Science & Technology Facilities Council

Technical Report
RAL-TR-2011-015

Guidelines for the development of HSL software, 2011 version

JD Hogg, JK Reid, JA Scott

August 2011

RAL Library
STFC Rutherford Appleton Laboratory
R61
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1358- 6254

Guidelines for the development of HSL software, 2011 Version[1]

by

J. D. Hogg, J. K. Reid and J. A. Scott[2]

Abstract

HSL is a collection of portable, fully documented, and tested packages for large-scale scientific computation. HSL is primarily written in Fortran; MATLAB and C interfaces are available for some packages. It has been developed by the Numerical Analysis Group at the Rutherford Appleton Laboratory, with additional input from other experts and collaborators.

The aim of this report is to provide clear and comprehensive guidelines for those involved in the design, development and maintenance of software for HSL. It explains the organisation of HSL, including the use of version numbers and naming conventions, the aims and format of the user documentation, the programming language standards and style, and the verification and testing procedures.

This version supersedes RAL-TR-2008-027.

Keywords: HSL, software development, software testing.

---

[1] Current reports available from 'http://www.cse.scitech.ac.uk/nag/reports.shtml'.

[2] The work of the third author was supported by the EPSRC grant EP/E053351/1.

Computational Science and Engineering Department,
Atlas Centre, STFC Rutherford Appleton Laboratory,
Oxon OX11 0QX, England.

August 17, 2011

Contents

1 Introduction

HSL is a collection of portable, fully documented, and tested Fortran codes for large-scale scientific computation. HSL began as the Harwell Subroutine Library back in 1963, making it one of the oldest such libraries. Over the past four decades, HSL has continually evolved and been updated as new algorithms and codes have been developed and older codes have been superseded. It moved to Fortran 77 in 1990 and now consists of a mixture of Fortran 77 and Fortran 95 codes that are all threadsafe. A number of packages are offered with MATLAB and/or C interfaces. The majority of HSL codes are written and developed by the Numerical Analysis Group at the Rutherford Appleton Laboratory (`http://www.cse.scitech.ac.uk/nag/`), with additional input from other experts and collaborators. Among the best known HSL packages are those for the solution of sparse linear systems of equations and sparse eigenvalue problems.

HSL offers users a high standard of reliability and, over the years, it has gained an international reputation as a source of robust and efficient numerical software. Furthermore, HSL software is fully supported and maintained.

Many of the older HSL codes have gradually been superseded by newer versions, with increases in functionality, improved interfaces, or speed of execution. As a result, HSL is now arranged in two parts, the main library (currently HSL 2011) and an archive, HSL Archive. The HSL Archive comprises older codes that were part of previous releases of HSL, many of which have been superseded by more modern codes. We do not maintain the codes in HSL Archive and the rest of this report refers to the main library except where HSL Archive is explicitly mentioned.

HSL 2011 and the HSL Archive are arranged as collections of Fortran 77 and Fortran 95 **packages**, each of which consists of either a single program unit or a set of program units. Almost all the Fortran 77 program units are subroutines, but there are also some functions and some HSL Archive packages contain block data subprograms. The Fortran 95 program units are mostly modules, but there are some external procedures. Each package performs a basic numerical task and has been designed to be incorporated into programs. Each has its own user documentation, which gives full details about how to use the package (see Section 3.1), and is available as a PDF file. Note that a number of optimization-related packages were removed from HSL with the release of HSL 2011. Replacement packages are available in the GALAHAD library (see `http://www.galahad.rl.ac.uk/`).

The aim of this report is to provide clear and comprehensive guidelines for those involved in the design, development and maintenance of software for HSL. Section 2 explains the organisation of HSL, including the use of version numbers and naming conventions. In Section 3, we discuss the user documentation, which is an important part of the Library. The programming language and style that is used by HSL is discussed in Section 4. Interfacing to non-Fortran languages is described in Section 5. Then, in Section 6, we explain the verification and testing procedures that we employ before a new package is accepted. The role of the Librarian is discussed in Section 7 and commercial issues are the subject of Section 8.

2 Organisation of HSL

2.1 Versions

Prior to 1990, HSL evolved continuously. New packages were added whenever they were ready. Since then, additions have been collected into releases, with intervals of around three years between releases, although packages are made available as soon as they ready. The original motivation for having formal releases was for bound sets of documentation (both the catalogue and the user documentation for the individual packages) to be printed. Nowadays, a online catalogue is available (`http://www.hsl.rl.ac.uk/catalogue`), but it remains convenient to construct a new PDF catalogue only occasionally and it is convenient for publicity, too, to have distinct releases. The PDF catalogue and PDF versions of the individual user documentation are available by links from the online catalogue.

With HSL 2004, we started to mark each version of a package with a version label of the form $l.m.n$ where

l is a digit that labels major changes that affect the interface,

m is a digit that labels a technical change, usually a bug fix, that does not affect the interface, and

n is a digit that labels an editorial change such as the correction of a spelling error in an output format or a non-trivial alteration to the documentation (typos in the documentation do not lead to a new label) .

The first version of a package is labelled 1.0.0. We did not attempt to mark the changes prior to HSL 2004 with version labels.

The documentation, which is described in Section 3, includes the version label. In addition, the leading comments in the source code for each package provide a copyright statement and, for each version, a brief description of the changes that led to it.

2.2 Naming conventions

Each Fortran 77 package has a sequence of two letters and two digits as its name. The name of each program unit within the package starts with these four characters. The fifth character, which is always alphabetic, is used to distinguish between the program units. The sixth character of the name (or its absence) identifies the type of its principal argument. The possibilities are

Absent	Single precision.
`D`	Double precision.
`I`	Integer.
`L`	Long integer.
`C`	Single precision complex.
`Z`	Double precision complex.

If the principal argument is always complex, the sixth letter is absent or `D`. If there are more than 26 procedures in a package, another four-character name is used for these but they are still regarded as a part of the package; for example, `EA16` has procedure names that start with `EA16`, `EA17`, and `EA18`.

The name of each Fortran 95 package starts `HSL_` and is followed by a sequence of two letters and two digits. In most cases, this is its whole name, but occasionally this is followed by a qualifier such as `_ELEMENT`. The letters in the name are written in upper case. The package usually consists of one or more modules, but may also contain some external procedures. Each module name starts with the package name and is followed by a qualifier to identify the type of its principal argument. The possibilities are:

`single`	Single precision.
`double`	Double precision.
`integer`	Integer.
`long_integer`	Long integer.
`complex`	Single precision complex.
`double_complex`	Double precision complex.

If the principal argument is always complex, the qualifier is `single` or `double`. Each public name is the sequence of four characters (excluding the leading `HSL_`), followed by a qualifier (if any), followed by further characters.

