SCIENTIFIC SOFTWARE DESIGN THROUGH SCIENTIFIC COMPUTING PATTERNS

Gustavo Rodríguez–Gómez
Ciencias Computacionales
Instituto Nacional de Astrofísica,
Óptica y Electrónica (INAOE)
Puebla, Tonantzintla, México
e-mail: grodrig@inaoep.mx

Jaime Muñoz–Arteaga and
Ricardo Fernández–del–Busto
Depto. de Sistemas de Información
Universidad Autónoma de Aguaescientes (UAA)
Aguascalientes, México
e-mail: jmunoz@correo.uaa.mx
Depto. de Ingeniería Eléctrica y Electrónica
Tecnológico de Monterrey,
Campus Ciudad de México (TM)
Tlapan Ciudad de México, D.F.
email: ricardo.fernandez@itesm.mx

ABSTRACT
An approach to facilitate the development and validation of simulation models is presented. We exploit the object-oriented implementation techniques together with scientific computing patterns for developing a basic solver engine to approximate numerical solution of a set of ordinary differential equations. The solver engine can be configured with different numerical methods according to the engineer needs.

KEY WORDS
Design patterns, scientific computing patterns, ordinary differential equations, Runge Kutta methods, simulation, model validation.

1 Introduction
A common problem in simulation of large-scale dynamic systems, like operator training simulators, is related to the numerical solution of the algebraic-differential equations of the mathematical model. We would like to select a method according to the next criteria: 1) that minimizes the computing time, and 2) retains suitable accuracy in the approximate solution. Commonly, the numerical method for the solution is selected from a set of possible methods and we choose the “best” method from comparison of the results obtained according to above criteria. This process may be performed in several ways, for example:

a) Block simulation, such as Simulink [1],

b) Off-the-shelf numerical routines, like ODEPACK [2].

Both approaches have at least two disadvantages: first, the difficulty of adapting application-specific program, such as training simulators, to the interface required by the numerical software, and second, when problem size increases. However, the great variety of numerical methods available in those types of packages is an enormous advantage to solve a specific problem.

We propose an approach to alleviate the above situations by applying the standard object oriented programming (OOP) characteristics together with scientific computing patterns, leading to the development of an basic solver engine for ordinary differential equation (ODE) systems. We consider that this approach has the following advantages:

- It inherits the OOP well known properties.

- It provides great flexibility to include new numerical methods or withdraw obsolete ones, and the interface of these methods is easy to adapt to the application-specific program, since the data structures are hidden to the user.

- The user may select different numerical methods, without regarding the implementation issues.

- The user may focus his efforts towards the analysis of the solutions obtained rather than worrying about programming issues.

However, one drawback present in this approach is the abstraction penalty which might be costly in some circumstances. There are some examples in the literature that this disadvantage can be alleviated by means of using special compilers such as KAI C++, BLITZ ++ [3].

The structure of this work is organized in six sections as follows: Section 2 states the problem definition present in solving numerical problems and how patterns could be applied. In section 3, we state the mathematical problem encountered in computing a numerical solution of algebraic-differential equations, and we propose to use Runge–Kutta methods to deal with this problem. Section
4 constitutes the core of this work since two new scientific computing patterns are defined, and the solver engine design, applied to Runge-Kutta methods, is discussed. OOP is used as the underlying paradigm in the design and implementation phases. Section 5 presents numerical results when applying the solver engine to a specific mathematical model. Section 6 gives some conclusions of the present work.

2 Problem Definition

Usually, the validation of a model requires an environment that has enough flexibility to employ different numerical tools. We need verify that the compute solution is efficient and sufficiently accurate to serve the purposes for which the model was constructed. Trustful results may be achieved if different numerical methods are available to help the engineer to select the right numerical method.

Traditionally, these numerical tools are available in a static library, a collection of object files that store different algorithms. This strategy is too rigid as far as it is concerned: adding a new method not previously considered, extending the capacity of an algorithm, changing the way a procedure is called. These types of changes are hard to carry out.

The lack of flexibility in the numerical software may originate in the delays in the development of a project, as could be the case in training simulators, which could seriously affect the termination of a project.

