Science & Technology Facilities Council

Technical Report

Code Coverage Analysis for Fortran

L.S. Chin, D.J. Worth, and C. Greenough

August 27, 2009

RAL-TR-2009-019

© Science and Technology Facilities Council

Enquires about copyright, reproduction and requests for additional copies of this report should be addressed to:

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

The STFC ePublication archive (epubs), recording the scientific output of the Chilbolton, Daresbury, and Rutherford Appleton Laboratories is available online at:
http://epubs.cclrc.ac.uk/

ISSN 1358-6254

Neither the Council nor the Laboratory accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigation

RAL-TR-2009-019

Code Coverage Analysis for Fortran

L.S. Chin, D.J. Worth, and C. Greenough

August 27, 2009

Abstract

This report introduces the concept of coverage analysis and provides a list of coverage tools available for Fortran. It also contains a step-by-step tutorial on using *gcov* and *LCOV*, and presents a case study on performing coverage analysis on an existing Fortran library (The Finite Element Library). The tools and steps laid out in this report focus on code written in Fortran, however the concepts should apply equally to code written in other programming languages.

Keywords: coverage analysis, software testing, software quality, QA tools, Fortran, SESP

Email: `shawn.chin@stfc.ac.uk, david.worth@stfc.ac.uk, christopher.greenough@stfc.ac.uk`

Reports can be obtained from `www.softeng.cse.clrc.ac.uk`

Software Engineering Group
Computational Science & Engineering Department
STFC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
Oxfordshire OX11 0QX

Contents

1 Introduction

This report introduces the general concept of coverage testing and provides a step-by-step tutorial on performing coverage analysis. The tools and steps laid out in this report focus on code written in Fortran, however the concepts should apply equally to code written in other programming languages.

Section 2 provides a brief discussion on what is meant by coverage analysis and its purpose followed by Section 3 which iterates through a non-exhaustive list of coverage tools available for Fortran.

In Sections 4 and 5, the report dives into the usage of one of the tools, namely gcov. A quick guide to using *gcov* for coverage analysis is provided for those in a hurry and this is followed by a more involved discussion on usage and the internals of *gcov*.

Finally, in Section 6 we present a walkthrough for the process of integrating a complete code coverage solution into the build process of an existing Fortran project – the Finite Element Library [8]. This section will cover the use of *gcov* and *LCOV* for generating a comprehensive coverage report that is user-friendly and navigable, as well as providing examples on how the whole process can be scripted and automated.

This report is one of the outputs of the Software Engineering Support Programme (SESP) [7].

2 Coverage Analysis

2.1 Overview

Code coverage is a quantitative measure of the degree to which the source code of a program has been exercised. It is most often used during the software testing process to determine the coverage achieved by the testing process and as such is often also referred to as test coverage. Code coverage and test coverage are synonymous.

Code coverage analysis is performed using tools that instrument the source code or intermediate binaries with instructions to output coverage information during its execution. The coverage tool would then gather the generated data and reference the original source code to produce a coverage report.

Coverage reports contain two types of information:

1. An annotated version of the original source code indicating elements of the code that have not been exercised and the frequency with which each element[1] has been exercised.
2. A percentage or score indicating overall coverage level. The coverage levels could be for the whole code project or broken down into that of individual directories, files, modules and/or functions. Analysing coverage levels at a lower granularity is important as overall coverage levels can hide large gaps in coverage.

The annotated source code can be used to identify cold and hot spots – portions of the code which are never (cold) or frequently (hot) exercised. Cold spots are useful for identifying blind spots within the code and providing hints on whether, and where, more tests are required; hot spots can be used to identify issues like redundant test cases and potential performance optimisation opportunities.

The coverage score gives a more general view of how much ground is covered by the tests. A low coverage score would indicate an inadequate level of testing which implies a lower probability of the test suite exposing any bugs in the code. In addition, a low initial coverage level can often indicate a deficiency in the test development process.

The inverse however is not true – while a good test suite is likely to achieve high coverage levels, even tests with the highest possible level of coverage are not guaranteed to catch all errors [1].

Take for instance the following Fortran module (Listing 1):

```
module xyz
contains
    function get_height(area, width)
    real, intent(in) :: area, width
    get_height = area  / 5.0
    end function get_height
end module xyz
```

Listing 1: Code snippet for module *xyz*

```
test_suite xyz

test height_calculation
    assert_real_equal(2.0, get_height(10.0, 5.0))
    assert_real_equal(0.0, get_height(0.0, 5.0))
end test

end test_suite
```

Listing 2: Test code for module *xyz* (written for FUnit [9])

[1]Depending on the granularity and level of coverage analysis performed, the term element is used loosely here to refer to a function, statement, branch or even a logical condition within the code.

The example test code (Listing 2) would run without any failures and provide 100% coverage of the code but it is by no means complete. It does not expose the bug where a constant divisor is used instead of the *width* argument, or that the code author may have unintentionally omitted checks for ensuring that the *area* and *width* are within acceptable values ($area \geq 0.0$, $width > 0.0$).

Therefore, while it is natural to treat coverage levels as a direct measure of test completeness, it is important to instead think of it as a tool to better understand the behaviour and shortcomings of the testing process and the tests at hand.

Furthermore, by including coverage analysis as a process within the software development cycle (see Figure 1), developers can avoid test rot – a situation where the code evolves well beyond what is being interrogated by the tests, resulting in an increasingly obsolete set of tests that may instil a misplaced sense of confidence in the code.

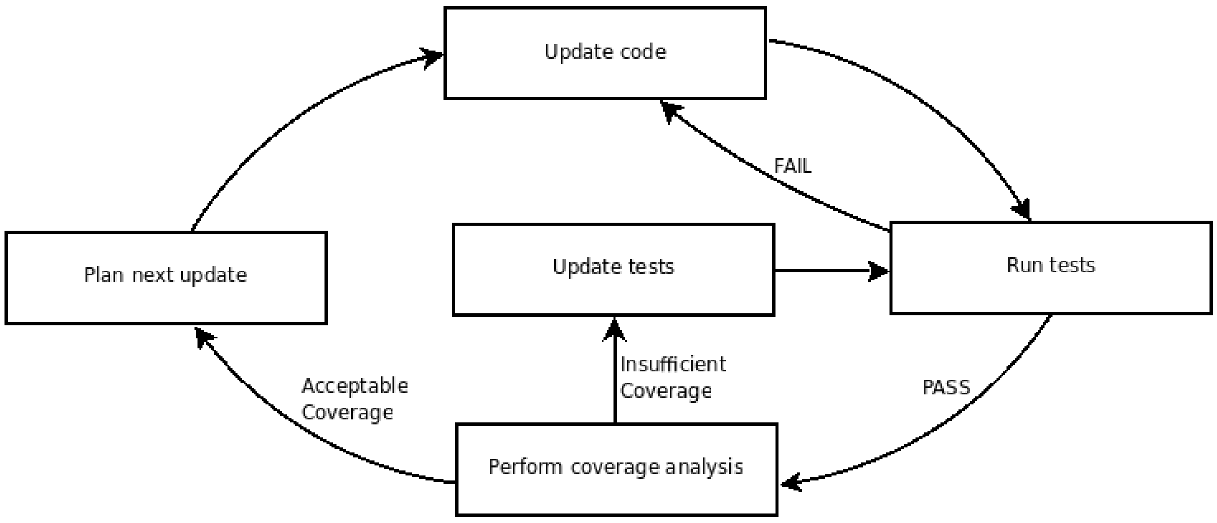


Figure 1: A software development iteration that incorporates a *coverage analysis* stage

2.2 Coverage criteria and targets

To measure coverage, one or more of the following criteria are commonly used. Each of these criteria involves different levels or granularity and would therefore require varying levels of complexity and effort to analyse and achieve.