With the appearance of hardware for which single precision working is significantly faster than double precision, an important use of single precision versions is in mixed precision computations where the single precision result is computed and then refined to double precision accuracy.

2.3 File naming conventions

The naming convention that is used for the files of a double-precision package that uses the free source form are summarized in Table 1. For files in the fixed source form, the suffix `.f` is use instead of `.f90`. All the Fortran 77 packages use the old source form, of course. All of the Fortran 95 packages except `HSL_MP01` use the new source form. By 'source', we mean the source code of the package itself, excluding any modules accessed from elsewhere (e.g. other HSL packages) or procedures invoked from elsewhere. By 'test', we mean the comprehensive test (see Section 6.2). By 'spec', we mean the first or only simple example in the user documentation (see Section 3.1.5). By 'other spec', we mean another simple example in the user documentation.

Single-precision versions are as for double with the `d` replaced by `s`. Integer versions are as for double with the `d` replaced by `i`. Where a package contains both real and complex versions, the complex version is as for double with the `d` replaced by `c` and the double complex version is as for double with the `d` replaced by `z`.

Table 1: The file naming convention for double-precision Fortran 95 packages

directory:	*package*
source:	*package*`d.f90`
test:	*package*`dt.f90`
test data:	*package*`dt.data`
test output:	*package*`dt.output`
spec:	*package*`ds.f90`
spec data:	*package*`ds.data`
spec output:	*package*`ds.output`
other spec:	*package*`ds1.f90`
other spec data:	*package*`ds1.data`
other spec output:	*package*`ds1.output`
other spec:	*package*`ds2.f90`
other spec data:	*package*`ds2.data`
other spec output:	*package*`ds2.output`

3 Documentation

Each HSL package is accompanied by user documentation that the user will need to read to find out how to use the package. For many of the more complex packages, which cannot be fully described in this way, a separate technical report is written and made available at `http://www.cse.scitech.ac.uk/nag/reports.shtml`. In this section, we discuss the user documentation in detail and then briefly comment on the aims and objectives of writing a report to provide users and other researchers with further information.

3.1 HSL user documentation

For most potential users, their first in-depth contact with the software is likely to be through the user documentation. It is therefore essential that it is clear, concise, and well written. Writing good user documentation is an integral part of the design process and developers of HSL software are strongly encouraged to draft the user documentation in the early stages of code development. This allows others in the Group to comment on the design and make suggestions on the options offered and on the user interface.

The user documentation is written in Latex or TSSD. Starting with HSL 2011, the PDF catalogue is written in Latex. New user documentation will usually be written in Latex. A template Latex file together

the necessary style files are provided in `/numerical/num/hsl2011/hsl_latex` to assist developers. As already noted, the typeset versions of the user documentation are made available to users in PDF format.

The complexity of the user documentation will, in part, depend upon the complexity of the package and the range of options that it offers. But, for each package, it follows a similar format. In the rest of this subsection, the various sections of the user documentation are explained.

3.1.1 SUMMARY

The summary begins with a single sentence that states the purpose of the package, with the most relevant part in bold. Often the reader should be able to decide whether or not the package is suitable by reading this one sentence.

It continues with a fuller description that introduces any mathematical ideas and notation that are needed in the rest of the document. This is usually less than about half a page in length, but sometimes it has to be longer than this. The summary ends with a list of attributes that include:

Version: The version label with the date that the version was released (see Section 2.1).

Types: The types offered (see Section 2.2).

Remark: (Optional) Any useful information not covered otherwise. Examples are that at least 8-byte arithmetic is recommended, that the package supersedes or is superseded by some other package, or that it is a front end for some other package. If the development of the package was supported by a grant (or other funding), this should be highlighted here.

Language: Fortran 77 (some packages call the Fortran 95 intrinsics `CPU_TIME`, `EPSILON`, `HUGE`, `TINY`, and some make use of `COMPLEX*16`), Fortran 95, or Fortran 2003 subset (we allow Fortran 95 + TR15581 + C interoperability).

Parallelism: Either 'MPI' or 'OpenMP available'. This must be noted for packages that use parallelism directly or through the procedures that they call.

Calls: The names of the packages containing procedures that are called or the generic names of other procedures that are called or used. This will include other HSL packages, BLAS routines and any LAPACK routines that are called (see Section 4.11).

Original date: The date that the package entered HSL.

Origin: The names of the authors and their affiliations at the time of writing the package.

Interfaces: For packages with more than one interface, a list of the available interfaces (Fortran, MATLAB, C).

3.1.2 HOW TO USE THE PACKAGE

The main body of the documentation explains how to use the package. This should be straightforward to understand, avoiding as far as possible technical terms and details of the underlying algorithms because the reader may not be an expert in the area. The input required from the user and the output that will be generated should be clearly described, with full details of any changes that the code may make to the user's data. This section is divided into a number of subsections.

If the package contains modules, the first subsection starts with an explanation of the `use` statements that are needed. Two modules of different types may contain the same public name if it is expected that they will rarely be used together. In this case, the documentation explains what renaming will be needed should they be used together.

If the package uses OpenMP, there should be a warning that the user who wishes to take advantage of this needs to tell the compiler about OpenMP at compile time. The standard wording is

OpenMP is used by *package* to provide parallelism for shared memory environments. To run in parallel, OpenMP must be enabled at compilation time by using the correct compiler flag (usually some variant of `-openmp`). The number of threads may be controlled at runtime by setting the environment variable `OMP_NUM_THREADS`.

The procedures that are available to the user are then listed, with a very brief description of each (this should just be one or two sentences). If reverse communication is used, this is explained here. The aim is to provide readers with a quick overview of the procedures and introduce them to the calls that they will need to make to use the package. It should be made clear whether a procedure must be called or is optional.

If the package contains public derived types, these are explained in the next subsection. A full description of each component that the user is expected to access is either given here or in a later subsection. Since Fortran 95 does not allow individual components to be declared `public` or `private`, sometimes there are additional components that are not documented.

For packages with several versions, it is convenient to introduce a package type. For example, we may define **package type** to mean default real if the single precision version is being used, double precision real for the double precision version, default complex for the complex version and double precision complex for the double complex version.

There follows a full description of all the argument lists of procedures that the user may call and their calling sequences. Usually, there is a separate subsection for each procedure. In each of these, the type, shape and intent of each argument is documented ahead of a description of its purpose. If a full description would be long, e.g. for all the possible values of an error flag, it is better to give a short description here and refer the reader to a full description in a later subsection. Any restrictions on the argument (e.g. $n > 0$) that are tested within the package are listed in bold at the end of the description of the argument. In each call, `OPTIONAL` arguments are listed as the last arguments and square brackets `[ ]` are used to indicate these in the description. For example, in the call

```
call MA77_open(n,filename,keep,control,info[,nelt,path])
```

the arguments `nelt` and `path` are optional. Since we reserve the right to add additional optional arguments in future releases of the code, we advise users that all optional arguments be called by keyword, not by position.

