GRAPHICAL SIMULATION OF NUMERICAL ALGORITHMS
An Aproach based on Code Instrumentation and Java Technologies

Carlos Balsa, Luís Alves, Maria J. Pereira, Pedro J. Rodrigues and Rui P. Lopes
Polytechnic Institute of Bragança, Campus de Sta. Apolónia, Apartado 1134, Bragança, Portugal

Keywords: E-Learning Tool, Numerical Methods, Octave, Code Instrumentation, Inspector Functions, XML, OpenGL, Website, Servlet, Open Source.

Abstract: We want to create a working tool for mathematics teachers and a corresponding learning tool for students, namely a graphical simulator of mathematical algorithms (GraSMa). To achieve it we try two different strategies. We started by annotating manually the original algorithm with inspector functions. Now we are testing a new approach that aims at automatically annotating the original code with inspector functions. To achieve this we are developing a language translator module that enables us to comment automatically on any code written in Octave language. The run of the annotated code gated by one of these two ways, records in a XML (eXtensible Markup Language) file everything that happened during the execution. Subsequently, the XML file is parsed by a Java application that graphically represents the mathematic objects and their behaviour during execution. The final application will be accessed on-line through a website (WebGraSMa) which is currently under development. In this paper we report and discuss about the procedures followed and present some intermediate results.

1 INTRODUCTION

The human visual system constitutes a massively parallel processor that provides the highest bandwidth channel into human cognitive centers. Considering this fact, we can easily assume that visual representations of data amplify cognition, leveraging perception and understanding of complex ideas with clarity, precision, and efficiency. As an extension to this concept, it is clear that geometric representation of mathematical concepts will provide a valuable method to help interpret and understand them. From this perspective, numerical methods can no longer be seen as a sequence of lines of code, but a rich set of moving graphical objects.

In this context, we are developing an open source tool (Graphical Simulator of Mathematical Algorithms - GraSMA) that can be used by teachers and students in different mathematical classes. GraSMA will help to understand concepts as approximated solution, iteration, convergence, error, etc.

Currently there are several software modules in the field of mathematics education. Some are commercial and other free. Most of them focus on secondary education. Subjects taught in graduate education, particularly on the subject of numerical methods, are scarce. In these series, we highlight the "Interactive Educational Modules in Scientific Computing," available online at the site http://www.cse.illinois.edu/iem/. In this software, each module is a Java applet that is accessible through a web browser. For each applet, we can select, from a list, problem data and algorithm choices interactively and then receive immediate feedback on the results, both numerically and graphically. Our approach differs from this because it is open source and generic, open to the inclusion of new mathematical methods that can be illustrated graphically

In previous work (Balsa et al, 2010), we put out several important questions namely: How to retrieve the information about the sequence of algorithm iterations (data flow and control flow)? How to represent internally that information? Is the representation in XML pretty generic? Which technology should be used to visualize graphically a mathematical algorithm (Java and OpenGL)?

We begin by addressing these questions in sections 2, where we describe the main steps that led to the current GraSMA implementation. After that, in sections 3, we illustrate the GraSMA utilization with the Newton Raphson's method. In section 4 we discuss about GraSMA improvements currently under development.

2 GRASMA IMPLEMENTATION

To build the information essential to animate algorithm iterations we choose to instrument the code (Code Instrumentation). This technique is well known in the area of program comprehension, see for instance (Berón et al, 2007) and (Cruz et al, 2009), and usually is adopted when the objective is to visualize programs written in a specific language. The main idea is to annotate the source code with inspector functions. This will allow retrieving static and dynamic information of the program execution.

To store this information, a Document Type Definition (DTD) was created in order to generate an intermediate representation in XML - eXtensible Markup Language (Ramalho and Henriques, 2002). That DTD describe how to represent information about the algorithm execution. One of the first challenges of this work is to determine the schema appropriate to describe a large number of different algorithms.

Finally, in order to visualize the algorithms, the Java programming language (Cadenhead and Lemay, 2007) and OpenGL (Shreiner, 2009) are used.

The software, based on Java and OpenGL, relies on the GNU Octave engine for numerical calculations. The visualization section is based on two predominant classes: the GLRenderer2D and the GLRenderer3D.

On the other side, the class OctaveCaller generates the XML file, based on the execution of the algorithm in Octave. The "Renderer" classes process and display on screen a series of mathematical object representing a step or iteration of a numerical method.

