Scientific programming with Java classes supported with a scripting interpreter

S. Papadimitriou

Abstract: jLab environment provides a Matlab/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. Moreover, the toolboxes of jLab can be easily implemented in Java and the corresponding classes can be dynamically integrated to the system. The efficiency of the Java compiled code can be directly utilised 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 the similar C/C++/Fortran-based open source environments (e.g. Scilab and Octave). Neuro-Fuzzy algorithms can require enormous computation resources and at the same time an expressive programming environment. The potentiality of jLab is demonstrated by describing the implementation of a Support Vector Machine toolkit and by comparing its performance with a C/C++ and a Matlab version and across different computing platforms (i.e. Linux, Sun/Solaris and Windows XP).

1 Introduction

Recently, with the growing speed and potentiality of computers, the popularity of integrated scientific programming environments has significantly risen. These environments, in general, demand much more time and space resources from the traditional compiled programming languages (e.g. 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 [1], Mathematica [2]) and 'matrix computation' systems that are oriented towards numerical computations and are well suited for engineering applications (e.g. the Matlab [3] that dominates at the commercial market and the open source 'clones' Scilab [4] and Octave [5]). An excellent recent comparative review of three well-established commercial products can be found in [6, 7].

These systems are usually implemented in C/C++/Fortran and they are available either in platform specific binary formats or also in platform specific build from source configurations (e.g. the open source Scilab and Octave systems). To the contrary, the Java programming language, in which the presented jLab environment is implemented, allows one to have 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.

The Java language offers an excellent framework for the construction of flexible scientific software with concepts as:

- *Reflection framework*: This allows the interpreter to flexibly interrogate the dynamically loaded extension toolboxes that contain Java classes, implementing specialised functionality (e.g. ODE solvers and neural network models) [8].
- *Parsing flexibility*: The Java programming language allows one to detect flexibly the type of the next scanned token. The *instance of* operator allows to check dynamically the token type and to take the corresponding actions [e.g. with statements like: if (nextToken instanceof VariableToken)...]
- Well-designed *portable and powerful graphical environment*. This allows the implementation of high-quality scientific graphics that are platform independent.
- *Object-orientation* that allows the modular and robust design that exploits the reusability of the code whenever possible.
- Robust *exception handling*. In a complex, flexible programming environment, a lot of errors can occur. jLab catches a lot of exceptions and in most cases it recovers gracefully without even distorting the flow of user computation, whenever this is possible.
- Reliable, simple and uniform *installation* on any platform (e.g. Unix/Linux and Windows) that supports a recent Java Runtime Environment (JRE).
- User friendly *graphical configuration* of the system's environment variables and the exploitation of the powerful abilities of Java's AWT/Swing for displaying both the program output and the program state.
- Support of concurrent and parallel computation with the multithreaded nature of the language and the extended support of distributed computation technologies [9].

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, whereas the Scilab or Octave sources take several minutes. Moreover, at the later environments, a lot

doi:10.1049/iet-sen:20060048

Paper first received 3rd September and in revised form 3rd December 2006

The author is with the Department of Information Management, Technological Educational Institute of Kavala, Kavala 65404, Greece

E-mail: sterg@teikav.edu.gr

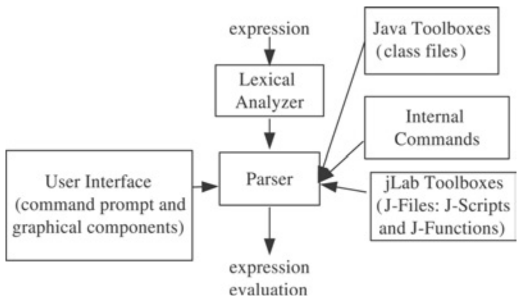

expression
Lexical Analyzer
Java Toolboxes (class files)
Internal Commands
User Interface (command prompt and graphical components)
Parser
jLab Toolboxes (J-Files: J-Scripts and J-Functions)
expression evaluation

Fig. 1 *Main components of the jLab*

of machine-specific details can perplex the building from source process.

2 Architecture of the system

The system at the top level consists of the following main components (Fig. 1):

1. The java Execution 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 consists of the following modules:

- The Lexical Analyser. It tokenises the input in order to permit the parsing phase to operate on a token stream instead of plain text.
- The Parser. The parser first checks the syntax of all the jLab's programming constructs, then executes each expression by building an expression tree and evaluates the nodes of the tree by a top-down recursive traversal (Fig. 1).

