Matlab-like Scripting for the Java Platform with the jLab environment

STERGIOS PAPADIMITRIOU, KONSTANTINOS TERZIDIS
Department of Information Management
Technological Educational Institute of Kavala
65404 Kavala,
GREECE

sterg@teikav.edu.gr, kter@teikav.edu.gr

Abstract: - The jLab environment extends the potential of Java for scientific computing. It provides a Matlab/Scilab like scripting language that is executed by an interpreter implemented in the Java language. The jLab environment combines effectively Groovy like compiled scripting with the interpreted jScript one. A special purpose modification of the Groovy language, called GroovySci is developed for effective compiled scripting.
The paper concentrates on the topic of using the jLab scripting engine from within a pure Java application, in order to allow the application to utilize the scientific scripting potential of jLab and its large scientific libraries. The implementation is inspired by the JSR 223 standard but it is much simpler. The same methodology for script invocation can also be used within the Groovy effective compiled scripting framework. We describe the basics of the Groovy scripting environment. To our knowledge, this is the first full Matlab like scientific scripting for Java. The jLab environment is open source and can be downloaded from https://jlab.dev.java.net

Key-Words: - Java, Scripting, Interpreters, Matlab, Scientific Programming, Groovy Scripting

1 Introduction

The Java platform provides rich resources for both desktop and web application development [4,11]. However, using those resources from outside the platform has been impractical unless you resort to proprietary software solutions. No industry standard has defined or clarified how developers can use Java class files from other programming languages. Scripting languages haven't had a standard, industry-supported way to integrate with Java technologies.

However, today things they are changed. One change is Java Specification Request (JSR) 223, which helps developers integrate Java technology and scripting languages by defining a standard framework and application programming interface (API) to do the following:

1. Access and control Java technology-based objects from a scripting environment
2. Create web content with scripting languages
3. Embed scripting environments within Java technology-based applications

This article focuses on the specification's third goal and demonstrates how to use a scientific Matlab-like scripting programming environment from a Java platform application. The later environment is the jLab environment described in detail in [12,13]. It is a Matlab/Scilab/Octave like programming environment [3,5,6,7,8,9] coded in pure Java. Its main component is a mathematical scripting engine implemented in pure Java with techniques similar to those described in [1,2]. Scientific programming issues have been addressed previously in [16,17,18].

The paper describes the utilization of jLab

scripting engine from within a pure Java application, in order to allow the application to utilize the scientific scripting potential of jLab and its large scientific libraries. The implementation is inspired by the JSR 223 standard but it is much simpler. The simplicity of the interface is a crucial factor for its adoption by the scientific community. To our knowledge, this is the first full Matlab like scientific scripting for Java. The jLab environment is open source and can be downloaded from https://jlab.dev.java.net

The paper proceeds by describing the advantages offered by the scripting languages in section 2. Section 3 outlines basic concepts of the JSR 223 specification and places our design in terms of it. Section 4 is the main core of the paper since it describes the most useful routines for parameter communication between the Java program and the jLab scripting engine and vice versa. Also, it presents a simple and illustrative code example. Section 5 describes the basics of the Groovy compiled scripting engine that coexists in jLab and can utilized in the same way for more effective compiled scripting. Finally, the conclusions are presented along with directions for future work.

2 Rationale for Scripting Languages

Most scripting languages are dynamically typed. You can usually create new variables without predetermining the variable type, and you can reuse variables to store values of different types. Also, scripting languages tend to perform many type conversions automatically, for example, converting the number 10 to the text "10" as necessary. Although some scripting languages are compiled, most languages are interpreted. Script environments generally perform the script compilation and execution within the same process. Usually, these environments also parse and compile scripts into intermediate code when they are first executed.

