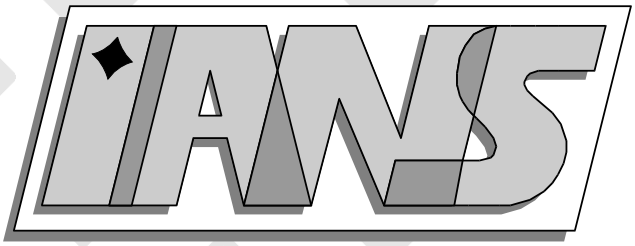
Universität Stuttgart



How to Access Matlab from Java

Andreas Klimke

Berichte aus dem Institut für Angewandte Analysis und Numerische Simulation

Preprint 2003/005

Universität Stuttgart

How to Access Matlab from Java

Andreas Klimke

Berichte aus dem Institut für Angewandte Analysis und Numerische Simulation

Preprint 2003/005

Institut für Angewandte Analysis und Numerische Simulation (IANS)
Fakultät Mathematik und Physik
Fachbereich Mathematik
Pfaffenwaldring 57
D-70 569 Stuttgart

E-Mail: ians-preprints@mathematik.uni-stuttgart.de
WWW: http://preprints.ians.uni-stuttgart.de

ISSN **1611-4176**

Abstract

Combining the strengths of the Java™ programming language and The Mathworks' MATLAB™ offers interesting new possibilities for application development. Java and its J2EE™ development environment are well established today as to provide solutions for dynamic Web services, large-scale distributed systems, and other network-centric applications. Matlab is a technical computing environment with a high-level programming language. Due to its ease of use and its strong graphics capabilities, it has become popular across many engineering disciplines.

Matlab includes an interface to Java. It allows Matlab to access Java (you may call Java classes from Matlab), but not vice versa. This introductory article presents two alternative solutions on how to interface Matlab with Java, one based on the Java Runtime class, the other one based on the Java Native Interface and Matlab's C Engine library. With the proposed strategies, a Java-based system can easily start a new Matlab session and communicate with it, thus employ Matlab as the computational engine. Advantages and disadvantages of each approach are presented.

Keywords: Matlab, Java, JNI, interface

1 Introduction

With Matlab, carefully designed numerical algorithms (Matlab is based on the LAPACK library [1]) are accessible through an intuitive user interface and an easy to learn, "4th generation" programming language. Both commercial organizations and educational institutions use Matlab to develop new algorithms, or perform graphical evaluation of data sets, to name just two of its many applications. Matlab is an open system, i.e. users can add functionality by providing their own routines, and it is available for all common operating systems. Routines and functions written in Matlab, the so-called "m-files", are transferable among different operating systems running Matlab.

Sun's Java 2 platform has become today's most versatile environment for distributed systems due to its emphasis on operating system independence (and therefore suitability for heterogeneous networks), reusability, and modularity [9].

The combination of the two systems can open up interesting new possibilities in application development. For example, one could use Java to provide a platform-independent framework (user interface, network integration, etc), while employing Matlab as the computational engine. Significant efforts have been made already to facilitate this approach. Recent releases of Matlab include a Java Virtual Machine to provide a graphical user interface, for instance. Furthermore, it is possible to access Java classes from Matlab functions. Unfortunately, the Matlab manual and other resources do not provide information on how to access Matlab from Java code. We would like to close this remaining gap with this paper.

Let us briefly review the available information regarding external interfaces (as of Matlab version 6.5 [7]):

- Matlab can interface with Fortran or C programs through a so-called C or Fortran Engine.

- DDE and COM objects are supported, which could be used to communicate with Java. However, these solutions are available under Windows™ only.
- Matlab can access Java programs [7, 5]. But there is no documentation available describing how to provide the link in the other direction (i.e. to "access Matlab from Java", with the Java application running as a separate process).

In the forthcoming sections, we present two ways of calling Matlab from Java. The Java program could be a regular class, but also a Servlet, a JSP™ page, or any other special form of a Java class. The examples are intentionally kept rather simple. But in general, all of Matlab's functionality is available.

2 Approach 1: Communication via the Java Runtime Class

This approach uses the Java Runtime class to start a new Matlab process. Communication is achieved through acquiring Matlab's standard input and output streams.

