Designing and Implementing a Java-based GUI for Automatic Generation of MATLAB Gateway Functions of Numerical Subroutines

Johan Sejdhage

June 7, 2005
Master's Thesis in Computing Science, 20 credits
Supervisor @ CS-UmU: Robert Granat
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

In numerical computing, especially in connection to linear algebra, Fortran is still very common as programming language. One reason is the speed of the Fortran language. A problem is that constructions of relevant test examples often requires extra implementations of test programs and matrix generators which can be quite cumbersome. In addition, these extra implementations sometimes have to be rewritten for different test examples. A solution to this problem is to use the interactive matrix constructions and visualization possibilities provided by MATLAB. By implementing a tool for automatic generation of Fortran MEX-files this can be achieved. The finished tool, called OpenFGG, is platform independent with a graphical user interface which simplifies its usage. OpenFGG contains functionalities like parsing Fortran 77 source files and automatic generation of Fortran gateways and MEX-files. Tests with large libraries as LAPACK and BLAS have been done with great results. The prototype implementation will be made available to the numerical community and further improved on requests.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Fortran is still a very common programming language in numerical computing and one of the reasons is the speed, but there are also a few drawbacks. A number of problems arises during the development of library-standard numerical software for different types of matrix computations. One is that construction of relevant test examples requires additional implementations of test programs and matrix generators. Sometimes these extra implementations must be partly rewritten and recompiled when going from one test example to another. Another problem is that there is no *simple* way to visualize the generated test matrices in a clever way in a terminal window.

By implementing a tool for automatic generation of MATLAB[1] gateway functions for the developed numerical routines, one can take advantage of all the interactive matrix construction and visualization possibilities that MATLAB provides in the development and testing phases of the numerical routines. By offering the functionality of such a tool through a GUI the usage would be simplified and users may save a lot of time and effort.

1.2 Outline

This report will cover the whole project and the outline is shown below

Chapter 2 describes the problem and defines the task for this project.

Chapter 3 gives an overview of the main areas that are concerned during the work on this project.

[1] The MathWorks - MATLAB and Simulink for Technical Computing, http://www.mathworks.com

Chapter 4 covers the resulting application, and the components are described together with all implemented classes.

Chapter 5 contains a description of the restrictions of the system and a discussion of possible extensions.

Appendix A is a User's Guide with a detailed description on how to use the system.

Appendix B contains the document type definition (DTD) for the utilized XML specification.

Appendix C is a covering abbreviations and other words related to this subject.

Chapter 2

Problem Description

This chapter will introduce the task of this Master's Thesis project and gives a short summary of related work in the area.

2.1 The task

This project is a sequel of a previous project conducted during the Fall of 2004 by Magnus Andersson. That project resulted in an application called FGG (Fortran Gateway Generator). FGG is a textbased Java program that generates MATLAB gateways to numerical Fortran procedures. It has support for matrices with dimensions larger than two. For details about FGG, we refer to [1] and the references therein. The task in this Master's Thesis is to add a graphical user interface to the Fortran Gateway Generator. The advantages with this new application will be

- Users can save their work into projects.
- Users will not have to understand or work with XML[1].
- Users will not have to call `mex` by themselves.
- The usability and simplicity will increase.

In the implementation there should also be a simple text editor that can be used to edit content in files directly. There are three major reasons why the GUI should be written in Java.

- First, FGG is written in Java.
- Secondly, Java as programming language is well suited for building graphics.
- Thirdly, Java results in platform independence.

The main goal is to make the application work in UNIX environments, but other platforms such as Windows are desirable.

[1] Extensible Markup Language, http://www.xml.com

Another part of this Master's Thesis is to implement a web page from where the application can be downloaded. There should also be instructions and some examples about the usage.

2.2 Related work

There is some existing softwares that dealt with this problems. One of them is the commercial NAGWare Gateway Generator [9]. It offers generation of gateways, but the drawbacks with this software is that it is text based (see Figure 2.1), and neither the gateways nor the software itself are platform independent. Another big drawback with NGG is however that it cannot handle matrices with a dimension larger than two. This is important in research where sequences of matrices are studied, e.g., [4].

Figure 2.1: Screenshot of NGG demo version.

Chapter 3

MATLAB and MEX-Files

According to Mathworks [8] and Pärt-Enander & Sjöberg [5], MEX-files are subroutines written in either C or Fortran that can be called directly from MATLAB. There is no distinction in the way of using a MEX-file compared to an ordinary M-file. When one have a program written in C or Fortran which one wants to use in MATLAB, it can be a convenient solution to create a MEX-file and avoid rewriting it into an M-file. One can also use this technique to speed up bottlenecks because of the speed of C and Fortran compared to the slow interpretation used in execution of MATLAB M-files.

In the following sections a MEX-file will refer to a Fortran MEX-file.

3.1 The components of a Fortran MEX-File

In [8] MEX-files are split into two distinct parts, a *computational routine* and a *gateway routine*. They are described as:

- The computational routine contains the code used to perform the computations.
- The gateway routine is an interface against MATLAB. It has one entry point which is a function called `mexFunction`. The computational routine is called by the gateway as an internal subroutine.

To call a MEX-file MATLAB uses the gateway entry point. This entry point must be named `mexFunction` and have a certain set of parameters, as follows:

```
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer plhs(*), prhs(*)
integer nlhs, nrhs
```

The parameters nlhs, nrhs, plhs and prhs refers to the following:

- `nlhs` - the number of left-hand side arguments

- `nrhs` - the number of right-hand side arguments
- `plhs` - an array of pointers to left-hand side arguments
- `prhs` - an array of pointers to right-hand side arguments

Below follows an example that illustrates how this works in practice. When MATLAB invokes a MEX-file with the command

```
result = foo(a,b)
```

the function `mexFunction` is called with the arguments

- `nlhs` = 1
- `nrhs` = 2
- `plhs` $\longrightarrow$ [•] $\longrightarrow$
- `prhs` $\longrightarrow$ [•] $\longrightarrow$ a
  [•] $\longrightarrow$ b

The output argument `result` is set when the subroutine executes, therefore `plhs` is pointing to nothing at this moment. When the execution is finished the argument `result` is set and the control is given back to MATLAB.