After the subsections that describe the argument lists there may be a number of additional sections describing, for example, a derived type that has many components, information that is returned to the user, possible error and warning diagnostics, and other features of the package that have not been described elsewhere.

3.1.3 GENERAL INFORMATION

The general information section has the following headings:

Input/output: This includes diagnostic printing plus any I/O to direct-access and/or sequential access files.

Restrictions: A list of the restrictions mentioned for one or more of the arguments and/or control parameters.

Changes from Version $l.m.n$**:** ($l.m.n$ is replaced by the version label). If a major change has been made (that is, l has changed), this should be explained here.

3.1.4 METHOD

The method section provides a brief self-contained description of the method, which should give the reader some understanding of it without delving into fine details. It is normally less than a page long.

For details, the reader should be given a short list of references to papers and to the accompanying report, where available (see Section 3.2).

3.1.5 EXAMPLE OF USE

The calling sequence for the package is illustrated through the inclusion of a simple example (called the spec example), which is supplied complete with the input data and expected output. In our experience, if it is well written and fully commented, this kind of example provides a template that is invaluable, particularly for first-time users of a package.

Some packages have a second example to illustrate some other aspect of the way the package is used. Very occasionally, for a complex package with a number of different possible calling sequences, there is a third example.

The spec examples that are included in the user documentation generally use the default settings (see Section 4.5) and do not attempt to illustrate the use of all the facilities of the package.

3.2 Accompanying reports

For many users, a report that provides full details of the capabilities of the package and the options it offers is useful. This will describe in greater depth the algorithms used, highlight important or novel implementation details, present theoretical results, and show numerical results for practical applications. It will also provide references to other relevant research reports and papers. The report should enable advanced users to select suitable control parameters (see Section 4.5) and to understand the ways in which the current package differs from other packages. For the software developer, explaining a new code in a report is a useful exercise; a careful review frequently leads to modifications and improvements and so should be undertaken before the package is formally included within HSL.

The reports that accompany HSL software are made available online at (`http://www.cse.scitech.ac.uk/nag/reports.shtml`). They are often also submitted (possibly in a modified form) for publication in a leading international journal, such as ACM Transactions on Mathematical Software.

3.3 HSL PDF catalogue

The PDF version of the catalogue contains a complete list of all the packages in HSL at the time of writing and gives for each one a brief outline of its purpose, method, origin, language, and other attributes. It was originally hoped that this data could be extracted automatically from the user documentation, but this has not proved possible and separate source files are maintained. It is the resposibility of the developer of a new package to write a catalogue entry for that package.

The catalogue also contains an extensive index which is designed to help a potential user identify the appropriate package for a particular task.

There is a separate PDF catalogue with the same format for HSL Archive.

A new version of the PDF catalogue is produced for each release of HSL; in addition, it may be updated from time-to-time at the discretion of the HSL Librarian.

4 Programming language and style

4.1 Use of Fortran 95

All the packages in HSL are written in Fortran 77 or Fortran 95. The first Fortran 90 package to be included in HSL was `HSL_MA42` in 1995. Since 1995, developers have individually chosen to use Fortran 77 or Fortran 95. A major consideration when choosing the language was, in the past, portability. Good quality free Fortran 95 compilers were slow to appear and was one reason why many HSL users were

reluctant to switch from Fortran 77 to Fortran 95. However, for some years now the free Fortran 95/2003 compilers `g95` and `gfortran` have been available. Our experiences of these compilers have generally been very positive and so we are happy to recommend them to HSL users. As access to `g95` and `gfortran` is widespread, new HSL packages should normally be written in Fortran 95.

Fortran 95 packages must adhere to the Fortran 95 standard except that we allow the use of allocatable structure components and dummy arguments. These are part of the official extension that is defined by Technical Report TR 15581(E) and is included in Fortran 2003. It allows arrays to be of dynamic size without the computing overheads and memory-leakage dangers of pointers. Addressing is less efficient in code that implements pointer arrays since it has to allow for the possibility that the array is associated with an array section, such as `a(i,:)`, that is not a contiguous part of its parent. Furthermore, optimization of a loop that involves a pointer may be inhibited by the possibility that its target is also accessed in another way in the loop. Starting with HSL 2011, we also allow C interoperability.

The Fortran 77 packages adhere to the Fortran 77 standard except that some packages call the Fortran 95 intrinsics `CPU_TIME`, `EPSILON`, `HUGE`, `TINY`, and some make use of `COMPLEX*16`. Since Fortran 77 has no dynamic memory, working memory required by a Fortran 77 package must be provided by the user and passed to the package as real and integer arrays. If the memory is insufficient, the code should terminate and give advice to the user on how far the computation has proceeded and how much additional memory is needed. The user will then need to provide larger work arrays and, very often will be required to restart the computation from the beginning (although sometimes it is possible to restart at some point before it was found that the memory was insufficient). This is much less convenient for the user than working with a Fortran 95 code that automatically allocates and deallocates memory as required (especially since once larger arrays have been chosen, there is still no guarantee that they will be large enough and the process may need to be repeated). Relying on the package to allocate and deallocate memory also simplifies the user documentation since the number of arguments can often be reduced and this helps restrict the number of possible errors the user can make.

The use of dynamic memory allocation does come with a memory-management issue: the developer should provide the means for all temporary storage that has been allocated by the package to be released when the computation terminates. In particular, if the code fails before the computation is complete (perhaps because of an error in the user-supplied data), the developer should ensure that all temporary storage is released. This can be difficult, especially if there are many possible return paths back to the user's calling program. For storage that cannot be released until the user has finished with the computation, a straightforward method of deallocating the storage should be included within the package. A sparse solver is very commonly used in the inner loop of an iterative process, such as in solving Jacobian linear systems in the solution of differential-algebraic equations. If pointer memory is used in such a setting, even a small memory leak can lead to memory exhaustion over the course of the iterative process.

4.2 HSL rules

While it is recognised that each software developer will have his or her own programming style, there are a small number of rules that all contributors to HSL must adhere to. These are currently as follows:

- A ChangeLog file must be maintained for each package, giving brief summaries of each change that has been made since the original version.
- Each package must start with comments that specify the Copyright, version and original date.
- Each package must be threadsafe. This allows a user to safely run multiple instances of the package simultaneously in different threads or on different processors. A consequence of this requirement is that no common blocks or Fortran `save` statements are allowed within HSL packages.
- The source code must conform to the Fortran 95 standard, except that allocatable components and dummy arguments are allowed (and encouraged), plus C interoperability; in particular, it must contain no tabs. `forchk` (see Section 9.4) checks for these.

