of Fortran 2008 (or higher) that are particularly advantageous, then detailed documentation, test cases and a similar alternative 'standard' coding should be provided. These extensions should be discussed with the authors.

2.3 Compiler/preprocessor directives

These should be avoided wherever possible in high-level 'driving' routines. If necessary, they should only be used in interfacing or utility modules (see Section 4) for machine-specific code. Note that files needing preprocessing should use extensions of the form `.F90` rather than `.f90`.

3 Style and Layout

3.1 Types

Do NOT use implicit typing. All routines should have an `IMPLICIT NONE` accessible. This should be followed by explicit declarations of any input variables, including an `INTENT(...)` attribute. Then, declarations of local variables should be made. See also Section 3.3.

However, a convention for naming variables that follows the default rules of implicit typing is strongly recommended, i.e. the names of integers are chosen to start with letters in the range

I–N. Exceptions can be made for global integer parameters with their own history or conventions (such as `wp` below).

Note: a utility module that contains only data and is named **precisn** contains definitions of certain parameters that may be used throughout the code, particularly the specification of 'kinds' of variables. Therefore, do not give type as `DOUBLE PRECISION`, but use `REAL(KIND=wp)` instead, where the integer parameter `wp` is defined in the **precisn** module. Similarly, constants should always be written as `1.0_wp` rather than `1.0D0`, and transformations are `REAL(a,wp)` and `CMPLX(a,b,wp)`, etc. Note that inclusion of the kind parameter is essential when using `CMPLX`. Any local machine-dependent operations then need only be applied to the **precisn** utility module. All precision definitions should come from the **precisn** module: compiler flags to alter precision are then not needed and should not be used.

3.2 Names

For clarity, don't change variable names on passing to a different routine wherever possible. Avoid short (1 or 2 letter) variable names except for simple loop counters. Avoid using the names of intrinsic functions as variables as this can be confusing (the popular text editors operating in Fortran mode as well as static analysers such as PlusFort can help identify intrinsic function names). Use the form `END SUBROUTINE <name>` etc. rather than the older form `END`.

We recommend one convention that has been used to good effect in other packages, whereby all subprogram names in a given module begin with the same stem (related to the module name) so that they may be easily identified. For example, routines in the **comms** module (Section 4.1) all begin with the stem `comms_`.

3.3 Comments and subroutine styling

The code should be commented liberally. Comment density should ideally approach code density. Commenters should ensure their remarks are attributable: unless the comment is from the (sole) main author of the code then their initials (or ID, etc.) should be used to mark it. We will adopt the convention that all comments are to be in English.

Each file containing source code should begin with a header section comprising comments. This should document at least:

- the overall purpose/function of the code within the file

- its relationship to other files

- the identity of the main author(s) with an approximate date for the development.

As further work is done on a file, a list in this header should be built up of changes that are 'major' (in the view of the developer). A brief explanation of the change should be given, as

well as a date and the identity of the developer. This in-situ information supplements but does NOT replace the log file from the version control system, which documents all changes. If more than one module is in a file then a description for each should be given – but note that such organization of modules is not recommended for anything other than collections of very small utility modules.

Each non-trivial subprogram must also begin with a comment section, containing at least the following information:

- an explanation of the purpose/function (including a description of / references to the underlying physics, where relevant)

- a note on any limitations of the routine

- a list of the module variables used (noting which if any might be modified by the subprogram)

- a list of any necessary conditions for entry

- the main author and date

Following the introductory comments in this Section, any `use` statements, in the form `use <module>, only:` (see section 5.2), should also be (briefly) commented. Input and output variables should be declared before local variables. The declarations of (at the very least) the input, output and important local variables should be commented with definitions – brief descriptions of what they are.

3.4 Layout

Indentation should be used for loops, logic, etc., to aid readability. Popular text editors can help with this (e.g. GNU-Emacs f90-mode).

3.5 Case

Developers should be aware that no restrictions on the use of upper/lower case have yet been made. However, switching case or using different capitalization patterns for the same variable in the same code is annoying, risky and should be avoided! One popular convention is to use lower-case, except for intrinsic functions and variables, which are written in upper case.

3.6 Format

Code may be in free format, with a maximum line length of 132 characters. Up to 39 continuation lines are allowed by the standard, but we do not recommend taking advantage of this as the code

will not be clear.