How are the parameters passed between MATLAB and Fortran? The answer is that data is copied using special routines and the special data type `mxArray`. MATLAB allocates memory and sends pointers as parameters to the MEX-file. To fetch data from the input parameters the `mxGet` functions is used. Memory for the output parameters are created with the `mxCreate` functions and the pointers in `plhs` are set to point to this memory space. To get a feeling of how it works we can study the following example (from [8]):

```
C Computational subroutine
subroutine timestwo(y, x)
real*8 x, y

y = 2.0 * x
return
end


C The gateway routine
subroutine mexFunction(nlhs, plhs, nrhs, prhs)

integer mxGetM, mxGetN, mxGetPr
integer mxIsNumeric, mxCreateDoubleMatrix
integer plhs(*), prhs(*)
integer x_pr, y_pr
integer nlhs, nrhs
integer m, n, size
real*8 x, y
```

```
C Check for proper number of arguments.
if(nrhs .ne. 1) then
   call mexErrMsgTxt('One input required.')
elseif(nlhs .ne. 1) then
      call mexErrMsgTxt('One output required.')
endif

C Get the size of the input array.
m = mxGetM(prhs(1))
n = mxGetN(prhs(1))
size = m*n

C Check to ensure the input is a number.
if(mxIsNumeric(prhs(1)) .eq. 0) then
   call mexErrMsgTxt('Input must be a number.')
endif

C Create matrix for the return argument.
plhs(1) = mxCreateDoubleMatrix(m, n, 0)
x_pr = mxGetPr(prhs(1))
y_pr = mxGetPr(plhs(1))
call mxCopyPtrToReal8(x_pr, x, size)

C Call the computational subroutine.
call timestwo(y, x)

C Load the data into y_pr, which is the output to MATLAB.
call mxCopyReal8ToPtr(y, y_pr, size)

return
end
```

First of all, one can see that the gateway contains the entry point `mexFunction` with all the required parameters. The validity of the parameters is checked and an error message is displayed otherwise. After that the size of the input parameter is fetched by calling `mxGetM(prhs(1))` and `mxGetN(prhs(1))`. The call to `mxCopyPtrToReal8` copies the data that `prhs(1)` points to into the variable x. This variable is sent as argument to the computational routine that calculates the product. When the subroutine `timestwo` returns the variable y holds the output data. Finally the output in y is copied into the memory used as output to MATLAB. MATLAB returns the control and fetches the output data.

3.2 Creating MEX-Files

MATLAB provides a script called `mex` which is used to create MEX-files from Fortran subroutines. The only requirement is that a Fortran compiler is installed

on the target computer. The calling sequence to `mex` looks like

```
mex [option1 ... optionN] sourcefile1 [... sourcefileN]
    [objfile1 ... objfileN] [libraryfile1 ... libraryfileN].
```

An example, where we use the routine `timestwo` described on the previous page, would be:

```
mex timestwo.f
```

This will create a MEX-file called `timestwo` with the platform specific extension. The user may find more information using `mex -help`.

3.3 Fortran Gateway Generator

The Fortran Gateway Generator was conducted as a Master's Thesis during the Fall of 2004 by Magnus Andersson [1]. FGG is a system that semi-automatically generates MATLAB gateways of Fortran functions and subroutines to make MATLAB usable as test platform. FGG is built upon two major components: a *source code analyzer* and a *gateway generator*.

To parse the source code FGG uses a parser generator called JavaCC[1]. JavaCC is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. When the parse step has been finished, the information is converted into an XML-specification. The XML file is used to pass information between the two components. FGG uses JDOM, which is a Java representation of an XML document, to create the XML-specification.

The gateway generator reads from the XML-specification and creates a gateway and M-files that corresponds to the specification. This gateway is compiled together with the source files into a MEX-file. The resulting MEX-file is called from the corresponding M-file.

The procedure is as follows:

```
java -jar fgg.jar foo.f

java -jar fgg.jar foogw.xml

mex foogw.f foo.f
```

After the first and before the second step it is possible to change properties for the parameters in the XML-specification. But to force the user to edit an XML file is not desirable because it should not be an requirement to know XML just to use the application. This is one reason why a GUI would simplify the usage. Even if the user have knowledge about XML, a GUI would probably be better and faster.

[1] JavaCC Home, https://javacc.dev.java.net/

Chapter 4

Results

This chapter will give an insight in how Open Fortran Gateway Generator (OpenFGG) is built. Components and classes will be described in close and the basic functionality will be explained.

During the development of OpenFGG some general guidelines were followed.

- The usability was very priorited, because that also affects what users think of the application. An application that is hard to use will not become very popular.
- The degree of difficulty to extend the application. The implementation strived to make extensions possible and keep the different functionality apart from each other. By having natural and clear boundaries it facilitates for possible extensions.

4.1 Components

The core of this system is the Fortran Gateway Generator (FGG). Proper modifications have been made to this component in purpose to make it fit in OpenFGG and to make data exchange between new and old components simple. An overview of the implemented system is presented in Figure 4.1. The main components of the system are:

OpenFGG - This is the main class. When the application starts an instance of the OpenFGG_GUI class is created to interact with the user. The class contains the fundamental functions. Actions performed by the user in the GUI is transferred into this class and then redirected to a more specific instance.

OpenFGG_GUI - The interface between the user and the system. Contains many kinds of components and listeners[1] to facilitate the usage.

[1] A listener is used by a component to dispatch events, e.g. mouse movements and keyboard input.

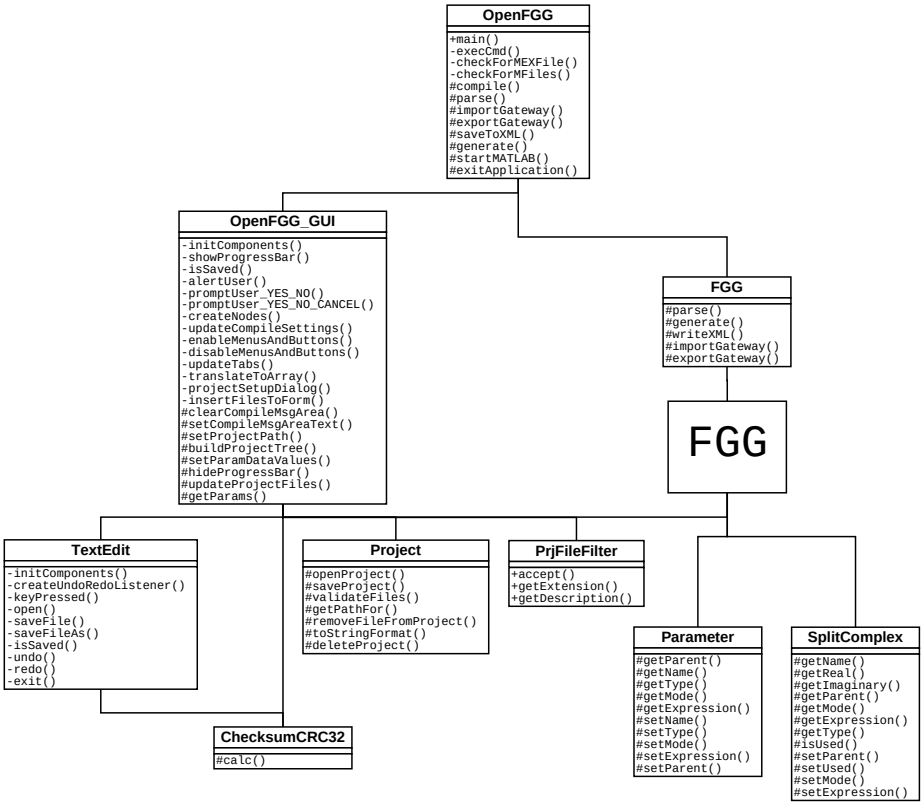

Figure 4.1: Overview of the components in OpenFGG.
The FGG package refers to the work done by Magnus A.

Project - Represents a project, including files and other properties. Projects are used to save users work, see Section 4.2.1.

PrjFileFilter - A file filter class used to filter the visible files depending on which action that is performed.

TextEdit - A text editor for making easy changes to files.

ChecksumCRC32 - A class that can be used to compute the CRC-32 of a data stream (see Appendix C for details).

FGG - This class acts as a gateway between the new part with the GUI and Fortran Gateway Generator (the FGG package in the Figure) with the core functions.

