Scientific programming with an environment that combines effectively compiled and interpreted scripting at the Java platform

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 scripting language supports the basic programming constructs with Matlab like matrix manipulation operators. The jLab "core" provides the general purpose functionality with an extensive set of built in mathematical routines that cover all the basic numerical analysis tasks.
The important advantage of jLab compared to other similar environments is the potentiality to dynamically and automatically integrate Java code to the system in order to obtain both execution speed and to reduce the programming effort. This task is supported both by an easy to use extension Java class wizard and by application specific class wizards that automate the utilization of jLab's scientific libraries.
However, the incorporation of external Java general purpose code is not as convenient as the scripting code development is. Also, j-scripting is relatively slow compared to Groovy scripting that operates by compiling the scripts to Java classes. This was the motivation for the adaptation of the general purpose Groovy "scripting SuperJava" language as a parallel and cooperative scripting option in the jLab environment. The paper concentrates on the issues involved in the implementation of the multiscripting environment and on the benefits that can be obtained by the combination of these two very different scripting frameworks. The Groovy agile scripting language for the Java platform is both very flexible and powerful. We describe the modifications to the Groovy language and some of the most basic extensions that we have implemented in order to build the GroovySci language, the compiled scripting language of the jLab platform.

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

1 Introduction

Recently with the growing speed and potentiality of computers the populatity 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 (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: a. *computer algebra systems* that perform extensively symbolic mathematical evaluations (e.g. Maple [8], Mathematica [7]) and b. *matrix computation* systems that are oriented toward numerical computations and are well suited for engineering applications (e.g. the Matlab [10] that dominates at the commercial market and the open source "clones" Scilab [3] and Octave [9]). 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 (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 platform independence. The j-Lab scripting engine is implemented according to the principles presented in [1,2]. We have tested jLab on Linux, Solaris, MacOS X, and Windows XP and it runs in the same way, on all

these different environments, without any change of the code and without even requiring installation. All that it requires is the installation of the free and open source Java Runtime Environment (JRE) of Sun Microsystems to provide the Java Virtual Machine. jLab is open-source and can be downloaded from https://jlab.dev.java.net/.

However, even though the j-Script engine presented in [12,13] operates generally efficiently, there are cases where the speed of the j-Scripting is not sufficient and/or a more powerful and general purpose scripting language is required. This fact has formed the motivation of integrating the new Groovy programming language [14] within the framework of jLab as an alternative scripting engine. Thus the user benefits an important performance optimization when Groovy scripting is exploited instead of j-Scripting.

But the Groovy Scripting also benefits a lot when it is executed at the context of jLab. In addition to its extensibility jLab has a large set of basic classes integrated within its "kernel". First of all it has a powerful programming environment for writing scripts and using the domain specific wizards (e.g. ODE wizard [12,13]). Second, j-Scripting is more convenient for plotting and for simple scripts. Third, and perhaps more important, the Groovy programmer has access to a large library of Java numerical routines since Groovy can directly utilize any Java class [14] .

The numerical analysis functionality provided by the jLab kernel (i.e. built-in) routines is in fact very strong: jLab is based on the NUMAL numerical library [16]. The construction of this library was directed by P.W. Hemker of the Mathematisch Centrum at Amsterdam (Mathematical Centre - MC) and carried out by a large group of (numerical) mathematicians from the MC and the Dutch Universities of Amsterdam and Groningen.

The library initially was developed in Algol 60. Recently, H. T. Lau, has ported the library to Java and the source is available with his book [16]. The book describes clearly the interface of the functions and facilitates significantly the incorporation of the powerful NUMAL numerical machinery in jLab. Although Lau does not give many details on the numerical analysis algorithms involved, a subset of those algorithms are explained at the classical "Numerical Recipes" text [17]. Additionally, jLab incorporates the functionality of the WEKA machine learning and data mining framework [15], and some others Java coded scientific libraries. The set of available libraries can be easily and dynamically extended as described in [14]. Also, both the *j-Script* engine and the *Groovy* script engine can fully utilize the available scientific code.