2.1 Implementation

In the following, we will give a simple main program calling the Matlab engine, and then review the construction of the underlying package providing the required functionality.

Code Segment 1 Main program, approach 1.

```
import MatlabRuntimeInterface.*;

public class Main {
  public static void main(String[] args) {
    Engine engine = new MatlabRuntimeInterface.Engine();
    try {
      engine.open("matlab -nosplash -nojvm");
      System.out.println(engine.getOutputString(500));
      engine.evalString("A = gallery('lehmer',10);");
      engine.evalString("f = ones(10,1);");
      engine.evalString("pcg(A,f,1e-5)");
      System.out.println(engine.getOutputString(500));
      engine.close();
    }
    catch (Exception e) {
    }
  }
}
```

Let us summarize code segment 1. We would like to import a package that contains the classes providing the communication capabilities (called `'MatlabRuntimeInterface'`). This will enable us to start a new Matlab process, send commands to this process, and receive

results back, which is illustrated by a simple example of solving a linear equation system using Matlab's Preconditioned Conjugate Gradients method.

Let us now have a look at the implementation of the `MatlabRuntimeInterface` package. It will consist of a single class called `'Engine'`. First of all, we need to declare a couple of private variables (see code segment 2). `p` is the handle to the Matlab process. `in`, `out`, and `err` will be used to get hold of Matlab's standard input/output streams. The remaining declarations allow us to store Matlab's output into a buffer of characters later on. We initialize the buffer in the constructor of the class.

Code Segment 2 Declaration of member variables, definition of constructor.

```
package MatlabRuntimeInterface;

import java.lang.Runtime;
import java.io.*;

public class Engine {
  private Process p;
  private BufferedReader in;
  private BufferedWriter out;
  private BufferedReader err;
  private char[] outputBuffer;
  private static final int DEFAULT_SKIP = 65536;
  private static final int DEFAULT_BUFFERSIZE = 65536;
  private int totalCharsRead;

  public Engine() {
    outputBuffer = new char[DEFAULT_BUFFERSIZE];
  }
}
```

Code segment 3 shows how to open the Matlab session. This is the first method of our class `Engine` that will provide the functions to access Matlab.

The public method `open` may be called from a main program. The argument `startcmd` that is passed should contain a valid start string, such as `'matlab'`. Under Unix-based systems, other possibilities for a start string would be `'matlab -nojvm -nosplash'` to suppress the Matlab splash screen and the startup of Matlab's Java Virtual Machine, or `'ssh hostname /bin/csh -c 'setenv DISPLAY <hostname>:0; matlab'` to run the Matlab process on another machine through a secure shell. The method itself does nothing but invoking the Matlab process by calling the `'exec'` method, and obtaining the input/output streams. Since Matlab will return some "welcome messages" when initially opened, we wait for those by calling the `receive()` method (see code segment 5).

Synchronizing access to Matlab's input and output streams seems to be a good idea to avoid possible conflicts. Therefore, we obtain a lock on the engine instance prior to each access by an enclosing `synchronize(this)` statement.

Code Segment 3 `open()` method.

```
public void open(String startcmd) throws IOException {
  try {
    synchronized(this) {
      p = Runtime.getRuntime().exec(startcmd);

      out = new BufferedWriter(new OutputStreamWriter
                               (p.getOutputStream()));
      in  = new BufferedReader(new InputStreamReader
                               (p.getInputStream()));
      err = new BufferedReader(new InputStreamReader
                               (p.getErrorStream()));
    }
    // Wait for the Matlab process to respond.
    receive();
  }
  catch(IOException e) {
    System.err.println("Matlab could not be opened.");
    throw(e);
  }
}
```

The next methods we provide are to send and receive information (see code segments 4 and 5). The input/output pipes of a runtime process are binary, thus, a conversion of the character-based information to a stream of bytes becomes necessary, and vice versa. Most conveniently, Java's `OutputStreamWriter` and `OutputStreamReader` classes automatically take care of the conversion by using the default character encoding of the operating system.

The method `evalString` sends a string to the Matlab process for evaluation, and waits for the Matlab process to respond.

```
public void evalString(String str) throws IOException {
  send(str);
  receive();
}
```

When sending information, we should terminate the string with the newline character sequence and flush the output buffer, so Matlab will execute the command.