Parameter - All ordinary parameters (see User's Guide in Appendix A) are stored as an object of this class to exchange data between Fortran Gateway Generator and the GUI.

SplitComplex - All split-complex parameters (see User's Guide in Appendix A) are stored as an object of this class to exchange data between Fortran Gateway Generator and the GUI.

4.2 System description

In Section 4.1, an overview of the implemented components was given. This section will in more detail describe the collaboration between components and their functionality.

4.2.1 Projects

The system is based upon a structure which allows users to save their work as *projects*. A project consists of basic information, such as included files and settings, and can be restored at a later point in time. This information is stored in a class called Project and a new object of this type is created for each new project. There were two different alternatives how to save a project to disk

1. A text file containing key values in a project, e.g., files and settings.
2. Save the whole project object using the Java interface `Serializable`.

An advantage with the first alternative is that a text file is readable to human beings, but a big drawback is the reading and writing to the file. Especially when it comes to reading there is a large overhead, because some sort of parse mechanism have to exist. It do not have to be an advanced parser but it still has to be implemented. Alternative two does not suffer from this problem. By implementing the interface `Serializable`[2] in the Project class, the reading and writing becomes very easy. Now the only thing to do is to read from and write to ordinary streams in Java. But changing the structure of a serialized class is very restricted [6], so one have to define the structure carefully.

Because of the simplicity and the lack of need to modify project structure, the second alternative was chosen. Every project file will be given the extension .prj.

To discover changes made to a certain project, a checksum is calculated every time a project is saved. The algorithm used is CRC-32, a 32 bit Cyclic Redundancy Check, and the data is a text representation of all files included in the project. When the user is about to close a project or exiting the application a new calculation is made and compared to the previous one. If the checksum differs the user will be notified and given an opportunity to save the project. If changes only are made to one of the parameter tables (see Section 4.3), a flag will inform the application about it and the user will be notified.

[2] Serializable (Java 2 Platform SE 5.0),
http://java.sun.com/j2se/1.4.2/docs/api/java/io/Serializable.html

4.2.2 Mechanisms

Behind the graphical user interface there exists several mechanisms that cooperate. Some of the these are hidden from the user to keep the usage simple and efficient.

Parse source files

The purpose with the parse mechanism is to extract information from the Fortran source files about procedure name, argument names, argument types etc. For every source file in the project that is parsed, an internal representation is created. For more information about the parser, see the report on FGG [1].

Parsing is only necessary to perform once: upon creation of a new project or when opening an existing project (for the latter case this is done automatically). The user may force OpenFGG to re-parse the source files if that is required.

Export and import to/from external format

An external format is used to make user modifications and persistent storage possible. The external format is an XML-specification which Document Type Definition (DTD) is presented in Appendix B. An XML document is easily read by both human beings and computers and that is what makes it very useful in cases like this. To process the XML files OpenFGG uses *JDOM*[3]. More about the implementation can be found in [1]. An XML-specification is created when source files are parsed and when the user saves a project. An existing file will be overwritten and it is always the properties currently visible in the GUI that is saved.

OpenFGG includes another mechanism to read from the external format into an internal representation. When a specification is imported, the content is translated and presented as a table in the GUI. From there the user may update and change properties and then save the project again. If the user makes changes outside the table to the specification, e.g., through the text editor, the specification has to be imported one more time to make the changes visible within the GUI. This is performed very easy by refreshing the parameter table (see Section A.2.3).

Generate gateway

Before a project can be compiled into a MEX-file, a gateway (file) must be generated. The gateway contains the entry point `mexFunction`, described in Section 3.1. This gateway is passed as an argument to `mex` together with the source files.

After the gateway has been constructed one M-file for each parsed subroutine is created. These are used by MATLAB when a call to a subroutine in the

[3] JDOM, http://jdom.org

MEX-file is made. For illustration, consider the following example:

A MEX-file called *my_sum* consists of two routines, *isum* and *dpcsum*. The corresponding M-files looks like this:

M-file for routine isum

```
function isum = isum(vector)
% function isum = isum(vector)
% Generated by the Fortran Gateway Generator
isum = my_sum(vector, 0);
```

M-file for routine dpcsum

```
function sum = dpcsum(vector)
% function sum = dpcsum(vector)
% Generated by the Fortran Gateway Generator
sum = my_sum(vector, 1);
```

As one can see both M-files makes a call into the same MEX-file (*my_sum*), but they are distinguished by the last argument, 0 or 1. This setup is performed automatically and the user only have to make a call to *isum* or *dpcsum* and the rest is solved be the system. There is only one MEX-file but it can contain several routines. By this procedure, the user may compile several library routines into a single *library gateway*.

Compile

From the GUI, the user can choose to compile a project. This step consists of several internal steps. First a gateway file is created, then all M-files is created and at last the call to `mex` is conducted. The call is performed using Java's `Runtime` class. This technique is used to get hold of external programs and at the same time keeping full control of the GUI. The syntax of the command string varies between platforms of UNIX type and platforms of Windows type. To solve this, the platform type is controlled before the command string is built. Below the two syntaxes is showned:

On UNIX:

```
mex -output name -outdir dir -f optionfile
    [option1 ... optionN] sourcefile1 [... sourcefileN]
    [libraryfile1 ... libraryfileN]
```

On Windows:

```
cmd /c mex -output name -outdir dir -f optionfile
    [option1 ... optionN] sourcefile1 [... sourcefileN]
    [libraryfile1 ... libraryfileN]
```

Extension	Platform
.mexaxp	Dec/Compaq Alpha
.mexhp7	HP 9000 PA-RISC
.mexrs6	IBM RS/6000
.mexsg	SGI
.mexsol	Solaris
.dll	Windows
.mexmac	Mac OS X
.mexglx	glnx86
.mexi64	glnxi64
.mexphux	HPUX

Table 4.1: Platform dependent extensions for MEX-files.

If the compilation is successful, a MEX-file is created and the system searches for a MEX-file with one of the platform dependent extensions showned in Table 4.1.

The first existing file that is found, is added to the project. Now the MEX-file can be called from MATLAB. It is possible to start MATLAB from the GUI by pressing the `Start MATLAB` button.

4.3 The Graphical User Interface

OpenFGG is built upon *Java Look and Feel Design Guidelines*, which promote flexibility and ease of use in cross-platform applications [7]. At the beginning of the project a tool called *NetBeans IDE*[4] was used for GUI development, but most work have been conducted manually.

In Figure 4.2 we show a screenshot of the GUI and some numbers representing the major components:

1. This component is a tree structure of all files in the current project. Files will be added here automatically if new files are added or created. At the bottom of the application window there is a status row that shows the absolute path to a file when the mouse pointer is moved over it.

2. Here are all parameters presented that were found by the parser. Every routine will have its own tab with the same name as the routine. Properties for each parameter is changed by editing the table. It is also possible to get a popup menu by pressing the right mouse button (see User's Guide in Appendix A).

