Seminar: The Interplay of Mathematical Modelling and Numerical Simulation

Using Fortran code in programs on other languages

Kernitsky Igor

July 3, 2005

Abstract

In this work method of utilizing of Fortran code is presented. Survey of possible methods of linking with Fortran is given. Among them are linking with object files, with dynamic link libraries and COM objects. Thier benefits and drawbacks are highlited. DLLs are considered to be most suitable to preserve original Fortran performance and precision. Main concepts of dynamic linking are described by the example of Windows OS Dynamic Link Libraries.

Their important feature – full independence from source code and from progamming language can be used to link any progarm with mathematical libraries on Fortran. But sometimes Fortran subroutines could not be easily called from other language. Possible problems are data misalignment, callbacks, data type mismatch. In this case special Shell DLL should be used. This method was successfully applied for LAPACK, FFTPACK and ODEPACK libraries. Some of their subroutines were integerated into new programming language BARSIC. Several performance benchmarks are listed.

Contents

1 Introduction

Present day scientists and programmers have access to a set of large, comprehensive and widely used mathematical libraries, which are intended to solve different problems. Among them are NAG, IMSL, and huge amount of libraries from NETLIB repository [1]. Some of them are fully free or distributed under GNU General Public License, so they can be used free of charge. Mathematical algorithms that are represented in these libraries have quite good performance and they have almost no errors, because they were tested and used by many people. These libraries allow programmers to write high-performance and highly relialble code.

Overwhelming majority of free code in the field of numeric scientific calculations is written and distributed in Fortran, because for a long time there were no acceptable alternative to this language. Primary goal was to increase performance and it was reached to the prejudice of interactivity, usability and stability of program code. Fortran itself, being a language with weak typification (even no type checks are made in a calls to subroutines) and unsafe syntax, can be a source of unreliability of program code. Nowadays Fortran is replaced more and more by programming languages that offer more usable and effective tools for application development (like Java, C++, C#).

2 Possible means of utilizing of Fortran sources

During last few years most popular mathematic libraries were translated from Fortran to C++ or Java, among them are LAPACK, FFTPACK and Jama, but the translation is not full or not effective enough, so programming with them is rather problematic.

It is significant to mention that translation from one language to another is not a best solution. It is obvious that development time will increase greatly for manual translation. To overcome this problem machine translation could be used. It means that Fortran source code is translated by special program translator. But translator is only a program so it cannot understand meaning of the code that is processed. It leads to unreadability of result code and as a consequence to problems with any modifications of translated sources. Loss of performance or precision also may occur. Another problem is that no one can guarantee that output code of manual or machine translation would be reliable and stable as the Fortran code was. The only benefit is that mathematical library will be fully compatible with the development tools of selected language (i.e. compiler or interpreter).

Even if it is acceptable to compile source code of the library with the project problem may appear from compilers mismatch. For example if some code was expected to be compiled with Intel C++ compiler there is no guarantee that it would be successfully compiled by Microsoft Visual C++ compiler (same mismatch may occur with Microsoft Fortran compiler and other Fortran compilers). These inconsistencies mostly occur with rare and small libraries. For big and popular solutions it is not a problem, because their source code usually satisfies one of language standards (Fortran77, Fortran90, ANSI C, etc.).

Sometimes math libraries are distributed as a set of object or library files. Object file is binary file that contains compiled code and information that is required for linker to build a project. Main benefit of such approach is that library sources were compiled on their native language (e.g. Fortran), so all features of Fortran library, like reliability, performance and high precision, are present. This approach is good enough but in some cases is cannot be applied. First it is a problem of use with languages that don't support object files or even have no linking process at all (like Java or C#).

Another trouble is possible incompatibility with destination language. It descends from different sets of standard subroutines in different languages. In general, object file does not contain all code that must be used for linking. It may contain references to subroutines that reside in other object files of the library and object files which are elements of Fortran standard

libraries. All references are resolved by subroutine names. So if project that is compiled with one language contains object files compiled with other language following linker errors may occur: duplicate reference, unresolved reference. They could be solved by removing, renaming and including appropriate subroutines into project or into set of language standard libraries, but in any case these operations produce unreliable and unstable result. One of possible reasons of instability is that different languages use different memory managers; also subroutines in language standard libraries can work in different ways and even produce different results, while having the same name.