These qualities of scripting languages help you write applications faster, execute commands repeatedly, and tie together components from different technologies. Special-purpose scripting languages can perform specific tasks more easily or more quickly than can more general-purpose languages. For example, many developers think that the Perl scripting language is a great way to process text and to generate reports. Other developers use the scripting languages available in `bash` or `ksh` command shells for both command and job control. Other scripting languages help to define user interfaces or web content conveniently. Developers might use the Java programming language and platform for any of these tasks, but scripting languages sometimes perform the job as well or better. This fact doesn't detract from the power and richness of the Java platform but simply acknowledges that scripting languages have an important place in the developer's toolbox.

Combining scripting languages with the Java platform provides developers an opportunity to leverage the abilities of both environments. You can continue to use scripting languages for all the reasons you already have, and you can use the powerful Java class library to extend the abilities of those languages. If you are a Java language programmer, you now have the ability to ship applications that your customers can significantly and dynamically customize. The synergy between the Java platform and scripting languages produces an environment in which developers and end users can collaborate to create more useful, dynamic applications.

3. Architecture of the JSR 223 and Discovery Mechanism

Version 6 of the Java Platform, Standard Edition (Java SE), does not mandate any particular script engine, but it does include the Mozilla Rhino engine for the JavaScript programming language. The Java SE 6 platform implements the `java.script` APIs, which allow you to use script engines that comply with JSR 223.

An important design goal of this API is portability. It means to serve as a standard API for embedding all kinds of scripting languages, although these are characterized

by diversity in terms of the functionalities that they provide. The API tends to cover all the features that a certain scripting engine can supply, and at the same time it enables simple engines, with just basic functionalities, to comply with the API.

Thus, the Scripting API provides the ability to application developers to determine the features that are implemented in certain scripting engines at runtim. The developers can adjust their code for specific cases, or fail correctly if the scripting engine does not implement certain optional features.

The discovery mechanism refers to the way with which the available scripting engines are detected. The Scripting API is based on the service provider mechanism described in the Jar File Specification. According to this specification, the service is a set of interfaces and (possibly abstract) classes. The **service provider** represents an implementation of the service, i.e. an implementation of the interfaces and abstract classes.

The mechanism allows to make service providers available to the application **dynamically**, by **adding them to the classpath**. The Jar File Specification achieves this by specifying that the files located in the META-INF/services folder of the JAR archieves should be used as the service providers configuration files. Furthermore, the configuration files should be named after the service interfaces (or abstract classes) they implement, and the name must include the service package as well. Finally, the files should contain a newline-separated list of particular classes that implement the service.

Scripting engines are created through the factory method defined in the **javax.script.ScriptEngineFactory** interface. So the **ScriptEngineManager** class searches through all the JAR files in the application's classpath and registers engine factories that are found in the **META-INF/services/javax.script.ScriptEngineFactory** files of those archives. For example the jLab .jar archive (e.g. The jLabLinSol.jar or the jLabWin.jar) contains the following entry:
com.sun.script.jLab.jLabScriptEngineFactory in its META-INF/services/javax.script.ScriptEngineFactory file. This implies that the class **com.sun.script.jLab.jLabScriptEngineFactory** implements the ScriptEngineFactory functionality for the jLab engine.

However, although initially we had implemented a design based on the ScriptEngineFactory concept, we observed that although such indirect designs are elegant and promote generality, they are rather difficult and inconvenient for the majority of the scientific community. Thus, we redesigned the interface in order to become much more direct and simpler.
The redesigned interface is based on direct instantiation of *the jLabScriptEngine* object. All that the user has to perform is to have the .jar of the jLab system on the classpath and to import the jLabScriptEngine class (i.e. import jLabScriptEngine.jLabScriptEngine).

The jLab Script engine (jLabScriptEngine) keeps a reference to the jLab interpreter object that will be used to evaluate the script. The Java Virtual Machine classloader loads the *jLabScriptEngine* class from the jLab .jar file (which we recall is accessible from the classpath). The *getClass().getResource()* method is used to load a simple resource placed within the directory of the *jLabScriptEngine* class (i.e. “resources/engine.gif”). The returned URL is used to extract the name of the jLab's jar file which is important. The *jLabScriptEngine* can then create a full jLab scripting engine object from the jar file. Subsequently it initializes the execution environment by clearing the bindings.