- For Fortran 95 packages, each line of the source must be at most 80 characters in length; for Fortran 77 packages, the limit is 72 characters (see Section 9.3).
- Implicit typing must not be employed. Fortran 95 packages must use `implicit none`.
- A `go to` statement must not branch backwards (that is, the branch target statement must not be earlier in the code than the `go to` statement). It should be used in Fortran 95 only when it makes the code clearer.
- Packages that allocate memory internally must either deallocate all such memory or must explicitly explain in the user documentation if there are any arrays that have been allocated but not deallocated. An example might be arrays that the package has allocated to hold matrix factors and which the user may wish to retain for later solves. If there remains allocated memory it should be possible for the user to successfully deallocate it.
- Packages that hold data in direct-access and/or sequential-access files must provide a means of releasing file storage once it is no longer required.
- Optional arguments are allowed and must be at the end of the argument list.
- The source code must contain no `stop` statements (it should always be possible for the calling code to recover from error situations).
- There must be a way to suppress the printing of warning and error messages and, if printing is offered, the user must be able to specify the unit number(s) for this.
- List-directed output (that is, `print` or `write` in which the format specification is an asterisk) must be avoided in the source code and in the simple and comprehensive tests (see Sections 6.1 and 6.2). This is because the output is not portable, and can vary with both compiler and/or computer platform.
- All new Fortran 95 packages must specify the intent of each argument of each user-callable procedure. All information parameters (see Section 4.7) must be of `intent(out)` on a initializing call and `intent(out)` or `intent(inout)` for other calls. Except in a procedure that sets default values for control parameters, each control parameter (see Section 4.5) must have `intent(in)`.

4.3 HSL recommendations

Beyond the rules of the previous subsection, we have the following recommendations:

- Automatic arrays should be replaced by allocatable arrays.
- Allocatable arrays should be used in preference to pointer arrays.
- Pointers should be used only where they offer a significant advantage.
- Neither the code itself nor any of its tests should have any variables that are not referenced or are referenced without being defined.
- Where a format is used only once, it is preferable to place it as a character constant within the output statement, e.g.

```
write(*,'(a,es12.4)') " 2-norm of residual =", resid
```

- Each derived type should be defined in a module, not in a procedure.
- Each intrinsic should be called by its generic name (e.g. `abs` rather than `dabs`).
- Each module should set default visibility to `private` and use an explicit `public` statement to make visible those entities that it is expected that the user will need to access.

4.4 Standard HSL format for sparse matrices

Starting with HSL 2011, it is our intention to enable a user to pass the sparse matrix input to one HSL package (such as an ordering routine) unchanged to another HSL package (such as a solver). We also want to avoid the overhead of checking the user's data multiple times. To facilitate this, we have introduced the standard HSL format. This is a compressed sparse column format with the entries within each column ordered by increasing row index. There is no requirement that zero entries on the diagonal are explicitly included. The package `HSL_MC69` may be used to convert from other sparse formats to standard HSL format and to check the sparse matrix data for errors. `HSL_MC69` handles duplicates (it sums them) and out-of-range entries (it removes them); in each case, a warning flag is set and information on the number of such entries returned to the user. `HSL_MC69` does not alter the user's data but returns the cleaned matrix data in separate arrays. The intention is that other HSL packages that use standard HSL format will make no further checks on the matrix data.

4.5 Use of control parameters

Since the early 1990s, many HSL packages have made use of control parameters, which have default values but may be set by the user to control the action; they are not altered by an HSL procedure unless it is initializing them with default values. Some HSL packages are designed to offer the user a large degree of control, while others leave few decisions open to the user. Clearly, a balance has to be achieved between flexibility and simplicity and this will, in part, depend on the target user groups/application areas and the complexity of the algorithms being implemented.

Apart from simple controls (such as those for diagnostic printing and whether or not the user's data is to be checked for errors), default values for control parameters should normally be selected on the basis of numerical experimentation as being in some way the 'best all round' values. Getting these choices right is really important because, in our experience, many users (in particular, those who would regard themselves as non-experts) frequently rely on the default settings and are reluctant to try other values even if their hardware is very different from the original test platform (possibly because they do not feel confident about making other choices). Thus when developing a new package experiments should be performed on as wide a range of computing platforms and practical problems as possible; this is discussed further in Section 6.3. There will inevitably be situations when the defaults may give far from optimal results and the user will then need to try out other values. Note that the ability to try different values can also be invaluable to those doing research. The developer may want to change the default settings at a later date as a result of hardware developments; within HSL this would lead to a new second digit in the version label.

HSL routines that are written in Fortran 77 use an array for integer controls and another for real controls. In general, the packages include an initialization subroutine that the user may call to assign default values to these control arrays. If other values are wanted, the relevant individual entries of the arrays can be reset by the user after the initialization and prior to calling other subroutines in the package. We recommend using arrays for the controls and allowing some extra space for controls that may be wanted in the future.

Many of the Fortran 95 packages in the Library replace the use of the arrays by a derived datatype whose components are the control parameters. Extra components can be added, if necessary, in a new version of the package. If their default values give the behaviour of the old code, only the second digit of the version label need change. Using a derived type can be more user-friendly as it allows meaningful names to be used for the controls. The components of the control derived type can be given default values when a variable of this type is declared and this is the approach used in recent HSL packages.

We note that it is important that the control parameters are fully documented within the user documentation. However, as many of them are primarily intended for tuning and experimentation by experts, they should not over complicate the documentation for the novice user. The user documentation needs to explain what, in broad terms, the effect of resetting a control parameter is likely to be.

If a parameter is important enough for the HSL developer to want the user to really consider what value to use (which may be the case if, for instance, the best value is too platform or problem dependent for a default to be confidently selected), then that parameter should not be a control but should be passed to the subroutines as an input argument. An example might be the use of scaling for a linear solver. Our experience has been that the benefits of different scaling strategies are highly problem dependent and so we feel a user should really be aware of what scaling, if any, is being performed and that it should not be 'hidden' away in a control parameter.

4.6 Checking of the user's data

A robust package needs to incorporate a means of checking of the user's data. Data checking is available (sometimes optionally) for most of the packages in the Library. The main exceptions are routines that are primarily intended for use by other HSL packages so that the data will have already been checked before they are called (and rechecking would not only be unnecessary and complicate the code but could also add an unacceptable overhead). If a package uses the standard HSL format (Section 4.4), the user should be encouraged to check his or her input matrix data using `HSL_MC69`; the package would then not need to carry out further checks (beyond trivial checks such as on the order of the matrix).