2. 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 utilised from jLab. We will demonstrate the construction of a Java class library in Section 6. The Java programmer that implements these toolboxes also has access to the wide set of numerical libraries and application-specialised toolboxes (e.g. fuzzy systems, neural networks). The popularity of the Java language makes it easy to utilise excellent libraries for specific domains, for example, the JOONE library for neural networks [10], the WEKA data mining system [11] and the fuzzy-expert system of Bigus [12].
3. The jLab toolboxes use the jLab interpreted language to implement program logic with text code files called J-Files. We selected it to follow the syntax of the Scilab language [4]. The similarity of the J-File syntax with Scilab facilitates the task of incorporating the repository of Scilab's numerical software. However, currently jLab supports a subset of Scilab syntax and, thus in many cases, it is not possible to execute Scilab files without modifications. We decided to base the syntax on Scilab for the following reasons: (a). Scilab is also an open source effort and can be a productive exchange of ideas between the developers/researchers of both systems and (b) jLab can accelerate significantly existent Scilab code by replacing the compute intensive parts with Java classes. Although the same can be accomplished within Scilab by linking external code, in jLab it is much more easier and modular.

The jLab is a programming environment that integrates the dynamic loading and execution of Java classes with the execution of J-Files (both J-Script files and J-Function files).

Also, we should note that the user interface resembles a Matlab type user interaction via a command prompt on which the user can type and edit commands. Also, the Java's Swing framework [13] is utilised extensively to provide elegant dialog boxes, trees for graphical display of hierarchically organised information and so on. We proceed by describing the jLab architecture that permits implementations of algorithms with both Java and scripting components.

3 Function handling

This section elaborates on the important subject of function handling. The jLab environment allows one to integrate both functions implemented as methods of Java classes and J-Script-based functions implemented as J-Files. From the former functions, the basic ones are implemented as a built-in class library, whereas specialised Java class libraries can extend the potential of the system at particular application domains. Also, the basic functions are handled internally by the system. The general function architecture is demonstrated by Figs. 2 and 3. We proceed by scrutinising the main components.

3.1 J-files, J-functions and extension J-classes

In jLab, a specific Java class, that is, the 'FunctionManager' class, is used to implement the functionality of function handling and to represent any functions used in an expression. The details of the 'FunctionManager' class are described in Section 3.3. 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 *extension J-Classes*). The J-Files are interpreted and they resemble the syntax of Scilab's .sce files. They implement either functions and are refered as *J-Functions* or they are simply batches of jLab code, the *J-Scripts*.

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

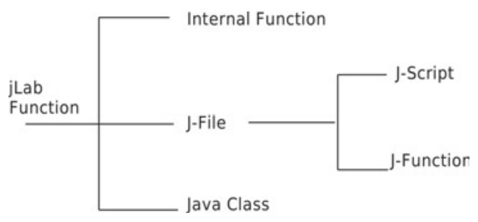


Fig. 2 *Types of functions supported by jLab*

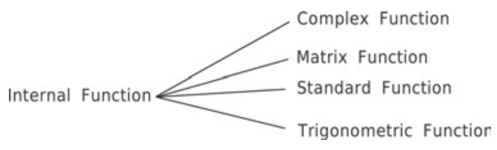


Fig. 3 *Basic types of internal functions*

machine classpath has for class loading. The J-Scripts serve as 'batch' files for jLab's commands.

The J-Functions can return multiple return parameters in a syntax $[rv_1, \; rv_2, \; \ldots] =$ some-J-Function $(\mathrm{arg}_1, \mathrm{arg}_2, \ldots)$, where rv_i denotes the return values and arg_i are the arguments of the function.

An example of a J-Function that returns multiple values is:

```
% computes many values
function [xAdd,xSub,zMult] = computeMulti
(a,b)
  xAdd = a + 100 * b;
  xSub = a - 100 * b;
  zMult = xAdd * Sub;
```

Their main disadvantage is their speed of execution – they are usually 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, that is, a J-Class, that can also be dynamically executed by the system. In this case, the code is very fast, since it is a compiled Java code, and can compete even with the corresponding C++ or Fortran library functions. Although some Java libraries perform better than native code libraries, we should expect a delay by a constant factor of about two to three, because of the virtual machine overhead.

We refer to the dynamically connected J-Classes, which aim to implement various toolboxes and are implemented with Java classes, as extension J-Classes. The extension J-Classes offer the potential to easily extend the functionality of the system at several application domains with Java code.

The interfacing with J-Functions is encapsulated with the ExternalFunction class. Each compiled extension 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.

3.2 Internal functions

In addition to the forementioned extension J-Classes 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. Some subclasses of InternalFunction class that further specialise the corresponding properties and behaviour of the function are (Fig. 3):

- ComplexFunction: A class representing a jLab Complex function. jLab has extensive provisions for complex arithmetic.
- MatrixFunction: A class implementing the mathematical functions for matrices.
- StandardFunction: A class implementing the standard mathematical functions (e.g. abs( ), exp( ), log( ), ln( )).
- TrigonometricFunction: A class implementing trigonometric functions. (e.g. sin( ), cos( ), tan( )).