The paper proceeds as follows: Section 2 describes the basics of the Groovy scripting framework. Section 3 describes the concept of the global variable workspace and the mechanics of variable sharing between the two scripting engines and the script engine switching. Section 4 presents an example where the two scripting engines are exploited for building efficiently a data mining application. The Java classes of the WEKA library are utilized. The Groovy scripting is generally more efficient for data preprocessing while the j-Scripting is more convenient for the interfacing to the WEKA libraries. Section 5 decribes the implementation of the Matlab-like matrix support in GroovySci. Section 6 lights on the GroovySci's internals by tracing some method invocations. Section 7 describes the basics of the autocompletion system, that is valuable for effective code development. Finally, the conclusions are presented along with directions for future work.

2 The Groovy Scripting Framework

The Groovy approach to scripting is very different from the j-Script one. 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) [1, 2] that fully represents all the code in the file.

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.
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 project http://groovy.codehaus.org/GroovyLab aims to achieve this. The Groovy version that is exploited in jLab benefits from the additional functionalities that are implemented with the GroovyLab project.

3. The Global Workspace and Script Switching

This section concetrates on the important subject of the implementation of the effective cooperative infrastructure between the two radically different scripting engines.
A global workspace keeps the variables that the user can directly evaluate and examine. The workspace for these variables is implemented within the j-Script engine implementation and they are shared by the Groovy environment.
The implementation allows Groovy scripting to coexist with jLab scripting and to handle a common workspace. The parameter sharing is accomplished with the ***binding*** mechanism of Groovy. A **Binding** object is used to keep the shared parameters.
The method **passBindingFromGroovyToJLab()** updates the jLab workspace with all the variables binded in the Groovy environment. This routine first obtains the reference to the jLab interpreter object. This reference then is used to retrieve the list of the jLab global variables. In order to get access to the Groovy's global variables we need to obtain a reference to the Groovy variable binding. This reference is used for obtaining a Map of the binded variables at the Groovy's context and subsequently a set view of the variables in the Map.
Subsequently this set view is used to iterate through the Groovy's variables. For each Groovy binded variable we get its name and we create a corresponding jLab global variable. Depending on the type of the Groovy variable we have to take different actions. If the Groovy variable is double [], i.e. a jLab vector, we need to create a jLab vector to take its value. This is performed by instantiating a *NumberObject* with the corresponding ***Vector*** (i.e. double []) as its value. Similarly the case of a ***Matrix*** (i.e. double [][]) return type is handled. The following code snipet helps to obtain a more specific view of the interfacing code.

```
// updates the jLab workspace with all the variables binded in the Groovy environment
public static void passBindingsFromGroovyToJLab() {
if (GlobalValues.enablePassParamsFromGroovyTojLab) {
Interpreter jLabInterpreterObj = jExec.jLab.jLab.interpreter; // object to jLab interpreter
VariableList currentVars = GlobalValues.getGlobalVariables(); // get the current list of variables
Binding groovyBinding = GlobalValues.groovyBinding; // get the reference to the Groovy variable binding
Map variables = groovyBinding.getVariables(); // get a Map of the binded variables at the Groovy's context
Set bindElemsSet = variables.keySet(); // return a set view of the variables in the Map
Iterator bindedElemsIter = bindElemsSet.iterator(); // iterate through the Groovy's variables
while (bindedElemsIter.hasNext()) {
String currentVarName = (String) bindedElemsIter.next(); // get the name of the Groovy's variable
Variable var = currentVars.createVariable(currentVarName); // create a jLab corresponding variable
Object currentVarValue = variables.get(currentVarName); // get the value of the Groovy's variable
if (( currentVarValue instanceof double [])) // a Vector
{
var.assign(new
```

```
jExec.Tokens.NumberObject((double
[])currentVarValue));
        }
    else
       if (( currentVarValue instanceof  double [][]))
    {
                                        var.assign(new
jExec.Tokens.NumberObject(((double  []
[])currentVarValue)));
    }
    else  {  // floating point or String
        if  (currentVarValue instanceof  Integer)
        {
                          int   varValue  =  ((Integer)
currentVarValue).intValue();
                                        var.assign(new
jExec.Tokens.NumberObject((double) varValue));
        }
        else if ( currentVarValue instanceof Long)  {
                          long   varValue  =  ((Long)
currentVarValue).longValue();
                                        var.assign(new
jExec.Tokens.NumberObject((double) varValue));
        }
        else if ( currentVarValue instanceof Float)  {
                          float   varValue  =  ((Float)
currentVarValue).floatValue();
                                        var.assign(new
jExec.Tokens.NumberObject((double) varValue));
        }
        else if ( currentVarValue instanceof Double)  {
                        double   varValue  =  ((Double)
currentVarValue).doubleValue();
                                        var.assign(new
jExec.Tokens.NumberObject((double) varValue));
        }
         else //  (currentVarValue instanceof String)  //
simple scalar numeric values  or Strings can be treated
as Strings and passed to jLab
           {
                                  String  varValue  =
currentVarValue.toString();  // get the value of the
Groovy's variable as a String
                                        var.assign(new
jExec.Tokens.StringObject(varValue));  // assign this
value to the jLab workspace
              }
          }  // floating point or String


      }
    }
  }

   //  makes the data of the jLab's workspace available
to Groovy
                          public  static  void
passBindingsFromJLabToGroovy() {
                                                  if
(GlobalValues.enablePassParamsFromJLabToGroovy)
{
                              Binding  groovyBinding  =
GlobalValues.groovyBinding;  // get the reference to
the Groovy variable binding
     Variable var;
                             VariableList  currentVars  =
GlobalValues.getGlobalVariables();  // get the current
list of variables
     Iterator jLabVarIter = currentVars.getIterator();
     while (jLabVarIter.hasNext())
               {
                                  Map.Entry  next  =
((Map.Entry)jLabVarIter.next());
     var = ((Variable)next.getValue());
          jExec.Tokens.DataObject  varData  =
(jExec.Tokens.DataObject)var.getData();
      if (varData != null) {
   String varName = var.getName();
   if (varData instanceof  NumberObject)  { // number
object check to see if it is an array, vector or simply
scalar
          double  [][]  valsAll  =  ((NumberObject)
varData).values;
   int valsXLen = valsAll.length;
   int valsYLen = valsAll[0].length;
   if ( valsXLen > 1 || valsYLen > 1) {  // array
        if  (valsXLen  ==  1  ||  valsYLen  ==  1)  {  // a
vector
       double [] valsVecAll = valsAll[0];
                groovyBinding.setVariable(varName,
valsVecAll);
      }  // a vector
       else  // a matrix
      groovyBinding.setVariable(varName, valsAll);
      }  // array
   else  {  // a simple scalar
                                        double  val  =
((NumberObject)varData).values[0][0];
        groovyBinding.setVariable(varName, val);
          }
        } // number object
        else
                 groovyBinding.setVariable(varName,
(String) varData.toString());
                }  // varData != null
      }  // jLabVarIter.hasNext()
                                      }                 //
GlobalValues.enablePassParamsFromJLabToGroovy
  }

}
```