As already noted in Section 3.1.2, the user documentation highlights what restrictions there are, if any, on the parameters that must be set by the user. The code should normally check that the user-supplied controls are feasible; if one is not, we suggest that either a warning is raised and the default value is used or a return is made after setting an appropriate error flag. The user's control parameters **should not** be overwritten; if one or more is unsuitable and is replaced by the default, this should be done by using an internal copy.

4.7 The role of information parameters

Most HSL packages provide the user with information, both on successful completion and in the event of an error. The information that is provided does, of course, depend very much on the package but, as a minimum, this information will include an error flag. In the event of an error, the user needs to be given sufficient details to understand what has gone wrong together with advice on how to avoid the error in a future run. All error and warning returns should be fully explained in the user documentation.

HSL routines that are written in Fortran 77 use 'information arrays' to return information to the user; in general, two arrays are used (one for real and the other for integer information) and should be slightly larger than needed to allow for later additional requirements. The arrays need not be set by the user, although some codes that have a reverse communication interface currently rely on the information in these arrays not being altered by the user between calls; HSL packages written since January 2007 do not do this. In Fortran 95 packages, derived types (with `intent(out)`) can be used in place of arrays. As with the control parameters, this can be more user-friendly as it allows meaningful names to be used. However, printing a simple list of the information generated is then less straightforward; one possibility to assist users is to provide a separate printing routine to do this or, alternatively, to use a control parameter to allow the user to select an option that prints the information parameters once the computation is complete.

4.8 Communication between procedures

It often happens that data that are constructed during an invocation of a procedure in an HSL package are needed on a later invocation of the procedure or on an invocation of another procedure. The requirement that the code be threadsafe (see Section 4.2) means that this data must be passed through the argument lists.

This can often be done naturally by changing one or more of the principal arguments; for example, an array might be overwritten by its LU factorization and a permutation array might be set. If this is not realistic or further data is need, variables with the name `keep` (or a name including these four characters)

are normally used (note that some existing HSL packages use the name `save` in place of `keep`). In Fortran 77, they are usually arrays; in Fortran 95, they are usually scalars of a derived type that is part of the package.

In Fortran 77, there is no way to stop users altering a keep array, which would probably have a disastrous effect, so the documentation tells users not to do it. In Fortran 95, the derived type may be defined in a module with its components declared `private`, which will mean that the user cannot alter them. However, it may be desirable that some of the data is visible to the user or to other HSL packages. In this case, only those components that are intended to be visible to the user should be mentioned in the user documentation.

4.9 Use of MPI

When writing a package that uses MPI, it is important to avoid interfering with communications in other libraries or on other processes.

We recommend creating a new communicator and using it exclusively within the package. The easiest way to do this is to duplicate the user's communicator by calling `MPI_COMM_DUP`. If a smaller communicator is needed, `MPI_COMM_SPLIT` is available. When the package has finished, it should destroy its communicators by calling `MPI_COMM_FREE`, in order to avoid a memory leak.

The use of a separate communicator means that the package's communication is not confused with the user's, which aids debugging for both.

4.10 Use of OpenMP

An HSL package that supports OpenMP parallelism may need to call some of the procedures that are part of OpenMP. In particular, `omp_get_num_threads()` may be used to find out whether the code is executing in a uniprocessor environment, which allows key parts to be written differently for this case.

As these procedures are provided by the OpenMP `omp_lib` module, they are not available if the package is not compiled with OpenMP. For this to be possible without user modifications to the package, all lines calling such routines must start with the compile sentinel `!$` in columns 1 and 2.

4.11 Use of other library routines

Where appropriate, we recommend that developers of new HSL packages make use of existing HSL packages. Many of the packages whose names begin with the characters `MC` are sparse-matrix manipulation routines that have been included as separate packages so that those designing, for example, sparse direct solvers can use them to simplify both the development and maintenance of what is inevitably very complex code. Another advantage of using existing packages is that they will have been tested separately and so can be used with confidence, without retesting. Sometimes when developing a new package it will be appropriate to put part of the code into a separate HSL package to improve the modularity of the design and to allow others access in the future. For example, the out-of-core solver `HSL_MA77` has at its heart a call to a routine that efficiently performs a partial factorization (and corresponding partial solve) of a dense matrix. It was felt that this would be useful to have as a separate package (`HSL_MA54`). Similarly, all the handling of the out-of-core working is done by a separate HSL package (`HSL_OF01`), which we anticipate reusing for future out-of-core codes.

We also encourage the use of BLAS (Basic Linear Algebra Subroutines). BLAS are an aid to clarity, portability, modularity, and maintenance of software, and they have become a *de facto* standard for elementary matrix and vector operations. Performance (speed) will typically be critically influenced by the use of appropriately tuned BLAS so, although the BLAS source code should be used when testing (and is supplied with HSL), users are advised to replace this with BLAS that are tuned for the target machine, for example, vendor-supplied BLAS

or ones generated by ATLAS (see `http://math-atlas.sourceforge.net/`) or GotoBLAS2 (see `http://www.tacc.utexas.edu/tacc-projects/gotoblas2/`).

We also allow the use of LAPACK routines. The main problem associated with this is that many of the LAPACK routines involve a large number of dependencies, which must be identified and the source compiled when testing. As with the BLAS, source code is provided with HSL but performance will be dependent on using a tuned version.

The packages in the EP and MP sections of HSL rely on the MPI (Message Passing Interface) library. If this is not already available on the user's computer, it may be downloaded from

`http://www.mcs.anl.gov/research/projects/mpi/`

The package `HSL_MP01` consists solely of a module with an `include` statement for the MPI constants. All Fortran 95 packages in HSL should use this module since this will permit them to be written in the free source form.

The only other external library that may currently be called from an HSL package is the ordering library METIS. This is optionally used by a number of packages, including `MA57`. If used, it must be downloaded separately by the user from the METIS website

`http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`

5 Interfaces

Fortran interfaces are available for all HSL packages. In addition, MATLAB and/or C interfaces are available for some packages.

5.1 MATLAB

The package `hsl_matlab` provides type-safe wrappers around the MATLAB API and should be used for all access to MATLAB. Should the MATLAB API change there is then a single package that requires updating. Further, more comprehensive testing can be conducted on this package to ensure compatibility with all versions of MATLAB.

MATLAB interfaces must be updated and retested whenever the main package is.

The report `RAL-TR-2010-013` provides further detailed guidelines on the development of HSL MATLAB interfaces.