In general, the above constraints have their origin in the type of computing language, which in most cases are procedural, that couples the numerical algorithms to specific data representations [4]. A strategy that could help to avoid these couplings is to use OOP, where the objects are defined by their responsibilities, not by their representations.

The approach used in this work is to develop the analysis and design of a basic solver engine employing patterns. They provide a fundamental tool to solve specialized problems [5]. We propose two scientific patterns that will help in the design and implementation of numerical methods to solve dynamic systems simulation problems. These patterns are Model Solver and System Modularization; see Section 4.3.

The system developed is flexible and extensible, and is not tied to a static library since the numerical method is instantiated at compilation time. It allows incorporation or deletion of numerical algorithms in an easy fashion and for exemplification purpose the standard Runge–Kutta class is used.

3 The Mathematical Problem

In general, physical systems are represented by mathematical models involving systems of algebraic-differential equations. The model designers should solve them numerically in order to analyze the model behavior.

If only ordinary differential equations are used, then the problem we want to solve numerically is the following:

\[ \dot{y} = f(t,y), \quad y(a) = y_0 \]  

where “\( \dot{y} \)” denotes \( d/dt \) and \( f: \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n, n \geq 1 \).

For simplicity in the presentation, we assume that \( n = 1 \). We wish to find a solution in the interval \( a \leq t \leq b \), with finite \( a \) and \( b \). It is assumed that the function \( f \) satisfies the conditions stated in the theorems of existence that guarantees a unique, continuous and differentiable solution [6]. In what follows, \( h \) denote the integration step and is constant, \( y_n \) is an approximation to the value of the exact solution \( y(t_n) \), and let \( f_n \equiv f(t_n, y_n) \).

3.1 Numerical Algorithm

A very popular class of methods to approximate the solution of (1) is the class of \( R \)-stage Runge–Kutta methods. These methods can be written in the form [6]:

\[ y_{n+1} - y_n = h \varphi(t_n, y_n, h). \]  

The general \( R \)-stage explicit Runge-Kutta process is given by:

\[ y_{n+1} - y_n = h \varphi(t_n, y_n, h), \]

\[ \varphi(t, y, h) = \sum_{r=1}^{R} c_r k_r, \]

\[ k_1 = f(t_n, y_n), \]

\[ k_r = f(t_n + ha_r, y_n + h \sum_{s=1}^{r-1} b_{rs} k_s), \]

\[ a_r = \sum_{s=1}^{r-1} b_{rs}, \quad r = 2, 3, \ldots, R. \]

These methods involve \( R \) evaluations of the function \( f(t, y) \) for each integration step. Each function \( k_r(t, y, h) \) for \( r = 1, 2, \ldots, R \), may be interpreted as an approximation to the derivative \( y'(x) \), and the function \( \varphi(t, y, h) \) is a weighted average of these approximations. The consistency of the method requires that \( \sum_{r=1}^{R} c_r = 1 \) [6].

4 Object Oriented Analysis and Design

The analysis and design of a solver engine covers two stages: those related with the numerical method, and those related with software engineering. The second stage have a relation with the attributes of high quality software.

In the following sections is presented the strategy to design and implement the basic solver engine.
4.1 Design Characteristics

Design must satisfy that the integrator-class allows reusability without modification, in such a way that can be used by different models. Consequently, the following characteristics must be satisfied

1. The integrator-class structure must be reusable without modification.
2. The integrator-class must not contain details of the integrated model.
3. The differential equations are located outside the integrator-class.
4. The integrator-class must be extensible. New integrations methods can be incorporated within the integrator-class structure with minimal coding.
5. The numerical methods are reusable for a large set of ordinary differential equations.
6. The selected numerical method is instantiated at compilation time.

4.2 Scientific Computing Patterns

Software design patterns are widely used in the development of different software systems [7]. In Gamma et al. [8], the idea to apply design patterns for the development of software is presented and a catalog of 23 patterns are introduced. These patterns are classified according to their purpose and scope. Furthermore, Gamma et al. postulate different object oriented strategies and provide alternate methods of development based on the design patterns proposed. However, in the present work, we set a reference in the conceptual stage where we develop a set of scientific computing patterns. We focus in the use of patterns at very early stages, mainly in the analysis of the problem.