In order to demonstrate the coexistence and cooperation of the two scripting frameworks we present a simple benchmark script that simply computes a large two-dimensional matrix with elements $a_{ij}=i*j, 0<i<2000, 0<j<500$. The j-Script code that performs this computation is listed below:

```
# tic; k=1; m=1; while (k<2000) { while (m<500) { a(k,m)=k*m; m++; } k++; } tm=toc;
```

The time for the execution of this loop is about tm = 1.769. This execution time can be much smaller with the exploitation of the compiled Groovy scripting. The symbol '@' is used to switch the jLab's mode to Groovy scripting and the symbol '`' to return to j-Scripting. The corresponding code and execution time follows:

```
# tic; @ k=1; m=1; a = new double[2000][1000]; while (k<2000) { while (m<1000) { a[k][m]=k*m; m++; } k++; } ` tm=toc;
```

The execution time at the same Pentium Dual-Core 2.0 Gz PC is tm = 0.052 i.e. about 35 times faster. For even more complex code, the speedup improvement of Groovy scripting becomes even more significant and improvements of 200 to 500 times faster are usually obtained.

4. Example of the multi-scripting at a data mining application

The example application demonstrates the utilization of the multiscripting facility for exploiting both the flexibility of j-Scripting and the generality and speed of Groovy scripting at the context of a data mining application.

Jlab utilizes the WEKA [17] machine learning and data mining environment in order to have an effective library plug-in for data mining and computational intelligence applications. Since WEKA is entirely in Java and open-source, it is a straightforward task to utilize its machine learning models both from the j-Scripting engine and from the Groovy framework. The example demonstrates the application of the WEKA's Multilayer Perceptron implementation, with a simple data preprocessing in Groovy. This data preprocessing simply keeps those training instances that have a positive third attribute, i.e. data[k][2] >0, for all k. Groovy is a powerful language for performing effectively complex data preprocessing operations and therefore it fits well in the jLab's multiscripting framework.