5.2 C

C interfaces are implemented entirely in Fortran, with C-language header files provided. The simple (spec) examples included in the documentation should be ported to C and provided to the user.

A typical C interface will consist of the following files

package`d_ciface.f90` containing the interface code written in Fortran 2003.

package`d.h` C-language header file detailing the interface.

package`ds.c` C-language spec example.

package`ds.data` Input for C-language spec example.

package`ds.output` Output from C-language spec example.

C interfaces must be updated and retested whenever the main package is.

More detailed guidelines on the preparation of HSL C interface will be provided in a forthcoming report.

6 Verification and testing

6.1 Simple examples

One or more simple (spec) examples are included in the user documentation, see Section 3.1.5, and are intended to provide a simple illustration of the use of the package. While they do provide a check for gross blunders, they rarely exercise enough of the code to give confidence in its correctness. As these examples will be compield and tested on a user's system, particular care shuold be made to ensure that portable examples are used (e.g. numbers close to zero should be avoided as these can vary by processor).

6.2 Comprehensive test

The comprehensive test is intended to provide a fuller check on the correctness of the code. It (and the simple examples) are rerun whenever a change is made to a package so it is important both that its execution time be not too long and that it explores as much of the code as possible. Furthermore, it should be designed so that the output is (as far as possible) machine and compiler independent. For example, for a linear equation solver, the output should not be the computed solution or the residual; instead, the test code should check that the solution and residual are as expected and then only perform printing in the event that it is not.

These objectives mean that it has to be designed specifically for testing the package. It would not be satisfactory, for example, to employ a code that was used to tune its performance or compare it with other codes. The aim is to check that it really is performing the algorithm that was finally chosen and this can usually be achieved with small test cases, perhaps constructed with a generator for pseudo-random numbers.

It should exercise all the modes that the user is permitted to employ (including testing of different settings for the control parameters, errors and warnings, and the different levels of printing). Ideally, it would exercise all possible execution sequences, but we recognise that this would usually not be practicable. Instead, we aim to exercise every statement at least once. Even this is often not practicable; for example, it is desirable to test the `stat=` variable of a `deallocate` statement, but it is probably impossible to provoke the statement to fail.

A tool should be used by the developer to discover which statements are not being exercised. Each such statement should be examined to make sure that it is not practicable to exercise it and to check by eye that the statement is correct and that the statements that would be executed following its execution are correct. In Section 9.1, we explain how we currently use the g95 compiler for this purpose.

Finally a memory-checking tool such as `valgrind` (Section 9.8) should be employed to check for usage of uninitialised memory, reading past the bounds of arrays and to detect any memory leaks.

6.3 Independent testing

In addition to the simple examples that are included in the user documentation and the comprehensive test, testing should be performed on a set of problems representing those at which the package is primarily aimed. These tests do not form part of the Library but they will typically be reported on in any accompanying report (see Section 3.2).

Testing on practical applications needs to be performed continually while the package is under development. Such tests often result in changes to the user interface, in particular, it may become apparent that further options are needed (perhaps via optional arguments or additional control parameters). The tests may be used to select defaults for the control parameters. The test set should be as large and varied as possible. If a package is intended to be general-purpose, the test set not only needs to include problems of different sizes but also problems from a range of application areas. For testing sparse linear solvers and sparse eigensolvers, matrices from the University of Florida Sparse Matrix Collection (`http://www.cise.ufl.edu/research/sparse/matrices/`) may be used. This is an extensive

set of matrices arising from real problems. It includes the original Harwell-Boeing Collection (which members of the Group were responsible for creating back in the 1980s) and is constantly being updated to include ever larger problems. For optimisation, there is the CUTEr suite of Fortran subroutines (see `http://www.cuter.rl.ac.uk/` for details), which is developed by the Group and collaborators. A key advantage of using these collections is that they are available via the web, which facilitates comparisons with other packages (both HSL and non-HSL packages). Where appropriate, comparisons with other packages needs to be part of the validation and testing process.

As well as testing on a range of problems, a package should be tested on as many different platforms as possible. This is to help ensure portability and robustness and also to check that the default settings for the control parameters are appropriate.

We highly recommend that the developer of an HSL package involves someone who has not themselves been closely involved in the design and development of the code in the testing. This may be another Group member who has an interest in the package or, ideally, a potential user from outside the Group. In our experience, feedback from an independent user is invaluable, both for the user documentation and the package.

6.4 Use of compilers

We are fortunate in having access to a range of up-to-date Fortran compilers (currently, Nag, gfortran, ifort, pgfortran). The developer should test his or her software using each of these compilers. The compilers g95 and lf95 are also available and useful, but are not considered up-to-date. We often find that different compilers will unearth errors that others failed to detect and can provide assistance in tracking down and correcting an error. When compiling the simple test(s) and the comprehensive test, the highest level of error checking offered by the compiler needs to be used. Any warnings returned at compile time should be checked. Ideally, no compiler warnings should be issued but, as we now allow allocatable structure components and dummy arguments, these will currently lead to a warning message. In our opinion, warnings can be off-putting for users, who may be concerned that they have made an error in the compilation process, and it is normally very easy to make simple changes to the code to eliminate most of these warnings.

7 Role of the Librarian

The Librarian (who is currently also the HSL Manager) is responsible for maintaining an up-to-date copy of HSL and associated tools. For HSL 2011, this is currently in the directory

```
/numerical/num/hsl2011
```

on the Group's Linux computer `nag`. This directory is maintained under version control with the Linux tool `svn`. Everything associated with a particular package is held in a subdirectory named after the package in the subdirectory `packages`. For example, everything associated with `HSL_MA48` is held in the directory

```
/numerical/num/hsl2011/packages/hsl_ma48
```

For the HSL package *package*, the names of the files for source, data, and output are summarized in Table 1. The output files are those produced by the Nag compiler on the Linux computer `nag`.

In addition, there are the following files:

- `makefile` or `Makefile`, whose purpose is explained in the next paragraph
- `blas` holds the names of the files for the source for the BLAS procedures called directly or indirectly in the single-length case. `dblas`, `cblas`, ... hold the names for other cases.
- `lapack` holds the names of the files for the source for the LAPACK procedures called directly or indirectly in the single-length case. `dlapack`, `clapack`, ... hold the names for other cases.

- `deps` holds the names of the files for the source for the HSL codes called or used directly or indirectly in the single-length case. `ddeps`, `cdeps`, ... hold the names for other cases.
- *package*`.tssd` or *package*`.tex` holds source for the user documentation.
- *package*`.yaml` may be used instead of `deps`, `blas` and `lapack` file to store the same information. It may additionally store information on extra files to include when building the tar archive distributed to the user. For example `hsl_ma87.yaml` specifies that the C interfaces for `HSL_MC68` and `HSL_MC69` should also be included.
- `catalog.tex` holds source for the catalog entry for the package.
- `ChangeLog` holds the version history for the package.
- *package*`.pdf` holds the user documentation in PDF format.