The important *eval*() method executes the script. There are overloaded versions but the most important is the method that takes a Bindings parameter that control the context at which the script at which the script will be executed. The *eval()* method initially retrieves the current context in order to restore it later after its execution. Then it sets the new bindings for the evaluation of the script. These bindings are passed to the

scripting interpreter object that is responsible for evaluating the expression with the bindings.

4. Evaluation

The utilization of the scripting facilities of the jLab scripting engine from a Java program is straightforward. We have provided a simple and effective interface for communicating parameters between the Java program and the scripting engine. This interface follows the spirit of JSR223 but we designed it different in two ways in order to have it more effective for scientific programs:

a. the implementation of the full generality of JSR223 is avoided. Although the design of JSR223 is excellent from the software engineering point of view, we believe that the average scientist likes a much simpler and straightforward set of routines, in order to keep the intefacing lines of code up to a few simple lines only. The JSR 223 is full of concepts such as script engine managers, script engine factories, script engine metadata etc., that although they are technically elegant, they usually confuse the average scientist that usually is not an expert in software engineering methodologies.

b. some additional variants of the *get()* method are implemented in order to retrieve easily the binded values of the main data types used in scientific programs, i.e. Strings, floats, vectors and matrices.

Thus the implemented interface consists of the following routines:

1.// put a binding for the variable key at the ENGINE_SCOPE
public void put(String key, Object value)

2. // get the binding for variable key from the ENGINE_SCOPE
public Object get(String key)
The two methods above can handle arbitrary objects. However, practically more usefull are the following methods that handle the basic types:

3. // get the double value computed for the variable by the Interpreter
public double getBindingAsDouble(String variableName);
This method returns to the Java program the value of the numeric variable *variableName* as a *double* type.

4. // get the double value computed for the variable by the Interpreter
public double getBindingAsDouble(String variableName);
This method returns to the Java program the value of the numeric variable variableName as a double type.

5. // get the String value computed for the variable by the Interpreter
public String getBindingAsString(String variableName);
This method returns to the Java program the value of the alphanumeric variable *variableName* as a *String* type.

6. // get the Vector value computed for the *double []* variable *variableName*
public double [] getBindingAsVec(String variableName);
This method returns to the Java program the value of the alphanumeric variable *variableName* as a *String* type.

7. // get the Matrix value computed for the *double [][]* variable *variableName*
public double [][] getBindingAsMatrix(String variableName);
This method returns to the Java program the value of the alphanumeric variable *variableName* as a *Matrix* type.

With this interface the utilization of the mathematical scripting potential of the jLab scripting engine from Java programs is kept very simple. For example, the following example Java code, first constructs a jLab scripting engine object, with the call: *jLabScriptEngine jLabSEng = new jLabScriptEngine();* Then it creates a Binding object in order to pass the binded variables to the interpreter, i.e. *Bindings binding = new SimpleBindings();*

At this object the code puts the variables that we have to transfer to the interpreter, i.e.
binding.put("freq1", new Double(10.0));

```
binding.put("freq2", new Double(4.0));
binding.put("amplitude", new Double(3.0));
```

Afterwards the expression that we have to evaluate is passed to the interpreter object, i.e.

```
LabSEng.eval("t=0:0.01:20; freq = freq1*freq2; x=sin(freq1*t)+amplitude*cos(freq2*t); plot(t,x); y=ones(20,20)*20; title('Evaluation with freq = '+freq);", binding);
```

Finally, the last interaction step is to retrieve the computed values that we are interested for. The example code retrieves both a double, a vector and a matrix.