The algorithm is represented, in Java, by the class Algorithm, which describes the Octave algorithm through a list of iterations (each iteration is itself a list of mathematical objects to be displayed). This information is stored in a list of iterations and is obtained via the Parser class that can process an XML file to retrieve the iterations data and thus place them in the corresponding field in the instance of the Algorithm class.

To visualize the algorithm execution two different panels are displayed: the first displays some standard elements that are always on the screen (named the "global" elements). The second draws some elements that are visible only for the iteration that is currently on display. These elements will be replaced at the next iteration. That is why one is able to see on the class diagram that the Algorithm class is linked to the MathObject interface by two different links: an iterationList (that is to mean a list of iteration where an iteration is a list of MathObject), and a global list is just a list of MathObject. A generic schema of all software components can be seen in Fig. 1.

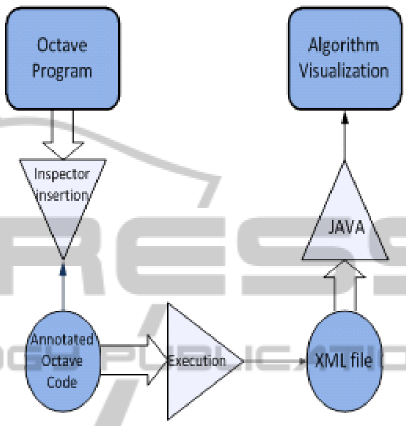


Figure 1: Generic scheme of all software components.

2.1 Annotation with Inspector Functions

The software can display on the screen any type of mathematical algorithm that uses some type of mathematical objects that will be detailed in section 2.3. For this, the algorithm coded in Octave language, must be changed a second time to allow record data by each iteration. This data is encapsulated in an XML file.

Two Octave functions, already defined, are added to the Octave code: `init_global()` and `end_global()`

These functions are to generate the early part of "global" algorithm, i.e., all elements that appear on the screen from iteration to iteration. For each iteration, other functions are used: `init_data()`, `end_data()`, `init_iteration()`, `init_iteration_with_information()`, `end_iteration()` and `end_data()`.

The `init_data` is called on the beginning of a list of iterations. This function will be followed by a series of successive calls of `init_iteration` function (with or without information) to declare the beginning of a new iteration and a call of `end_iteration` function to complete the annotation. When all the iterations have been reported with their mathematical objects inside, we can call the `end_data` function to close the iterations list.

Finishing the iteration annotation it is time to declare the mathematical objects that appear in this iteration. To do this, the following functions are available in Octave:

```
    new_curve
    new_ellipse
    new_circle
    new_curve_with_parameters
    new_integral
    new_integral with parameters
    new_parameter
    new_point2d
    new_point3d
    new_vector
    end_curve_with_parameters
    end_integral_with_parameters
```

The Octave basic function should be amended to bring up a parameter `file_id` as the first parameter of the function. This `file_id` is the file identifier for the XML backup of the execution of the algorithm. This file identifier is created automatically by GraSMA that will itself launch Octave script with this parameter.

The Octave basic function should be amended to bring up a parameter file_id as the first parameter of the function. This file_id is the file identifier for the XML backup of the execution of the algorithm. This file identifier is created automatically by GraSMA that will itself launch Octave script with this parameter.

If we wish to run the script manually in Octave, you must have an identifier file.

All these Octave functions simply write lines of XML in a file. Finally the XML file follows the document type definition (DTD), allowing a valid XML file to be correctly parsed by the Java code.

2.1.1 Example of Code Instrumentation

We present below the sequence of procedures done for the graphical representation of the Newton-Raphson's method.

The end users (students and teachers) are not concerned with code annotation; they just chose the algorithm and watch the generated visualizations.

In this approach, code instrumentation is performed by us in each Octave algorithm. It occurs only once throughout the software lifetime. Octave inspector function calls are added to code in order to register in the XML file "what is happening".

We start first by changing the header function to add a new argument `file_id`.