To retrieve information, we wait until characters are available (indicated by `in.ready()`), and then continuously read from the input stream until all available data is read into the output buffer. We need to keep track of the number of characters retrieved to avoid a buffer overrun. Additional characters are dropped by calling the `skip()` method.

Finally, we present the `getOutputString` method (see code segment 6). We read a specified number of characters from the output buffer and return the result as a string. This enables the main program to access the results obtained from Matlab.

Code Segment 4 `send()` method.

```
private void send(String str) throws IOException {
  try {
    str+="\n";
    synchronized(this) {
      out.write(str, 0, str.length());
      out.flush();
    }
  }
  catch(IOException e) {
    System.err.println("IOException occured while sending data to the"
                       +" Matlab process.");
    throw(e);
  }
}
```

Our core communication class `Engine` is now complete. We can run any set of Matlab commands and retrieve the results. A drawback, however, is the fact that we will eventually have to parse the output stream to interpret the results instead of just displaying them. This would become necessary if we would like to further process the results in Java, for example. To facilitate this, one could write a small Matlab method that would format the contents of a variable in Matlab in a way that is easier to parse, and possibly more efficient in transfer size (i.e. an encoding capable of transferring more than a single digit of a number for each character).

2.2 Advantages and Disadvantages

Let's look at the advantages and disadvantages of this approach:

Advantages:

- Matlab can be started directly from a Java class, either on the same machine, or, under Unix-based systems, even on another machine.
- Platform-independent.
- Very little Java code required.
- Will most likely work with future versions of Matlab.
- No additional installation procedures or system setup changes required.

Disadvantages:

- Transferring of data is character- and stream-based, and therefore inefficient, especially for larger amounts of data (high latency, low transfer rates compared to direct memory access).
- Parsing of the Matlab output stream has to be done manually.

Code Segment 5 `receive()` method.

```
private void receive() throws IOException {

  int charsRead = 0;
  int numberToRead;

  totalCharsRead = 0;
  // System.err.println("Receiving...");
  try {
    synchronized(this) {
      while (totalCharsRead == 0) {
        while (in.ready()) {
          if ((numberToRead = DEFAULT_BUFFERSIZE-totalCharsRead) > 0) {
            charsRead = in.read(outputBuffer,totalCharsRead,numberToRead);
            if (charsRead > 0) {
              totalCharsRead += charsRead;
            }
          }
          else {
            skip();
            return;
          }
        }
      }
    }
  }
  catch(IOException e) {
    System.err.println("IOException occured while receiving data from"
                       +" the Matlab process.");
    throw(e);
  }
}
```

Code Segment 6 `getOutputString()` method.

```
public String getOutputString (int numberOfChars) {
  if (totalCharsRead < numberOfChars) numberOfChars = totalCharsRead;

  char[] result = new char[numberOfChars];
  System.arraycopy(outputBuffer, 0, result, 0, numberOfChars);
  return new String(result);
}
```

3 Approach 2: Using a JNI Wrapper for Matlab's C Engine

With Java's Native Interface (JNI) [2, 3, 4, 10], we can write a wrapper class for The Mathworks' C Engine library [7]. While this is somewhat cumbersome to do, we can take advantage of all the functionality that this library provides for its use with C programs.

Building a Java application that contains native libraries is more difficult than compiling a pure Java application. Therefore, we describe this process in more detail.

3.1 Implementation

We would like to keep the main program the same – the complexity of the native interface does not need to bother the application programmer. Let us just change the package name to account for the use of native methods with the implementation of Approach 2. Thus, we change two lines compared to the main program of Approach 1, the `import` statement and the creation of the `Engine` instance.

```
import MatlabNativeInterface.*;
Engine engine = new MatlabNativeInterface.Engine();
```

Let us now take a look at the implementation of the `Engine` class of the `MatlabNativeInterface` package (code segment 7). It basically contains only the function headers of the native wrapper library (which is written in C).

Code Segment 7 Native function declarations of class `Engine`.

```
package MatlabNativeInterface;

import java.io.*;

public class Engine {
  public native void open(String startcmd) throws IOException;
  public native void close();
  public native void evalString(String str);
  public native String getOutputString(int numberOfChars);

  static {
    System.loadLibrary("engineJavaMatlab");
  }
}
```

As with Approach 1, we provide the same four functions `open`, `close`, `evalString`, and `getOutputString`. The function header of the `open` method indicates that we have provided error handling for this function. This would apply to the other functions, too. However, we did not include this here, since it is repetitive and can be easily added by the reader. A good resource regarding error handling with JNI can be found in [3].