The program listing that follows is the complete Java program. In order to compile and execute this program we need only place the .jar archive of the jLab system (i.e. the jLabLinSol.jar or the jLabWin.jar) in a place accessible by the Java classpath, e.g. a specific example using the -cp option, for compile:

```
javac -cp /export/home/sterg/NBProjects/jLab/jLabPr/dist/jLabLinSol.jar:. testJLAB.java
```

and for running the example:

```
java -cp /export/home/sterg/NBProjects/jLab/jLabPr/dist/jLabLinSol.jar:. testJLAB
```

The complete program listing follows:

```
import java.util.Collection;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import javax.script.ScriptEngineFactory;
import javax.script.ScriptEngineManager;
import jLabScriptEngine.jLabScriptEngine;
import javax.script.Bindings;
import javax.script.ScriptException;
import groovy.lang.Binding;
import javax.script.SimpleBindings;
import javax.swing.JOptionPane;

public class jLabEngineTest {
        public static void main(String[] args) {
    jLabScriptEngine jLabSEng = new jLabScriptEngine();
        try {
 Bindings binding = new SimpleBindings();
  binding.put("freq1", new Double(10.0));
  binding.put("freq2", new Double(4.0));
  binding.put("amplitude", new Double(3.0));
    jLabSEng.eval("t=0:0.01:20; freq = freq1*freq2; x=sin(freq1*t)+amplitude*cos(freq2*t); plot(t,x); y=ones(20,20)*20; title('Evaluation with freq = '+freq);", binding);
       double freqRes = jLabSEng.getBindingAsDouble("freq2");
 double [] x = new double [1];
 x = jLabSEng.getBindingAsVec("x");
  for (int k=0; k<100; k++)
   System.out.println("x["+k+"]= "+x[k]);
 double [][] y = new double [1][1];
 y = jLabSEng.getBindingAsMatrix("y");
   for (int k=0; k<10; k++)
      for (int m=0; m<10; m++)
           System.out.println("y["+k+"]["+m+"]="+y[k][m]);
         System.out.println( "freqRes = "+freqRes);
         }

      catch (ScriptException ex) {
         ex.printStackTrace();
      }
  }
}
```

The figure that follows (Figure 1) illustrates the signal computed and plotted by the jLab interpreter on behalf of the Java program.

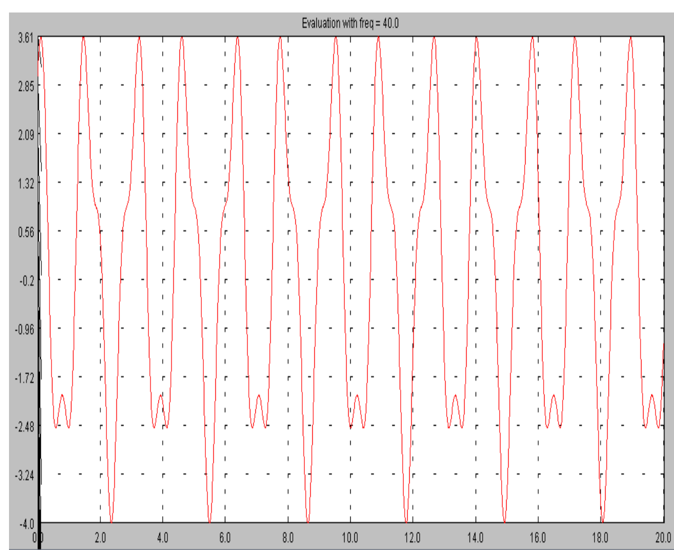

Figure 1: The jLab scripting engine computes and plots this graph, when it is

invoked from the simple Java program described above

5 The Groovy Scripting Framework

The Groovy approach to scripting is very different from the j-Script one presented previously. We developed also a Groovy based compiled scripting framework that works in the same way as the previously described interpreted jScripts and we call this as *GroovySci*.
In order to understand the mechanics involved suppose that a Groovy script named groovyScript.groovy is to be evaluated. The following steps then are involved [14]:

a. The file groovyScript.groovy is fed into the Groovy parser.