Some of the basic coverage criteria are:

- **Function coverage** – full coverage involves having every function or procedure invoked at least once.
- **Statement coverage** – full coverage involves having every executable statement run at least once.
- **Branch coverage** – full coverage involves having every control structure evaluated to both true and false at least once.
- **Condition coverage** – full coverage involves having every logical condition within a control structure evaluated to both true and false at least once.
- **Path coverage** – full coverage involves having every possible route through the program taken at least once. A path is defined as a unique sequence of logical conditions, which means that full coverage requires attempting every combination of logical conditions within the program.

All the criteria listed above are related such that coverage of each criterion would imply coverage of the ones before it. For example, full branch coverage would imply full statement and function coverage (since all statements would be reached if every branch is taken at least once) while full path coverage is the most comprehensive and would imply that all the above coverage criteria are also met.

All available coverage tools are capable of analysing statement coverage while some (such as *gcov*) can perform branch coverage analysis. There is a small handful of commercial coverage tools for Fortran that claims to provide anything beyond branch coverage[2].

Writing test suites to achieve full function coverage is often trivial and as such yields minimal benefit while achieving full path coverage, except for that of trivial programs, is usually impractical if not impossible[3] [1]. Therefore, the most common approach is to aim for target levels of statement and/or branch coverage. While not the most comprehensive, they provide a good balance between achievability and benefit.

The decision on a coverage level to target is a common subject of contention. Some practitioners advocate maintaining a 100% coverage target while others believe that to be an impractical target especially for complex and mature code projects [2, 5, 13].

In general, the most common recommendation is that one should adopt a sensible coverage target that is appropriate to the code in question and make it a point to peruse (and document!) sections which are left ‘uncovered’ [2, 3, 4].

[2] *LDRA Testbed* claims to perform condition coverage analysis while *TCAT/PATH* from TestWorks claims to support path coverage analysis

[3] A program with n branches can have up to 2^n paths while loops can lead to an infinite number of paths

3 Coverage Tools Available for Fortran

The following is a non-exhaustive list of coverage analysis tools available for Fortran. Some of these tools are freely available while others are commercial applications that can be accessed through the Software Engineering Support Program (SESP).

3.1 gcov

gcov is an open source tool that is part of the GNU compiler suite. It is simple to use and supports statement and branch coverage analysis.

Subsequent chapters of this report will focus on gcov as it is freely available and easily integrated within the build process of most projects.

Performing coverage analysis on individual source files using the gcov command is straight forward. However, the task of perusing the results can be quite daunting for large projects especially when code is split across multiple files and directories. Therefore, to move beyond analysing individual source files, we use tools like *LCOV* to simplify the process and present the coverage analysis results in a more digestible format.

3.1.1 LCOV

LCOV was developed as part of the Linux Test Project (http://ltp.sourceforge.net) and can be seen as an extension to *gcov*. This extension consists of a set of Perl scripts which collect *gcov* data for multiple source files and create HTML pages containing summary information and the source code annotated with coverage information.

The use of *LCOV* is extremely useful for coverage analysis of larger projects as it provides a birds-eye view of the overall coverage at three different levels: project view, directory view, and file view. Screenshots of the coverage report produced by *LCOV* are available in Section 6.2.6.

LCOV is available from its project page [10] or through the software package manager of many Linux distributions.

This tool will be explored further in Section 6.

3.2 ggcov

ggcov is a prettier alternative to *gcov*. It consumes the same coverage data files as *gcov* but presents the output in a graphical interface. It provides a coloured listing of annotated source code (including branch coverage data) and summary information with addtional views such as call graphs and bar charts.

While not advertised as working for Fortran, a simple patch to the source code can allow it to be used. More information on this can be found on the SEG Blog [11].

ggcov is available from the project page: http://ggcov.sourceforge.net/

3.3 FCAT

FCAT (Fortran Coverage Analysis tool) is a tool written by Dr. YiFan Hu while he was based at Daresbury Laboratory back in 2001.

It is available as a Perl script or a pre-compiled binary (available for Linux and a few Unix variants). *FCAT* works by pre-processing source code and producing an instrumented version which is to be compiled and run instead of the original. It supports only statement coverage.

FCAT can be downloaded from:

- http://www.dl.ac.uk/TCSC/UKHEC/FCAT/
- http://research.att.com/~yifanhu/SOFTWARE/FCAT/

More usage and installation details can be found in the documentation file (http://www.dl.ac.uk/TCSC/UKHEC/FCAT/README.html).

3.4 nag_coverage95

nag_coverage95 is the coverage tool for *NAGWare F95* produced by the Numerical Algorithms Group (NAG).

nag_coverage95 provides statement coverage support and is very easy to use for code within a single source file. The tool is used in place of a compiler and it will automatically output an instrumented executable that produces coverage information at run-time.

For a more substantial project with multiple source files, users must compile all files using the *NagWare* Fortran 95 compiler into modules files before linking the executable using *nag_coverage95*.

More usage information can be found in the tools documentation (http://www.qaportal.cse.clrc.ac.uk/html/NagWare/nag_coverage95.html).

3.5 CVRANAL

CVRANAL is part of the *plusFort* package produced by Polyhedron Software (http://www.polyhedron.co.uk/pfqa0html#coverage).

It is meant to be used in conjunction with *SPAG* (also part of *plusFort*) which is a very powerful tool but not the easiest to pick up for first-time users.

More usage information can be found in the tools documentation (http://www.qaportal.cse.clrc.ac.uk/html/plusFORT/manual/spag.html#S2112).

4 Using *gcov* – A quick guide for those in a hurry

The following steps illustrate the basic procedures involved in performing coverage analysis using *gcov*. Subsequents chapters of this report will give a more involved discussion on *gcov* and will provide examples of how coverage analysis can be made more manageable for larger code projects.

Step 0: Ensure you have a suitable version of *gcov*

Use a compiler from the GNU compiler suite (*gcc*, *g77*, *gfortran*, etc.) and a version of *gcov* that came packaged with the suite. If you have multiple versions of the GNU compilers installed or wish to use another GNU-compatible compiler such as *g95*, ensure that you use a *gcov* version which is equal to or higher than your compiler version.

