VOL. 10, NO 20, NOVEMBER, 2015 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences

www.arpnjournals.com

MATLAB INTEGRATION INTO JAVA-BASED EXPERT SYSTEMS FOR PHYSICAL PROCESSES MODELING

Dmitry Potapov[1,2]
[1]National Research Nuclear University "MEPHI", Kashirskoe highway, Moscow, Russia
[2]The European Laboratory for Particle Research CERN, Route de Meyrin, Geneva, Switzerland
E-Mail: div-x15@yandex.ru

ABSTRACT

A technique for dynamic Matlab functions integration into Java is proposed; overcoming the constraints imposed by languages involved, such as the need to recompile Java-project, the conversion of m-functions into Matlab scripts, and passing Matlab array as a list of parameters into Java methods.

Keywords: matlab, Java integration, mathematical modeling, dynamic code integration, reflection.

1. INTRODUCTION

The need for dynamic integration of Matlab code into Java was faced by the author while developing an expert system for the electrolyte solutions' properties' modeling and prediction (Potapov *et al*., 2011). The system provides the following functionality:

a) storing and providing access to the tables of experimental data, containing the values of physicochemical properties (vapor pressure P, the activity of the solvent a, osmotic coefficient φ, the activity coefficient γ ±, solution density ρ, the excess volume V, Gibbs energy G, excess enthalpy H) at various concentrations (molality m, molarity c, molnodolnoy x, weight percent w) and fixed under the experimental conditions (pressure P, temperature T);

b) access to the results of modeling - model's fitness characteristics and parameters' estimates;

c) access to the models of the thermodynamic properties containing model equations and data processing software;

d) modeling of electrolyte solutions' thermodynamic properties;

e) plotting various correlations;

f) calculation of the solutions' physicochemical properties at concentrations for which no experimental data exists.

Considering the requirement for the system to be cross-platform, Java was selected as the programming language for the implementation of the basic modules of the system and writing the GUI. We chose MySQL as DBMS for experimental values' and simulation results' storage.

The main distinction of the described system from existing analogues (Elliott, Lira, 1999; Koretsky, 2004; Kyle, 2005; Sandler, 2006) is the possibility of adding new models into it. Thus, the system is not only designed to simulate chemical processes with existing models, but also to check the adequacy of new models.

Chemical solution simulation might be a very time-consuming procedure, which often involves complex mathematical operations, such as differentiation (not necessarily numerical, but sometimes analytical), integration, local and global multi-variable function optimization, etc. (Rudakov *et al*., 2011) to perform such operations requires sufficiently powerful mathematical package with a built-in ability to integrate with Java for cross-platform portability. Matlab complies with these requirements, providing functions which allow for arbitrary complexity of the mathematical models (Kotkin, Cherkasskiy, 2001). Therefore, we chose it as the language in which the model equations imported into the system are written. In addition, the system is chemist-oriented, so the user often has experience with Matlab but is unlikely to be familiar with the multi-disciplinary programming languages.

Despite the availability in the latest version of Matlab of a tool to convert m-files containing Matlab functions to Java classes, building a system with the possibility of adding new models poses a number of issues, which are further discussed in detail along with suggested solutions.

2. MATERIALS AND METHODS

2.1. m-files to Java-files conversion in Matlab

In Matlab a tool called "deploytool" allows to convert the functions contained in the m-files in the Java classes. The original function is transformed into resulting class' method of the same name. Matlab compiler is used

for the conversion, which adds only those functions and modules to the resulting Java project which are used in the m-file being converted. However, information on the use of a particular function is not always available at compile time, because Matlab supports mechanisms for dynamic function calls at runtime using the functions eval / evalin. In the case of their use in deploytool user must explicitly specify which other m-files can be called from the m-file in question (Ksu, 2005). The conversion process is as follows: first, the compiler finds all the functions called from the converted file. After that it iterates through each of them individually and looks for a deeper level functions that are called from them. Thus, the compiler implements the wave-algorithm, and the final project includes only those functions that are necessary for it to run. However, in the case of a system for new mathematical models adequacy verification it is impossible to say at compile time what functions will be called, because model for the calculation is determined interactively. In this situation the user enters the equations of the newly created model, which does not exist at compile time. Therefore, a technique to overcome these limitations is required.