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

3.3 Function manager

The functionManager class is an essential component with respect to the dynamic class execution. It is implemented by means of a Java class. It uses a method evaluate( ) to evaluate each function. The evaluation code first checks if the function name is overloaded by a variable name. If a variable overloads the function, then a variable is created and the parameters of the function are treated as the limits of the variable. The variable with these limits in turn is evaluated and the corresponding result is returned as the result of an attempt to evaluate the function.

Otherwise, that is, when the function name is not overloaded by a variable name, 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 User Function 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 corresponding J-script or the Java 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). The evaluation is performed according to some priority rules explained below.

In order to evaluate an InternalFunction, the system first checks whether the function by itself is an expression. In the affirmative case, all the child-expression are evaluated recursively. Having evaluated all the child-expressions, the root node, which represents the InternalFunction object obtains its value. This value corresponds to its return value, which is returned. When the InternalFunction is not an expression it represents a numeric value, 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 and matrix) and manages the dynamically expanded set of User functions (both extension Java Classes and J-Files). The 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 an Internal Function (i.e. a built-in Java piece of code). In the case where the search outcome is negative, the extension J-Classes becomes the target. Finally, the UserFunctions are scrutinised. This order of function evaluation is illustrated by Fig. 4. Also, Fig. 5 illustrates the stages of expression parsing. We should stress the point that even the J-Files are processed into Java UserFunction classes and then are handled uniformly.

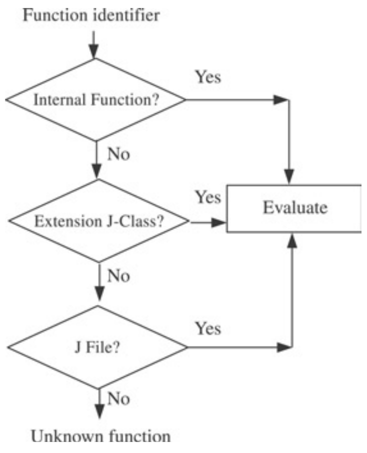


Fig. 4 *Order of function evaluation by the FunctionManager*

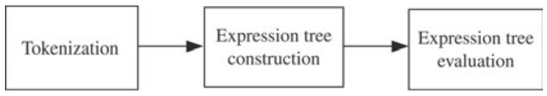


Fig. 5 *Stages for the expression parsing*

The configuration of jLab is simple: as we have 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. These parameters set up the environment for the code loaders that are elaborated in the following section.

4 Code loaders

The custom code loaders are essential to the flexibility and extensibility of the system. Contrary to similar systems, as Scilab [4] and Matlab [3], jLab can be easily extended with specialised Java toolboxes that run as fast as the Java runtime (i.e. the Java Virtual Machine implementation) 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 jClassLoader) that resembles the functionality of flexible java-class loaders [13], whereas 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 for fast lookup at any loaded class. Thus, although the time to locate a new class is linear in the number of extension classes, the subsequent calls to the same class cost only O(1) time. The J-File namespace is handled similarly.

The jClassLoader maintains a root directory for the available jLab extension Java class files (i.e. the extension 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 */javaApps/jLab*) that can also be supplied as a command line argument at the jLab's execution. The `jClassLoader` can locate and execute any Java class file located under this 'root'. With this design, we can obtain modular tree-based organisation of the jLab's classes, extensibility and exploitation of the excellent file-handling facilities of the current operating systems.

The baseClassDir parameter is very significant and is expected as a command line argument. It is, in essence, the root directory where the classes of the jLab system are installed at the local filesystem. At the baseClassDir, there can exist two other important but optional configuration files: 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 utilised 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 where 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, for example, 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 in 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. We remind that the `j-Function` files implement jLab functions, whereas the j-Scripts simply organise a batch of commands, that is, they are just a couple of commands that are typed in a text file. The jFileLoader, in turn, calls the FunctionParser 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 the Java programmers to look and handle the fields of objects that were not known at the compile time [13]. The Java's reflection mechanism allows one to add new classes to the jLab system at the runtime. 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, which 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, preferably the number crunching numerical routines should be coded in Java.

5 Parser design

This section elaborates on the important issue of parsing. The first subsection deals with the issue of function parsing, that is, how jLab deals with the various types of functions. The next subsection analyses the expression parsing, which includes the handling of the programming constructs of the language (e.g. `if-then`, `for-loop`, `while-loop`).