- **Hint**: Version 4 of *gcov* is sometimes distributed with the binary renamed as `gcov4`. Do also look out for that command.
- **Hint**: Use the `--version` flag to check the versions of gcov and your compiler. For example:

```
[lsc@softeng]$ gcov --version
gcov (GCC) 4.1.2 20080704 (Red Hat 4.1.2-44)
[lsc@softeng]$ gfortran --version
GNU Fortan 95 (GCC 4.1.0 20050311 (experimental))
```

Step 1: Use the right compiler flags

Include the `-fprofile-arcs -ftest-coverage` flags when compiling and linking your code. Leave out all optimisation flags. The additional flags will inform the compiler to generate the additional information needed for performing coverage analysis.

- **Hint**: If you use a *Makefile* to compile your project, you can add the flags to the `FFLAGS` and `LDFLAGS` parameters. Watch out for optimisation flags within `FFLAGS`.
- **Hint**: Starting from version 4 of the GNU compiler suite, you can use the `--coverage` flag instead. This flag will automatically be expanded to the required flags for performing coverage analysis.

Step 2: Run your program

Run your program as you would normally. You can repeat the run multiple times (ideally under different test conditions); the gathering of coverge data is cumulative.

Step 3: Use *gcov* to view coverage

Run *gcov* on each of your source file to explore the coverage. For example:

```
[lsc@softeng]$ gcov matmul.f
File 'matmul.f'
Lines executed: 100.00% of 20
matmul.f:creating 'matmul.f.gcov'
[lsc@softeng]$ gcov trim3.f
File `trim3.f'
Lines executed:95.83% of 24
trim3.f:creating `trim3.f.gcov'
```

- **Hint**: To retrieve branch coverage information, include the `-b` option when calling *gcov*.

Step 4: View annotated source code generated by *gcov*

Peruse the **.gcov* files generated in Step 3. These files contain annotated versions of your source code. This will help you identify hot and cold spots in the code.

5 *gcov* in a Nutshell

5.1 Overview

gcov is a test coverage program that can be used in concert with the GNU Compilers (*gcc*, *g++*, *gfortran*, etc.) and other compatible compilers such as *g95*.

To perform coverage analysis, the source code must be compiled with specific compiler flags which instruct the compiler to:

- instrument the intermediate binaries such that the resulting executables produce data which tracks the flow through the call graph during run time
- output the application call graph

All these data can be read by *gcov* to reconstruct the basic block call graph and tally the traversal frequency of each basic block. By mapping each basic block to the corresponding source line number a full coverage analysis can be obtained.

At present, *gcov* is capable of performing statement coverage as well as branch coverage analysis.

5.2 Compiling and linking

The target code must be compiled and linked using the `-fprofile-arcs -ftest-coverage` compiler flags.

```
[lsc@softeng]$ ls
gol.f90  gol_io.f90  gol_util.f90
[lsc@softeng]$ gfortran -fprofile-arcs -ftest-coverage -c gol_util.f90
[lsc@softeng]$ gfortran -fprofile-arcs -ftest-coverage -c gol_io.f90
[lsc@softeng]$ gfortran -fprofile-arcs -ftest-coverage -c gol.f90
[lsc@softeng]$ gfortran -fprofile-arcs -ftest-coverage -o run_gol *.o
[lsc@softeng]$ ls gol_util*
gol_util.f90  gol_util.gcno  gol_util.mod  gol_util.o
```

The `-fprofile-arcs` flag instructs the compiler to instrument the compiled code and generate a call graph, while the `-test-coverage` flag instructs the compiler to identify and track each basic block within the source file [6].

Notice that apart from the standard **.o* and **.mod* files, the compiler has also created **.gcno* files. These files are where the compiler stores data on the generated call graphs and basic blocks; they are required by *gcov* to reconstruct the program flow and map coverage data to specific lines in the source code.

Older versions of the GNU compilers may generate **.bb* and **.bbg* files instead, a combination of which serves the same purpose as the current **.gcno* format

5.3 Analysing coverage

Running the executable compiled with the `-fprofile-arcs` flag will create a **.gcda* file (or **.da* for older versions of GCC) in the directory where the original source file was located. This file is used to record profiling information generate at run-time. The recorded information is cumulative so running the executable multiple times will accumulate all coverage data into the same **.gcda* file.

```
[lsc@softeng]$ ./gol input/glider.dat 3
Running step 1
Running step 2
Running step 3
[lsc@rsofteng]$ ls gol_util*
gol_util.f90  gol_util.gcda  gol_util.gcno  gol_util.mod  gol_util.o
```

Once coverage data is accumulated, *gcov* can use the information in the **.gcda* and **.gcno* files to calculate code coverage and produce an annotated version of the source files. By passing any source filename to *gcov*, the coverage score will be printed out and an annotated version of the source will be created with the **.gcov* file extension.

```
[lsc@softeng]$ gcov gcov_util.f90
File gol_util.f90
Lines executed: 94.29% of 35
gol_util.f90:creating gol_util.f90.gcov
```

The **.gcov* files are written with the following format:

```
   <execution count>: <line number>: <source line text>
```

Example snippet from *gol_util.f90.gcov*:

```
      -:   40:        ! Allocate new memory
      3:   41:        allocate(data1(w,h), data2(w,h), stat=alloc_stat)
      -:   42:
      -:   43:        ! if allocation failed, returned 'stat' would be > 0
      3:   44:        if (alloc_stat .ne. 0) then
  #####:   45:            call gol_cleanup
  #####:   46:            stop "gol_init: Memory Allocation Failed. Quitting!"
      3:   47:        end if
```

A hyphen (`-`) is used to represent source lines that contain no executable code, while a row of hashes (`#####`) indicate lines that were never executed.

5.4 Obtaining coverage summaries for each subroutine

The `--function-summaries` option can be used to output coverage levels per subroutine in addition to the file level summaries.

```
[lsc@softeng]$ gcov --function-summaries gol_util.f90
Function '__gol_util__calculate_stat'
Lines executed:100.00% of 12

Function '__gol_util__update_grid'
Lines executed:100.00% of 5

Function '__gol_util__count_neighbours'
Lines executed:100.00% of 2

Function '__gol_util__gol_step'
Lines executed:100.00% of 2

Function '__gol_util__gol_cleanup'
Lines executed:100.00% of 3

Function '__gol_util__gol_init'
Lines executed:81.82% of 11

File 'gol_util.f90'
Lines executed:94.29% of 35
gol_util.f90:creating 'gol_util.f90.gcov'
```

5.5 Analysing branch coverage

The `--branch-probabilities` option can be used to output branch coverage summaries. This will also annotate the **.gcov* entries with information on branch frequencies. Unconditional branches will not be included unless the `--unconditional-branches` option is also used.

```
[lsc@softeng]$ gcov --branch-probabilities gcov_util.f90
File gol_util.f90
Lines executed: 94.29% of 35
Branches executed: 100.00% of 84
Taken at least once: 85.71% of 84
Calls executed: 86.67% of 30
gol_util.f90:creating gol_util.f90.gcov
```

The '*Branches executed*' field refers to the number of branches encountered, while the '*Taken at least once*' field refers to the number of branches that were taken at least once. In the above example, all 84 branches were encountered and evaluated but only 85.71% (72) of them were taken at least once. The remaining 12 branches have conditions that always evaluated to False and the statements within them never executed, hence never tested.