3. In this field messages about the execution is showned. During compilation the output from the `mex` script is presented here and also output

[4]Welcome to NetBeans, http://www.netbeans.org/

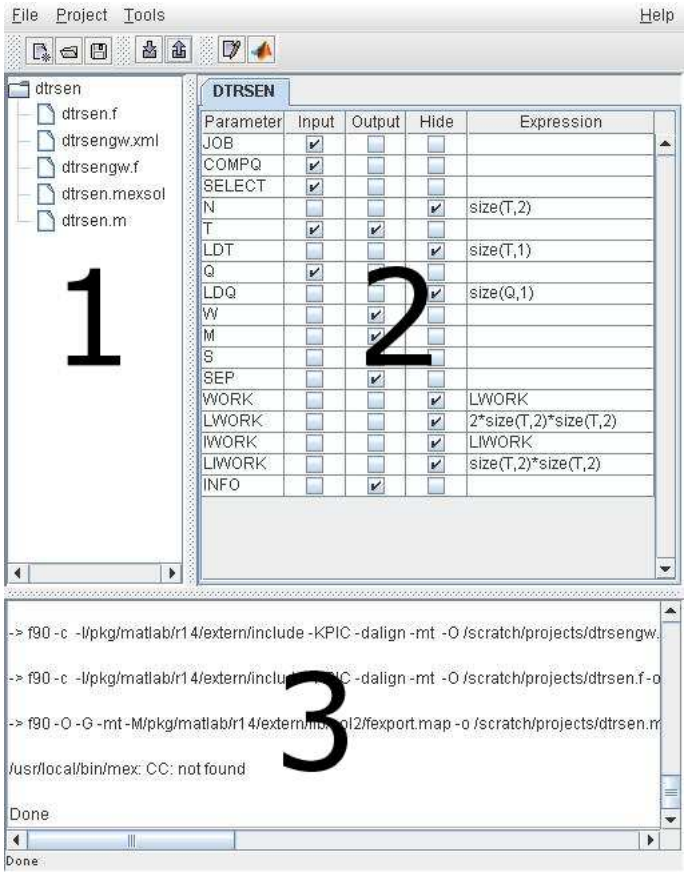

Figure 4.2: Components in the GUI.

from FGG. To fetch output from external applications the GUI uses the `InputStreamReader` and `ErrorStreamReader` in Java.

The toolbar at the top have buttons with icons and these have the following meaning (the shortcut inside parenthesis):

- Create a new project, same as menu alternative `File/New` (Ctrl-N)
- Open an existing project, same as `File/Open` (Ctrl-O)
- Save current project, same as `File/Save` (Ctrl-S)
- Parse sources, same as `Tools/Parse`
- Compile project, same as `Tools/Compile`
- Open the text editor, same as `Tools/Text editor`

–  Start MATLAB, same as `Tools/Start MATLAB`

4.4 Test results

To make sure the application works as expected, some larger test runs have been conducted with some routines in *BLAS*[5] *(Basic Linear Algebra Subprograms)* Level 3 and a routine called *DTRSEN* from *LAPACK*[6] *(Linear Algebra PACKage)*.

The tests were performed on computers with the following characteristics:

– SunOS 5.8
– f90 Fortran compiler
– MATLAB version 7.0.1 (R14)

and

– Windows XP Professional version 2002
– Compaq Visual Fortran Compiler version 6.5
– MATLAB version 7.0.1 (R14)

All tests turned out successfully.

4.4.1 BLAS - DGEMM

BLAS Level 3 performs matrix operations that are commonly used in computational linear algebra. The routines *DGEMM, DSYMM, DSYR2K, DSYRK, DTRMM, DTRSM* were compiled into a lib, i.e., one MEX-file containing several routines. The test was made by calling the routine DGEMM from MATLAB, which is illustrated in Figure 4.3.

The test was performed in MATLAB with the following values:

```
>> transa = 'N';
>> transb = 'T';
>> alpha = 1.0;
>> beta = 1.0;
>> lda = 10;
>> ldb = 10;
>> ldc = 10;
>> a = rand(10,10);
>> b = rand(10,10);
>> c = rand(10,10);
>> c = dgemm(transa,transb,alpha,a,lda,b,ldb,beta,c,ldc);
```

[5]BLAS, http://www.netlib.org/blas
[6]LAPACK, http://www.netlib.org/lapack

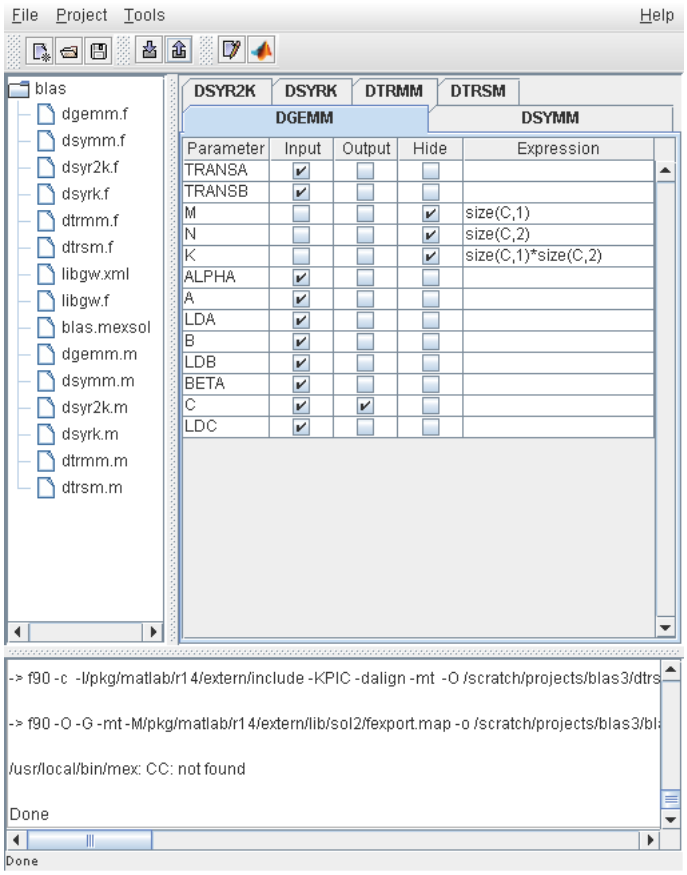

Figure 4.3: Screenshot from the test with DGEMM.

The calling sequence to the MEX-file may be simplified by using the `hide` property in OpenFGG. However, the result was compared to the right answer and it turned out to be correct. This test shows that OpenFGG can deal with large routines from BLAS and create a single MEX-file that contains many routines. It is possible to create even larger MEX-files that contains even more routines.

4.4.2 LAPACK - DTRSEN

One purpose with this test was to control the functionality of the complex data type split-complex (see Section A.2.3). In LAPACK there is a routine called *DTRSEN*. In the source code for this routine one can read that DTRSEN

reorders the real Schur factorization of a real matrix

$$A = QTQ^T \tag{4.1}$$

so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T, and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace. The selected cluster of eigenvalues is returned in two distinct arrays, one for the real part and one for the imaginary part, respectively. We want to build a gateway that uses a single array to store the output of complex eigenvalues.

Figure 4.4 shows the appearance of OpenFGG during the test. The parameter `W` represents the split-complex parameter. There is no visible difference between `W` and another parameter in the list, but `W` has been set as split-complex (described in Appendix A.2.3).