The `makefile` is used by the Librarian for the following tasks:

- Run the comprehensive test. This relies on a fresh compilation of all the code invoked (including any BLAS and/or LAPACK routines called by the package), which allows full tests to be made for errors such as out-of-range subscripts or argument list mismatches wherever they occur.
- Run the simple tests. Again, all the invoked code is compiled.
- Check the 80-character line length for Fortran 95 code or the 72-character line length for Fortran 77 code.
- Run a coverage check of the comprehensive test.
- Construct all the files needed for distribution.
- Clean the directory of all temporary files.
- Expand the master files (usually after editing them) back to their original form, overwriting existing versions.

For each release of a package, the Librarian's responsibilities are:

- to check the leading comments in the code;
- to check the ChangeLog has been brought up-to-date
- to check conformance with the relevant language standard;
- to run the comprehensive and simple tests under the Nag compiler and check that the output remains unchanged, or that the changes are acceptable;
- to check that the tests do not have any unused variables or references to undefined variables;
- to check the version number has been correctly amended in the user documentation and catalog entry;
- to check that the line-length limits have not been exceeded;
- to construct the PDF files of the user documentation; and
- to construct the files needed by users.
- update the webpages

Updating the webpages and constructing the distribution tarball have been automated using a Python script `hsladmin` located in `/numerical/num/hsl2011/bin`.

The Librarian is also responsible choosing new HSL names (in consultation with Group members) and for maintaining lists of HSL names. All names, including those under development and those that have been deleted from HSL, are listed in

`/numerical/num/hsl2011/hslnames.txt`

A slightly shortened version, containing only those currently in HSL and HSL Archive, is contained in

`/numerical/num/hsl2011/schedulec.txt`

In addition, the Librarian has responsibility for all aspects of the PDF catalogue, which is changed at a new release of the whole Library and may be updated from time-to-time.

The Librarian is not responsible for constructing the code and its tests, eye-ball checking of the code and its comments (other than the leading comments), performing performance checks, running the tests on compilers other than Nag, constructing and checking the user documentation, or correcting bugs. These are all the responsibility of the developer. Of course, the Librarian is a member of the Group, so he may perform these tasks as a developer or as a colleague of the developer.

7.1 Check lists

Starting in January 2007, before passing a new package to the Librarian for inclusion in HSL, a pre-release check list needs to be completed. Similarly, after a level one revision is accepted (the first digit in the version label is incremented), a revision check list must be completed. Sample pre-release and revision check lists are in Appendices 9.10 and 9.11. These check lists are intended to remind the developer of the steps that need to be gone through when preparing the code and the accompanying documentation and should prevent packages from being passed to the Librarian prematurely. In particular, the user documentation and the main source code must be signed off by a senior member of the Group (currently Duff and Scott may approve a package).

8 Licence issues

All use of HSL requires the user to have a valid licence. Three types of licence are available for HSL 2011:

- **Academic:** HSL packages are available at no cost for academic research and teaching. Access is via download links for individual packages in the online catalogue `http://www.hsl.rl.ac.uk/catalogue/`.
- **Commercial (per seat)** For use internal to a company, per seat licences are offered.
- **Commercial (incorporation)** Commercial (incorporation) For incorporation in software that is distribute to third parties,an incorporation licence is offered. Pricing is done on a case-by-case basis.

Packages in the HSL Archive are available at no cost for personal academic or commercial (non incorporation) use.

Each package is distributed as Fortran source code (in some cases with a MATLAB and/or C interface), starting with comments that detail the Copyright and conditions of use, followed by comments on the version history (see Section 2.1). There follows the code itself. Since HSL 2011, packages are distributed complete with all comments.

The Copyright lines specify all the institutions and/or individuals that have any Intellectual Property Rights in the package. Many of the earlier codes are copyright (solely or jointly) as Aspen Technology Inc. (or Hyprotech, AEA, etc). Where the copyright extends beyond Science & Technology Facilities Council (or CCLRC, RAL, etc) and Aspen Technology Inc. (or Hyprotech, AEA, etc), there should be

an agreement in place that specifies the rights and how any revenues will be shared. In the case of a collaboration in which the outside party does not claim any Intellectual Property Rights, there should be a memorandum signed by the outside party that makes this clear. Starting with HSL2011, such agreements should be in place before a new package can be consideered for inclusion in the library.

9 Appendices

9.1 Coverage with gfortran

The tool that we recommend is part of the gcc package and may be run on any of the Group's computers as is illustrated by the example

```
gfortran -fprofile-arcs -ftest-coverage \
          fa14d.f90 of01d.f90 of01dt.f90 ddeps.f
./a.out > temp
gcov of01d
gedit of01d.f90.gcov
```

The steps in this example are as follows:

1. Compile the program, adding counting instructions to the executable. This also creates files with suffix `gcno`.
2. Run the program. This also creates files with suffix `gcna`.
3. Construct an annotated listing of `of01d`. This needs the files `of01d.gcno` and `of01d.gcna`.
4. Examine the annotated listing with an editor. Executable lines that were not executed are marked `#####`.

9.2 Coverage with nag_coverage95

An alternative is the Nag tool `nag_coverage95`. The way the Librarian uses it is illustrated by the example

```
nag_coverage95 -source -load all_together.f90 -first ddeps.o
all_together.inst.exe
```

The code to be profiled must include the main program. Here it has been collected into `all_together.f90` and the rest has been compiled under `nagfor` into `ddeps.o`. The first line creates instrumented code in `all_together.inst.f90`, compiles it, and links it with `ddeps.o` to `all_together.inst.exe`. The second line runs this executable and produces counts in `all_together.out`.

For other ways to use `nag_coverage95`, see its documentation

```
http://www.nag.co.uk/nagware/nq/man/f95_tools/nag_coverage95.html
```

9.3 Checking line lengths

The commands

```
g95 /numerical/num/hsl2007/checks/check72.f90
a.out <ma48ad.f
```

check `ma48ad.f` for lines longer than 72 characters, copying any such lines to unit 6. The code `check80.f90` in the same directory checks for lines longer than 80 characters.

9.4 Debugging and checking conformance with standards

For debugging and checking conformance with standards, it is important to run several compilers. Our experience is that different compilers can find different problems. The following command lines may be used on the Group's computer `nag`:

Nag compiler:

```
nagfor  -C=all -C=undefined -nan -u -gline
```

For Fortran 77 packages with double complex versions, the option `-dcfuns` may be needed.