2.2. Adding new model equations dynamically

We have described above the general algorithm for converting m-files into Java classes. This section discusses its inherent limitations and methods of overcoming them.

2.2.1. The need to re-compile the project after adding new models

Since it is necessary to explicitly specify the functions to be included into the project, if new m-files are added after the compilation, a full re-compilation of all Matlab code used in the system is required after which the classes in the corresponding jar-archive should be accordingly updated. This procedure significantly reduces the flexibility of the application and, moreover, requires an installed version of Matlab. Moreover, following all these steps may be rather difficult for a user-chemist.

When using eval () function in Matlab code for compilation it is not always required to explicitly specify all functions called from it. If it is expected that a valid Matlab statement will be passed to eval () as an argument, containing only calls to the functions of the main Matlab module (i.e., without references to additional packages of instruments, such as the Image Toolbox, Database Toolbox, etc.), eval () can be compiled without any explicit indications of called functions. Inability to use additional tools is not a big limitation since usually equation models are mathematical functions with standard operations, implemented in the main Matlab module (including integration and differentiation). Thus, instead of calling a new user-defined function, its whole body (without signature) can be passed to eval ().

2.2.2. Matlab functions to Matlab scripts conversion

In the case of complex model equations, they are often separated into several sub-functions contained in one m-file. These sub-functions might be nested into the outer ones. Passing the main function body with the inner functions contained within it as an argument to eval() leads to a runtime error. The argument to eval() should be a valid Matlab statement, which is not the case for a string that contains within itself a function (declared using the word "function"). A Matlab statement is any sequence of characters that can be successfully executed from Matlab command line. It is not allowed to declare functions (defined with the word "function") from the command line in Matlab. Thus, the initial objective is to convert the function described in the m-file into Matlab statement (Matlab script).

This problem can be solved using anonymous functions supported in Matlab. They do not contain the word "function" in their declaration, and can be declared from the command line. This creates a function handle, which can be subsequently used with the same syntax as the functions defined in the m-file. The anonymous function declaration is done as follows:

$F = @(x_1,x_2,\ldots)\ f(x_1,x_2,\ldots)$

where F is an arbitrary name of the variable, storing the function handle, $(x_1,x_2,\ldots)$ – list of function arguments, $f(x_1,x_2,\ldots)$ - required function.

Consider an arbitrary inner Matlab function, contained in the same m-file which, in its turn does not contain nested functions. We shall assume that it does not contain loops and branching statements. This is based on the assumption that the function in the developed system represents the equation of a mathematical model written using standard mathematical operations. In addition, we assume that the values of the arguments and global variables do not change during the function execution. This somewhat limits the range of applicability of this approach, but in the case of arbitrary m-file a general conversion algorithm has not been found, moreover, this restriction only concerns inner functions and does not affect the main function. General view of the function has a structure similar to the following:

```
function f = F(x1,x2,...)
  global b1, b2, ...
    %...
    tmp1       =       f1(b1,b2,...,a1,a2,
...,x1,x2,...);
    tmp2       =       f2(b1,b2,...,a1,a2,
...,x1,x2,...);
    %...
    tmpn       =       fn(b1,b2,...,a1,a2,
...,x1,x2,...);
    f = fn+1(tmp1,tmp2,...);
end
```

where xi - function arguments, bi - global variables, ai - local variables. We will now transform this function into anonymous. Using eval() for this transformation is not allowed, as a nested call of eval () inside eval () causes numerous problems. First, the function has to be rewritten in the form where each local variable changes after initialization not more than once. To achieve this, variables that do not meet this condition need to be replaced by new ones starting from the place at which they change the second time. After that we shall consistently reduce the number of variables used by specifying the corresponding expressions inline. As a result, there will be only one line, which is an explicit expression for the return value, depending on the arguments and environment variables (the latter are converted into ordinary function arguments). Then, the variable representing the return value is replaced with an anonymous function of the same name. For example, the following line

$F = f(in_1, in_2, ...)$

will be transformed into:

$F = @(in_1, in_2, ...) f(in_1, in_2, ...)$

The described procedure is applied for all inner functions. In case of meeting the above-mentioned requirements, no collisions or errors occur. After that, all generated anonymous functions are placed in the body of the main function immediately after the signature. Their order must match the order of their appearance in the generated script. The line in which the variable is assigned the return value is also converted to an anonymous function. In the resulting file the signature and the reserved word "end" are deleted. Finally, the name of the function handle in the resulting script is replaced by "model_equation". The purpose of this replacement will be explained in section 2.4.