More suitable and flexible way of embedding of external executable code into a program is in linking a program with dynamic link libraries. They are named DLLs (dynamic link libraries) in Windows OS and SOs (shared objects) in Linux OS family. Dynamic libraries allow to eliminate all problems connected with compilers and languages incompatibility and to keep all benefits of Fortran source code (i.e. performance and stability) intact.

COM objects and .NET components also must be mentioned. They support all DLL features and even allow more rigid control of linking process, data types and component versions, but in order to use them programmer have to select language that allows working with COM objects or .NET framework. Process of encapsulating of Fortran code into COM object is complex enough, too. Programmer is expected to be an expert in object oriented programming. According to all mentioned properties and requirements DLL is considered to be most suitable for linking programs with functions from scientific and mathematical libraries. Some libraries already have their codes linked into DLL, for example FFTW library (www.fftw.org).

3 Principles of dynamic linking

DLL module in OS Windows is a file on disk with *.dll extension. It consists of some global data, set of compiled functions and some resources. Among them is the information about functions that DLL exports and about other DLLs that are imported. Process of loading and initialization of a DLL is carried out by means of OS. Each process in Windows can use any DLL it needs. All DLL global information is unique per process and could be read or written only by the process into whose context DLL was loaded. To acquire fully global and shared data (shared between all processes) special data segment must be declared in DLL.

To perform runtime linking with DLL operating system needs to identify functions that are exported. For this purpose each exported function must have name or ordinal value. Function name may or may not be the same as in DLL source code. All function identification information (exports) is placed by linker into special table. It is named exports table and is placed at the end of DLL. Another table imports table can be found in all executable modules that are in PE (portable executable) format. Import table directs OS what functions should be loaded to resolve all references. Each imported DLL have corresponding imports table (at least import table for kernel32.dll could be found in most executables). Part of import table holds names or ordinals of functions to be imported another part is filled by OS at runtime by their addresses in memory context of the process. Linking method, described above is called implicit linking.

To perform implicit linking with some dynamic library programmer need to supply special import library that contains information for linker about how to build import table. This library could be obtained with DLL or generated by utilities supplied with programming language, for example lib.exe for Microsoft Visual Studio C++ compiler.

Another way to link with DLL is to perform explicit linking. This process is more complex, but it gives more flexibility for linking and error handling. Windows kernel exports three functions that are used for explicit linking.

LoadLibrary - is used to map a DLL module into address space of calling process. It

returns a handle that can be used in GetProcAddress to get the address of a DLL function. LoadLibrary can also be used to map other executable modules. For example, the function can specify an .exe file. If the module is a DLL not already mapped for the calling process, the system loads and maps it. Also DllMain function is called to initialize DLL.

LoadLibrary is often followed by several calls to GetProcAddress. This function allows to obtain address of function, exported from DLL by DLL handle and function's identification information (i.e. name or ordinal). DLL is unloaded automatically, when the last handle to it is closed by a call to FreeLibrary function [2].

The main benefit of explicit linking is that the program is at least started, so some special measures to handle errors of function import could be taken. For example if implicitly linked DLL could not be found at program startup, program's code even will not get control, so programmer cannot handle these errors in any way. Operation system will simply display a message like following: "Application failed to start because mydll.dll was not found" or "The ordinal N could not be located in the dynamic link library MyDLL.dll". Sometimes exact DLL is not needed for the task that program is expected to perform (plug-in technology). In this case only explicit linking could be used.

Significant problem of dynamic linking approach is that DLL modules have no information about parameters of exported functions. So no type and even count checks could be performed at runtime. Sometimes DLLs that are linked with a program are replaced by newer versions which have different function ordinal numbers or parameters. After such a replacement program can be unable to start at all or may produce unpredictable results. This problem is also known as DLL Hell. Solution lies in use of component object model (COM) and interfaces.

All responsibility for correct parameters passing and function call lies on a programmer. When calling a function by its address programmer must know not only types of parameters, but also a calling convention. It consists of order in which parameters are pushed on to stack and information about who will free the stack after call ends [3]. Most common calling conventions are:

- stdcall – parameters are pushed on to stack from right to left. Stack is freed by invoked procedure.
- cdecl – parameters are pushed right to left, stack is freed by caller
- fastcall – first two parameters are passed through CPU registers, others from right to left
- safecall – used in Delphi for correct handling of raised exceptions. Analogue of stdcall

Calling conventions impose some restrictions on language specific features. For example function with variable count of parameters (like C function printf) could be implemented only using cdecl calling convention, because it's first parameter is pushed on to stack last and stack is freed by caller (only caller exactly knows how many parameters were put on to stack). Inconsistency between calling conventions that are used by caller and subroutine in DLL will, in general, lead to runtime errors.