3.7   Spaces

Use spaces between keywords e.g. `END IF` not `ENDIF`.

3.8   Code cleanly

This is crucial, to ensure correctness and make it easier for others to maintain the code. If the obvious way to code a routine is inefficient, whilst the efficient way is obscure, then the efficient way is allowed as long as the routine is well documented, particularly in terms of comments in the code.

3.9   Code robustly for errors

The user should never see a Fortran run-time error message – your code should check for possible error conditions, and either correct the error or print a nice, meaningful error message and exit gracefully.

3.10   Be aware of vectorisation

Code with a style that enables good vectorisation, e.g. don't put `IF` statements inside nested loops where possible. This is very important for modern processors such as those used on the national resource ARCHER (`http://www.archer.ac.uk`), which rely on taking advantage of 'micro-vectorization' to achieve high performance, and which charge for time assuming this high performance has been achieved. We recommend you check the documentation and guides for local machines.

3.11   Use of allocatable arrays

Code explicitly to avoid memory leaks: allocatable arrays should be deallocated after they are no longer needed, and certainly before exiting the routine if they are local, whether or not the compiler will do this for you automatically (strict Fortran95 does perform this task, but explicit deallocation aids clarity of code). Allocated arrays that are required over several calls to the routine should be defined with the `SAVE` attribute (as should all such data). It is recommended that allocate/deallocate statements only occur once in a routine for a particular array. Avoid placing them inside loops as this can result in a significant slowdown. Allocatable arrays should be used rather than pointers (see Section 6.2).

3.12 Flow of control

A `CASE` construct should be used in preference to an `IF` construct where appropriate (i.e. there is a single expression to be evaluated). Use `DO ... END DO`, with name-labelling of long 'do loops' recommended for clarity. Within do-loops, use `CYCLE` and `EXIT` where appropriate, i.e. if the loop will not vectorize and placing a test inside the loop is necessary, but only to move a maximum of one level of nesting. Avoid the use of `GO TO` (but note that some may remain in the legacy code).

3.13 Functions

A function subprogram should be 'pure' in nature, i.e. generate no side-effects through modification of variables in the argument list or module variables through use association.

3.14 Line numbers

Avoid where possible. For example, use `DO ... END DO` with indentation rather than `DO 10 i = 1, n ... 10 CONTINUE`. Line numbers should ideally only be used for format statements, and `END=` or `ERR=` in file handling. Do loop name labels may be used for clarifying multiply nested do loops or for do loops with large amounts of content between `DO ... END DO`.

3.15 Revision number Note: this is yet to be implemented for the RMT code in GitLab.

One of the many advantages of a version control system for source code, if used as intended, is that one should easily be able to retrieve the exact code used in an earlier calculation.

Each program when run should include the revision number of the code from which it was built as part of the 'header' information written as human-readable output. For a particular 're-lease' version of the suite, this revision number will be meaningful and constant (e.g. v1.0). For code under development, the necessary information is currently extracted from the Subversion repository and added to an intermediate source file as part of the build system; the developer should ensure that a write statement in the appropriate form is present in the code. The existence of local modifications with respect to the repository will also be recorded in the output via this mechanism, as will the distinguishing part of the name of the appropriate development stream/branch (e.g. /trunk/source or /dev-branches/<branch-name>/source). Note, however, that using an executable built from source that does not correspond exactly to a revision somewhere in the repository (and therefore cannot easily be retrieved in the future) is poor version control practice for a production run.

4   Organization of the code

4.1   Interfacing and utility modules

All access to low-level subprograms in 'libraries' must be wrapped. That is, the specific subprogram name must not be used anywhere in the code apart from within an interfacing subprogram. For example, instead of invoking directly the LAPACK subroutine `DSYEV` to calculate the eigenvalues (and optionally eigenvectors) of a real symmetric matrix, a wrapping routine should be called. The wrapping routine ensures that the required data is present in a suitable form, calls `DSYEV`, and then manages the returned data. This also applies to libraries such as PetSci and slepsi, with direct calls in a separate localized module.

The following are considered to be low-level subprograms:

- Routines from external maths libraries (e.g. linear algebra). It is a debatable point whether BLAS routines should be included in this rule: we do not insist on it for the moment. On the one hand they are an external library; on the other hand they should be installed in machine-optimized form on any machine on which it is worth running the code. Whilst we definitely recommend the use of Level 2 and Level 3 BLAS routines for all appropriate operations, we probably don't recommend Level 0 BLAS routines as the calls can make the code confusing and there is very little if any gain in performance on modern machines.