5.6 Caveats

5.6.1 Compiler optimisation

When performing coverage analysis, compiler optimisation should always be disabled as some optimisation may eliminate, reorder or combine statements and influence the coverage results. Take for example the following C code example adapted from the GCC documentation [12]:

```c
int main(int argc, char ** argv) {

    int i, int_a, int_b, result, sum;

    sum = 0;
    int_a = 10;
    int_b = 20;

    for (i = 0; i < 10; i++)
    {
        if (int_a == int_b)
            result = 42;
        else
            result = 24;

        sum += result;
    }

    printf("%d", sum);
}
```

Listing 3: Sample C code adapted from the GCC documentation

Different coverage results will be obtained when the code above is compiled with and without optimisation.

```
[lsc@aphek]$ # Without optimisation, we get 90% coverage
[lsc@aphek]$ gcc -fprofile-arcs -ftest-coverage -o test test.c
[lsc@aphek]$ ./test
240
```

```
[lsc@aphek]$ gcov test.c
File 'test.c'
Lines executed:90.00% of 10
test.c:creating 'test.c.gcov'

[lsc@aphek]$ # Now with optimisation, coverage will go up to 100% !!
[lsc@aphek]$ gcc -fprofile-arcs -ftest-coverage -O3 -o test test.c
[lsc@aphek]$ ./test
240
[lsc@aphek]$ gcov test.c
File 'test.c'
Lines executed:100.00% of 10
test.c:creating 'test.c.gcov'

[lsc@aphek]$ # reproducibility of this example may depend on compiler version used
[lsc@aphek]$ gcc --version | head -n1
gcc (GCC) 3.4.6 20060404 (Red Hat 3.4.6-11)
```

Looking at the annotated source code for both cases, note that the compiler has combined the `if/else` block into a single basic block during the optimisation phase.

```
       10:   10:         if (a == b)
       10:   11:                 c = 0;
        -:   12:         else
       10:   13:             c = 42;
```

Listing 4: test.c.gcov of Optimised Code

```
       10:   10:         if (a == b)
    #####:   11:                 c = 0;
        -:   12:         else
       10:   13:             c = 42;
```

Listing 5: test.c.gcov of Unoptimised Code

5.6.2 Version incompatibility

The version of *gcov* must be compatible with the compiler used to compile the source code as it needs to understand the output produced by the compiler.

This requirement may be an issue for users who maintain several GNU-based compilers (*gfortran*, *g77*, *g95*) on their system. If these compilers are installed and updated separately, the compiler versions may go out of sync. It will then be up to the users to choose a suitable version of *gcov* based on the compiler used.

Using a version of *gcov* that is older than the compiler will lead to errors as such:

```
[lsc@softeng]$ gcov -v | head -n1
gcov (GCC) 3.4.6 20060404 (Red Hat 3.4.6-8)
[lsc@softeng]$ g95 --version | head -n1
G95 (GCC 4.0.3 (g95 0.90!) Aug  9 2006)
[lsc@softeng]$ g95 -fprofile-arcs -ftest-coverage -o test test.f90 && ./test
[lsc@softeng]$ gcov test.f90
test.gcno:version '400*', prefer '304R'
test.gcda:version '400*', prefer version '304R'
test.gcda:corrupted
```

As *gcov* is designed to be backward compatible, a possible solution to this issue is to always maintain a *gcov* version that is higher than or equal to all installed compilers. A mismatching (but higher) *gcov* version would result in a warning message, but the analysis will still be successful.

```
[lsc softeng]$ gcov4 -v | head -n1
gcov (GCC) 4.1.1 20070105 (Red Hat 4.1.1-53)
[lsc@softeng]$ g95 --version | head -n1
G95 (GCC 4.0.3 (g95 0.90!) Aug  9 2006)
[lsc@softeng]$ g95 -fprofile-arcs -ftest-coverage -o test test.f90 && ./test
[lsc@softeng]$ gcov4 test.f90
test.gcno:version '400*', prefer '401p'
test.gcda:version '400*', prefer version '401p'
File 'test.f90'
Lines executed:100.00% of 3
test.f90:creating 'test.f90.gcov'
```

5.6.3 Premature program termination

Coverage data files (**.gcda*, or **.da* for older compilers) are written out only at the end of a program execution. Therefore, if a program terminates prematurely due to conditions such as a segmentation fault or a termination signal, the data files will not be output and coverage analysis cannot be performed.

5.6.4 Multi-statement lines

The lowest resolution that *gcov* can accumulate coverage statistics is on a per-line basis. Therefore, when more than one executable statement is on a single line (e.g. when using elaborate macro expansions) the statistics will be less helpful as *gcov* will attribute all activity to that one line. For example, it will not be obvious if any of the statements defined on that line is not executed if some other statement was executed.

6 Case Study: Coverage Analysis of the Finite Element Library (FELIB) Using *gcov* and *LCOV*

In this section, we will go through the motions of setting up a comprehensive test coverage analysis workflow for an existing software library which contains multiple source directories and several test programs.

We will look at how the use of *gcov* and *LCOV* can be seamlessly integrated into the existing build process. We will then take a step further by exploring how *LCOV* can be used to gather the coverage analysis results and present them in a more user-friendly and navigable format

6.1 About the FELIB source code

For the purpose of this walkthrough, we have chosen the Finite Element Library (FELIB) – a program subroutine library for the solution of partial differential equations using the finite element method. We will be using version 4.0 of FELIB which is written in Fortran77.

The source code of FELIB is freely available from the project page on CCPForge [8] and it includes a set of example programs and input data. The provided source is organised in the following directory structure:

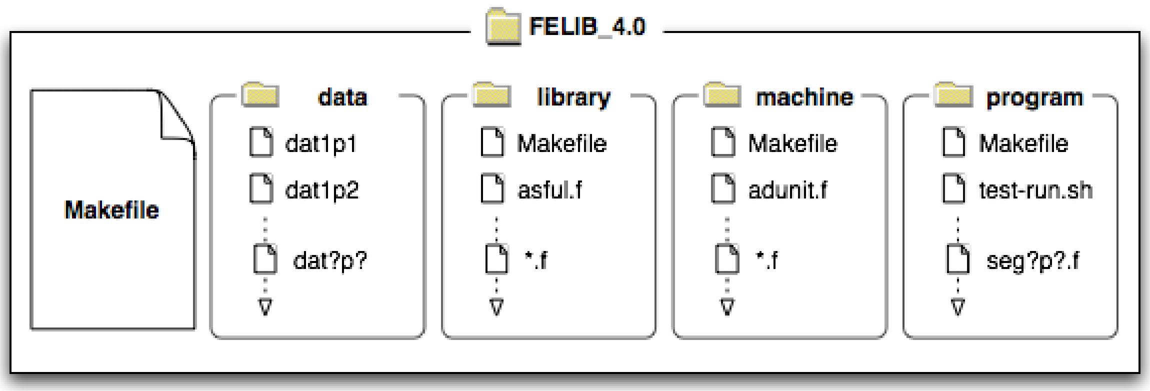


Figure 2: The FELIB4.0 source layout

⇒ `library/` – basic library routines

⇒ `machine/` – machine dependent routines

⇒ `program/` – example programs

⇒ `data/` – example data

⇒ `Makefile` – *make* configuration file used to build the library and example programs

For more information, refer to the `install.txt` file also included with the source.