Following steps are performed when DLL is created:

- Programmer compiles source (*.f) files, if he has some, to obtain object files
- Programmer writes definitions file (*.def) where he declares all exported functions, their export names and ordinals (at least ordinal or name must be specified)
- Then object files from first step and may be some other are passed to linker to obtain dynamic library module. Also library for implicit linking is produced.
- This step takes place only if linker is not capable of producing static library. In this case programmer needs to find special utility that will wrap static library into simple DLL. For example dllwrap tool is needed to make DLL when using GNU Compilers Collection in version for Windows (MinGW).

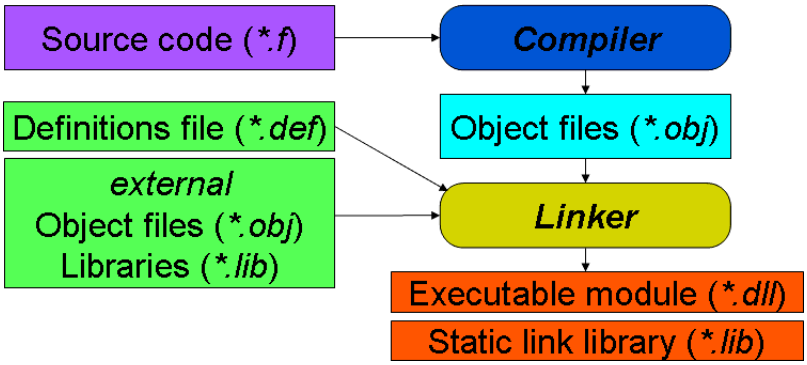

Figure 1: DLL creation schema

Minimal information that one will need to link a program with DLL is (*.dll) file itself and knowledge about exported functions parameters and names. All other files can be generated automaticaly or written by programmer, but it is better to obtain import library (*.lib) if implicit linking is needed, and function headers for C++ (*.h) with a DLL.

Below is the sample code of function S(X) that calculates square of X **(*.f file)**:

```
REAL FUNCTION S(X)
    REAL X
    S = X*X
END
```

Definitions file defines export function named Sqr with ordinal value 1 that have name S in source code **(*.def file)**:

```
LIBRARY SQR
EXPORTS
    Sqr = S @1
```

These two files could be compiled and linked into DLL by set of utilits from MinGW. Then it could be used from other language.

Here is C++ code for implicit linking, it is assumed that import library was added to project. At first function must be declared. It is better to put declarations of functions from a DLL into header file **(*.h file)**:

```
extern "C" double _cdecl Sqr(double X);
```

This declaration means that function Sqr is external function, named in C style and it uses cdecl calling convention. Being once declared it could be called as ordinary function from any C++ code **(*.cpp file)**:

```
#include "our .h file"
    ...
double Y = Sqr(10);
```

After execution of this code value of Y will be 100.

For explicit linking no import library is needed **(*.cpp file)**:

```
typedef double (_cdecl *LPFNSQR)(double* X);
    ...
```

```
HANDLE hLib = LoadLibrary("Sqr.dll");
    ...
double Y = 0;
    if (hLib)
    {
    LPFNSQR lpSQR = GetProcAddress(hLib, "Sqr");
        if (lpSQR) Y = lpSQR(10);
    }
    ...
    if (hLib) FreeLibrary(hLib);
```

This code will produce same results as code for implicit linking, but if OS would be unable to find DLL or export function by name program will execute successfully and value of Y will be 0.

4 Complex DLL linking scheme. Improvements over the simple DLL

In some cases interface usability or resource allocation in dynamic library that could be obtained from simple linking of Fortran code is not good enough. Some measures could be held to improve simple DLL. Among them is reducing of number of function paramters, allocating temporary memory buffers and different extended input and output paramter checks. Though all these improvements could be implemented in Fortran it is better to implement them in more structured and reliable programming language like C++. One more DLL must be created. It will act as intermediate (Shell) DLL between program and Fortran (Core) DLL.