5.1 Function parsing

As was already emphasized, jLab is an environment that can be efficiently utilised with mixed mode programming – the high-level structure of the program should be coded as a j-Script and the number crunching routines in Java. The Java-based extension code is implemented as extension 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 jLab easily by following a few simple rules for the interfacing of the new functions. 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. Up on evaluation, it returns the token itself. It is very suitable for passing alphanumeric information in jLab routines.

The `FunctionParser` class parses user functions. We recall that user functions are implemented as J-Files. The latter either contain 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 `OperandToken` class. A standard Java `ArrayList` class maintains the values of the input parameters 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.

The 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 initialised with the values of the actual parameters. After passing of the parameter 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. The function evaluation code assigns the corresponding values to the return variable. When multiple return variables exist, they are collected within a matrix and this matrix is returned.

5.2 Expression parsing

The Interpreter (jExec) starts by separating the expression into tokens and then it constructs 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 utilised in order to store information about a possible error on the expression evaluation as a special variable.

The `Expression` class implements a tree where each node has a variable number of child-expressions. Each node keeps information for the operator that it implements. Also each expression keeps track of the index of the child expression being executed. The operator 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 the evaluation outcome to the left side variable.

The tokeniser, as is well known is one of the first phase of compiler processing [14]. Although tools, like the `lex` (or `flex`) and `yacc` (or `bison`) are valuable, we implemented manually a lexical analyser and a parser, in order to have maximum speed and flexibility. Moreover, these tools fit better for code generators and not for the interpretation of the code that the jLab performs. Finally, they are most suited for C code generation.

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 also used to represent complex jLab's programming language constructs as the `while-do`, `if-then-else`, `for-loop`. For example, the syntax of the for-loop construct is

```
 for (forInitialization;  forRelation;
forUpdate)
   forCode
```

Let us consider some concerns involving the implementation of the for-loop. The ForOperatorToken consists from four other tokens: the forInitialisation represents the initialisation of the construct, and similarly the forRelation, the forUpdate and the forCode represent the condition test, the updating of the contents of the variables across successive iterations and the code block, which the for-construct executes repetitively.

Subsequently, the evaluation code of the ForOperatorToken evaluates first the forInitialization token for the initialisations 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 a 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 an array of Tokens as parameters 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 utilises 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 a 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 [14].

Having presented some concepts related to the jLab implementation, we proceed by presenting an application to the development of Support Vector Machine (SVM) Learning software.

6 Application for support vector learning

This section demonstrates the potentiality of the jLab for the implementation of complex computational tasks by using its flexibility to directly incorporate the available Java numerical software. In particular, we will deal within the field of Computational Intelligence, with the SVMs, and we will explore the machinery of the LibSVM Java library [15]. Initially, we briefly present the principles of the SVM model. Subsequently, we present the jLab class interface and the jLab code. Finally, we elaborate on the computational performance issue and we perform some comparative tests.

6.1 SVM principles

The SVMs are a relatively new machine learning model that is based on the Statistical Learning Theory of Vapnik [16]. Numerical algorithms for the efficient solution of the quadratic programming problem involved at the SVMs training have been developed recently [17–20]. Although the sophisticated numerical algorithms have realised the practical application to large data sets, the involved computation is still heavy for scripting languages as Matlab/Scilab, and therefore the compiled languages (e.g. C++ and Java) are still necessary for acceptable performance.

We illustrate how easy it is to interface the powerful SVM software with the jLab and to utilise it at application domains. First, we outline the basic SVM theory.

6.1.1 Linear separability of data and linear SVMs: Suppose we are given a set of examples $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)$, where $\boldsymbol{x}_i \in \Re^N$ and $y_i \in \{\pm 1\}$ are the input patterns and their class labels, respectively. Initially, we assume that the two classes of the classification problem are linearly separable. In this case, we can find an optimal weight vector $\boldsymbol{w}_0$ such that $\|(\boldsymbol{w}_0)^2\|$ is minimum (in order to maximise the margin $\Delta = 2/\|(\boldsymbol{w}_0)\|$ of separation [21, 22]) and $y_i \cdot (\boldsymbol{w}_0 \cdot \boldsymbol{x}_i + b_0) \geq 1,\ i = 1, \ldots, l$.