b. The parser generates an Abstract Syntax Tree (AST) that fully represents all the code in the file. Actually, the Groovy parser is based on the ANTLR (ANother Tool for Language Recognition) tool, described in detail in [15].

c. The Groovy class generator takes the AST and generates Java bytecode from it. Depending on the contents of the script file, this phase can create multiple classes. These classes are then available through the Groovy classloader.

d. The Java runtime is invoked in a manner equivalent to running **java groovyScript.**
Thus, contrary to j-Script scripting, Groovy does not directly interprets the scripts. Classes are always fully constructed before runtime and do not change while running.

- Scripts in Groovy are represented by the class Script. If a Groovy file does not contain any class declaration, it is handled as a Script. This is performed by transparently creating a Script class from the script code. The generated class has the same name as the source script filename. The source code of the script is mapped into a run method, and an additional main method is constructed for the initiation of the script execution.

- When a Groovy file declares one class with the same name as the file, then there exists the same one-to-one relationship as in Java.

- A Groovy source file may contain multiple class declarations of any visibility. Different from Java is the fact that there is no enforced rule that any of these classes must match the filename. The groovyc compiler generates class files for all the declared classes.

- A Groovy source file can have both class declarations and scripting code. The scripting code in this case is wrapped into a main class to be executed with a name equal to the Groovy's script source filename. Thus, it is important in such a situation to avoid declaring a class with the same name as the source filename.

However, Groovy can be characterized as a “scripting super-Java”. It is a lot easier and quicker for the programmer to work with Groovy instead of Java. One of its strongest features is that it is a **dynamic** language [14].

Dynamic languages have the potential of seemingly modifying classes at runtime. We can for example add new methods to a running class. Groovy can achieve this flexibility even though the bytecodes of the generated classes cannot change. The bytecode that the Groovy class generator produces is necessary different from that of the Java compiler, not in format but in content. The dynamic nature of Groovy makes relatively easy to extend the language with Matlab like matrix operations.

The concept of the workspace is implemented in GroovySci with the use of undeclared variables. In this case these variables are assumed to come from the script's binding and are added to the Hashtable data structure

that implements the binding. The binding is a data store that enables transfer of variables to and from the caller of a script. This trick gives the illusion to the user that the code is directly interpreted but isn't. Classes that are not scripts, do not participate in the variable binding technique, thus they should not have undeclared variables.

The declared variables of these classes are kept local to the class that implements the script and are not exported to the calling environment through the binding.

Naming in the Groovy system becomes very important when we do not compile explicitly. In this case, Groovy finds a class by matching the class name to a corresponding *.groovy source file. Thus the source file name becomes important in the class resolving process. After finding such a file, all its declared classes are parsed and become known to Groovy.

The *default field visibility* has a special meaning in GroovySci. When no visibility modifier is attached to a field declaration, a property is generated for the respective name.
At GroovySci's methods, declaring explicit return parameter types is optional. If the return type declarations are omitted, Object is used. For example, the classic Java declaration:

public static void main(String [] args)

can be recasted in Groovy as:

static main (args)

We note that the *public* modifier is optional and was omitted since public is the default visibility for GroovySci's methods. Also, because return types are not used by the dynamic method *dispatch()*, the *void* declaration is also omitted.

Usually most programming environments support positional parameters, where the meaning of each argument is determined from its position in the parameter list. GroovySci in addition offers the named parameters, that facilitate the handling of methods that accept many parameters.