Process of building a Core DLL could be automated scince it always consitsts of the same steps. The only thing needed from Core DLL is to export all functions from Fortran code to make them accessable from other programs. Here the sequence of operations for creating Core DLL with GCC (MinGW) is listed [4]:

1. Compile (*.f) files into a set of (*.obj) files (gcc.exe)
2. Create (*.lib) file from *.obj files (ar.exe)
3. Use dlltool.exe to list all function names that could be found in (*.lib) file info (*.def) file
4. Modify (*.def) file to export only necessary functions
5. Use dllwrap.exe to create (*.dll) from (*.lib) using (*.def)
6. Third party tool could be used to create import library from (*.def) file (Microsoft lib.exe)

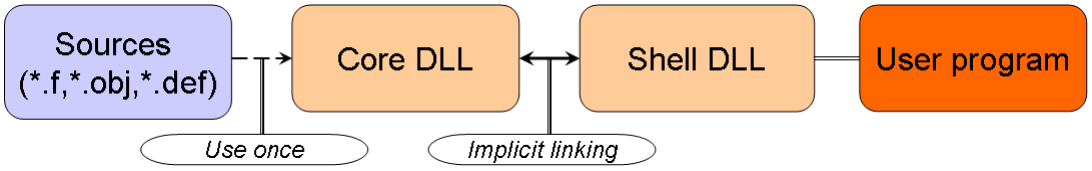

Figure 2: Complex scheme of linking DLLs

Shell DLL could be written in any language capable of producing DLLs. It is easier to use implicit linking to connect Core and Shell DLLs. Explicit linking should be used only if

different implementations of Core DLL could be loaded at runtime (e.g. loading of numerical core optimized for best performance on current CPU).

Even by using a complex scheme some difficulties with no simple solution may arise. First of them is that in some languages floating point numbers are stored in extended precision format (Float80) and majority of Fortran versions use double precision datatype (Float64). Also reverse problem may appear when special Fortran compiler with support of Float80 datatype is used. Another place where type conversion is needed is link between Shell DLL and program. C++ language supports only Float64 and Dephi language allows to use Float80. It must be mentioned that all modern FPUs use only Float80 datatype in internal operations, so in case of Delphi higher precision is obtained almost for free. To deal effectively with all floating point datatypes conversion problems following simple assembler code could be used:

```
lp:
    fld TBYTE PTR [esi]
    fstp QWORD PTR [edi]
    add esi, 10
    add edi, 8
    dec ecx
jnz lp
```

This code converts array of Float80 into array of Float64 Main idea is to push data on to FPU stack as one datatype and then pop as another. ESI is assumed to hold a pointer to source array, EDI to destination, ECX is counter register and is set to number of elements in array before the loop.

In some cases calls to Fortran functions in a DLL can produce instabilities or drops in performance in comparison with pure Fortran code. Possible reason is input or output buffer misalignment. Root of performance penalty lies in IA32 architecture. Each processor cache line is 32 bytes long. When Float64 array is aligned at 4 byte boundary each 4th array element is split between two cache lines. When CPU performs computations with these elements it is forced to work with two cache lines, so performance is reduced. On tested algorithms performance drop was from 1.2 to 2.5 times. Insure that your Float64 data is aligned at least at 8 byte boundary.