The support vectors are those training examples $\boldsymbol{x}_i$ that satisfy the equality, that is, $y_i \cdot (\boldsymbol{w}_0 \cdot \boldsymbol{x}_i + b_0) = 1$. They define two hyperplanes. One hyperplane goes through the support vectors of one class and the other through the support vectors of the other class. The distance between the two hyperplanes is maximised when the norm of the weight vector $\|(\boldsymbol{w}_0)^2\|$ is minimum. This minimisation can proceed by maximising the following function with respect to the variables α_i (Lagrange multipliers) [16, 23, 24]

$$W(\alpha) = \sum_{i=1}^{l} a_i - \frac{1}{2}\sum_{i=1}^{l} \times \sum_{j=1}^{l} \alpha_i \cdot \alpha_j \cdot y_i \cdot y_j \cdot \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle \tag{1}$$

subject to the constraints: $0 \leq \alpha_i$ and $\sum_{i=1}^{l} \alpha_i \cdot y_i = 0$. If $\alpha_i > 0$ then $\boldsymbol{x}_i$ corresponds to a support vector. The classification of an unknown vector $\boldsymbol{x}$ is obtained by computing

$$F(\boldsymbol{x}) = \operatorname{sgn}\{\boldsymbol{w}_0 \cdot \boldsymbol{x} + b_0\}, \text{where } \boldsymbol{w}_0 = \sum_{i=1}^{l} \alpha_i \cdot y_i \cdot \boldsymbol{x}_i \tag{2}$$

and the sum accounts to only $N_s \leq l$ non-zero support vectors (i.e. training set vectors $\boldsymbol{x}_i$, whose α_i are non-zero). Clearly, after the training, the classification can be accomplished efficiently by taking the dot product of the optimum weight vector $\boldsymbol{w}_0$ with the input vector $\boldsymbol{x}$.

6.1.2 Nonlinear separability of data and nonLinear SVMs: The case in which the data is not linearly separable, it is handled by introducing slack variables $(\xi_1, \xi_2, \ldots, \xi_l)$ with $\xi_i \geq 0$ such that, $y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i + b_0) \geq 1 - \xi_i,\ i = 1, \ldots, l$. The introduction of the variables ξ_i, allows the misclassified points, which have their corresponding $\xi_i > 1$. Thus, $\sum_{i=1}^{l} \xi_i$ is an upper bound on the number of training errors. The corresponding generalisation of the concept of optimal separating hyperplane is obtained by the solution of the following optimisation problem

$$\text{minimise } \frac{1}{2}\boldsymbol{w} \cdot \boldsymbol{w} + C \cdot \sum_{i=1}^{l} \xi_i \tag{3}$$

subject to

$$y_i \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_i + b_0) \geq 1 - \xi_i \text{ and } \xi_i \geq 0, \quad i = 1, \ldots, l \tag{4}$$

The control of the learning capacity is achieved by the minimisation of the first part of (3) whereas the purpose of the second term is to punish for misclassification errors. The parameter C is a kind of regularisation parameter that controls the tradeoff between learning capacity and training set errors. Clearly, a large C corresponds to assigning a higher penalty to errors.

Finally, the case of nonlinear SVMs should be considered. The input data in this case are mapped into a high-dimensional feature space through some nonlinear mapping Φ chosen a priori [20–22]. The optimal separating hyperplane is then constructed in this space. Further details on the mathematical method can be found in the references – an excellent reference is [24].

6.2 jLab SVM class interface

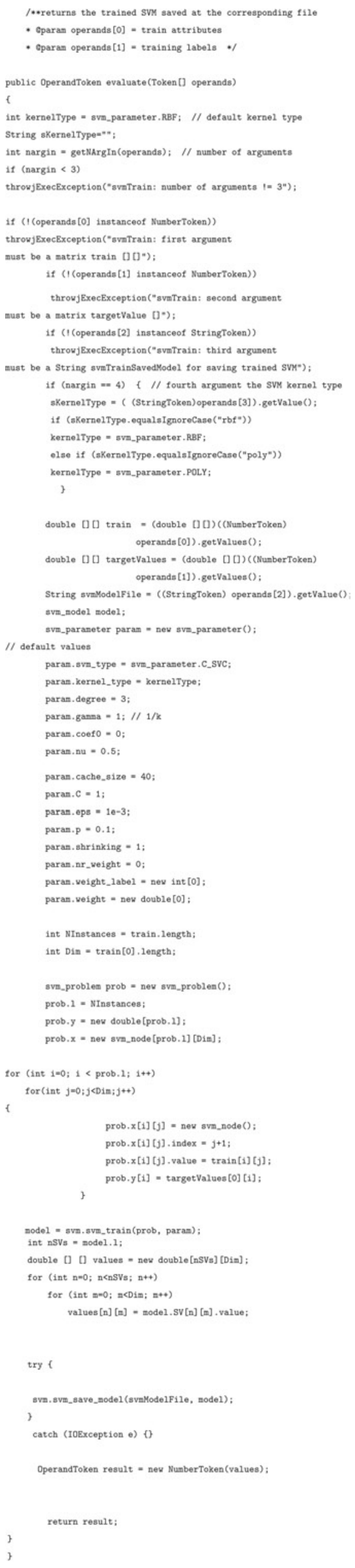
On account of space limitations, we will not present the whole SVM class interface, but instead we will limit ourselves to the SVM training routine. This routine demonstrates the general method of interfacing Java classes in jLab as extension J-Classes. Each extension J-Class is available as a jLab function and its functionality can be directly utilised from within jLab's scripting machinery.