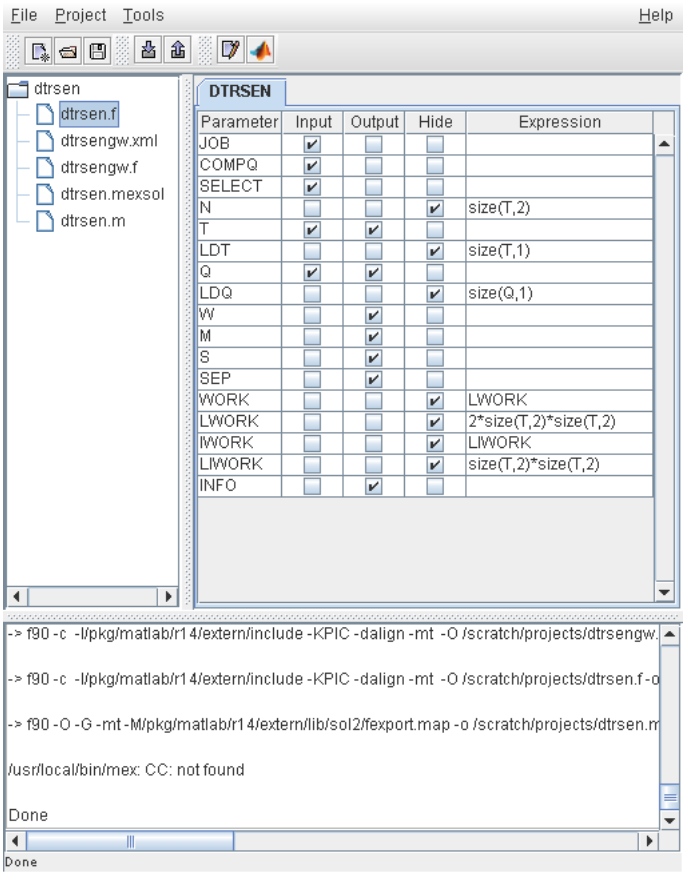

Figure 4.4: Screenshot from the test with DTRSEN.

When the MEX-file was created it was tested in MATLAB with the following

values

```
>> job = 'B';
>> compq = 'V';
>> select = [0,0,1,1];
>> t = triu(rand(4,4));
>> q = eye(4);
>> t(2,2) = t(1,1);t(2,1)=-t(1,2);t(4,4)=t(3,3);t(4,3)=-t(3,4);
>> w0 = eig(t);
>> w0

w0 =

   0.9501 + 0.8913i
   0.9501 - 0.8913i
   0.6154 + 0.1763i
   0.6154 - 0.1763i

>> [tt,qq,w,m,s,sep,info] = dtrsen(job,compq,select,t,q);
>> w

w =

   0.6154 + 0.1763i
   0.6154 - 0.1763i
   0.9501 + 0.8913i
   0.9501 - 0.8913i
```

All the parameters in the table in Figure 4.4 that had mode `hide` are not visible in the calling sequence. These parameters have been given values in the background to simplify the usage.

As one can see, the split-complex parameter `W` contains complex values which shows that the functionality in the application is working. The output has been controlled and it is correct.

Chapter 5

Restrictions

5.1 Limitations

As said in the problem description in the beginning, the main goal was to make the application work in UNIX environments. According to the results of the tests that have been made this seems to be fulfilled. Also running under Windows works as expected. Tests under Macintosh has not been conducted, but some features in OpenFGG will not work right now. The problem is the calls to external applications, such as compilation with `mex` and starting MATLAB will not work in MacOS. This means that a fully executable MEX-file cannot be created on Macintosh, but OpenFGG will be able to create gateways and M-files so there is in practice only one missing step. A Macintosh user can use OpenFGG to parse the sources, set up the parameter properties and create gateways and M-files, and then make the call to `mex` manually.

When using a UNIX platform there is a limitation regarding search paths. Right now it is not possible to use whitespaces inside a path, because the call to `mex` cannot handle it. On the other hand is it possible to generate gateways and M-files. Under Windows this problem does not exist.

OpenFGG is built to support most parts of Fortran 77[1]. It does not have support for any features from Fortran 90/95, because the parser does not recognize the keywords and will not allow them. This might be a limitation because routines may be written in mostly Fortran 77, but include just a few additions from Fortran 90/95, and this will result in incompatibility with OpenFGG. Somehow, this problem has to be addressed in the future.

[1] http://www.fortran.com/

5.2 Future work

To make OpenFGG even more useful, some extensions can be incoorporated. When Fortran 90 was released it contained some new features like pointers and recursion. To be able to support this kind of features, the Fortran source code parser has to be extended. Some extensions that are relevant to add are described below. Of course, the list can be made much longer.

5.2.1 Intent

Intent is a mechanism that tells the compiler about how variables are used within a subprogram [3]. A variable can be set to one of the following modes: `IN`, `OUT` or `INOUT`. If the compiler is aware of this information, it can provide additional checking and optimization. The modes can then be set in the source code and the user will not have to provide this information to the GUI. In releases of OpenFGG, the parser might use the `INTENT` declarations to lock the modes in the parameter tables.

5.2.2 Optional arguments

According to [3] and [2], subroutines and functions can have optional arguments which are declared with the attribute `OPTIONAL`. To know if an argument is used or not, Fortran uses the function `PRESENT(ARG)`, which returns true if the argument exists and false otherwise.

5.2.3 Pointers

Pointers are interesting because one will be able to represent a three dimensional matrix as an array of pointers to two dimensional arrays instead of todays three dimensional arrays, thus giving the potential of saving storage space in cases where the matrices in the sequences are of different sizes.

Pointers in Fortran points to a specific object. That object must be a `target`[2] or another pointer of the same type [3].

5.2.4 Recursion

If a subroutine or function is recursive this must be specified in the declaration. The attribute `RECURSIVE` is used for this purpose. Also the attribute `RESULT` can be used to specify the variable that holds the function result [3], [2]. A recursive function declaration is generically described as

```
RECURSIVE FUNCTION FOO (ARG) RESULT (RES).
```

[2]Variables with the TARGET attribute can become pointer targets.

5.2.5 Colon and double colon syntax

The parser should be able to recognize the colon and double colon notations and allow that they are used in the source code. The colon syntax means that one can write `A(1:M,1:N)` as in MATLAB. Type declaration statements in Fortran 90/95 use double colon syntax (`::`).

5.2.6 Dummy procedures

FGG lacks support for building gateways which can handle dummy procedures, i.e., input arguments that are functions themselves. It is not clear whether it can be supported in OpenFGG, but some attempts to achieve this will be done.

Chapter 6

Acknowledgements

I would like to thank Robert Granat for making this Master's Thesis available and for his guidance and support as my supervisor. I would also like thank Magnus Andersson that has been helpful with his detailed knowledge about FGG.

References

[1] Magnus Andersson. Semi-Automatic Generation of MATLAB Gateway Functions of Numerical Fortran Routines. *Master's Thesis*, Report UMNAD 561/05, Dept. Computing Science, Umeå University, 2005.

[2] Bo Einarsson. *Lärobok i Fortran 90/95*. Linköping University, first edition, 2004.

[3] Australian Partnership for Advanced Computing. Advanced Fortran Programming. Web page, 13 April 2005. `http://nf.apac.edu.au`.