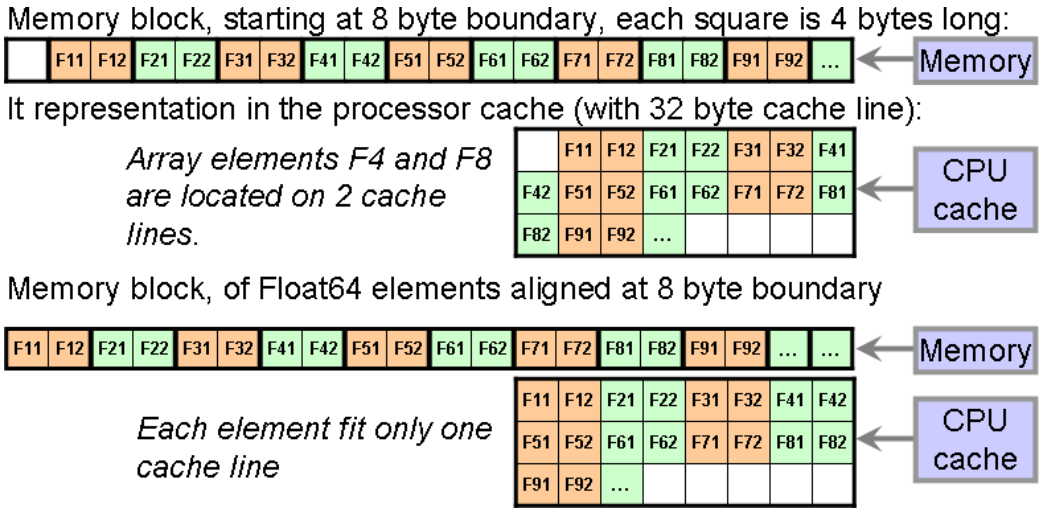

Figure 3: Example of misaligned and aligned arrays of Float64

Fortran subroutine may expect one of it's paramters to be a pointer to user-defined callback function. Problem occurs when callback function implementation in the destination language

could not be made as it is declared in Fortran. Shell DLL again helps in the solution of this problem. Langauge for Shell DLL ought to be selected to fulfil all requirements that are implied both from Fortran and from program language. In most cases C++ is enough to resolve any conflicts.

5 Practical applications. BARSIC programming language

Described method of linking was applied to functions from LAPACK, FFTPACK, SLEIGN2 and ODEPACK packages. It's efficiency could be shown by the example with a function from LAPACK. Subroutine DSYEVX is double precision (D) algorithm for symmetric matrices (SY) that allows to find eigen vectors and eigen values (EV) of a matrix. Version with exetended (X) parameter set is used. In Fortran it is declared as following:

```
SUBROUTINE DSYEVX(JOBZ, RANGE, UPLO, N, A, LDA, VL, VU, IL, IU,
                  ABSTOL, M, W, Z, LDZ, WORK, LWORK, IWORK)
  CHARACTER JOBZ, RANGE, UPLO
  INTEGER IL, INFO, IU, LDA, LDZ, LWORK, M, N
  DOUBLE PRECISION ABSTOL, VL, VU
  INTEGER IFAIL(*), IWORK(*)
  DOBULE PRECISION A(LDA,*), W(*), WORK(*), Z(LDZ,*)
```

Paramters JOBZ, RANGE and UPLO describe task parameters. JOBZ is used to define task type (find eigenvalues only or both eigenvectors and eigenvalues). RANGE defines type of range where subroutine have to compute eigen values. UPLO defines what triangle should be used (upper or lower). All these parameters could be combined in one. LDA and LDZ are first dimensions of arrays. For full matrix they could be calculated from N. Pairs IL, IU and VL, VU set range for eigenvalues for different JOBZ values. They could also be grouped in one pair. WORK and IWORK are pointers to memory buffers that could be allocated in a Shell DLL.

After all optimizations result will look as following C++ function:

```
long stdcall EV_DC_X_EX(long n, long type, double *m, double* eval, double* evect,
                        double abstol, double min, double max, long* lpevcnt);
```

Here n is matrix size; m is matrix itself; eval, evect - eigenvalues and eigenvectors that were found; abstol defines tolerance; min and max - range of search; lpevcnt is a number of eigenvalues that were actually found. One can easily see that total count of paramters for this function in Shell DLL is 9 in contrast to 18 in Fortran. For most cases interface given by Shell DLL is quite enough. Declaring of original Fortran function in a Shell DLL exports can also help user, when interface provided by adapted function is not acceptable for some reasons.

Set of LAPACK, ODEPACK and FFTPACK functions was successfully integrated into new programming language BARSIC that is developed in a research group leaded by Associate Prof. V.V.Monakhov (http://www.niif.spbu.ru/~monakhov/) at the Department of Computational Physics of Saint-Petersburg State University.

BARSIC (Buisness And Research Scientific Calculator) is programming language for education, research and business. It is a powerfull tool to develop applications for mathematical simulation, data processing and visualization, numerical calculationas and computer animations. Main field of BARSIC applications is Physics and Mathematical Physics. Graphic user interface (GUI) is one of main features of BARSIC. Visual design is similar to Visual BASIC and Delphi [5].

BARSIC is interpreting programming language. High flexibility of the programs for BARSIC is on the one hand but on the other are significant performance penalties for huge computational algorithms because of BARSIC's interpreting nature. This problem was overcomed by use of linking scheme that was introduced in conjunction with BARSIC. As a result we got programming language that allows to process data with high effeciency and then to work with it or to view a results via user-friendly interface.