For the particular example, the svmTrain class provides to the jLab the function

```
  double [] [] svmTrain( double [] []
trainData, int [] trainLabels, String
svmModelFile, String svmType);
```

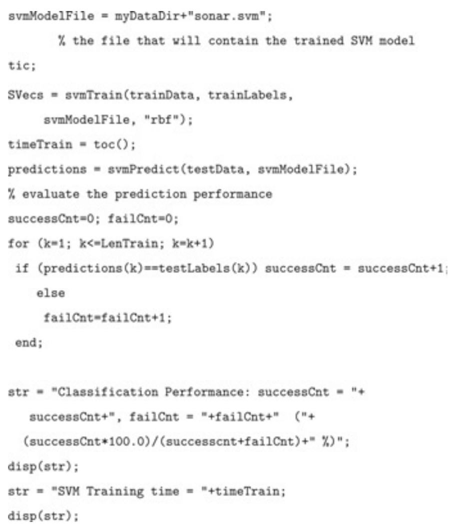
This jLab function triggers the functionality of the evaluate( ) method of the Java class with the same name.

At the first stage of processing the Interpreter localises the class file svmTrain at the jLab's class path (unless the class was already cached either because it is already used or by the class preload mechanism). Then the interpreter



```
    /**returns the trained SVM saved at the corresponding file
    * @param operands[0] = train attributes
    * @param operands[1] = training labels  */

public OperandToken evaluate(Token[] operands)
{
int kernelType = svm_parameter.RBF;  // default kernel type
String sKernelType="";
int nargin = getNArgIn(operands);  // number of arguments
if (nargin < 3)
throwjExecException("svmTrain: number of arguments != 3");

if (!(operands[0] instanceof NumberToken))
throwjExecException("svmTrain: first argument
must be a matrix train [][]");
        if (!(operands[1] instanceof NumberToken))
         throwjExecException("svmTrain: second argument
must be a matrix targetValue []");
        if (!(operands[2] instanceof StringToken))
         throwjExecException("svmTrain: third argument
must be a String svmTrainSavedModel for saving trained SVM");
        if (nargin == 4)  {  // fourth argument the SVM kernel type
         sKernelType = ( (StringToken)operands[3]).getValue();
         if (sKernelType.equalsIgnoreCase("rbf"))
         kernelType = svm_parameter.RBF;
         else if (sKernelType.equalsIgnoreCase("poly"))
         kernelType = svm_parameter.POLY;
           }

        double [][] train  = (double [][])((NumberToken)
                                 operands[0]).getValues();
        double [][] targetValues = (double [][])((NumberToken)
                                 operands[1]).getValues();
        String svmModelFile = ((StringToken) operands[2]).getValue();
        svm_model model;
        svm_parameter param = new svm_parameter();
// default values
        param.svm_type = svm_parameter.C_SVC;
        param.kernel_type = kernelType;
        param.degree = 3;
        param.gamma = 1; // 1/k
        param.coef0 = 0;
        param.nu = 0.5;

        param.cache_size = 40;
        param.C = 1;
        param.eps = 1e-3;
        param.p = 0.1;
        param.shrinking = 1;
        param.nr_weight = 0;
        param.weight_label = new int[0];
        param.weight = new double[0];

        int NInstances = train.length;
        int Dim = train[0].length;

        svm_problem prob = new svm_problem();
        prob.l = NInstances;
        prob.y = new double[prob.l];
        prob.x = new svm_node[prob.l][Dim];

for (int i=0; i < prob.l; i++)
    for(int j=0;j<Dim;j++)
{
                    prob.x[i][j] = new svm_node();
                    prob.x[i][j].index = j+1;
                    prob.x[i][j].value = train[i][j];
                    prob.y[i] = targetValues[0][i];
               }

    model = svm.svm_train(prob, param);
     int nSVs = model.l;
     double [] [] values = new double[nSVs][Dim];
     for (int n=0; n<nSVs; n++)
         for (int m=0; m<Dim; m++)
             values[n][m] = model.SV[n][m].value;


     try {

      svm.svm_save_model(svmModelFile, model);
     }
      catch (IOException e) {}

       OperandToken result = new NumberToken(values);


         return result;
 }
 }
```

