A Java based scientific programming environment with a scripting interpreter supported with Java classes

STERGIOS PAPADIMITRIOU, KONSTANTINOS TERZIDIS

Department of Information Management,
Technological Educational Institute of Kavala,
65404 Kavala, Greece,

Abstract

The jLab environment provides a Scilab like 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 efficiency of the Java compiled code can be directly utilized for any computationally intensive operations. Since jLab is coded in pure Java the build from source process is much cleaner, faster, platform independent and less error prone than similar C/C++/Fortran based open source environments (e.g. Scilab, Octave).

keywords: Programming Environments, Java, Scientific Software, NeuroFuzzy Systems

1 Introduction

Recently with the growing speed and potentiality of computers the populatity of integrated scientific programming environments has significantly rising. These environments in general demand much more time and space resources from the traditional compiled programming languages (i.e. C++ and Fortran). However, they greatly facilitate the task of creating quickly reliable scientific software, even from scientists with little programming expertise.

Two categories of general scientific software can be identified: *computer algebra systems* that perform extensively symbolic mathematical evaluations (e.g. Maple [11], Mathematica [10]) and *matrix computation* systems that are oriented toward numerical computations and are well suited for engineering applications (e.g. the Matlab [13] that dominates at the commercial market and the open source "clones" Scilab [1] and Octave [12]). An excellent recent comparative review of three well-established commercial products can be found in [5, 6].

These systems are usually implemented in C/C++/Fortran and they are available in platform specific binary formats or in also platform specific build from source configurations (for the open source Scilab and Octave). To the contrary, the Java programming language in which jLab is implemented allows platform independence. We have tested jLab on Linux, Solaris and Windows XP and it runs in the same way, on all these different environments, without any change of the code.

Contrary to some other Fortran and C based open source numerical computing environments such as Scilab and Octave, the compilation of the jLab's source is extremely fast,

simple and platform independent. It compiles in only a few seconds, while the Scilab or Octave sources take several minutes. Moreover, at the later environments a lot of machine specific specific details can perplex the building from source process.

The paper proceeds as follows: Section 2 presents the architecture of the components that constitute the jLab system. Section 3 deals with the important subject of function definition, and elaborates on the two different ways to define functions: as Java class files and as jLab j-scripts. Section 4 outlines the main points involved with the modules that perform the dynamic loading and execution of either Java class files, or jLab coded j-script modules. Some important issues related to parsing of jLab programs are discussed in Section 5. Section 6 discusses the performance of jLab. Finally, section 7 concludes the paper and presents some basic directions for future work.

2 The architecture of the system

The system at the top level is consisted of the following main components :

a. The *j*ava *Exec*ution engine (*jExec*), is the part that translates dynamically the jLab programming language and executes the user's commands. It is actually a flexible interpreter coded in Java that is consisted of the following modules:

- The *Lexical Analyzer*. It tokenizes the input in order to permit the parsing phase to operate on a token stream instead of the plain text.
- The *Parser*. The parser first checks the syntax of all the jLab's programming constructs. Then the parser executes each expression by building an expression tree and evaluating the nodes of the tree by a top-down recursive traversal.

b. The *Java toolboxes*. These toolboxes consist of Java class libraries that need to adhere only to a small set of conventions in order to be directly utilized from jLab. The popularity of the Java language makes it easy to utilize excellent libraries for specific domains, e.g. the JOONE library for neural networks [7], the WEKA data mining system and the fuzzy expert system of Bigus [8].

c. The *jLab toolboxes* use the *jLab interpreted language* to implement program logic with text code files called *J-Files*. We selected to follow the syntax of the Scilab language [1].

3 Function Handling

In jLab a specific Java class (the *FunctionToken* class) is used to implement the functionality of function handling and to represent any functions used in an expression. A function can be implemented either as a compiled Java *class file* or as a jLab *J-File*. We will refer to the former functions as *compiled Java functions* (abbreviated *J-Classes*). The later (i.e. the *J-Files*) implement either functions and are refered as *J-Functions* or they are simply batches of jLab code, the *J-Scripts*. The *J-Scripts* serve as "batch" files for jLab's commands. The *J-Files* are interpreted and they resemble the syntax of Scilab's .sce files.

The *J-Functions* can return multiple return parameters in a syntax $[rv_1, rv_2, \ldots,] =$ *some-J-Function*$(arg_1, arg_2, \ldots)$, where rv_i denotes the return values and arg_i are the arguments of the function.