Groovy's method invocation is accomplished with the ***InvokerHelper*** *class* [14]. This class has an ***invokeMethod*** method that has the prototype:
public static Object invokeMethod(Object object, String methodName, Object arguments)

This method accomplishes the important task of invoking the given method on the object. It works as follows:

If the *object* is a *Class* its calls a static method from the class. This is accomplished by retrieving the metaclass of the class from the metaclass registry object. The metaclass registry keeps globally the association between the class objects and their metaclasses. Thus, the call:
MetaClass metaClass =
metaRegistry.getMetaClass(theClass);

retrieves the metaClass object that corresponds to the class object that is passed as parameter. Obviously, if the object is a class we can only call its static methods. This is accomplished with the call:
metaClass.invokeStaticMethod(object,
methodName, as Array(arguments));

If the object is a class instance then the invocation code distinguishes two cases:

a. if the object does not implement the GroovyObject interface (i.e. it is not a builder, closure etc.) then the method is invoked with the call:
invokePojoMethod(object, methodName,
arguments);

This method retrieves first the class of the object, i.e.:
Class theClass = object.getClass();

Then the metaRegistry object is queried in order to obtain the metaclass of the former class object, i.e.
MetaClass metaClass =
metaRegistry.getMetaClass(theClass);

Finally, this metaclass object is used to invoke the method:
metaClass.invokeMethod(object, methodName,
asArray(arguments));

b. if the object implements the GroovyObject

interface the invoking protocol is somewhat different and it is undertaken by the *invokePogoMethod()*.

This method first casts the object as an instance of GroovyObject in order to exploit the pecularities of the later. Subsequently, the object is asked if it implements the GroovyInterceptable interface.

If it's a pure interceptable object (*even intercepting toString(), clone(), ...*) then the invocation is performed with the call:
groovy.invokeMethod(methodName, asUnwrappedArray(arguments));

The Interceptor class allows to specify code that is executed before and after the method itself is optionally called. These code chunks are specified with the *beforeInvoke()* and *afterInvoke()* methods.
Object beforeInvoke(Object object, String methodName, Object[] arguments);

The ***Delegating*** metaclass is a metaclasss that delegates the metaclass functionality to the ***delegate*** metaclass. The later metaclass is kept in a field of the delegating metaclass. The purpose of the delegating metaclass is to delegate the metaclass related operations to its delegate. The basic operations that are delegated include:
addNewInstanceMethod(), addNewStaticMethod(), addMetaMethod(), getMethods(), getProperty(), getProperty(), invokeConstructor(), invokeMethod(), invokeStaticMethod(), setMetaClass(), getMetaClass().

The ***ExpandoMetaClass*** is the metaclass that allows the addition of new methods on the fly. Thus, it constitues a basic component of the dynamic object orientation of the Groovy system. By default methods are only allowed to be added before *initialize()* is called. In other words you create a new ***ExpandoMetaClass***, add some methods and then call *initialize()*. If you attempt to add new methods after *initialize()* has been called an error will be thrown. This is to ensure that the MetaClass can operate appropriately in multi threaded environments as it forces you to do all method additions at the beginning, before using the MetaClass. In order to be able to expand dynamically the methods of a class object an **ExpandoMetaClass** object should be constructed for it. The method ***invokeMissingMethod()*** overrides the default missing behaviour and adds the capability to look up a method from super class. This class also has the potential of recursively searching for a particular method in a class hierarchy.

A **ProxyMetaClass** is a subclass of MetaClass, ProxyMetaClass manages calls from Groovy Objects to POJOs (Plain Old Java Objects). It enriches Metaclass with the feature of making method invocations interceptable by an Interceptor. To this end, it acts as a decorator (decorator pattern) allowing to add or withdraw this feature at runtime.

A MetaClass within Groovy defines the behaviour of any given Groovy or Java class. The MetaClass interface defines two parts, the client API which is defined via the MetaObjectProtocol interface and the contract with the Groovy system.

In general the compiler and Groovy runtime engine interact with the methods of the MetaClass class, while MetaClass clients interact with the methods defined by the MetaObjectProtocol interface.

6 Conclusion

The paper has presented a powerful scripting language that is executed by an interpreter implemented in the Java language. This language supports all the basic programming constructs and an extensive set of built in mathematical routines that cover all the basic numerical analysis tasks.