The `System.loadLibrary` statement is an essential part of each class containing native functions. Here, we can tell the Java Runtime Environment which library has to be loaded at

runtime that contains the declared native methods. You might wonder how the JRE knows about the path where this library can be found. Under Unix-based systems, you can specify the path with the `LD_LIBRARY_PATH` system variable. The JRE also looks in default system directories.

Let us now look into the wrapper class written in C, `NativeEngine.c`. We start with the include statements and variable definitions (see code segment 8). The first include statement `#include <jni.h>` provides the header file for the Java Native Interface (this header file is part of any standard Java distribution). Furthermore, we include `MatlabNativeInterface_Engine.h`. This header file can be generated automatically from the `NativeEngine.class` file with the `javah` utility (see Section 3.2). Finally, we need `engine.h` (this file is included in a standard Matlab distribution) that declares the Matlab C engine functions. The remaining lines are some global variable definitions, notably a pointer `ep` to store the handle of the Matlab process.

Code Segment 8 Include statements and variable definitions.

```
#include <jni.h>
#include "MatlabNativeInterface_Engine.h"
#include <stdio.h>
#include "engine.h"

#define DEFAULT_BUFFERSIZE = 65536
Engine* ep;
char outputBuffer[DEFAULT_BUFFERSIZE];
```

The method `Java_MatlabNativeInterface_Engine_open` takes care of opening a Matlab session (see code segment 9). The long function name corresponds to the declaration in the automatically generated header file `MatlabNativeInterface_Engine.h`. It includes the package name and class name to avoid naming conflicts with other methods. The interface pointer `env` and the pointer to this object `obj` are passed with any JNI function (they are required to make JNI Interface functions available from the native subroutines [10]). The `startcmd` argument is a Java string object, as indicated by the variable type `jstring`.

Let us go through the function body. The C Engine function `engOpen` that we would like to call to open a Matlab session takes a C-style string as input argument (i.e. an array of characters terminated by zero). We can use the JNI function `GetStringUTFChars` to perform the necessary conversion of our Java string object. We have to remember to free the allocated memory after use with the `ReleaseStringUTFChars` method to avoid memory leaks (C does not provide garbage collection!). Next, we try to open a Matlab session with the converted start command `c_string`. If successful, `engOpen` returns a handle to this Matlab session. We store this pointer to `ep` for later use. Otherwise, we throw an `IOException` that can be processed by the calling Java routine. Finally, we call the Matlab C Engine method `engOutputBuffer` to store the Matlab output stream data to our own buffer (by default, the Matlab C Engine discards all generated output, so we need to do this to receive and process results).

Let us proceed with the native wrapper methods for sending commands to Matlab and for receiving results back (see code segment 10). These methods are a bit shorter since we have omitted the error handling, which could be similar to the one of the `engOpen` method from

Code Segment 9 Native `open()` method.

```
JNIEXPORT void JNICALL
Java_MatlabNativeInterface_Engine_open(JNIEnv *env, jobject obj,
    const jstring startcmd) {
  const char *c_string;
  c_string = (*env)->GetStringUTFChars(env, startcmd, 0);
  if (!(ep = engOpen(c_string))) {
    jclass exception;
    (*env)->ReleaseStringUTFChars(env, startcmd, c_string);
    exception = (*env)->FindClass(env, "java/io/IOException");
    if (exception == 0) return;
    (*env)->ThrowNew(env, exception, "Opening Matlab failed.");
    return;
  }
  (*env)->ReleaseStringUTFChars(env, startcmd, c_string);
  /* indicate output should not be discarded but stored in outputBuffer */
  engOutputBuffer(ep, outputBuffer, DEFAULT_BUFFERSIZE);
}
```