- Routines and library calls devised to make use of heterogeneous architectures and accelerators (GPUs, Xeon-Phi etc). While use of available accelerator technology is encouraged for performance and future-proofing (provided it is not being prioritized over essential functional development), the actual accelerator code needs to be in a distinct utility-level module so that it can be called when the hardware is available. We do not want completely separate bulk code for (each) accelerator application. The wrapper calling routines should able to interface with standard code and each kind of accelerator code, either by choice at compile time or by a flag parameter to be set as input, or by use of overloaded interfaces (section 4.3).

- MPI for parallelism: all MPI routines and variables should be in a **comms** module with only the wrapper names used outside the module. This also applies to MPI-defined communicators (e.g. `MPI_COMM_WORLD`), which should never appear outside the **comms** module. Communicators used outside the **comms** module should all have wrapper names which are passed into the **comms** module in order to define/use the communicator. This allows a serial module with dummy routines to be substituted for the parallel module if required. It also allows the possibility of performing multiple parallel runs relatively easily as a single job, by introducing a new higher-level communicator and 'master' control code.

- I/O functions (including `OPEN, READ, WRITE`). Depending on the system, different I/O methods will be appropriate: formatted, unformatted, direct-access, portable XDR,

external I/O libraries, automated file number allocation, etc. By keeping all specific I/O calls in one place, only one module needs to be modified or expanded to increase the I/O possibilities for the whole code.

- Integer packing: this is highly system-dependent and goes against the portable, standards-conforming ethos.

- Timers.

- Calls to routines written in C code, accessed via 'iso_c_binding' constructs: all the references to these constructs and types are isolated in the module containing the wrapper routines.

The interfacing subprograms should be collected in modules, (at least) one for each necessary item in the above list.

Some of the interfacing subprograms simply involve calls to routines in external libraries (e.g. LAPACK, ARPACK, MPI) and packaging of the data; others require code that is specific to the suites (e.g. UKRmol-in) in the project (such as the integer packing and some of the I/O).

Data and remaining procedures which are common to different programs in the suite should be placed in 'utility' modules, in separate files from the programs themselves. This includes the **precisn** module that is used for defining types. (Note that the **precisn** module is *shared* the RTM and UKRmol/UKRmol+ suites. Therefore, no changes should be made to it without making sure that they are appropriate for both suites. And, when changes are made, the module should be updated in *both* repositories.)

This use of interfacing and utility modules will aid portability, as the routines that need to be changed to use a more efficient linear algebra library (for instance) are localized rather than being spread throughout the code. See also Section 5 for notes on the use of modules.

4.2   OpenMP and mixed-mode MPI/OpenMP

The RMT code contains advantageous and carefully targeted use of OpenMP parallelism. OpenMP basic commands appear as comments in the code and are switched on by compiler flags. The chain of public or private variables affected by OpenMP use can become obscure if this use involves several called subroutines and module variables rather than variables strictly contained in a subroutine. We recommend that OpenMP code remains reasonably simple with clear lists of PRIVATE/SHARED passed variables and THREADPRIVATE module variables, with tests included to check there are no race conditions and that everything has the value it should have. The chain of affected variables and routines should be clearly documented. In mixed-mode code, the OpenMP should normally be fully contained within a particular MPI task. More ambitious OpenMP coding and newer concepts as introduced in OpenMP 3 and above should generally only be attempted when distinct sections of code are being written from scratch, with the agreement of the senior developers.

4.3 Overloading

As programming gets more sophisticated, the use of calls to 'generic' and overloaded routines in utility and interfacing modules will simplify the code structure in the higher-level modules and routines. The lower-level modules should contain `INTERFACE` constructs which gather together several similar routines which, for example, perform (effectively) the same operations on different types of argument, within the generic interface. For example:

```
module module_name
use precisn, only: wp

private
public my_task

interface my_task
  module procedure my_task_integer_scalar
  module procedure my_task_real_scalar
  module procedure my_task_real_vector
  module procedure my_task_complex_symmetric_matrix
  module procedure my_task_hermitian_matrix
end interface
```

The correct routine is automatically identified from the arguments at the time of invocation.