The example code is presented below:

```
clear("all");
dataFile = getFileNamePathOpenDialog("Please specify your data file");
data = ReadARFFFile(dataFile);
[N M] = size(data);
// switch to Groovy mode in order to perform efficient data filtering
@
// use Groovy scripting in order to perform flexible and efficient data filtering
cnt=0; k=0;
while (k<N) {
  if (data[k][2] > 0)  cnt++;
  k++;
}
filteredData = new double[cnt][M];
k=0; cnt=0;
while (k<N) {
 if (data[k][2] > 0)
   for (m=0; m<M; m++)
      filteredData[cnt][m] = data[k][m];
    k++;
 }
// the WEKA preprocessing stage has finished
// switch now to j-Script mode
% use half of the data for training and half for testing
trainData = filteredData(1:2:N, :);
testData = filteredData(2:2:N,:);
ClassIdx = M;
TestingClassLabel = testData(:,ClassIdx);
//learningRate = getNumberDialog("Enter learning rate", "0.1");
//momentum = getNumberDialog("Enter momentum", "0.01");
  % This will set what the hidden layers are made up of when auto build is
  % enabled. Note to have no hidden units, just put a single 0, Any more
  % 0's will indicate that the string is badly formed and make it unaccepted.
   % Negative numbers, and floats will do the same. There are also some
  % wildcards. These are 'a' = (number of attributes + number of classes) / 2,
  % 'i' = number of attributes, 'o' = number of classes, and 't' = number of
```

```
% attributes + number of classes.
% param h A string with a comma seperated list of numbers. Each number is
% the number of nodes to be on a hidden layer.
//hiddenLayerString = getStringDialog("'a' = (number of attributes + number of classes) / 2, 'i' = number of attributes, 'o' = number of classes, and 't' = number of
//attributes + number of classes. param h A string with a comma seperated list of numbers. Each number is the number of nodes to be on a hidden layer.");
nepochs = 500;
decayString = "false";
configureParameters = 1;  // present the MLP network parameter configuration screen prior to training
uiOnString = "true";
% train the Multilayer Perceptron
rs = MLPNet(trainData, configureParameters); // hiddenLayerString, learningRate, momentum, nepochs, decayString, uiOnString);
% now evaluate its performance on the test data
evalNet = MLPNetEval(testData);
classesPredicted = round(evalNet);
successCnt = sum(TestingClassLabel==classesPredicted');
successRatio = successCnt/length(TestingClassLabel);
disp('MLPNet successRatio = '+successRatio);
figure(1); xaxis = 1:1:length(evalNet); plot(xaxis, TestingClassLabel','g');
hold("on"); plot(xaxis, evalNet,'r'); title("green: actual, red: predicted");
```

The example above demonstrates the switching to Groovy scripting with the control character '@' (at the boldfaced text) in order to preprocess the data. After, this preprocessing we return again to j-Script mode, where it is more convenient to call the WEKA classes for performing data mining.

5. The Matlab like matrix support

The support for high-level Matlab like matrix operations is essential for any mathematical programming environment.

Groovy provides the hooks on which to implement effective matrix operations at the language level. The language bases its operators on method calls and allows to override them in order to implement subtype specific behavior.

Thus, the addition operator, i.e. A + B, is overrided to perform matrix addition, by implementing the method plus, i.e. *A.plus(B)*. Similarly, the other basic mathematical operations, A – B, A * B, A / B, A % B, A ** B are overloaded by implementing the methods:

A.minus(B), A.multiply(B), A.div(B), A.mod(B), A.power(B).

In order to provide convenient array indexing operations it is necessary for the Matrix class to override the array indexing, i.e. the subscript operation []. This is achieved in Groovy by implementing the methods *A.getAt(idx)*, *A.putAt(idx, elem)*, where *idx* denotes the array index and *elem* is the element to put.