As an example, we are going to present the basic Newton-Raphson in Octave. The original implementation of this method is:

```
   function[x, res,
nbit]=nle_newtraph(f, df, x0, itmax,
tol, varargin)
   x=x0;
   nbit = 0;
   err=tol+1;
   fx= feval(f,x,varargin{ : });
   if  fx==0;
       x=x0;  res=0; nbit=0;
       return
   end
   while err > tol & nbit <= itmax
       aux = x;
       fx = feval(f,x,varargin{ : });
       dfx = feval(df,x,varargin{ : });
       x = x-fx/dfx;
       err = abs(aux-x);
       nbit = nbit+1;
   end
   res = feval(f,x,varargin{ : });
   if nbit > itmax
       printf( ["nle_newtraph stopped
without converging, maximum number of
iteration was reached .\n"] );
   end
   endfunction
```

The programming user must decide what he wants to display on the screen. Let's imagine that he wants to show the target function $f(x)$ and to display different tangent lines representing the evolution of the algorithm in each iteration.

Then, before the first iteration of the algorithm, we declare the elements that will be global, i.e. the mathematical elements that will be continuously displayed on the screen. The functions `init_global` and `end_global` must imperatively be called even if the list of elements inside is empty:

After the declaration of global elements, `init_data` function is called in order to prepare the annotation of the iterations. Next, at each iteration we will find at first the `init_iteration` call (or `init_iteration_with_parameter`, which can also take a string that represents the additional information to be displayed by the application) and, at the end, the `end_iteration` function call.

All these functions (which records data in an XML file) have always as first parameter `file_id`. After the end of the list of iterations, a call to `end_data` function is necessary.

Finally, it remains only to make a call to `init_error` and end_error before closing the data tag (`end_data`). One can put a list of errors (`new_error_point`) after the call to `init_error`.

2.2 Document Type Definition

Document Type Definition (DTD) is a structure of mark-up declarations that defines a document type for SGML-family languages (SGML, XML, HTML). A DTD is a kind of XML schema.

DTDs use a brief formal syntax that declares the structure and the elements and its attributes of one type of document. Each case of the DTD will follow the same organization and it has the same elements.

2.3 Plotting Mathematical Objects

The fundamental mathematical objects that we can visualize are: vectors, lines, curves (functions), integrals, circles, ellipses and 3D surfaces. Each of these objects corresponds to a Java class that implements the interface MathObject.

For instance, the semantic representation in the XML file that matches with the mathematical object Integral is:

```
<integral value="@(x) sin(x)"
 color="green" lowerbound="-2"
 upperbound="8"> </integral>
```

The display of integrals was necessary to see the evolution of the Simpson method (Fig. 2), used in numerical analysis, for numerical integration. The first attempt to draw the integrals was based on polygons (because the polygons are one of the basic designs of OpenGL). This was not conclusive because the full draw on the basis of a polygon is possible only if, on the interval over which the integral is calculated, the function does not change its sign. So we used even more basic integrals: using only lines, and different colours that work in all cases. Use 15-point type for the title, aligned to the center, linespace exactly at 17-point with a bold font style and all letters capitalized. No formulas or special characters of any form or language are allowed in the title.

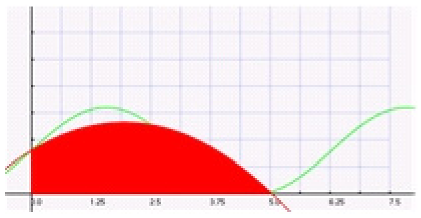
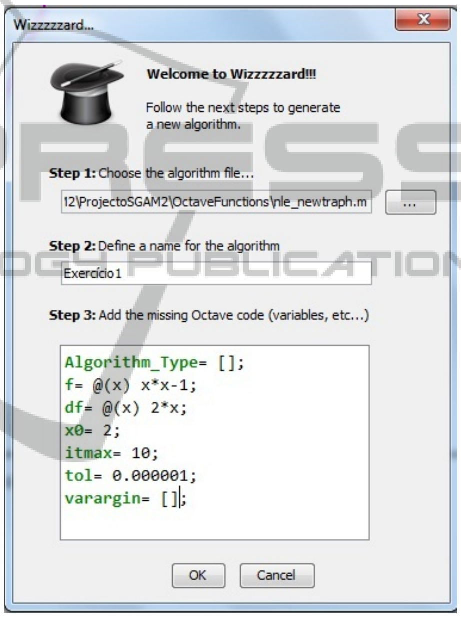


Figure 2: Visualization of the Simpson's method.

3 GRASMA UTILIZATION

At the opening of GraSMA system we select Files, then New, and it simply shows the steps on a new window. In step 1 we must choose an annotated Octave algorithm a give it a name in step 2. Finally, in step 3 we indicate the input parameters of the numerical method. If we need to refer a function, we must think about writing this function in Octave format (for example @(x) sin(x) for the sinus function).