6.2 The walkthrough

6.2.1 Preparation

To begin, we first ensure that we have *gcov* and *LCOV* installed. We should also check that our version of *gcov* is equal to or higher than that of the compiler we intend to use.

```
[lsc@aphek]$ which lcov && lcov --version
/usr/bin/lcov
lcov: LCOV version 1.8
[lsc@aphek]$ which gcov && gcov --version | head -n1
/usr/bin/gcov
gcov (GCC) 3.4.6 20060404 (Red Hat 3.4.6-10)
[lsc@aphek]$ which g77 && g77 --version | head -n1
/usr/bin/g77
GNU Fortran (GCC) 3.4.6 20060404 (Red Hat 3.4.6-10)
```

Next, we extract the source code for FELIB.

```
[lsc@aphek]$ tar zxf felib4.0.tar.gz
[lsc@aphek]$ cd felib4.0
[lsc@aphek]$ ls
data   install.txt   library   machine   Makefile   programs   results
```

We are now ready to build FELIB and analyse the coverage of its test programs.

6.2.2 Compiling the project and running the tests

To perform coverage analysis on a project, we need to include the `-fprofile-arcs -ftest-coverage` flags to the compilation and linking steps. We can do so for FELIB by running "`make all`" with the necessary flags specified in the arguments. This will compile the library and example programs with coverage support built-in.

```
[lsc@aphek]$ COVFLAGS="-fprofile-arcs -ftest-coverage"
[lsc@aphek]$ make all FFLAGS="${COVFLAGS}" LFLAGS="${COVFLAGS}"
cd machine;make FC=g77 FFLAGS="-fprofile-arcs -ftest-coverage" LFLAGS="-fprofile-arcs -ftest-co
make[1]: Entering directory '/misc/lsc/work/coverage/felib4.0/machine'
g77 -fprofile-arcs -ftest-coverage  -c -o adunit.o adunit.f
g77 -fprofile-arcs -ftest-coverage  -c -o maxint.o maxint.f
# .... (remaining output omitted for brevity sake) ....

[lsc@aphek]$ ls machine/adunit*
machine/adunit.f    machine/adunit.gcno    machine/adunit.o
```

Now that the library and example programs are built, we can use the provided test script to execute all the example programs.

```
[lsc@aphek]$ cd programs/
[lsc@aphek]$ ./test-run.sh
********************************************
Running  seg1p1
Data=dat1p1 Results=#res1p1
Differencing results (old/new)
*** Differences found: see file #diff1p1
********************************************
Running  seg1p2
Data=dat1p2 Results=#res1p2
Differencing results (old/new)
*** Differences found: see file #diff1p2

# .... (remaining output omitted for brevity's sake) .... #

[lsc@aphek]$ ls ../machines/adunit*
machine/adunit.f    machine/adunit.gcda    machine/adunit.gcno    machine/adunit.o
```

After the execution of each test program a **.gcda* file is generated for each source file. This file contains the analysis data that will be used by *gcov* to analyse code coverage.

6.2.3 Generating a report using *LCOV*

Now that the analysis data is available we can proceed with compiling the results using *LCOV*.

LCOV will recursively search a given directory for relevant files and invoke *gcov* to generate an 'info' file. The 'info' file can then be passed on to *genhtml* – a command that comes packaged with *LCOV* – to produce a web-based report

```
[lsc@aphek]$ pwd
/home/lsc/felib4.0
[lsc@aphek]$ ls
data  install.txt  library  machine  Makefile  programs  results

[lsc@aphek]$ lcov --capture --directory . --output-file felib.info
Capturing coverage data from .
Found gcov version: 3.4.6
Scanning . for .gcda files ...
Found 26 data files in .
Processing ./machine/maxint.gcda
Processing ./machine/adunit.gcda
## .... (some output omitted for brevity's sake) .... ##
Processing ./library/chosol.gcda
Processing ./library/prtvec.gcda
Finished .info-file creation

[lsc@aphek]$ genhtml --output-directory lcov_html felib.info
Reading data file felib.info
Found 26 entries.
Found common filename prefix "/home/lsc/felib4.0"
Writing .css and .png files.
Generating output.
Processing file library/matnul.f
Processing file library/matmul.f
## .... (some output omitted for brevity's sake) .... ##
Processing file machine/maxint.f
Processing file programs/seg5p2.f
Writing directory view page.
Overall coverage rate: 631 of 731 lines (86.3%)
```

The resulting report will be in the form of HTML files and will be written in the directory specified by the `--output-directory` option. It can be viewed by loading the `index.html` file within the output directory using a web-browser.

LCOV invokes *gcov* behind the scenes to parse coverage results generated by the runs. Should there be a problem with version incompatibility (see Section 5.6.2), the `--gcov-tool` can be used to select a newer version of *gcov*. For example:

```
[lsc@aphek]$ which gcov4
/usr/bin/gcov4
[lsc@aphek]$ lcov --capture --gcov-tool /usr/bin/gcov4 --directory . -o felib.info
```

6.2.4 Managing multiple tests

In the previous example, the *test-run.sh* script was used to run every test program. This resulted in our coverage results being cumulative and representative of an overall coverage achieved by all tests.

In order to obtain per-test statistics, coverage data has to be captured after each test and reset before the next test is executed. This can be achieved using a script such as the one included in Appendix A. This script does the following:

```
FOR EACH test
 CALL lcov --zerocounters --directory <dir>
 RUN <testname>
 CALL lcov --capture --directory <dir> --output-file <testname>.info
END FOR EACH

CALL genhtml --output-directory <outdir> *.info
```

Listing 6: Pseudocode for gathering coverage statistics of separate tests

6.2.5 Integrating coverage analysis into FELIB's *Makefile*

To simplify coverage analysis for FELIB, we can automate the process by including another target in its *Makefile*. This target will append the necessary flags to `FFLAGS` and `LFLAGS` before performing the standard build, followed by a call to the coverage script. For example:

```
# New make target for performing coverage analysis
COVFLAGS= -fprofile-arcs  -ftest-coverage
coverage:
        # clean up so everything is rebuilt using coverage flags
        make clean
        # Add coverave flags to FFLAGS and COVFLAGS
        # Also dsable optimisation by removing all occurences of -O* in FFLAGS
        FFLAGS=``$(FFLAGS:-O%=) $(COVFLAGS)" # Set shell vars using Makefile vars
        LFLAGS=``$(LFLAGS) $(COVFLAGS)"       # Set shell vars using Makefile vars
        make all FFLAGS=``${FFLAGS}" LFLAGS=``${LFLAGS}"
        ./compile_coverage.sh # script to run test and compile coverage as html
```

Listing 7: Entries added to FELIB's *Makefile*

With that target in place, developers (and users) can generate the coverage report by simply calling "`make coverage`".