The *J-Files* can be easily programmed since the jLab language is untyped and their syntax is kept simple, Scilab-like and to a large extent Scilab compatible. They can be directly executed in the jLab environment by placing them in directories accessible by the *jLabScriptPath* jLab's environment variable that has a similar role for J-File loading that the Java's virtual machine classpath has for class loading.

Their main disadvantage is their speed of execution: they are usually much slower than the equivalent Matlab or Scilab functions. However this drawback can be bypassed when

the programmer implements the equivalent functionality with a Java class file, i.e. a *J-Class*, that can also be dynamically executed by the system. At this case the code is very fast, since it is compiled Java code, and can compete even corresponding C++ or Fortran library functions. Although some Java libraries perform even better than native code libraries we should expect a delay by a constant factor of about 2 to 3, due to the virtual machine overhead.

J-Classes offer the potential to easily extend the functionality of the system at several application domains with Java code. We refer to the dynamically connected J-Classes that aim to implement various toolboxes and are implemented with Java classes, as *extension J-Classes*. The interfacing with *J-Functions* is encapsulated with the *ExternalFunction* class. Each *compiled J-Class* operates on a list of objects of the *Operand* abstract class type. As we will see, this design allows for maximum flexibility in parameter passing.

However, there are several other important classes that also represent Java class code, although this type of code is integrated with the system. These are represented by the *InternalFunction* class that is the base class for all the internal function types.

It is important to emphasize the basic distinction between *Internal* and *External* functions: *Internal* functions are "hardwired" to the system while the *External* can be dynamically extended by the user. We should note at this point that *External* classes are loaded by a special class loader (i.e. the *ExternalFunctionClassLoader*).

A method *evaluate()* defined in a *FunctionToken* class is used to evaluate each function. The evaluation code first checks if the function name is overloaded by a variable. If so, it evaluates the variable. Otherwise it calls the *function manager* (implemented with the class *FunctionManager*) in order to find the function. The *FunctionManager* tracks dynamically the *extension J-Classes*. The potentiality of the Java language for dynamic class loading and execution allows jLab to incorporate easily with its "kernel" any number of Java classes without any recompilation of the system. All that is required is to place the compiled class files in directories visible from the *jLabClassPath* variable.

The evaluation of an extension function is very fast since it is compiled Java code. However, a user with missing or limited Java experience is not expected to be able to implement extension classes. These users can use the jLab's scripting language and implement *J-Files* (*J-Functions, J-Scripts*). A function is referred as *UserFunction* if it is implemented as a J-File.

The evaluation task of each function whether *ExternalFunction* (i.e. Java code) or *UserFunction* (i.e. J-File) starts by first evaluating the operands of the function. Then the *J-script* or the function is evaluated by calling first the *clone()*, so the original functions stay untouched.

Although the evaluation code depends on the function type, each *evaluate* function adheres to the same signature in order to permit flexible evaluation of expression trees, comprised of functions of various types (e.g. both Internal and External functions).

In order to evaluate an *InternalFunction* the system first checks whether the function by itself is an expression. In the affirmative case all the childs of the expression are evaluated recursively. Having evaluated all the childs, the root node, which represents the *InternalFunction* object obtains its value. This value corresponds to its return value, that is returned. When the *InternalFunction* is not an expression it represents a number, which is returned as the function's return value.

The *FunctionManager* maintains the set of functions for the forementioned categories of *Internal* functions (e.g. trigonometric, standard, matrix), and manages the dynamically expanded set of *User* functions (both *J-Functions* and *J-Files*). Java class files that implement external extension J-Classes are loaded by a specific class loader, the *JClassLoader*. Another type of loader, the *J-File loader*, loads the J-Files (i.e. the *UserFunctions*). The *FunctionManager* starts by constructing a number of *internal functions*. A function is processed by first checking whether it is a *UserFunction* (i.e. a jLab piece of code coded as J-File). In the case that the search outcome is negative, the *external J-Classes* becomes the target. Finally, the internal functions are scrutinized. We should stress the point that even the J-Files are processed into Java *UserFunction* classes and then are handled uniformly.

The configuration of jLab is simple: as we already mentioned, two environment variables are used to set the search path for J-Files (i.e. executable scripts) and Java classes (i.e.

executable bytecodes) respectively. The first one is the already mentioned *jLabScriptPath* variable and the other the *jLabClassPath* variable. Both are settable and adjustable from within the graphical interface.

4 The Code Loaders

The custom code loaders are essential to the flexibility and extensibility of the system. Contrary to similar systems, as Scilab [1] and Matlab [13], jLab can be easily extended with specialized Java toolboxes that run as fast as the Java runtime permits. In order to achieve this, jLab owns two types of code loaders implemented with different classes. The first one is the *Java class loader* (abbreviated *jLoader*) that resembles the functionality of flexible java class loaders [2], while the second the *J-File loader* accomplishes the elaborate handling of *J-Files* (either *J-Functions* or *J-Scripts*).