Lahey-Fujitsu compiler:

```
lf95 --chk aesux
```

Gnu g95 compiler, allowing allocatable components and dummy arguments:

```
g95 -std=f95 -ftr15581 -Wall -Wimplicit-none -fbounds-check -ftrace=full
```

Intel compiler:

```
ifort -C -u
```

Forcheck syntax analysis for Fortran 77:

```
forchk -f77 -nwarn -ninf -nff
```

The following command lines may be used on the Group's computer `fox`:

Gnu compiler:

```
gfortran -std=f95 -fbounds-check -Wall -pedantic
```

Since each of these find different problems, they should all be used.

9.5 Polishing Fortran 95 code

Fortran 95 code may be polished with the Nag tool `nag_polish95`, as in the example

```
nag_polish95 temp.f90
```

which copies `temp.f90` to `temp.f90.original` and overwrites `temp.f90` by a polished version. It uses a default set of options unless it finds a polish options file with name `.polish_options`. It looks first for the environment variable `NAG_POLOPT95` or `NAG_POLOPT90` containing the full name of the file, then in the current directory, then in the home directory. We recommend that you copy the file `/numerical/num/hsl2007/tools/.polish_options` to your home directory. You can edit it there with `nag_polopt95`. There are man pages for both `nag_polish95` and `nag_polopt95`.

9.6 Polishing Fortran 77 code

Fortran 77 code may be polished with the Nag tool `nag_polish`, as in the example

```
nag_polish -po ~/polish_options temp.f
```

which copies `temp.f` to `temp.f.orig` and overwrites `temp.f` by a polished version. It uses the polish options file `polish_options` in the home directory. We recommend that you copy the file `/numerical/num/hsl2007/tools/polish_options` to your home directory. You can edit it there with `nag_polopt`. There are man pages for both `nag_polish` and `nag_polopt`.

9.7 Time Profiling

Time profiling reveals the amount of CPU time that each subroutine uses and can indicate which statements take the longest to execute. This in contrast to the coverage profiling of Subsections 9.1 and 9.2, which give the number of times each statement is executed.

There are two approaches – code instrumentation and hardware performance counters. We describe the latter, since we have found it to be the more useful. Both provide only approximate timings because of compiler optimizations and because they sample the execution.

Hardware performance counters work by counting specific events on each CPU core. When a count reaches a configurable value, it triggers an interrupt which is recorded against the currently executing instruction. The interrupts slow the execution slightly, but the approach has the benefit that code does not need to be instrumented and therefore runs at near-production speed.

To obtain output that is easy to read, the user must compile the source with the `-g` option.

We now explain how to use `oprofile` on the Group's computer `fox`. We need a separate window in which `root` is logged in and two windows in which the user is logged in normally. On one of the user windows, execute

```
top
```

This will dynamically show how the machine is being used. If anyone else is making significant use of the machine when you are profiling, you will be monitoring (and slightly slowing) their use as well as your own. **Therefore, all members of the Group should be warned beforehand.**

The profiling may be controlled by a GUI. To start this, execute

```
oprof_start &
```

in the `root` window. The GUI may be used to change settings. The most useful event is the unhalted clock tick, `CPU_CLK_UNHALTED`, which essentially measures time.

To profile a program, first use the start button in the GUI to start the profiler. Next, in a user window, run the program as normal. Use the GUI stop button to stop the profiler when the run has completed. The results of the profile run can be viewed by executing commands in the user window. To see a list of counts in procedures, execute

```
opreport -l a.out > temp
```

and view `temp` with an editor. To see a limited callgraph report, compile with `-fno-omit-frame-pointer` and execute

```
opreport -l -c a.out > temp
```

To see annotated source code, execute

```
opannotate --source a.out > temp
```

For further profiling, the counts will be accumulated unless the 'Reset sample files' button in the GUI is used to reset the counts to zero.

`oprofile` has many more advanced options. These are detailed in its own documentation, available on the web.

9.8 Valgrind

Valgrind can be used to simulate the program and detect any memory errors. Typically these include access to unallocated memory, use of unitialized values and memory that has not been deallocated.

To run a program under valgrind, compile with `-g` and invoke the program as

```
valgrind ./a.out
```

Valgrind has many more advanced options. These are detailed in its own documentation, available on the web.

9.9 Other tools

For advice on other tools see the web site of the Software Engineering Group:

`http://www.cse.scitech.ac.uk/seg/`

The Group has developed a tool

`http://www.softeng.rl.ac.uk/p/software/pyqa-forcheck`

that acts as a front end to the static analysis tool Forcheck. It

- automates most of the steps needed to run Forcheck, including searching for relevant source files, sorting them based on module dependencies, selecting program options, and building the Forcheck execution command; and
- parses the text-based output from Forcheck, and presents the results in a more manageable and comprehensible format.

9.10 Check list for new packages

Check List for a New Package

Package name:	.....
Start date:	.....
Initial documentation:	**Read by** .....
Check HSL rules (section 4.2):	YES/NO
Simple tests:	YES/NO
Comprehensive tests:	
Profile:	**Coverage:** YES/NO **Different modes:** YES/NO
Compilers:	**g95:** YES/NO **Nag:** YES/NO **ifort:** YES/NO **lf95:** YES/NO **gfortran:** YES/NO **pgfortran:** YES/NO
Use source BLAS/LAPACK:	YES/NO
Output portability:	YES/NO
Valgrind:	YES/NO
Independent testing:	**Tested by** .....
Generate other versions (e.g. single):	YES/NO
Simple tests:	YES/NO
Comprehensive testing:	YES/NO
Finalize documentation:	YES/NO
Signed off date:	.....

9.11 Check list for revised packages

Check List for a Revision

Package name:	.....
Start date:	.....
Version:	.....
Start with the latest HSL version of codes and documentation:	YES/NO
Update documentation:	
Version number:	YES/NO
Comments on changes:	YES/NO
Changes:	YES/NO
Detail update in ChangeLog:	YES/NO
Simple tests:	YES/NO
Comprehensive tests:	
Profile:	**Coverage:** YES/NO **Different modes:** YES/NO
Compilers:	**g95:** YES/NO **Nag:** YES/NO **ifort:** YES/NO **lf95:** YES/NO **gfortran:** YES/NO **pgfortran:** YES/NO
Use source BLAS/LAPACK:	YES/NO
Output portability:	YES/NO
Valgrind:	YES/NO
Independent testing (optional):	**Tested by** .....
Generate other versions (e.g. single):	YES/NO
Simple tests:	YES/NO
Comprehensive testing:	YES/NO