Code Segment 10 Native `evalString()` and `getOutputString()` methods.

```
JNIEXPORT void JNICALL
Java_MatlabNativeInterface_Engine_evalString
    (JNIEnv *env, jobject obj, const jstring j_string) {
  const char *c_string;
  c_string = (*env)->GetStringUTFChars(env, j_string, 0);
  engEvalString(ep, c_string);
  (*env)->ReleaseStringUTFChars(env, j_string, c_string);
}

JNIEXPORT jstring JNICALL
Java_MatlabNativeInterface_Engine_getOutputString
    (JNIEnv *env, jobject obj, jint numberOfChars) {
  char *c_string;
  jstring j_string;
  if (numberOfChars > DEFAULT_BUFFERSIZE) {
    numberOfChars = DEFAULT_BUFFERSIZE;
  }
  c_string = (char *) malloc ( sizeof(char)*(numberOfChars+1) );
  c_string[numberOfChars] = 0;
  strncpy(c_string, outputBuffer, numberOfChars);
  j_string = (*env)->NewStringUTF(env, c_string);
  free(c_string);
  return j_string;
}
```

above. As we did before, when transferring strings from Java to Matlab and vice versa, we have to take care of the proper string conversion to C format.

Sending strings to Matlab is simple via the Matlab C Engine library. The method we need to access in the library is called `engEvalString`. This method takes care of the I/O operation, i.e. transfers the data to the Matlab session. To do this, the method needs a handle to the session that we stored in `ep` when opening Matlab with `engOpen`.

Returning the results to the Java program is easy, too. We do not even need to call a function of the Matlab C Engine to do this, since we have instructed the Engine to store results in our buffer `outputBuffer` when opening the Matlab session. So we just need to copy the specified number of characters from our allocated output buffer to a string that we pass back to the calling Java routine.

3.2 Compilation

As mentioned earlier, compiling and linking a Java application containing native libraries requires more work than translating a pure Java application to byte code. A detailed explanation on how to do this can be found in [3], for instance. Here's a brief summary:

1. Translate the Java source files to byte code (class files) with the `javac` compiler.
2. Generate the native library header file (here: `NativeEngine.h`) with the `javah` utility.
3. Compile and link the native library with a C compiler (e.g. the GNU Compiler Collection `gcc` under Linux) as shared library.

For your convenience, Appendix A provides a download link where you may obtain the complete source code including a Makefile for Linux. Before executing Approach 2, make sure you provide a valid `LD_LIBRARY_PATH` variable that points to the directories containing the shared libraries.

3.3 Advantages and Disadvantages

Advantages:

- Matlab can be started directly from a Java class, either on the same machine, or, under Unix-based systems, even on another machine.
- The Matlab C Engine library functions are accessible. Special functions to retrieve arrays or send arrays to Matlab are already implemented and can be easily used with little additional programming effort.

Disadvantages:

- Data transfer is character- and stream-based, therefore inefficient especially for larger amounts of data (high latency, low transfer rates compared to direct memory access).
- Platform-dependent. Different compilation, installation and setup routines apply to different operating systems.
- Will most likely not work with future versions of Matlab without small adjustments and re-compilation.
- Cumbersome implementation. Hard to debug because of native code parts.

4 Related Work

JMatLink [8] is a free library that provides an implementation similar to Approach 2. JMatLink uses a multi-threading approach to improve performance and handle multiple Matlab sessions at a time. However, there is no clear separation of the JNI library and additional Java-specific functionality. Error handling is not provided.

5 Conclusions

It has been shown that both approaches are capable of interfacing Matlab with Java. While the first approach has its main advantage in being pure Java code, the second approach can make use of the C Engine library with its additional functionality. However, Apporach 2 is cumbersome to install and platform-dependent. With both strategies, data is transferred via character streams. This is not very fast, but there is no other way but to follow this procedure. For most applications, however, the amount of input and output data can be kept rather small, since communication is often required only at the beginning and at the end of a computation.

A Obtaining the source code

The complete source code for both approaches described in this paper is available for download at `http://matlabdb.mathematik.uni-stuttgart.de/search_result.jsp?Search=Java&SearchCategory=All` [6]. Under Linux, the provided Makefiles should help you to compile the code quickly. The source code is arranged as illustrated in Figure 1.