The class loaders keep all the loaded classes in a global hashtable (implemented with the *Hashtable* standard JDK class). The hashtable allows fast lookup at any loaded class.

The Java class loader maintains a *root* directory for the available jLab extension Java class files (i.e. the external J-Classes). The String *baseClassDir* maintains the path of this "root" at the local file system and is a configurable parameter (e.g. for Unix/Linux filesystems can be */home/user5/javaAppl5/jLab*) that also can be supplied as a command line argument at the jLab's execution. The jLab class loader can locate and execute any Java class file located under this "root". With this design we can obtain modular tree-based organization of the jLab's classes, extensibility and exploitation of the superb file-handling facilities of current operating systems.

The *baseClassDir* parameter is very significant and is expected as a command line argument. The jLab is invoked with a command line
java *jLab* "baseClassDir"
In Unix/Linux the command jLab can be executed by:
java *jLab* **$PWD**

The *baseClassDir* is in essence the directory where the jLab system is installed at the local filesystem, i.e. the location of the file *jLab.class* which is the class that initiates the system.

At the *baseClassDir* there can exist two other important but optional configuration files: a.) the *jLab.unix.properties* and the *jLab.win.properties*. Whenever these files exist, jLab initiates automatically the *jLabScriptPath* parameter. Depending on the operating platform (Unix/Linux or Windows) the corresponding file is used. These property files are utilized by the **JFileLoader** class that has the task of locating and retrieving the code implemented in the jLab's interpreted language.

The **JClassLoader** attempts first to locate a class in the formerly mentioned hashtable. In the case that the class is not in this hash table, a search process follows. It uses a simple and effective algorithm to locate the dynamically loaded Java class files: it expects them at the subdirectory *./jExec/Functions* in the jLab directory tree, e.g. at the previous example it will be: */home/user5/javaApps/jLab/jExec/Functions*. Whenever the search at the basedir *./jExec/Functions* fails, the system tries to locate the class at all the directories associated with the jLab's *jLabUserClasses* environment variable. This order of class searching allows the user to extend the existing class names with his/her own classes or J-Files and to keep his/her classes separately from those supplied within the jLab system.

The **JFileLoader** is a class that can load and execute *J-Files* (both the *J-Scripts* and the *J-Functions*) of the jLab language. Recall that the *J-Function* files implement jLab functions while the *J-Scripts* simply organize a batch of commands, i.e. they are just a couple of commands that are typed in text file. The *JFileLoader* in turn calls the *Function-Parser* to parse the text of the *J-File* and to return a *UserFunction* class to *jExec* ready for computation.

The **ReflectionFunctionLoader** is a class that calls a function from an external class using reflection. The reflection system allows Java programmers to look and handle the fields of objects that were not known at compile time. The Java's reflection mechanism

allows to add new classes to the jLab system at run time. With this mechanism the system can dynamically inquire about the capabilities of the classes that were added. The Java runtime system maintains *runtime type identification* on all objects, that keeps track of the class to which each object belongs. This information is used by the virtual machine to select the proper methods for execution.

Since it is quite easy to incorporate Java code into the jLab environment, at the extension *j-Class* framework, the scripting code fits usually only for the implementation of the high level application logic, while the number crunching numerical routines should be coded in Java.

5 Parsing of Functions

As we already emphasized jLab is an environment that can be efficiently utilized with mixed type programming: the high level structure of the program should be coded as a J-Script and the number crunching routines in Java. The Java functions are implemented as *external J-classes* with the *ExternalFunction* class and are important since they are the basic means for the efficient extension of jLab's functionality. Every Java programmer can extend easily jLab by following a few simple rules for their format. The interface for passing parameters to an external function (class *ExternalFunction*) is quite flexible allowing the implementations of arbitrary functions.

Each user specified external function extends the *ExternalFunction* class. It returns a generic structure of type *OperandToken* and accepts parameters in an array of *Token* classes. Numeric parameters can be easily passed with a *NumberToken* structure. The Java runtime object type checking operator *instanceof* is valuable for discovering the types of parameters at runtime. Also, the *StringToken* is the class that represents Strings. Upon evaluation it returns the token itself. It is very suitable for passing alphanumeric information in JLab routines.

The *FunctionParser* parses *user functions*. We recall that user functions are implemented as *J-Files*. The later contain either functions (i.e. the *J-Functions*) or they are simple script files (i.e. the *J-Scripts*).