2.2.3. The case of initial presence of anonymous functions in the m-file

In case of initial presence of anonymous functions in the m-file, the procedure described above causes an error. Within the outer function, they have full access to its arguments, but by removing the signature the information about the arguments is lost. Adding removed arguments into the parameter list cannot be done, since often such functions represent integrands. Their single parameter is the variable of integration, and in case of multiple parameters an error is thrown. A possible solution to this problem is to put the whole expression of the anonymous function inline without allocating an additional variable for it. Then, the arguments of the outer function will be accessible within the scope of this inner anonymous function without adding them to its arguments list.

2.2.4. Passing an array as a list of function arguments in Java

After the final m-file conversion, its resulting code can be stored in the mathematical models database. The procedure for using stored equations is simple. A java-class, generated by compilation of the following m-file is included into the project:

```
function f = eval_equation(s,varargin)
eval(s);
f = model_equation(varargin{:});
end
```

A string *s* containing the equations of the model, taken from the database is passed to this function as an argument, along with the model parameters values (number of which can be variable). Due to the fact that the name of the model function stored in the database is always "model_equation", its specific implementation is not important. Thus this code works for any model retrieved from the database and calling it does not require project re-compilation.

Here we face a problem of passing the parameters to this function in Java. The parameter values are retrieved from the database as an array, and the number of elements in it can vary. However, to call the described function, one must specify the required parameters sequentially, separated by commas. There is no direct mechanism to convert an array of arbitrary length into a list of function arguments. One option would be to change eval_equation () signature, making it accept an array as an argument, but this approach is inconvenient for users of the developed system, since using single array-variable instead of parameters list does not allow assigning intuitive names to the arguments, corresponding to their physical meaning (those are parameters of a mathematical model describing some physicochemical processes). In addition, often the user-chemist does not have the experience of working with arrays in Matlab. Given the above, the problem of passing parameters in Java requires a different approach.

The issue was solved by using the mechanism of reflection, whose methods are contained in java.lang.reflect package. It has a function of dynamic invocation of a class method, and it takes as an argument an array of parameters of the method to be called. Thus, the transformation of the array into the list of arguments is not necessary. The function call using reflection looks as follows in Java:

```
result = (Object[])
equationEvaluatorInstance.getClass().ge
tDeclaredMethod("eval_equation",
int.class,
Object[].class).invoke(equationEvaluato
rInstance, 1,params);
```

The equation Evaluator Instance object is an instance of the class created from m-file function eval_equation (). The class created by Matlab contains a method whose name matches the original function name ("eval_equation"). Using reflection, the method is found by calling getDeclaredMethod (), after which it is invoked by calling invoke (). invoke () takes as arguments the instance of the class whose method is invoked dynamically, the size of the output array of results and an array of arguments (*params*). Without reflection one might have used conditionals like:

```
if (params.length == 1) {
result           =           (Object[])
equationEvaluatorInstance.eval_equation
(1,params[0]);
}
else if (params.length == 2) {
result           =           (Object[])
equationEvaluatorInstance.eval_equation
(1,params[0],params[1]);
}
//etc.
```

However such structure is cumbersome and is limited to small maximum number of parameters while reflection supports an arbitrary number of them.

3. RESULTS

The technique described was implemented and integrated into the water solution modeling system. It is currently used to study a newly proposed cluster model of hydration in electrolytes. In this section an example of m-file conversion into Matlab script for storing it in the database is provided.