6.2.6 Sample coverage report generated using LCOV

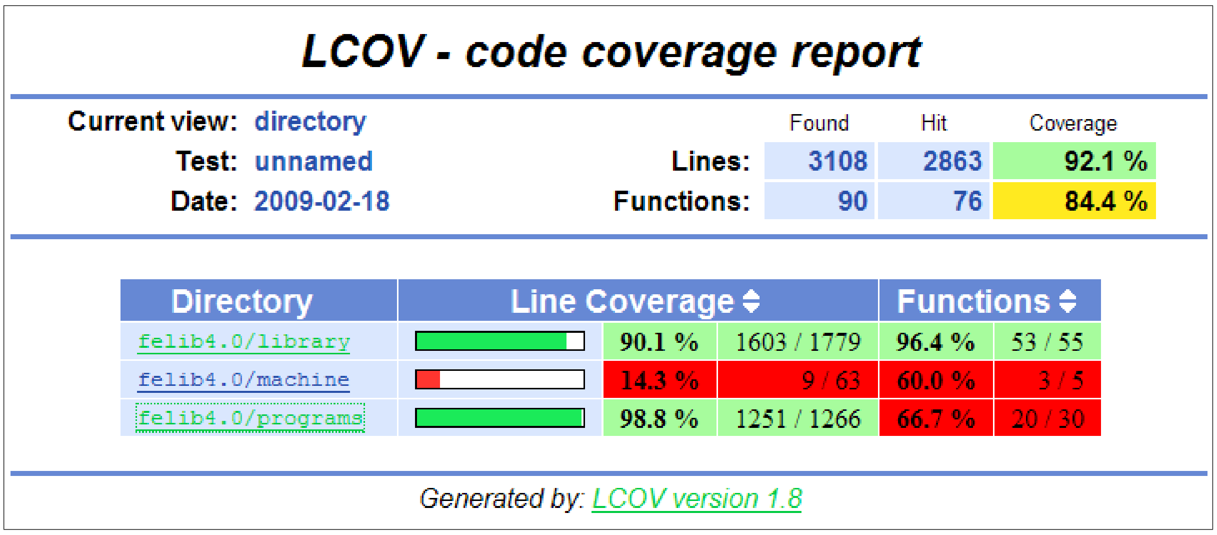
The following screenshots illustrate the different granularity of information provided by *LCOV*. An online version is also available on: http://www.sesp.cse.clrc.ac.uk/Analysis/felib4.0/coverage

Index page (Figure 3): the index page provides a summary of overall code coverage achieved as well as code coverage per source directory. Clicking on the directory name will lead us to the file listing page.

File listing page (Figure 4): this page provides the coverage achieved by the selected directory as well as coverage achieved for each source file. Clicking on the source file name will lead us to the source code listing page. Additionally, if the `--show-details` option was used with *genhtml*, we will have the option to view coverage by tests for each source file (see Figure 5).

Source code listing page (Figure 6): The source listing page provides a marked up view of the source code, with unexecuted lines highlighted in red and executed ones in blue. Optionally, if the `--frames` option was used with *genhtml*[4] a side navigation bar with a birds-eye view of the source code will be available. Clicking on the different sections of the image will bring us to the specific sections of the code.

[4]The `--frames` option requires the Perl GD module to be installed



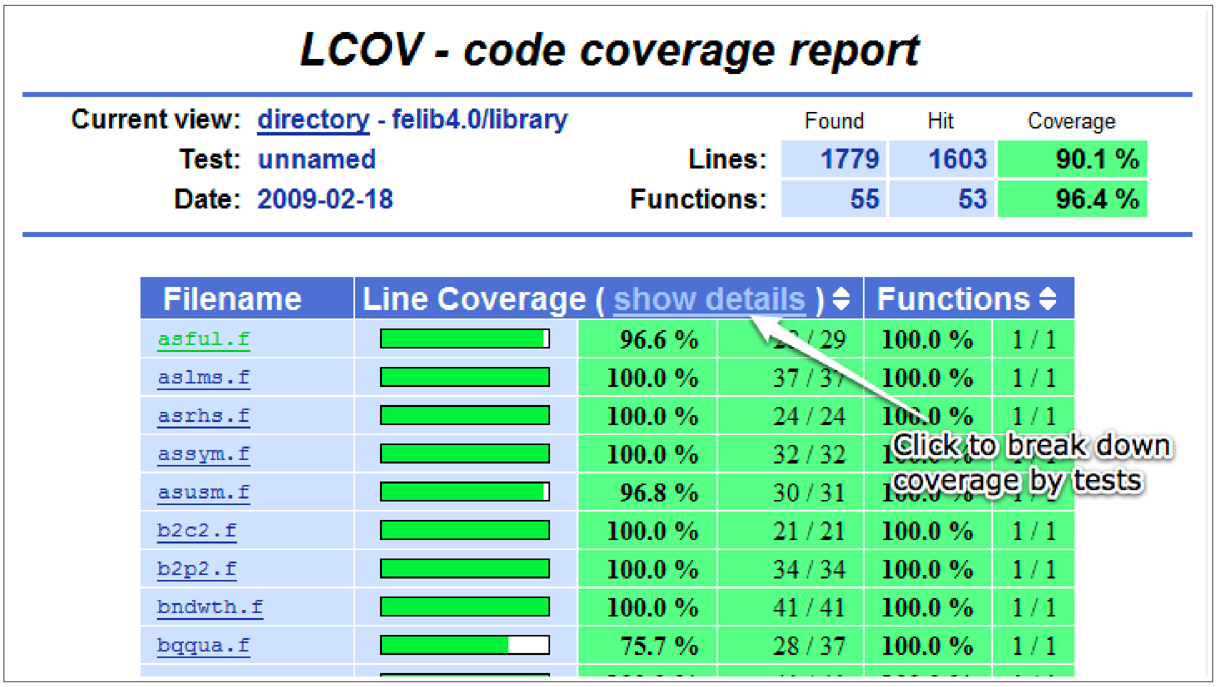
LCOV - code coverage report

			Found	Hit	Coverage
Current view:	directory				
Test:	unnamed	Lines:	3108	2863	92.1 %
Date:	2009-02-18	Functions:	90	76	84.4 %

Directory	Line Coverage			Functions	
felib4.0/library		90.1 %	1603 / 1779	96.4 %	53 / 55
felib4.0/machine		14.3 %	9 / 63	60.0 %	3 / 5
felib4.0/programs		98.8 %	1251 / 1266	66.7 %	20 / 30