Other public subroutines (see Section 5.1) within a utility or task module may be overloaded, i.e. via optional arguments which need not always appear in the calling routine.

4.4 Checkpointing

Developers are encouraged to consider strategies for checkpointing their code, where appropriate, to allow restarting. This is essential for programs that are to be run on shared-resource systems with batch queue submission, and for local systems that are subject to power cuts or automated energy-saving switch-offs. It is also relevant in the most time-consuming parts of the program package, taking account of scaling with system size, for which intermediate manual checks of output may be desirable during very long runs. When writing checkpoint strategies, take into account the demands of storage (file sizes) and time for the extra I/O (on very modern machines, starting and finishing I/O can take up significant fractions of the total execution time.

For example, a timer could be used to automatically determine when the code has reached a user-defined time limit (set via an input parameter), in which case a checkpointing file is written and the program exits gracefully.

4.5 Debug mode

Coding a debug mode is recommended, i.e. the addition of debug statements that are optionally executed, under input control via a keyword. Care should be taken over the amount of information generated and the time for extra I/O. The debug mode should be written so that it can be switched off with minimal computational effort for production runs.

5 Modules

5.1 Data hiding

Modules should use an initial `PRIVATE` statement to force all entities in the module to be private by default. Variables, datatypes and routine names should then be explicitly exposed via the use of `PUBLIC` as required. Exceptions may be simple modules containing global data (e.g. modules that have directly replaced common blocks) or modules containing many simple utility routines, all of which are public. `SAVE` attributes should be used whenever assigned data is needed over several calls to a routine, and hence in data modules.

5.2 Use of `USE`

Where routines use variables and subprograms from other modules, these should be noted explicitly via a `USE <name>, ONLY :` statement. Exceptions to this, for which the `, ONLY :` may be omitted, are simple modules containing global data (e.g. modules that have directly replaced common blocks) when the majority of the contents are needed, or utility modules in which (nearly) all of the public routines are utilized in the same calling module. It is recommended that appropriate `USE` statements should go into the subprogram, rather than a module header, for clarity and in order to keep track of module variable modification. Exceptions may be made for 'global' modules such as **precisn**.

6 Performance Optimization

We have included performance hints and recommendations in the bulk of the document where appropriate, for example, use of BLAS routines, inner-loop vectorizability, etc. We note the contradiction raised in section 3.8 concerning simple code and optimized code. To a large extent compilers will optimize code for you, but not as much as the manufacturers will claim. Nevertheless, test the code against increasingly high levels of optimization, checking that the results are safe, and you will see performance gains.

We require high-performance code for ambitious science. Take as much time testing the

code and profiling it as you do writing it. Parallel code should be tested for load-balancing, serial hot-spots and general scaling properties using internal timers and any parallel profiling tools that are available. Make sure that where code is necessarily obscure, the commenting is not obscure. Avoid repeating calculations unnecessarily, especially with calls to other functions and subroutines where variables that could be saved are time-consummingly recalculated. Avoid heavy use of 'raising to a power'. These practices are a major cause of bad serial performance.

Also, be aware of the memory limits of different 'target' machines at different cache levels and main memory. Performance will be hit as the required memory goes above these levels. This is again a contradiction as we want long-lasting code and system performance requirements change continuously. The use of interfacing and utility modules is one way to try and make sense of this. Attend 'how-to' courses on running codes on your target machines.

Note: recent (2009) advice on optimization as a response to new hardware technologies recommends the use of 'single precision' (kind `sp` in **precisn**) real variables for operations where floating point arithmetic is required but full accuracy is not. This can be very dangerous if `sp` is used where `wp` is needed but can lead to substantial performance improvement and gains regarding cache-memory use when constructed safely. We do not forbid this but any such use should be clearly commented and the developer should provide evidence of the improved performance. If this framework for floating point operations proves to have major benefits on modern and future hardware, we will recommend it more strongly. Note that 'old' considerations of floating point performance and memory which used to be applied in the context of 'large' computers, are resurfacing as considerations in the context of level 1 cache and internal (possibly simplified/streamlined for pure number-crunching) cores of individual 'processors'.

This section will be added to over the course of the project.

6.1   Passing arrays to subprograms

There are various ways of passing arrays, each of which has different consequences in terms of bounds-checking, memory use and compiler optimization.

The best practice is to always pass the whole array and make its extent in each dimension available to the subprogram by some means (e.g. as one of the other arguments or via a module).