Fig. 6 *Java interface to the LibSVM package*



```
svmModelFile = myDataDir+"sonar.svm";
        % the file that will contain the trained SVM model
tic;
SVecs = svmTrain(trainData, trainLabels,
     svmModelFile, "rbf");
timeTrain = toc();
predictions = svmPredict(testData, svmModelFile);
% evaluate the prediction performance
successCnt=0; failCnt=0;
for (k=1; k<=LenTrain; k=k+1)
 if (predictions(k)==testLabels(k)) successCnt = successCnt+1;
    else
     failCnt=failCnt+1;
 end;

str = "Classification Performance: successCnt = "+
   successCnt+", failCnt = "+failCnt+"  ("+
  (successCnt*100.0)/(successcnt+failCnt)+" %)";
disp(str);
str = "SVM Training time = "+timeTrain;
disp(str);
```

Fig. 7 *jLab code for SVM training*

utilises the parameters of the jLab's method svmTrain in order to prepare the call to evaluate( ).

Subsequently, the interpreter exploits the Java's reflection mechanism [13] in order to call the evaluation method. The corresponding Java interface to the LibSVM package [25] is very simple and is shown in Fig. 6.

6.3 jLab scripting SVM code

The part of the jLab code that performs the SVM training and evaluation is Fig. 7.

6.4 jLab 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 a 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 described also in [26, 27]. The jLab based on the LibSVM Java implementation [25] is on an average about ten times faster than the pure Matlab version. However, the LS-SVM Matlab toolbox of [28] 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 two to three 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 the C++ LibSVM implementations on a Pentium-4 PC at 2.6 GHz clock speed, 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. At both platforms, we have used the recent version of the JRE (i.e. JRE '1.5.0_07'), supplied by Sun Microsystems and the GNU C++ compiler. We have also tested 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

Table 1: Some averaged results from the performance of jLab across various platforms (time in seconds)

	Windows XP	Linux	Solaris 10
JDK 1.5, SVM training (Java)	0.36	0.41	0.35
JDK 1.5, jLab script data preprocessing	39	31	32
JDK 1.6, SVM training (Java)	0.3	0.31	0.29
JDK 1.6, jLab script data preprocessing	34	27	26

the JRE version – we have observed notable improvement in execution speed by using the recent improvement 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 are: (a) Windows XP: 0.36 s for the main training accomplished by the Java class file, and 39 s for J-Script preparation of data for training, (b) Linux (Fedora Core 5, with 2.6.13 kernel: 0.41 s for Java class and 31 s for J-Script preparation, respectively, and (c) Sun/Solaris 10: 0.35 s for Java class and 32 s 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 it does with the Sun's Java Virtual Machine. The memory requirements and overhead cost of the script interpreter are very small when no class preloaded is performed. In this case, only the accessed classes are loaded in the memory. However, the class preload operation loads all the extension classes beforehand in the memory and therefore consumes size proportional to the number of extension classes. Table 1 presents some comparative averaged performance results. The averaging was performed across 30 trials. The results clearly illustrate the advantage of the Java compiled code over the scripting jLab, which was used only for data preprocessing.

The Java code of jLab is open source and can be downloaded from https://jlab.dev.java.net/.

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 one to obtain both the speed efficiency and flexibility, when at the same time allows the utilisation of the vast amounts of scientific software that is implemented in the Java language. The implementation of jLab in pure Java allows a much cleaner, faster, platform independent and less error prone build from source process, than similar C/C++/Fortran-based open source environments (e.g. Scilab and Octave). Specifically, the clean and build-all process takes only ~5–8 s at the Netbeans 5.5 IDE. Similar is the required build time at the Eclipse development platform. We can contrast this with several (about 15–20) minutes required to run the configure script and the making process on a Linux Fedora Core 5 installation on a 3.2 GHz Pentium-4.

We have demonstrated the potentiality of jLab with the implementation of a SVM toolkit. Also, we have compared its performance with a C/C++ and a Matlab version and across different computing platforms (i.e. Linux, Sun/Solaris and 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 [10] and the WEKA data mining system that can easily provide an excessive set of routines for data preprocessing and visualisation [11]. Furthermore, we work on improving the parser in order to allow more flexible contructs, and improve the efficiency of the parsing phase, in order to be able to compete with C/C++ parser implementations (e.g. Scilab and Octave).