The Fig. 3 illustrates the case of the call to Newton-Raphson's method to find the zero of the function $f(x) = x^2 - 1$, with initial approximation $x_0 = 2$.


Wizzzzzard...

Welcome to Wizzzzzard!!!

Follow the next steps to generate a new algorithm.

Step 1: Choose the algorithm file...

12\ProjectoSGAM2\OctaveFunctions\nle_newtraph.m

Step 2: Define a name for the algorithm

Exercício 1

Step 3: Add the missing Octave code (variables, etc...)

```
Algorithm_Type= [];
f= @(x) x*x-1;
df= @(x) 2*x;
x0= 2;
itmax= 10;
tol= 0.000001;
varargin= [];
```

Figure 3: Example of a call to Newton-Raphson method.

Once this information is supplied, the algorithm visual representation appears on the application left side.

The user interface is very simple with icons for:

-Go to the next iteration

-Go to the previous iteration

-Make the animation of the algorithm

The progression of the algorithm is shown on the application right frame and if any information has been filled for a particular iteration in the modified Octave file, then it will be displayed on the list box.

Fig. 4 and Fig. 5 correspond to the two first iterations of the Newton-Raphson method with setup parameters shown in Fig. 3. The dashed line corresponds to the approximate solution obtained by the tangent function (straight pink) in the previous iteration.

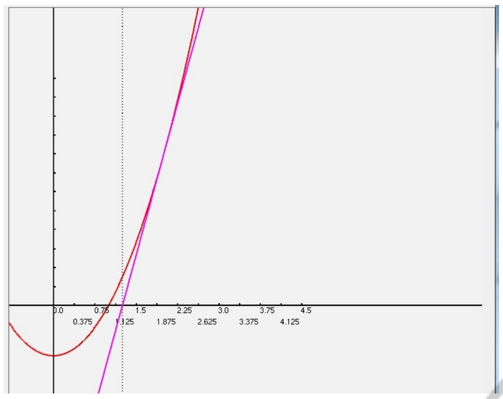


Figure 4: First iteration of Newton-Raphson method.

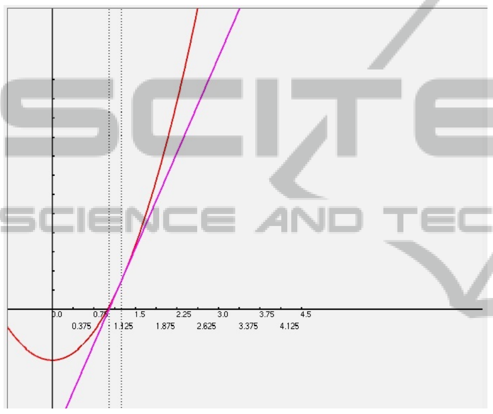


Figure 5: Second iteration of Newton-Raphson method.

We can increase or decrease the zoom level of the visualization by clicking with the left button of the mouse.

4 GRASMA IMPROVEMENTS

We are planning to add new functionalities to GrasSMa. Currently we are working in the automatic annotation of Octave code by means of a compiler that generates annotated code. Other current issue is the development of a website that enables the online access to GrasSMa.

4.1 Automatic Code Instrumentation

The main idea is to turn GrasSMa easily adaptable to other algorithms. As these algorithms are implemented in Octave language and we can extract information in run-time in order to visualize the execution process. For that, a language processor will be used to automatically annotate Octave programs with inspector functions. Until now, this task was performed manually for each algorithm but with this new frontend, GrasSMa can generate visualizations of any algorithm coded in Octave without modifying manually the source code.

To implement code instrumentation (Cruz et al, 2009) we insert inspection functions (or inspectors) in strategic places of a program to capture its execution flow. The information extracted along this inspection can be used to show different views to help understanding program behaviour. This is a well known technique for Program Comprehension.

To define a strategy to annotate the source code we have to know: which information we need to extract; and what are the strategic points in the source code.

To answer these questions we conceptualize the program execution process as a state machine (SM). The input values represent the initial state and the final state can be represented by the variable values after execution. The intermediate states are represented by the variable values reached during the program execution.

The transition between states is carried out through the Octave program functions. The values reached in each algorithm step with be saved internally to produce evolution graphical visualizations.

To implement this strategy we are building a parser for the source language extended with semantic actions. These actions insert into the program new statements that will allow tracing the state and the transitions.