This is example of BARSIC program that was used to test 2D FFT feature:

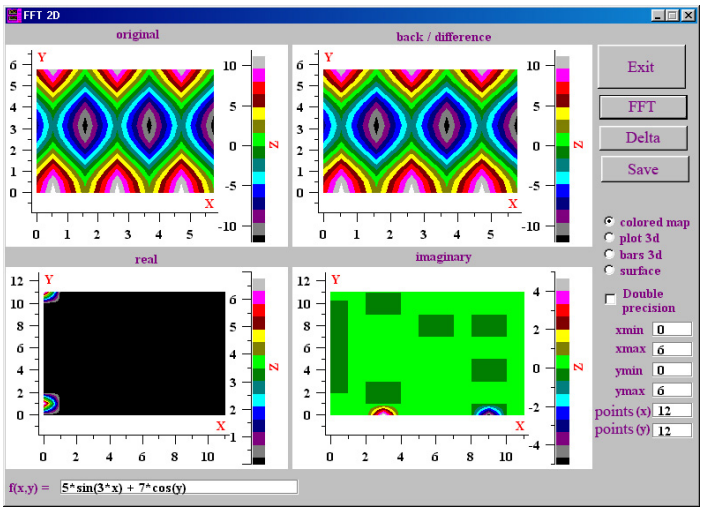

Figure 4: Typical BARSIC program interface

Performance is also kept at a good level. On the following diagrams performance level in percents is listed for several LAPACK subroutines in comparison with Maple and MATLAB.

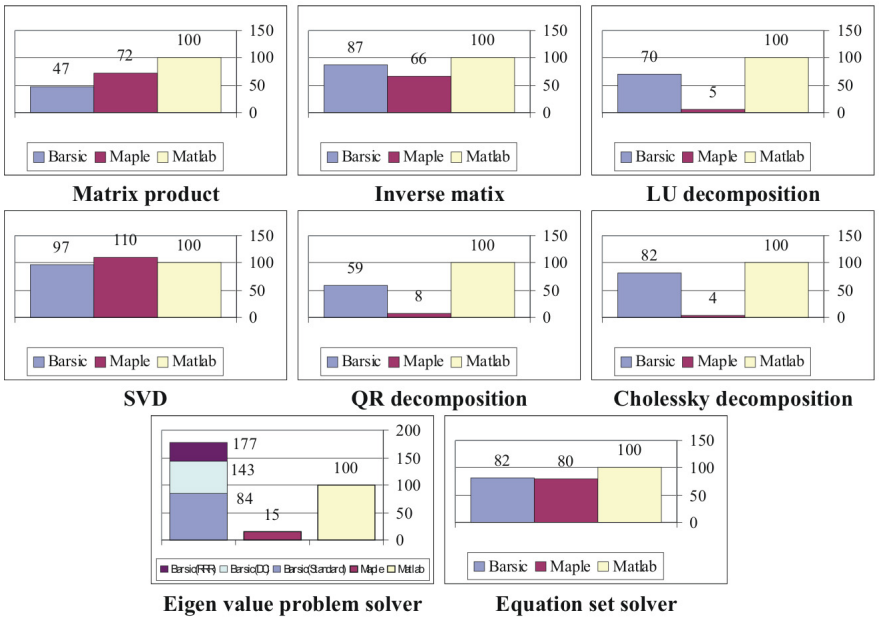

Figure 5: BARSIC LAPACK and Maple relative to MATLAB performance diagrams (the more the better)

6 Conclusion

Technique desribed above:

- keeps performance level of the original code
- is the safest way to use Fortran code
- provides maximum compatibility when compiling Fortran
- allows linking with most programming languages
- is the easiest way to enhance program functionality with powerful numerical algoithms

References

[1] NetLib repository at www.netlib.org

[2] G. Shepherd, D. Kruglinski Programming Microsoft Visual C++.NET, 2003.

[3] Microsoft Developers Network CDs, April 2003.

[4] Help files for Gnu Compilers Collection http://gcc.gnu.org/onlinedocs

[5] V. Monakhov , A. Kozhedub BARSIC - new software tool for education and research. In abstr. Int. Workshop on Computational Physics Dedicated to the Memory of Stanislav Merkuriev. St.Petersburg, 2003, p.49-50.