Generated by: LCOV version 1.8

Figure 3: *LCOV* report – Index page

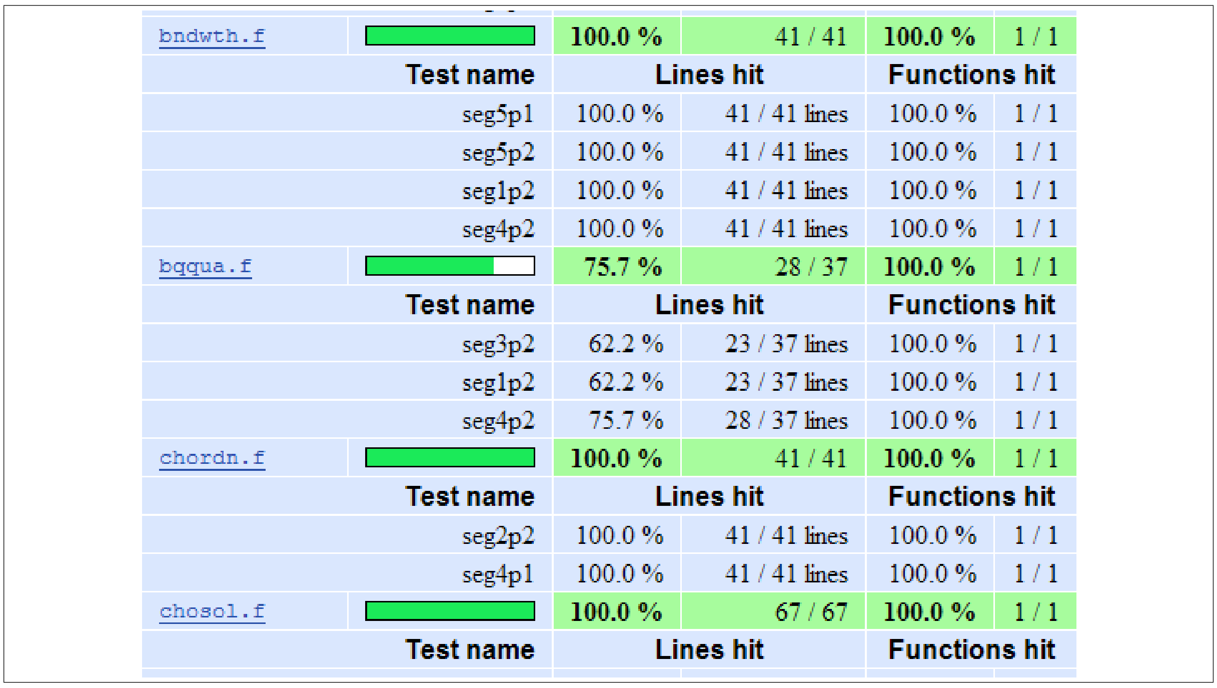


LCOV - code coverage report

			Found	Hit	Coverage
Current view:	directory - felib4.0/library				
Test:	unnamed	Lines:	1779	1603	90.1 %
Date:	2009-02-18	Functions:	55	53	96.4 %

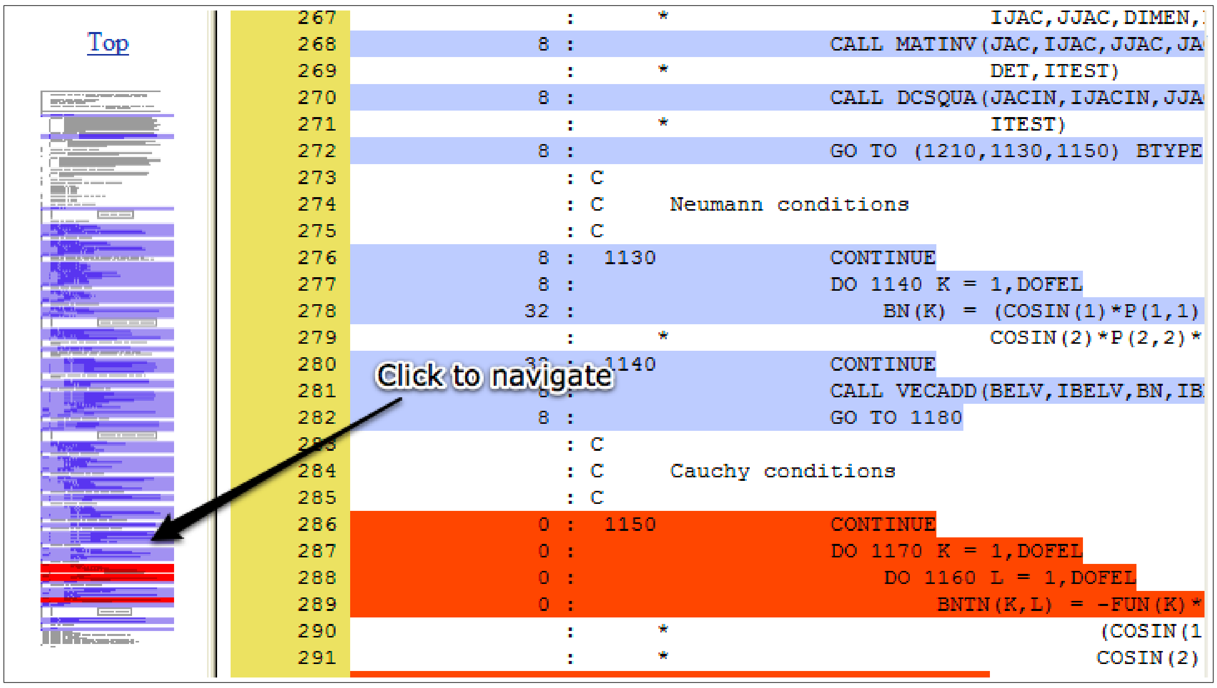
Filename	Line Coverage (show details)			Functions	
asful.f		96.6 %	[illegible] / 29	100.0 %	1 / 1
aslms.f		100.0 %	37 / 37	100.0 %	1 / 1
asrhs.f		100.0 %	24 / 24	100.0 %	1 / 1
assym.f		100.0 %	32 / 32	[illegible]	[illegible]
asusm.f		96.8 %	30 / 31	[illegible]	[illegible]
b2c2.f		100.0 %	21 / 21	100.0 %	1 / 1
b2p2.f		100.0 %	34 / 34	100.0 %	1 / 1
bndwth.f		100.0 %	41 / 41	100.0 %	1 / 1
bqqua.f		75.7 %	28 / 37	100.0 %	1 / 1

Figure 4: *LCOV* report – File listing page



bndwth.f	🟩	100.0 %	41 / 41	100.0 %	1 / 1
	Test name	Lines hit		Functions hit	
	seg5p1	100.0 %	41 / 41 lines	100.0 %	1 / 1
	seg5p2	100.0 %	41 / 41 lines	100.0 %	1 / 1
	seg1p2	100.0 %	41 / 41 lines	100.0 %	1 / 1
	seg4p2	100.0 %	41 / 41 lines	100.0 %	1 / 1
bqqua.f	🟩	75.7 %	28 / 37	100.0 %	1 / 1
	Test name	Lines hit		Functions hit	
	seg3p2	62.2 %	23 / 37 lines	100.0 %	1 / 1
	seg1p2	62.2 %	23 / 37 lines	100.0 %	1 / 1
	seg4p2	75.7 %	28 / 37 lines	100.0 %	1 / 1
chordn.f	🟩	100.0 %	41 / 41	100.0 %	1 / 1
	Test name	Lines hit		Functions hit	
	seg2p2	100.0 %	41 / 41 lines	100.0 %	1 / 1
	seg4p1	100.0 %	41 / 41 lines	100.0 %	1 / 1
chosol.f	🟩	100.0 %	67 / 67	100.0 %	1 / 1
	Test name	Lines hit		Functions hit	