Also a large number of high level matrix operators are implemented in GroovySci. Some of these operators are:

Matrix + Matrix
Matrix + Number
Matrix - Matrix
Matrix - Number
*Matrix * Matrix*
*Matrix * Number*
Matrix / Matrix
Matrix / Number
*Matrix ** int"*

Static operators on the *Matrix* object also implement the basic linear algebra operators. Some of these static operators are:

sum(Matrix)
prod(Matrix)
cumsum(Matrix)
cumprod(Matrix)
inverse(Matrix)
solve(Matrix A, Matrix b), returns X Matrix verifying A*X = b.
rank(Matrix)
trace(Matrix)
det(Matrix)
cond(Mtrix)
norm1(Matrix)
norm2(Matrix)
normF(Matrix)
normInf(Matrix)
Cholesky_L(Matrix)
Cholesky_SPD(Matrix)
QR_Q(Matrix)
QR_H(Matrix)
QR_R(Matrix)
LU_L(Matrix)
LU_U(Matrix)
LU_P(Matrix)
Singular_S(Matrix)

Singular_U(Matrix)
Singular_V(Matrix)
Singular_values(Matrix)
Eigen_D(Matrix)
Eigen_V(Matrix)
random(int, int), independant random values (between 0.0 and 1.0) Matrix of given size, alias to rand(int, int) *random(int, int, double min, double max)*
randomUniform(int m, int n, double min, double max)
randomDirac(int m, int n, double[] values, double[] prob)
randomNormal(int m, int n, double mu, double sigma)
randomChi2(int m, int n, int d)
randomLogNormal(int m, int n, double mu, double sigma)
randomExponential(int m, int n, double lambda)
randomTriangular(int m, int n, double min, double max)
randomTriangular(int m, int n, double min, double med, double max)
randomBeta(int m, int n, double a, double b)
randomCauchy(int m, int n, double mu, double sigma)
randomWeibull(int m, int n, double lambda, double c)

The following static sort/find methods are available:
sort(Matrix)
sort(Matrix, int columnIndex)
min(Matrix)
max(Matrix)

The following static transformation methods are available:
transpose(Matrix)
alias to t(Matrix)
resize(Matrix, int, int)
rowsMatrix >> Matrix
appends rowsMatrix to Matrix at last position (i.e. add last row)
columnsMatrix >>> Matrix
appends columnsMatrix to Matrix at last position (i.e. add last column)
Matrix << rowsMatrix
appends rowsMatrix to Matrix at first position (i.e. add first row)

The following static statistic sample methods are also available:
mean(Matrix)
variance(Matrix)
covariance(Matrix,Matrix)
correlation(Matrix,Matrix)

6. Tracing Invocations

In order to understand the mechanics involved at the GroovySci's compiled scripting we trace some distinctive method calls. Suppose that the user requires the computation of the expression:

*b = sin(4*a)*

where a was computed previously with the command:

a = rand(4,5)

i.e. is a Matrix object, of dimension 4x5, filled with pseudorandom numbers. Since a was computed exists in the global binding of variables. The following steps are involved:

1. A script is prepared named ScriptN.groovy where N corresponds to an auto-generated script numbering, e.g. the name is Script2.groovy for the second generated script.
2. The ScriptN.groovy script is compiled.
3. The compiled class is cached and is returned.
4. At InvokerHelper:
4a. The InvokerHelper creates an object of ScriptN class.
4b. It sets the current binding for the execution of the current script.(Script Execution Phase)
5. Resolve the class Matrix for the method sin().
6. Resolve the class Integer for the int argument 4.
7. Get the variable a from the current binding.
8. Invoke the method multiply() on the Groovy object Int{4} and with as arguments of the method the matrix a.
9. the object is of class Integer. The metaclass registry is invoked to find the metaclass for the Integer class.
10. the metaclass is used to invoke the method “multiply” on the object “4” and with

arguments the Matrix a.
11. the invoke() method on the Integer object "4" delegates the call to the invoke() method of the Integer class.
12. the method "multiply" is retrieved for the "Integer" target class and with arguments the Matrix.
13. the method is retrieved from the hash map of meta-methods. These methods should be defined in the DefaultGroovyMethods class.