3.1. Using the proposed technique of Matlab code dynamic integration into Java

Consider an m-file with the following source code:

```
function f=model8_G(q1,q2,y,h1,r1)
global q
    q=q1+q2;
    f=exp(lng(q1,q2,y,h1,r1));
end

    function f2=lng(q1,q2,y,h1,r1)
    global q
        J=@(t)
(model8_FEE(t,h1,r1,h2,r2,As,B)-1-
fe_D(B,t))./(t);
f2=   q1/q   +   q2/q   +   h1   +   r1   +
quadv(J,0.0000001,y);
    end
```

quadv () is a built-in numeric integration function from the main Matlab module.

The conversion starts with the inner function:

```
J=@(t) (sin(t+h1+r1)-1/t))./(t);
lng = @(q1,q2,q,h1,r1,y) q1/q + q2/q +
h1 + r1 + quadv(J,0.0000001,y);
```

Such code is not correct. To make it valid, we put the integrand inline:

```
lng = @(q1,q2,q,h1,r1,y) q1/q + q2/q +
h1   +   r1   +   quadv(@(t)(sin(t+h1+r1)-
1/t))./(t),0.0000001,y);
```

At each step of the algorithm the code is becoming more difficult to read, but since it is not intended for viewing by a user but is stored in the database for calculations, its readability is not important. It should be noted that the original code is also stored in the system so that the user may view and modify it if necessary.

After the processing of inner functions the main function should be transformed:

```
function f=model8_G(q1,q2,y,h1,r1)
    lng  =  @(q1,q2,q,h1,r1,y)  q1/q  +
q2/q      +      h1      +      r1      +
quadv(@(t)(sin(t+h1+r1)-
1/t))./(t),0.0000001,y);
    q = q1 + q2;
    f                  =@(q1,q2,y,h1,r1)
exp(lng(q1,q2,q,y,h1,r1));
end
```

The return variable is renamed into "model_equation", the signature of the function and the word "end" are discarded, resulting in:

```
lng = @(q1,q2,q,h1,r1,y) q1/q + q2/q +
h1 + r1 + quadv(@(t)    sin(t+h1+r1)-

      1/t))./(t),0.0000001,y);
q = q1 + q2;
model_equation    =    @(q1,q2,y,h1,r1)
exp(lng(q1,q2,q,y,h1,r1));
```

Finally all "carriage return" symbols are deleted from the obtained script, since they are not allowed in valid Matlab statements and the code is saved in the database.

4. DISCUSSIONS

In this paper a technique for dynamic integration of mathematical models equations implemented in Matlab

into Java is proposed. When using m-files directly with Java in developing expert systems, a complete project recompilation is required after adding a new model written in Matlab, resulting in strong system usability limitations. Converting the source code of models into Matlab scripts using the described approach with their subsequent passing to the Matlab eval () function as an argument makes it possible to overcome this issue. The procedure in question is based on consequential replacement of inner functions by anonymous. The built-in Java mechanism of reflection enables passing the retrieved Matlab array as a list of parameters to the called function. The proposed technique has been implemented in the water solution modeling system and is currently used to study the newly proposed cluster hydration model.

REFERENCES

Elliott J.R., Lira C.T. 1999. Introductory Chemical Engineering Thermodynamics, Prentice-Hall. pp. 693-712.

Koretsky M.D. 2004. Engineering and Chemical Thermodynamics, John Wiley and Sons. p. 553.

Kotkin G.L., Cherkasskiy V.S. 2001. Computer modeling of physical processes using MATLAB-Novosibirsk: Novosibirsk university. pp. 89-92.

Ksu D. 2005. Matlab interaction with ANSI C, Visual C++, Visual BASIC and Java. - M.: Piter. pp. 292-312.

Kyle B.G. 2005. Chemical and Process Thermodynamics, Prentice-Hall, Englewood Cliffs. pp. 746-751.

Potapov D.A., Modyaev A D., Rudakov A.M. 2011. Computer modeling and prediction of electrolytes solution properties. Information systems and technologies. № 6 (68): 57-66.

Rudakov A.M., Maikova N.S., Sergievskiy V.V. 2011. A cluster-model-based study of solvation and association in binary solutions. The problems of solvation and complexing in solutions: Proceedings of XI international conference - Ivanovo. p. 14.

Sandler S.I. 2006. Chemical, Biochemical, and Engineering Thermodynamics, Wiley, 4th edition. p. 945.