This does not cause the array to be copied and allows the compiler to do bounds-checking. The recommended method for passing arrays is therefore into explicit-shape array dummy arguments.

Assumed-size arrays, such as `myarray(*)`, are to be avoided where possible as their use precludes bounds-checking. The use of assumed-shape arrays, such as `myarray2(:,:)` is also less efficient as these have an explicit rank, but an implicit shape (the use may also lead to explicit copies of the array being created in the called routine).

Thus, if a section of an array is to be passed, pass `array(i_start:i_fin, n)` and either both `i_start` and `i_fin` or `i_fin - i_start + 1`.

6.2 Pointers

Generally, pointers should not be used unless there is a very good reason for doing so, such as the `iso_c_binding c_ptr` (isolated within the interfacing module), or if a 'legacy Fortran 90' utility module requires them (again the pointers should be isolated in the code by wrapper routines in a module).

For expert users, this rule can be relaxed where pointers are particularly useful in additional interfacing with C code, looking at decision trees (eg for parallel I/O and passive remote memory access for parallel work that wouldn't otherwise be load-balanced), linked lists and setting up shared memory segments for node-accessible memory if the said expert user is happier. However the influence of the pointers should be strictly contained within the module/routine and they should only be used for specific purposes clearly described by the programmer.

7   Documentation

The code should be self-documenting internally via comment statements (see Section 3.3). 'Automatic' documentation generators (such as doxygen, robodoc) may be used in the future to generate a developers' manual.

Additionally, external documentation in the form of a user guide is essential for each program. These should detail at least the input namelists (with default values and ranges), changes to any input variables or module variables (very important), and the outputs. The current user documentation can be taken as a starting point – this can be found in the /doc subdirectory of each branch in the version control repository, i.e. alongside /source. It is the individual developer's responsibility to ensure that a change in the code affecting the I/O is correctly documented.

8   Testing

Implementation of automated testing is planned for the UKRmol+ and RMT suites in the near future.

We have mentioned (though not exhaustively) testing for performance in Section 6; we now consider testing for precision and the correctness of results. A comprehensive test suite, under version control, is currently under construction (one is available for the UKRmol suite). Once the test suite has been built up, the author of a new piece of code should construct and run specific tests for that code, as well as running regression tests (i.e. the standard test cases) that aren't explicitly for the innovation, but which check that nothing pre-existing has been spoiled. New code, and hence new test cases, should be executed where practical using available debugging tools and all debugging compiler flags (i.e. flags that ruin performance but give valuable information on how robust the code is), to remove as many bugs as possible before running optimization tests and the existing test suites. It is most helpful if the code is tested on different platforms

and with various compilers, in order to maintain portability. When committing a new revision of the code to the version control repository, please describe the extent to which the code has been tested.

It is of paramount importance that code that either fails to compile or fails some of the accepted test cases is NOT committed to the 'trunk' of the version control repository (svn) or merged with the master version. Create your own branch for long-term and/or potentially destabilizing development work.

Future upgrades of this document will include notes about 'unit testing' of particular modules, 'coverage analysis' of tests for larger sections or the whole code (to be introduced with help from SESC as part of the 'SESC Build Service').

8.1 Reference data

Each test case should comprise input datafiles and reference output results. Correct operation of the test will be verified by comparing the actual output against the reference output (to within numerical tolerances etc).

Any information on the coverage (in terms of lines of code executed) of a test case would be welcomed. If available, the output from a coverage tool can be added to the reference data, for instance.

9 Committing (pushing) code to the main repository

Whenever code is committed to the repository, developers must ensure the commit log contains all necessary information for others to understand and follow the changes that have been implemented. The amount of detail required will depend on the extent of the modifications. However, even very small changes need at least some basic information to be provided. We suggest that, for commits of small changes, developers provide the information currently required by the UKRmol/UKRmol+ repository on CCPForge:

- changes to functionality and/or bur or problem solved: what is the change for

- subroutine, module, program, etc, affected: a clear statement of what part of the code(s) have been modified

- dependencies: if the changes will affect another program and how

- changes to required input: if changes to the input are needed (even if they are optional) this should be stated

- status: a description of how and to which extent the code has been tested and whether the work is completed

A commit template with these headings is available in the RMT repository in GitLab. More extensive changes may call for a more detailed description.