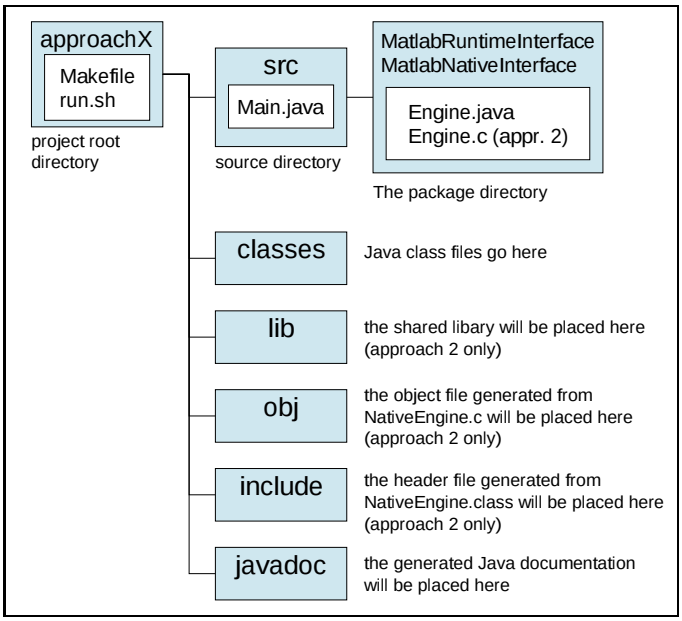


Figure 1: Directory structure of provided source code.

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1999.

[2] C. Austin and M. Pawlan. *Advanced Programming for the Java 2 Platform*, chapter 5: Java Native Interface (JNI) Technology. Addison-Wesley, 2000.

[3] M. Campione, A. Huml, K. Walrath, and T. Team. *The Java Tutorial Continued: The Rest of the JDK*, chapter Java Native Interface. Addison-Wesley, 1998.

[4] D. M. Epp. Interfacing Java with C in Linux. *CScene (online publication)*, 1998. Available from World Wide Web: `http://www.gmonline.demon.co.uk/cscene/CS4/CS4-04.html` [cited April 10, 2003].

[5] D. Hanselman and B. Littlefield. *Mastering MATLAB 6: A Comprehensive Tutorial and Reference*, chapter 35: Extending Matlab with Java. Prenctice Hall, 2001.

[6] Institute of Applied Analysis and Numerical Simulation, University of Stuttgart. Scientific/Educational Matlab Database [online]. 2002 [cited April 10, 2003]. Available from World Wide Web: `http://matlabdb.mathematik.uni-stuttgart.de`.

[7] The Mathworks, Inc. *MATLAB Documentation, External Interfaces/API*, 1994-2003. Available from World Wide Web: `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.shtml` [cited April 10, 2003].

[8] S. Müller. JMatLink [online]. 2001 [cited April 10, 2003]. Available from World Wide Web: `http://www.held-mueller.de/JMatLink/`.

[9] A. Nakhimovsky, T. Myers, M. Wilcox, S. Zeiger, J. Diamond, J. Griffin, K. Avedal, M. Holden, S. Tyagi, G. Damme, S. Allamaraju, A. Longshaw, D. O'Connor, R. Browett, R. Johnson, T. Karsjens, L. Kim, G. Huizen, and A. Hoskinson. *Professional Java Server Programming, J2EE Edition*. Wrox Press Ltd., Birningham, UK, 2000.

[10] Sun Microsystems, Inc. *Java Native Interface*, 2002. Available from World Wide Web: `http://java.sun.com/j2se/1.4.1/docs/guide/jni/index.html` [cited April 10, 2003].

Andreas Klimke

Institute of Applied Analysis and Numerical Simulation
University of Stuttgart
Pfaffenwaldring 57
70550 Stuttgart, Germany

E-Mail: `klimke@ians.uni-stuttgart.de`

Erschienene Preprints ab Nummer 2003/001

Komplette Liste: `http://preprints.ians.uni-stuttgart.de`

2003/001 *Lamichhane, B. P., Wohlmuth, B. I.*: Mortar Finite Elements for Interface Problems.

2003/002 *Dryja, M., Gantner, A., Widlund, O. B., Wohlmuth, B. I.*: Multilevel Additive Schwarz Preconditioner For Nonconforming Mortar Finite Element Methods.

2003/003 *Klimke, A., Hanss, M.*: On the Reliability of the Influence Measure in the Transformation Method of Fuzzy Arithmetic.

2003/004 *Klimke, A.*: RANDEXPR: A Random Symbolic Expression Generator.

2003/005 *Klimke, A.*: How to Access Matlab from Java.