[4] Robert Granat and Bo Kågström. Direct Eigenvalue Reordering in a Product of Matrices in Extended Periodic Real Schur Form. Report UMINF 05.05, Dept. Computing Science and HPC2N, Umeå University, 2005.

[5] Eva Pärt-Enander and Anders Sjöberg. *The MATLAB 6 Handbook*. Uppsala University, 2001.

[6] Inc. Sun Microsystems. Java™ Platform Standard Edition 5.0 API Specification. Web page, 15 March 2005. `http://java.sun.com/`.

[7] Sun Microsystems, Inc. *Java Look and Feel Design Guidelines*, second edition, 2001.

[8] The MathWorks, Inc. *MATLAB External Interfaces*, 2005.

[9] The Numerical Algorithms Group. *NAGWare Gateway Generator*, second edition, 1994.

Appendix A

User's Guide

The system is packed into a single JAR file, including a Main-class directive and dependency information. This file can be downloaded from OpenFGG's web page[1] which contains information about how to install and run the system.

This section explains how to use OpenFGG and which software and settings that are required.

A.1 Requirements

To run OpenFGG Java 1.5.0 or later must be installed. Java can be found on Sun Microsystems web site[2].

OpenFGG depends upon two external JAR files.

1. OpenFGG uses *JDOM*[3] to process XML files, hence the classes contained in the file `jdom.jar` must be available to the system.
2. OpenFGG uses the Jakarta Project's[4] *commons-math* to handle complex numbers which Java has no native support for. The classes contained in the file commons-math-1.0.jar must hence be available to the system.

To run the system these JAR files must either be included in the class path or, when using the the JAR version of the system, be placed in a directory called *lib* in the same directory as OpenFGG's JAR file.

A.2 Usage

This section describes how to use OpenFGG to create MEX-files. Creating a MEX-file involves several steps and these steps will be gone through and

[1] http://www.cs.umu.se/∼c00jos/openfgg
[2] Java Technology, http://java.sun.com/
[3] JDOM, http://jdom.org
[4] The Jakarta Site, http://jakarta.apache.org

illustrated with figures and examples.

When all dependent JAR files exists, OpenFGG can be started. This can be done in one the following ways:

1. The most simple way is to use the JAR as an executable. For Windows users it's possible to just double-click on the JAR file directly. To start from the command line one have specify the `-jar` option, as follows:
   `java -jar openfgg.jar`
2. One can add the JAR file to the class-path and call the main class, which is OpenFGG, as follows:
   `java -cp openfgg.jar OpenFGG`
3. If it doesn't exists a JAR file but one have all the class files, there is a third alternative. The class-path has to be updated so the class files can be found on the system and the main class has to be specified, as follows:
   `java -cp` *`classpath-definition`* `OpenFGG`

A.2.1 Get started

The GUI is shown in Figure A.1. To the left there is a tree structure that represents the files in the project. The main table contains a representation of the XML specification file. All parameters are listed together with their modes and possible expressions. Every subroutine in the source gets its own tab.

A.2.2 Create a project

To build a MEX-file, one first have to create a project and specify files and options. This is done by selecting `New` from the `File` menu or by using the toolbar button. A new dialog window with two tabs will now appear, see Figure A.2.

Files used in projects must not reside in the same directory as the project file. The absolute path is used for each file separately. Here is a description of the different fields in the dialog window

- *Source files to parse:* All Fortran source files that should be parsed and included in the gateway have to be added to this list. Files are added to the list by using the `Add` button. To remove files, the button `Remove` is used.
- *Source files NOT to parse:* The files in this list is only used at compile time. They will be sent as arguments to `mex` as ordinary source files. Used when ordinary sources uses outside routines. This field is optional.
- *XML specification file:* This file is an XML description of the parameters in the Fortran source files. By specifying this file it means that one already have a specification file. If this field is empty a specification file will be created automatically. The document type definition for the XML file can be found in Appendix B.

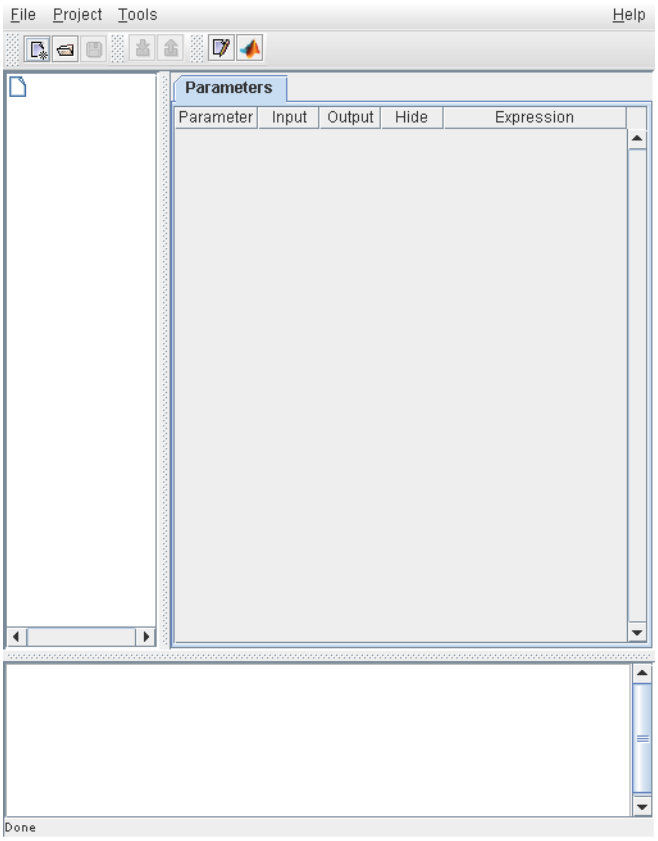

Figure A.1: Screenshot of OpenFGG.

– *MEX option file:* It is possible to specify an option file to `mex`, but if this part is left empty the default option file will be used. The default file can be set by running

```
mex -setup
```

– *Libraries:* If the Fortran subroutines are dependent on certain libraries, the absolute paths can be specified here separated by a whitespace. If paths contains whitepaces, double quotes have to be used around them to make the calling syntax to `mex` accepted. Libraries are useful for users with newer MATLAB versions that do not support direct calls to LAPACK or BLAS.

– *Name for MEX-file:* This field specifies the name of the resulting MEX-file. This field may be left empty. If the field is empty a default name will set by the application. If only one routine is used in the project the

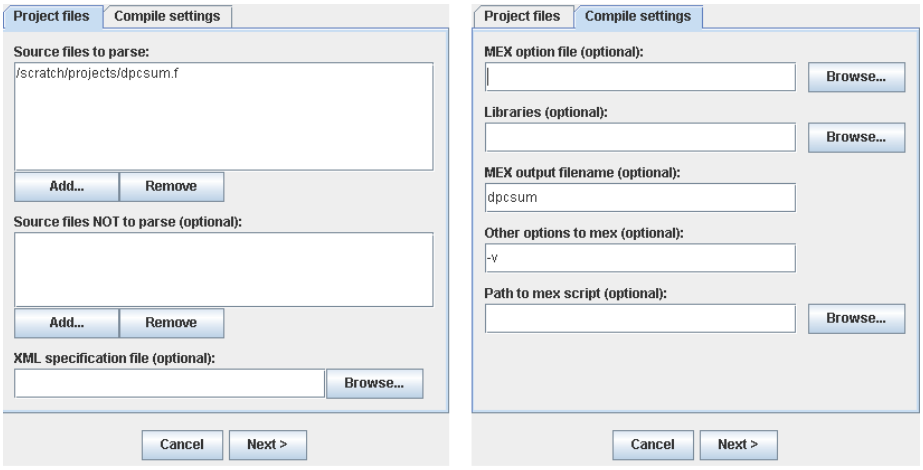

Figure A.2: Dialog window for project properties.

name of the MEX-file will be set to the name of that routine. If multiple subroutines are used the default name `libgw` is used.