8 References

1 Kreyszig, E.: 'Maple computer guide for advanced engineering mathematics' (Wiley, 2000, 8th edn.)
2 Trott, M.: 'The Mathematica guidebook: programming' (Springer, 2004)
3 Higham, D.J., and Higham, N.J.: 'Matlab guide' (SIAM Computational Mathematics, 2005, 2nd edn.)
4 Campbell, S.L., Chancelier, J.-P., and Nikoukhah, R.: 'Modeling and simulation in Scilab/Scicos' (Springer, 2006)
5 Eaton, J.W.: 'GNU octave manual' (Network Theory Ltd, 2002)
6 Chonacky, N., and Winch, D.: '3Ms for instruction.: reviews of maple, mathematica and Matlab', *Comput. Sci. Eng.*, 2005, **7**, (3), pp. 7–13
7 Chonacky, N., and Winch, D.: '3Ms for Instruction.: Reviews of Maple, Mathematica and Matlab', *Comput. Sci. Eng.*, 2005, **7**, (4), pp. 14–23
8 Sreedhar, V.C., Brurke, M., and Choi, J.-D.: 'A framework for interprocedural optimization in the presence of dynamic class loading'. ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2000, pp. 196–207
9 Maassen, J., van Nieuwpoort, R., Veldema, R., Bal, H., Kielmann, T., Jacobs, C., and Hofman, R.: 'Efficient Java RMI for parallel programming', *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2001, **23**, (6), pp. 747–775
10 Heaton, J.T.: 'Introduction to neural networks with Java' (Heaton Research, 2005)
11 Witten, I.H., and Frank, E.: 'Data mining: practical machine learning tools and techniques' (Morgan Kaufmann Series, 2005, 2nd edn.)
12 Bigus, J., and Bigus, J.: 'Constructing intelligent agents using java: professional developer's guide' (Wiley, 2001, 2nd edn.)
13 Horstmann, C., and Cornell, G.: 'Core Java 2, Vol I – Fundamentals, Vol II – Advanced Techniques' (Sun Microsystems Press, 2005, 7th edn.)
14 Aho, A.V., and Ullman, J.D.: 'Principles of compiler design' (Addison-Wesley, 1977)
15 SVM-Matlab toolbox http://asi.insarouen.fr/~arakotom/toolbox/index.html
16 Vapnik, V.N.: 'Statistical learning theory,' Wiley, New York, 1998: 'An overview of statistical learning theory', '*IEEE Trans. Neural Netw.*, 1999, **10**, (5), pp. 988–999
17 Chang C.-C. and Lin C.J. LIBSVM: a library for support vector machines', 2001, http://www.csie.ntu.edu.tw/čjlin/libsvm
18 Osuna, E., Freund, R., and Girosi, F.: 'An improved training algorithm for support vector machines'. Proc. 1997 IEEE Workshop Neural Networks for Signal Processing VII, Amelia Island, FL, 1997, pp. 276–285
19 Joachims, T.: 'Making large-scale SVM learning practical' in Scholkopf, B., Burges, C.J.C., and Smola, A.J. (Eds.): 'Advances in

kernel methods – support vector learning' (MIT Press, Cambridge, USA, 1998)

20 Scholkopf, B., Smola, A.J., Williamson, R.C., and Bartlett, P.L.: 'New support vector algorithms', *Neural Comput.*, 2000, pp. 1207–1245

21 Scholkopf, B., Mika, S., Burges, J.C., Knirsch, P., Muller, K.-R., Ratsch, G., and Smola, A.: 'Input space versus feature space in kernel-based methods', *IEEE Trans. Neural Netw.*, 1999, **10**, (5), pp. 1000–1027

22 Cortes, C., and Vapnik, V.: 'Support vector networks', *Mach. Learn.*, 1995, **20**, pp. 1–25

23 Haykin, S.: 'Neural networks' (MacMillan College Publishing Company, 1999, 2nd edn.)

24 Scholkopf, B., and Smola, A.J.: 'Learning with kernels: support vector machines, regularization and beyond' (MIT Press, 2002)

25 Fan, R.-E., Chen, P.-H., and Lin, C.-J.: 'Working set selection using the second order information for training SVM', *J. Mach. Learn. Res.*, 2005, **6**, pp. 1889–1918

26 Rakotomamonjy, A., and Canu, S.: 'Learning, frame, reproducing kernel and regularization'. Technical Report TR2002–01, Perception, Systemes et Information, INSA de Rouen, 2002

27 Canu, S., Mary, X., and Rakotomamonjy, A.: 'Functional learning through kernels' in Suykens, J.A.K. (Ed.): 'Advances in learning theory: methods, models and applications, NATO Science Series iii: computer and systems sciences edition' (IOS Press, 2003) vol. 190, Ch. 5 pp. 89–110

28 Suykens, J.A.K., Van Gestel, T., De Brabanter, J., De Moor, B., and Vandewalle, J. (Eds.): 'Least squares support vector machines' (World Scientific, Singapore, 2002)