Figure 5: *LCOV* report – File listing showing details of individual tests



```
         :     *                      IJAC,JJAC,DIMEN,
       8 :                       CALL MATINV(JAC,IJAC,JJAC,JA
         :     *                      DET,ITEST)
       8 :                       CALL DCSQUA(JACIN,IJACIN,JJA
         :     *                      ITEST)
       8 :                       GO TO (1210,1130,1150) BTYPE
         : C
         : C      Neumann conditions
         : C
       8 :  1130                 CONTINUE
       8 :                       DO 1140 K = 1,DOFEL
      32 :                           BN(K) = (COSIN(1)*P(1,1)
         :     *                      COSIN(2)*P(2,2)*
      32 :  1140                 CONTINUE
       8 :                       CALL VECADD(BELV,IBELV,BN,IB
       8 :                       GO TO 1180
         : C
         : C      Cauchy conditions
         : C
       0 :  1150                 CONTINUE
       0 :                       DO 1170 K = 1,DOFEL
       0 :                           DO 1160 L = 1,DOFEL
       0 :                               BNTN(K,L) = -FUN(K)*
         :     *                                  (COSIN(1
         :     *                                  COSIN(2)
```

Figure 6: *LCOV* report – Source code listing page

7 Conclusion

Code coverage analysis is a vital component in any software testing process. It provides developers with a measure of how well their source code is being exercised by the tests which can in turn be used to estimate how effective the tests would be in detecting errors in the code.

A 100% pass rate for a test procedure with low coverage would be meaningless since most of the errors may lay dormant in sections of the code that were not exercised by the tests. In fact, not only would such tests be useless, they are also dangerous as they lull developers into a false sense of confidence in their code.

While full coverage is a desirable target it is not always viable. We believe that developers should decide on a high but practical target that is suitable for their project, and most importantly, ensure that any sections of the code that are not exercised by the tests be manually inspected by the developer.

Furthermore, merely looking at overall coverage targets can sometimes be insufficient as it can mask huge uncovered blocks in a large project. Therefore, it is often useful to occasionally inspect the coverage levels at a lower granularity, i.e. per-source-file coverage levels. The inspection process can be made easier using post-processing tools such as *LCOV* which can present the detailed coverage report in a user-friendly and easily navigable format.

As presented in Section 6, coverage analysis using tools such as gcov and *LCOV* can be integrated into the build process and potentially automated. It may involve a small investment in effort to have such a system in place, but it would certainly be worth while.

References

[1] G.J. Myers, *The Art of Software Testing, 2nd Edition*, John Wiley & Sons inc., (2004).

[2] E. Dustin, *Effective Software Testing: 50 Specific Ways to Improve Your Testing*, Addison-Wesley, (2003)

[3] R.V. Binder, *Testing Object Oriented Systems: Models, Patterns and Tools*, Addison-Wesley, (2000)

[4] R.D. Craig, S.P. Jaskiel, *Systematic software testing*, Artech House, (2002)

[5] B. Marick, *How to Misuse Code Coverage*, Reliable Software Technologies, (1999)

[6] W.V. Hagen, *The Definitive Guide to GCC, Second Edition*, Apress, (2006)

[7] *Software Engineering Support Programme*, http://www.sesp.cse.clrc.ac.uk

[8] C. Greenough, *The Finite Element Library*, http://ccpforge.cse.rl.ac.uk/projects/felib

[9] *FUnit Documentation*, http://nasarb.rubyforge.org/funit

[10] *LCOV - the LTP GCOV extension*, http://ltp.sourceforge.net/coverage/lcov.php, Linux Test Project

[11] *Using ggcov with Fortran*, http://www.softeng.cse.clrc.ac.uk/blog/2009/07/using-ggcov-with-fortran/, SEG Blog

[12] *Using the GNU Compiler Collection (GCC)*, http://gcc.gnu.org/onlinedocs/gcc, Free Software Foundation, Inc., (2008)

[13] S. Cornett, *Code Coverage Analysis*, http://www.bullseye.com/coverage.html#release

A Coverage script example

```
#!/bin/bash
#==================================================
# Sample Coverage Testing script for felib4.0
#==================================================

#-----------------CONFIG VARS-----------------

# Project title
TITLE="FELIB4.0"

# Directory to begin searching for project files
SOURCE_DIR="." # use current working directory

# Tests to run
TESTS="1p1 1p2 2p1 2p2 3p1 3p2 4p1 4p2 5p1 5p2"

# Output directory for results
OUTDIR="lcov_html"

# Path to executables
GCOV="/usr/bin/gcov"
LCOV="/usr/bin/lcov"
GENHTML="/usr/bin/genhtml"

# Log files
LOG_FILE="coverage.log"
ERR_FILE="coverage_err.log"

# Do we have the Perl GD module installed?
HAVE_PERL_GD=1

#-----------FUNCTIONS---------------------------

# Error handling function
function quitOnError {

    RC=$1; MSG=$2

    if [ $RC -ne 0 ]; then
        echo "ERROR: $MSG" >&2
        exit 1
    fi
}

#----------------------------------------------------

# Empty log files (create if does not exist)
> $LOG_FILE
> $ERR_FILE

# For each test
for T in $TESTS
do
    # Set required variables
    PROG_EXE="programs/seg${T}"  # Test Executable
    DATA_FILE="data/dat${T}"     # Test Input Data
    RES_FILE="programs/#res${T}" # Result file

    echo "Running test (${T})"
    echo "---------------------------"

    # Zero gcov counters
    echo -n "* Resetting execution counters ... "
    $LCOV --zerocounters -d $SOURCE_DIR >> $LOG_FILE 2>> $ERR_FILE
    quitOnError $? "See $ERR_FILE for details"
    echo "DONE"

    # Run test
    echo -n "* Running program ($PROG_EXE) on data ($DATA_FILE) ... "
    $PROG_EXE < $DATA_FILE > $RES_FILE 2>> $ERR_FILE
    quitOnError $? "See $ERR_FILE for details"
    echo "DONE"

    # Capture coverage data
    echo -n "* Capturing coverage data ... "
    OUTFILE="gcov_${T}.info"
    OUTFILELIST="$OUTFILE $OUTFILELIST" # Append filename to list
    $LCOV --capture -d $SOURCE_DIR --gcov-tool $GCOV -t seg$T \
          -o $OUTFILE >> $LOG_FILE 2>> $ERR_FILE
    quitOnError $? "See $ERR_FILE for details"
    echo -e "DONE\n"
done


# if Perl::GD installed, include the --frames option for genhtml
EXTRA_OPT=""
if [ $HAVE_PERL_GD -ne 1 ]; then EXTRA_OPT="--frames"; fi

# Generate HTML from list of output files
echo -n "* Generating HTML ... "
$GENHTML $EXTRA_OPT --title "${TITLE}" --show-details \
         -o $OUTDIR $OUTFILELIST >> $LOG_FILE 2>> $ERR_FILE
quitOnError $? "See $ERR_FILE for details"
echo "DONE"

# Clean up
rm $OUTFILELIST

echo -e "\nAnalysis output available in $OUTDIR"

# // SCRIPT END
```