The parser relies on several compiler constructions (Aho et al, 1986) tools: Lex&Yacc (Levine et al, 1992), AntLR (Parr, 1999) or LISA (Henriques et al, 2005).

These tools are based on the language grammar and they allow specifying the automatic recognition and transformation of the program written in that language. In our case the language to be recognized is Octave and the transformation consists in adding the inspector functions.

4.2 WebGRASMA

The GrasSMa prototype is a desktop application, running in a Java Virtual Machine installed in the user workstation. This approach limits the possibility to update the existing algorithms as well as the possibility to deploy new algorithms. Moreover, the use of the software requires that each student installs the software on his own PC or laptop, making the tool unpractical to use in lectures.

To deal with these limitations, we are developing a Web based application able of displaying the algorithms directly on a web browser.

The web application is structured in two main sections: the server and the client. We are making the effort to use standard HTML, to allow using regular web browsers as the client.

The server side concentrates the application logic, responsible for storing the algorithms, executing them on demand on the Octave engine, retrieving the execution resulting XML file and producing the visualization, which will be sent to the client.

In the desktop version, a single instance of the Octave engine is sufficient to respond to the user operation. However, in the Web based version, each user has, potentially, allocated a different instance. This can pose scalability and stability issues.

To prevent, as much as possible, these problems, we are considering a server side cache mechanism, to allow responding to user operations without requiring on-the-fly interpretation of algorithms.

To leverage the code already developed for the desktop operation, we are building the Web application in Java Servlets, for the "hardcore" logic, and JSF (Java Server Faces) and HTML for the complementary website structure, such as forms and documentation pages.

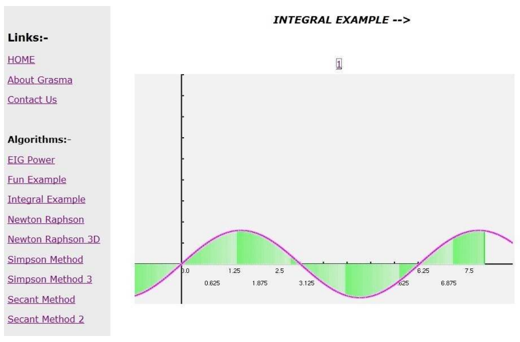


Figure 6: Snapshot of the web-based GraSMa.

5 CONCLUSIONS

This paper described GRASMA implementation. The approach followed by this graphical simulation tool was semi-automatic because it implied the manual insertion of code inspector functions.

Here a big improvement is presented: the numerical algorithm is automatically instrumented and the algorithm visualization is generated with any effort from the user.

The process is based in language processing techniques and it allows simultaneously the generation of XML code for further application domains.

REFERENCES

Balsa, C., Alves, L., Pereira, M. J., Rodrigues, P.J. 2010. Graphical Simulator of Mathematical Algorithm (GraSMA). In *Teaching and Learning 2010, Advances in Teaching and Learning Research*. IASK.

Berón M., Henriques P. R., Pereira M. J. V., Uzal R., 2007. Static and Dynamic Strategies to Understand C Programs by Code Annotation, In *OpenCert'07, 1st Int. Workshop on Fundations and Techniques for Open Source Software Certification.*

Cruz D., Béron M., Henriques P. R, Pereira M. J. V.,2009. In *Code Inspection Approaches for Program Visualization*, Editor Linus Michaeli, Faculty of Electrical Engineering and Informatics, Technical University of Kosice.

Ramalho J. C, Henriques P. R, 2002. *XML & XSL: da teoria à prática*. FCA Editor. Lisbon, 1st Ed.

Cadenhead R. and Lemay L., 2007. *Teach Yourself Java 6 in 21 Days*. Sams, 5th Edition.

Shreiner D., 2009. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th Edition.

Aho A. V., Sethi R., and Ullman J. D., 1986. *Compilers Principles, Techniques and Tools*. Addison-Wesley.

Levine J. R., Mason T., and Brown D., 1992. *Lex & Yacc*. Ed. Dale Dougherty. O'Reilly & Associates Inc.

Parr T., 1999. *Practical computer language recognition and translation – a guide for building source-to-source translators with antlr and java*. http:// www.antlr.org/ book/index.html.

Henriques P., Varanda M. J., Mernik M., Lenic M., Gray J., and Wu H., 2005. *Automatic generation of language-based tools using lisa system*. IEE Software Journal, 152(2):54–70, ISSN: 1462-5970.