– *Other options to mex:* Options in this field will be sent to `mex`, use `mex -help` to list available options, e.g. `-v` (verbose) or `-g` (debug).

– *Path to mex script:* If one want to give the absolute path to the `mex` script it is done here. This may be suitable when having trouble with the system search path or want to use a specific version of the script.

When this step is finished use the button `Next` to proceed. Now the user must specify the root directory of the project and a name of the project, see Figure A.3.

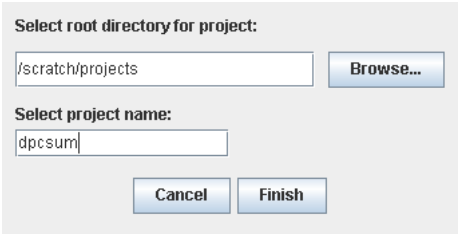

Figure A.3: Dialog window for selecting project root directory and name.

– *Select root directory for project:* In this folder the project file (.prj) and all output files from the application will be stored.

– *Select project name:* The name of the project.

When pressing `Finish` a new project is created and it is possible to start building a MEX-file.

A.2.3 Change parameter properties

OpenFGG allows the user to make changes to the XML specification through the GUI. This is done by editing the parameter table, marked as component 2 in Figure 4.2. It is also possible to make changes directly to the file. If changes are made directly to the file, one has to reload the content in the table by right-click on it and by choosing `Refresh`.

Change parameter mode and dimension expressions

The default value for the parameter mode is `input`. This can easily be changed by clicking the checkboxes. With support for intent this would be simplified, because then the mode is set in the source code and the GUI would not have to be updated (unless you want to do something else, like hiding a variable). One of the following combinations can be set

```
input OR output OR (input AND output) OR hide
```

If both `input AND output` is set this will be treated as `inout` in the source. The mode `hide` removes a parameter from the calling sequence. It corresponds to the term 'workspace' (see Appendix C) in Fortran 77. The value of a hidden parameter must initialized in the corresponding expression field. If the parameter is a variable, the expression will determine the value that the parameter should hold during the execution (seen as an <`init`> element in the XML-specification), but if it is an array, the expression will determine the size of the last dimension (seen as an <`dim`> element in the XML-specification). The value can consist of a whole expression, see Figure 4.4. Examples from the figure is that N will be given the size of the second dimension of T and the size of the last dimension of WORK will be set to the value of LWORK. For further details about the size-operator, we refer to the report of FGG [1].

When an array is declared as A(LDA,*)[5] the message `<DIM REQUIRED>` will be displayed in the expression field. This means that an expression for the last dimension of the array has to be set before a MEX-file can be created. After an expression has been set, the field will be cleared.

Proper parameter modes can often be found in the source code documentation of the actual routine.

Set up split-complex parameters

A split-complex parameter is used to represent a complex value and it contains two elements. The first one represents the real value and the other represents the

[5] A declaration of an array where the last dimension is left empty.

imaginary value. Figure A.4 describes how to manage split-complex parameters in OpenFGG. These windows are made visible by a right-click on the parameter table and choosing `Split-complex`.

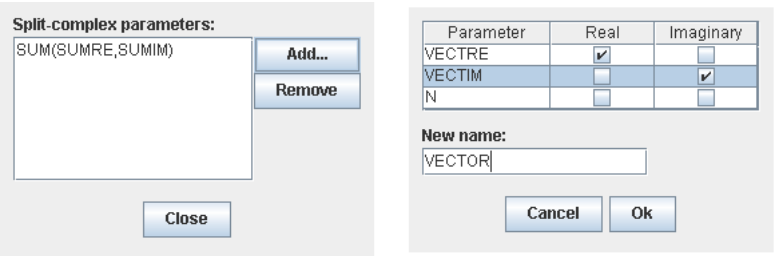

Figure A.4: Set split-complex parameters.

The right window in Figure A.4 will appear when the `Add` button is pressed. By selecting the real and imaginary part and specify a new name, a new split-complex parameter is created. The new name has to unique for the subroutine. All split-complex parameters will be presented on the form `NAME(REAL,IMAGINARY)`.

A.2.4 Compile

In Figure A.5 some new files have been added to the tree structure. These files were automatically created during the compilation. The files created are

- *A gateway:* Before `mex` is called a gateway is created by the application, based on the parameter properties set by the user. This file is a Fortran file with the same name as the subroutine in the source, but also the letters `gw` appended to it. An exception is made when there is multiple subroutines in the source, then the gateway will be given the name `libgw`.
- *M-file(s):* One M-file is created for every subroutine in the project. The M-file describes the call syntax to the MEX-file and makes the call if the corresponding MEX-file has a different name. If an M-file and a MEX-file have the same name the MEX-file has higher priority and will be called by MATLAB.
- *A MEX-file:* This is the resulting MEX-file. Optionally, the name of the file can be specified, see Section A.2.2. The filename extension depends on the platform, see Table 4.1.

Output from the `mex` script is displayed in the textarea at the bottom of the application. Upon a successful compilation the message `Done` is displayed. This means that everything went as it should and all files were created. If compilation fails the message will be `Failed` instead. A good thing to do if this happens is to add the option `-v` under the menu alternative `Project/Setup`, which will print each compile step and facilitate debugging, and compile again.

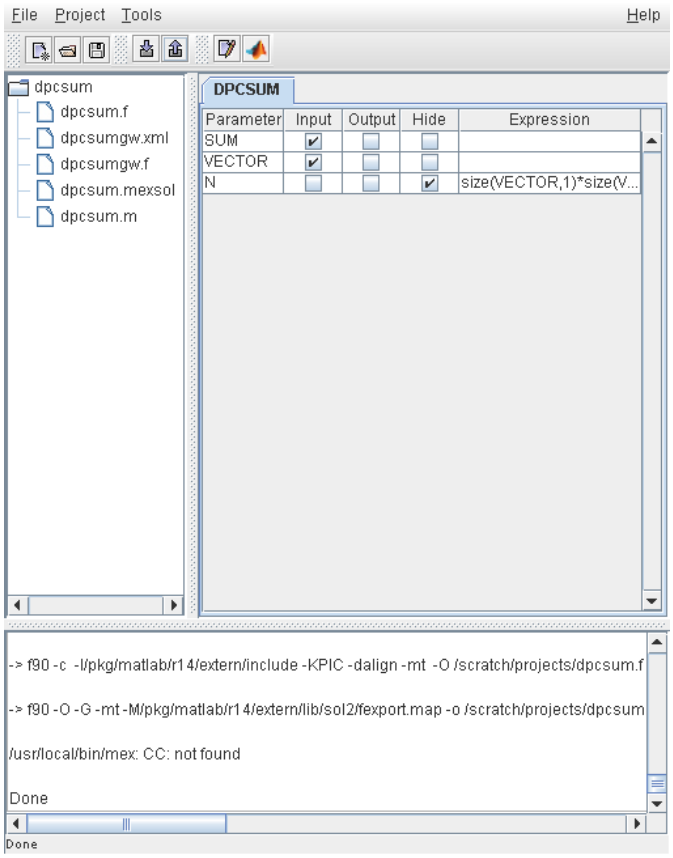

Figure A.5: Screenshot of OpenFGG after compilation.