Let's also trace what happens with the command
d[7][9] = 8.5
1. Load class BigDecimal for the right hand side
2. The method *getGroovyObjectProperty* for the property d sets a parameter *GlobalValues.currentPropertyProcessed= messageName.*
In the particular case the messageName is the variable being processed i.e. the "d" variable.
3. The *receiver.getProperty(messageName)* call returns the value of the matrix d.
4. An invocation of the *getAt()* method is performed on the object d and arguments 7.
5. The invoke *Plain Old Groovy Object* method is called with parameters object, methodName, and {7}, i.e. (invokePogoMethod(object, methodName, {7})
6. Call the
InvokerHelper(receiver, messageNAme, messageArguments)
where
receiver: the matrix M
mesageName: getAt
messageArguments: 2
7. the InvokerHelper calls
invokePojoMethod(Object object, String methodName, Object arguments)
object: Matrix M
methodName: "getAt"
arguments: 2
8. The Matrix M is a GroovyObject. Thus the InvokerHelper calls
GroovyObject groovy = (GroovyObject) object; groovy.getMetaClass().invokeMethod (object, methodName, asArray(srguments));
9. The metaclass of the GroovyObject is implemented with the MetaClassImpl class. This metaclass object calls
i*nvokeMethod("M" "getAt", "{2}")*

Subsequently, the invokeMethod() code calls the important routine
getMethodWithCaching()
in order to try to use the cache to find the method.
The caching system uses the MetaMethodIndex class where the important DefaultGroovyMethods are cached.

6. Figure Handling

A very important issue for mathematical programming environments is to have effective figure plotting facilities.

In *GroovySci*, a figure "manager" class performs the necessary bookeeping and updates a set of structures in order to present to the user a Matlab-like figure interface. Each figure is represented by a *FrameView* object. The Matlab like subplots are represented with *PlotPanel* objects. A 2-D array of *PlotPanel* objects holds the subplots associated with each figure, i. e. *FrameView* object. A handle where plot operations are directed is kept with the currentPlot variable that points to the current FrameView object.

Figure objects are distinguished as either 2D Figure objects or 3D ones. The *newPlot2D()* creates a new 2D figure object if either we do not have a current 2D ploting panel or a new figure is explicitly requested. A figure panel that is not splitted in subplots it is handled as if an explicit subplot2D(1,1,1) command was issued.

The Plot2DCanvas class controls whether the mouse operations perform a rotation, a zoom or a translation. Each axes can be set either to LINEAR or to LOGARITHMIC mode. The Canvas class implements the operations for adding plots to it, e.g. addGridPlot(), addCloudPlot() etc. An *AWTDrawer*() component is used for plotting within a canvas. The AWTDrawer() class uses a Projection class for the implementation of the necessary projections.

7. Autocompletion

Java is fast, robust, extensible, portable and an open platform for the academic community (recently it become also and open-source). These

facts are some of the basic reasons for the leading popularity of the Java language and platform. Large high quality scientific libraries by the academic and engineering community in Java. The aim of jLab is to exploit effectively this huge base of outstanding work.

Therefore, jLab explores three famous libraries of scientific code:

a. the *NumAl* library for numerical analysis
b. the WEKA machine learning framework.
c. the jSci scientific library.

The AutoCompletion system automatically retrieves information for the computational classes of these libraries. The Java reflection API is used to dynamically interrogate the classes for their methods and to provide the method prototypes dynamically to the user. Also, since methods are available with their short names (e.g. eig() for eigenvalue computations) and methods of similar functionality many times exist in different libraries (e.g. eigenvalue related methods exist both in NumAl and in jSci), the public methods of the scientific libraries are indexed and with their short name first then the package name.

Shell control commands can be implemented easily in GroovySci. A class *groovySci.BasicCommands* is used to implement a set of basic shell control commands. This class is then imported implicitly by the interactive shell and thus the implemented methods become available as user commands.

For example to implement the *dir* command a static method dir is implemented in the groovySci.BasicCommands class.

8 Conclusion

The paper concerned a powerful scientific programming environment that explores two cooperative scripting languages: the *j-Script* scripting engine that is executed by an interpreter implemented in the Java language and *the Groovy* "scripting super-Java" scripting Engine.

The main scope of the paper was to demonstrate the potential of the cooperation of these much different scripting engines.

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 j-scripting code and in stages replacing script code with Groovy code that can be compiled to Java class code or 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, *The 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] Ian H. Witten, Eibe Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*, Morgan Kaufmann Series, 2005.

[16] Hang T. Lau, *A Numerical Library in Java for Scientists and Engineers*, Chapman & Hall/CRC, 2003.

[17] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes in C++, The Art of Scientific Computing*, Second Edition, Cambridge University Press, 2002.