The *UserFunction* class is the class that handles the user edited J-File functions. This class implements a method that takes the jLab code of the function as a string and returns the *UserFunction* created. The J-File code of the function is represented with an *Operand-Token* class. A standard java *ArrayList* class maintains the names of the parameter values of the function. Similarly, the names of the return values are kept in a return variables *ArrayList*.

A flag indicates whether the *UserFunction* class represents a *J-script* or a *J-Function*. For *J-Functions*, the number of parameters that the function defines within its text body should match the number of parameters at the calling sequence. *J-Scripts* can be evaluated directly from their text code.

However, jLab has harder work in order to execute *J-Functions*. For *J-Functions*, a local context of their local variables is first created. At the next processing step the formal parameters of the function are initialized with the values of the actual parameters. After the parameter passing has been performed, the execution of the function code can be accomplished. The function code must be *cloned* so that the original code remains untouched. That function evaluation code assigns the corresponding values to the return variable. When multiple return variables exist those are collected within a matrix and this matrix is returned.

The *Interpreter (jExec)* starts by separating the expression into tokens by using and constructing an *expression tree*. These actions are performed with the aid of the *parser*. This expression tree is subsequently evaluated. The flexible exception handling capabilities of Java are utilized in order to store information about a possible error on expression evaluation as one special variable.

The *Expression* class implements a tree where each node has a variable number of children. Each node keeps information for the operator that it implements. Also each expression keeps track of the index of the child being executed. The operator being held within the

node is used in order to evaluate the expression accordingly. If this operator is an assignment then we evaluate the right side and we assign to the left side variable the evaluation outcome.

The tokenizer as is well known is one of the first phase of compiler processing [3]. Although tools, as the *lex* (or *flex*) and *yacc* (or *bison*) are valuable, we implemented manually a lexical analyzer and a parser in order to have maximum speed and flexibility.

The class that represents a number used in an expression is the *NumberToken* class. This class holds a 2D array of complex numbers in a 3D array of real values, since each complex number is represented by a $2X1$ array to hold the corresponding real and imaginary value. A wide variety of operations is supported on *NumberTokens*. These operations add, subtract, multiply, raise to a power, scalar multiply, scalar divide, perform trigonometric functions (e.g. $\sin, \cos, \tan$ etc.), exponentiations and logarithms.

Tokens are used also to represent complex jLab's programming language constructs as the *while-do*, *if-then-else*, *for-loop*. The syntax of the *for-loop* construct is:

for (*forInitialization* ; *forRelation*; *forUpdate*)
foreCode

Let us consider some concerns involving the implementation of the for-loop. The *ForOperatorToken* is consisted from four other tokens: the *forInitialization* represents the initialization of the construct, and similarly the *forRelation*, the *forUpdate* and the *forCode* represent the condition test, the updating of the contents and the code block that the for construct executes repetitively.

Subsequently, the evaluation code of the *ForOperatorToken* evaluates first the *forInitialization* token in order the initializations to take effect and then implements the logic of the for-loop by repeatedly evaluating the *forCode* as long as the *forRelation* is true, updating also the increment/decrement (i.e. evaluation of the *forIncrement* token).

Another important token type is the *FunctionToken* that is used to represent any functions used in an expression. The *FunctionToken* class implements all the required functionality for executing the function. Specifically, it first checks if the function is overloaded by a variable name. If so, the system creates a variable and sets the parameters of the function as the limits of the variable. Next, it evaluates the variable with the limits and returns the results. If the function name is not overloaded by a variable the system calls the *FunctionManager* in order to find the function. If the *FunctionManager* detects that the function is *UserFunction* it proceeds by evaluating it, by first evaluating its operands and then the function code.

The evaluation of operators resembles the evaluation of functions. Each operator is evaluated by the function *evaluate* that takes as parameters an array of *Tokens* and returns an *OperandToken*.

Since jLab is untyped an effective mechanism for handling dynamically the current set of variables and the objects to which they refer is required. jLab utilizes the built-in *Hashtable* Java's data structure in order to perform fast lookups. The dynamic class inspection facilities of Java allow to test easily the type of data that is associated with a variable (with the *instanceof* operator).

The system implements *local variables* by using the concept of nesting. In the case of an J-File that does not have its own parameters it is executed at the global context. The contexts are implemented with the well known *pop()* and *push()* stack operators [3].