A.2.5 Change project properties

Changes to the project can be made under the menu alternative `Project/Setup`. The dialog window is the same as setup for a new project, see Section A.2.2 for details.

The tree structure to the left allows the user to remove files from the project by right-click on a specific file. When the mouse pointer is moved over one of the files its absolute path is shown in the status field at the bottom of the application window.

A.2.6 The text editor

From the `Tools` menu one can start the text editor, see Figure A.6. By a double-click or a right-click on a file in the file-tree, the file can be opened in the text editor directly. The text editor also offers some basic functionalities such as `undo/redo` actions and `copy/cut/paste` text.

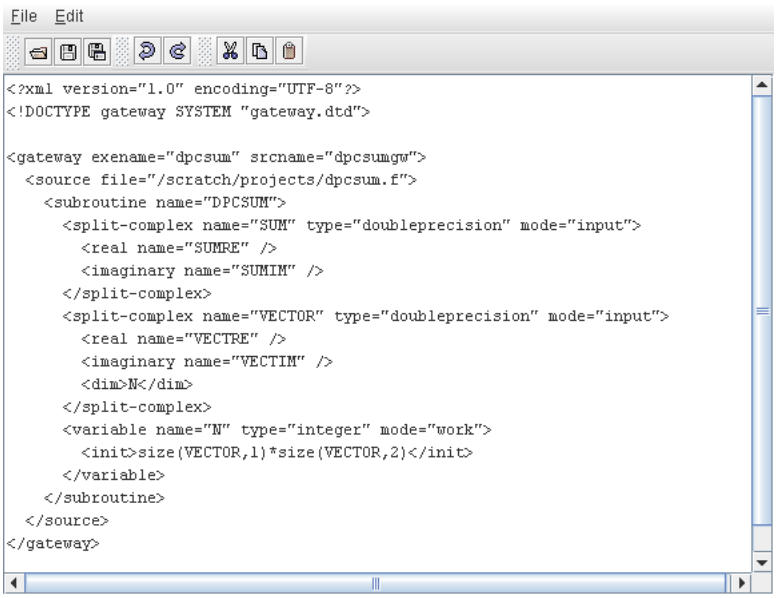

Figure A.6: Screenshot of the built-in text editor.
In the figure one can also see a typical XML-specification.

Appendix B

XML Document Format

This is how the document type definition (DTD) is formed. It was defined during the work of Magnus A and it is directly taken from [1].

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!ENTITY % mode "input|output|inout|work">
<!ENTITY % ncntype
    "INTEGER|REAL|DOUBLEPRECISION|integer|real|doubleprecision"
>
<!ENTITY % cntype
    "COMPLEX|DOUBLECOMPLEX|complex|doublecomplex"
>
<!ENTITY % ntype "%ncntype;|%cntype;">
<!ENTITY % nchtype "%ntype;|LOGICAL|logical">
<!ENTITY % type "%nchtype;|CHARACTER|character">

<!ELEMENT gateway (options?, source+)>
<!ATTLIST gateway
    exename NMTOKEN #REQUIRED
    srcname NMTOKEN #REQUIRED
>

<!ELEMENT options (xerbla?)>

<!ELEMENT xerbla EMPTY>

<!ELEMENT source (function|subroutine)+>
<!ATTLIST source
    file CDATA #REQUIRED
>
```

```
<!ELEMENT subroutine
    (description?, (variable|array|split-complex|returnspec)*)
>
<!ATTLIST subroutine
    name CDATA #REQUIRED
>

<!ELEMENT function
    (description?, (variable|array|split-complex)*)
>
<!-- len and bound attributes are only used when type is CHARACTER.
If left out 1 is implied.
-->
<!ATTLIST function
    name CDATA #REQUIRED
    type (%type;) #REQUIRED
    len CDATA #IMPLIED
    bound CDATA #IMPLIED
    mode (output|work) #REQUIRED
>

<!ELEMENT description (#PCDATA)>

<!ELEMENT returnspec EMPTY>

<!ELEMENT variable (init?)>
<!-- len and bound attributes are only used when type is CHARACTER.
If left out 1 is implied.
-->
<!ATTLIST variable
    type (%type;) #REQUIRED
    len CDATA #IMPLIED
    bound CDATA #IMPLIED
    name CDATA #REQUIRED
    matlabname CDATA #IMPLIED
    mode (%mode;) #REQUIRED
>

<!ELEMENT init (#PCDATA)>

<!ELEMENT array (dim+)>
<!ATTLIST array
    type (%nchtype;) #REQUIRED
    name CDATA #REQUIRED
    mode (%mode;) #REQUIRED
>
```

```
<!ELEMENT dim (#PCDATA)>

<!ELEMENT split-complex (((real,imaginary)|(imaginary,real)),
                         (dim*))>
<!ATTLIST split-complex
    type (%ncntype;) #REQUIRED
    name CDATA #IMPLIED
    mode (%mode;) #REQUIRED
>

<!ELEMENT real EMPTY>
<!ATTLIST real
    name CDATA #REQUIRED
>

<!ELEMENT imaginary EMPTY>
<!ATTLIST imaginary
    name CDATA #REQUIRED
>
```

Appendix C

Glossary

Argument
Element used to pass data between two subprograms or between MATLAB and a gateway. Synonym for **Parameter**.

BLAS
Basic Linear Algebra Subprograms. Routines for performing basic vector and matrix operations.

CRC-32
32 bit Cyclical Redundancy Check, used to check the integrity of information.

FGG
Fortran Gateway Generator

Gateway
A routine that makes communication between MATLAB and Fortran possible.

Inout parameter
Parameter used to pass data from MATLAB into a MEX-file and also pass a return value back to MATLAB.

Input parameter
Parameter used to pass data from MATLAB into a MEX-file, no return value is set.

JavaCC
A parser generator that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar.

JDOM
A Java representation of an XML document.

LAPACK
Linear Algebra PACKage, provides routines for solving linear systems of equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems, etc..

M-file
A MATLAB script used to call MEX-files.

MEX-file
A MATLAB executable that contains the Fortran routine and the gateway entry point, `mexFunction`.

Mode
A classification describing whether a parameter is used for input, output, both input and output or Fortran 'workspace'.

NGG
NAGWare Gateway Generator

Output parameter
Parameter used to pass output data from a MEX-file into MATLAB, the return value.

Parameter
Element used to pass data between two subprograms or between MATLAB and a gateway. Synonym for **Argument**.

Workspace parameter
Because all memory in Fortran 77 is statically allocated there is sometimes a need for some extra workspace. A workspace parameter is just allocated to achieve this.

XML-specification
An XML file that contains information about name, type and mode for subroutines and parameters.