The main scope of the paper was to demonstrate the potential of utilizing this scripting engine from Java code. This allows the programmer to exploit the full potential of a mixed mode programming paradigm:

- Java compiled code for the computationally demanding operations and

- Scripting code for fast implementation of the program's structure.

Although the creation of j-Script code is easier, and quicker, the alternative option of implementing compute intensive code in Java, offers to the competent Java programmer the advantages of execution speed, enhanced code robustness and the potentiality to exploit existing open-source Java libraries.
Thus, jLab can serve also and as an environment for the gradual development of complex systems, starting from an initial prototype that consists mostly of scripting code and in stages replacing script code with pure Java code.

This design permits to obtain both speed efficiency and flexibility while at the same time allows the utilization of the vast amounts of scientific software that is implemented in the Java language.

Acknowledgment
The authors wish to thank the Research committee of the Technology Education Institute of Kavalas, Greece, for the partial financial support of this research.

References:

[1] David A. Watt and Deryck F Brown, "*Programming Language Processorsin Java*", 2000, Pearson Education

[2] Steven John Metsker, "*Building Parsers with Java*", Addison-Wesley,2001

[3] Stephen L. Campbell, Jean-Philippe Chancelier, Ramine Nikoukhah, "*Modeling and Simulation in Scilab/Scicos*", Springer, 2006

[4] Cay Horstmann, Gary Cornell, "*Core Java 2*", Vol I Fundamentals, Vol II - Advanced Techniques. Sun Microsystems Press, 7th edition, 2005

[5] Norman Chonacky, David Winch, "*3Ms for Instruction: Reviews of Maple, Mathematica and Matlab*", Computing in Science and Engineering, May/June 2005, Part I, pp. 7-13

[6] Norman Chonacky, David Winch, "*3Ms for Instruction: Reviews of Maple, Mathematica and Matlab*", Computing in Science and Engineering, July/August 2005, Part II, pp. 14-23

[7] Michael Trott, "T*he Mathematica Guidebook: Programming*", Springer, 2004

[8] Erwin Kreyszig, "Maple Computer Guide for Advanced Engineering Mathematics (8th Ed.)", Wiley, 2000

[9] John W. Eaton, "*GNU Octave Manual*", Network Theory Ltd, 2002

[10] Desmond J. Higham, Nicholas J. Higham, "*Matlab Guide*", Second Edition, SIAM Computational Mathematics, 2005

[11] Budi Kumiawan, *A Tutorial: Java 6 New Features,* BrainySoftware, 2006

[12] S. Papadimitriou, *Scientific programming with Java classes supported with a scripting interpreter*, IET Software, 1, (2), pp. 48 - 56, 2007

[13] Papadimitriou S, Terzidis K., *“jLab: Integrating a scripting interpreter with Java technology for flexible and efficient scientific computation”*, Computer Languages, Systems & Structures (2008), Elsevier, in print

[14] Dierk Konig, Andrew Glover, Paul King, Guillaume Laforge, Jon Skeet, *Groovy In Action, Manning* Publications, 2007

[15] Terence Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages,* The Pragmatic Programmers, 2007

[16] Stergios Papadimitriou, Konstantinos Terzidis, Seferina Mavroudi, Skarlas Lambros, Spiridon D. Likothanasis, *“Fuzzy rule based classifiers from SV Learning”*, *WSEAS Transactions On Computers*, Issue 7, Vo 4, July 2005

[17] Stergios Papadimitriou, Konstantinos Terzidis, The Design and Implementation of a Java Based Open Source Mathematical Programming Environment, *WSEAS Trans. On Information Science and Applications*, Issue 7, Vo 4, April 2007, pp. 836-840

[18] Stergios Papadimitriou, Konstantinos Terzidis, Seferina Mavroudi, Skarlas Lambros, Spiridon D. Likothanasis, Fuzzy rule based classifiers from SV Learning, *WSEAS Transactions On Computers*, Issue 7, Vo 4, July 2005, pp. 661-670