6 Performance

The execution speed of an algorithm implemented in jLab depend heavily on the proportion of processing performed in Java related to that implemented as a J-Script. Clearly, the number crunching code should be coded in Java and only the control logic should be coded as a J-Script in order to obtain rapid and flexible experimentation. We have performed experiments with an SVM-Matlab toolbox downloaded from http://asi.insa-rouen.fr/ arakotom/toolbox/index.html, that implements in pure Matlab various current kernel and SVM algorithms. The jLab based on the LibSVM Java implementation is on the average about ten times faster than the pure Matlab version. However, the LS-SVM Matlab toolbox incorporates .MEX code compiled in C++ and is of comparable speed to our Java based

LibSVM implementation. We should note that the "pure" C++ implementation of the LibSVM algorithms is only 2 to 3 times faster than the Java version. This fact surprised us initially, and it can be explained by the significant advance at the design and implementation of the Java virtual machine environment. We have tested both the Java and C++ LibSVM implementations on a Pentium 4 PC at 2.6 GHz clock speed, both using the Fedora Core 5 Linux (based on 2.6.15 Linux kernel) and the Sun Solaris 10 operating system running at the same PC also. At both platforms we have used the recent version of the Java Run time Environment (i.e. JRE "1.5.0_07") supplied by Sun Microsystems and the GNU C++ compiler. We have tested also the jLab on the Windows XP platform, and the important point that we have derived is that the execution speed is similar to the Linux and Solaris based experiments. The only significant factor that affects the execution speed is the JRE version: we have observed notable improvement in execution speed by using later improved versions of the Sun Microsystems JRE. In particular the average training time for the data of the classic UCI Sonar dataset, on a Pentium 4 1.8 GHz PC capable of multibooting all the three tested operating systems were: a. **Windows XP:** 0.36 sec for the main training accomplished by the Java class file, and 39 sec for J-Script preparation of data for training b. **Linux (Fedora Red-Hat 5, with 2.6.13 kernel:** 0.41 sec for Java class and 31 sec for J-Script preparation respectively. c. **Sun/Solaris 10:** 0.35 sec for Java class and 32 sec for the J-Script. All the evaluated platforms have used the Sun Microsystems Java Vitual Machine and JDK, version 1.5. Also, the GNU supplied JRE (gcj, gjava) succeeds in compiling most of the jLab system (although there are problems in compiling all the integrated system) but the resulting Java code does not run as efficiently as with the Sun's Java Virtual Machine. Also, the code of jLab can be downloaded from page: http://infoman.teikav.edu.gr/∼stpapad/

7 Conclusions

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. These toolboxes can be easily implemented in Java and the corresponding classes can be dynamically integrated to the system.

The jLab is based on a mixed mode programming paradigm:

- Java compiled code for the computationally demanding operations and
- Scripting code for fast implementation of the program's structure.

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. Also, the implementation of jLab in pure Java allows a build from source process is much cleaner, faster, platform independent and less error prone than similar C/C++/Fortran based open source environments (e.g. Scilab, Octave).

We have compared its performance of jLab with a C/C++ and a Matlab version and across different computing platforms (i.e. Linux, Sun/Solaris, Windows XP). Neuro-Fuzzy algorithms can require enormous computation resources and at the same time an expressive programming environment.

Future work will proceed with the porting of the JOONE library for neural networks [7] and the WEKA data mining system. Furthermore we work on improving the parser in order to allow more flexible contructs, and we improve the efficiency of the parsing phase, in order to be able to compete with C/C++ parser implementations (e.g. Scilab, Octave).

References

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

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

[3] Principles of Compiler Design, Alfred V. Aho, and Jeffrey D. Ullman, Addison-Wesley, 1977

[4] Chang, C.-C., Lin, C.J, "LIBSVM: A library for support vector machines",2001, Available on-line: http://www.csie.ntu.edu.tw/ cjlin/libsvm

[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] Jeff T. Heaton, "Introduction to Neural Networks with Java", Heaton Research, 2005

[8] Joseph Bigus, Jennifer Bigus, "Constructing Intelligent Agents Using Java: Professional Developer's Guide", 2nd Edition, Wiley 2001

[9] Maassen J., Van Nieuwpoort, Veldema R., Bal H., Kielmann T., Jacobs C., Hofman R., "Efficient Java RMI for Parallel Programming", ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 23, Nr. 6, ACM 2001, pp. 747-775

[10] Michael Trott, "The Mathematica Guidebook: Programming", Springer, 2004

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

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

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

[14] Vugranam C. Sreedhar, Michael Brurke, and Jong-Doek Choi, "A framework for interprocedural optimization in the presence of dynamic class loading" In ACM SIGPLAN Conference on Programming Language Design and Implementation, pp 196-207, 2000

[15] Web Page of SVM-Matlab toolbox: http://asi.insa-rouen.fr/ arakotom/toolbox/index.html