The approach follows the proposal made by Fowler [7], but we identify these conceptual patterns as Scientific Computing Patterns. From these patterns, we build our software system, the basic solver engine, following OOP. After this conceptual analysis and with the aid of existing design patterns available in the software community, a class hierarchy is derived, according to our needs; see $\Delta$ (Delta) effect [9].

Scientific computing patterns capture the fundamental characteristics of the problem to be solved. They are focused on “what” and not in “how”. So we prevent to take design or implementation decisions at analysis stage. Besides, the patterns help analysts indicating what information must be gathered and how it must be structured. Then, they provide a common language that allows communication among scientific experts, analysts and users. In this way, we can set the requirements in terms of scientific computing patterns making future similar developments easier to carry out.

From the above design characteristics, Section 4.1, we propose two scientific computing patterns. They are described in the classical tabular form. For each proposed pattern, the following characteristics are defined: name, problem, intent, context, forces, solution, rationale.

The Model-Solver pattern emphasizes the decoupling between models and numerical methods. It allows the models and the numerical methods to vary independent of each other.

<table>
<thead>
<tr>
<th>Name: Model–Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem:</strong> A physical system represented by a mathematical model, involving systems of algebraic–differential equations, is required to be solved numerically in order to analyze the model behavior.</td>
</tr>
<tr>
<td><strong>Intent:</strong> Give a feedback between the model and the solver module during the simulation loop.</td>
</tr>
<tr>
<td><strong>Context:</strong> The implementation of continuous process simulation is well-defined problem by an ordinary differential equations system, which is solved by an iterative fashion during the execution time.</td>
</tr>
<tr>
<td><strong>Forces:</strong> The decoupling between models and numerical methods increases extensibility. The models are not aware of the numerical issues.</td>
</tr>
<tr>
<td><strong>Solution:</strong> Specify the mathematical model and the solver by modules, and define the communication between them in order to give continuity to the simulation loop.</td>
</tr>
<tr>
<td>Rationale: The model and the solver separation by modules allow their reusability and their weak coupling.</td>
</tr>
</tbody>
</table>

The System Modularization pattern emphasizes the input provides by the user: the model, initial conditions, simulation parameters, and the output variables to display. The solution is hidden to the user, giving great flexibility in combining different sequencing strategies with diverse numerical methods.

249
Name: System Modularization

Problem: A system is required to perform flexible simulations of continuous processes.

Intent: Propose a conceptual model in terms of modules for a continuous process simulator system.

Context: The system is represented by algebraic-differential equations. The input parameters of this solution are clearly identified.

Forces: The model of the system is possible to be decomposed into separate sub-modules in order to identify new sub-systems. The solution is hidden to the user given great flexibility to select new solution strategies.

Solution: The simulation module should allow interactions between the input, output and model solver modules in order to get all the necessary information during the simulation execution.

Rationale: The current simulation systems based on continuous processes can be specified in terms of a set of modules independently of their implementations.

4.3 Solver Engine Design

In this section, we will describe how we used the scientific computing patterns and the design patterns to design and implement the basic solver engine.

The design diagram is shown in Figure 1, the class hierarchy illustrate the decoupling between the mathematical model and the numerical methods. It should be observed that the numerical method is decoupled in the concrete part, numerical method implementation, and in the abstract part, the essence of the numerical method. It allows the abstraction and the implementation to vary independent of each other. Each numerical method is derived from the classes CsODE and IntODE. Specific integrators are derived from the intermediate cnrctRKGral class, which is an interface common to all supported algorithms of Runge-Kutta classes. The mathematical models are derived from the class ModeODE, which is the interface to derivative information of the models.

Figure 2 shows the relations between the client and the basic solver engine. The Facade pattern provides a simple interface to the system and gives a decoupling of the system from clients and other subsystems; hence, we get flexibility. Besides, we have applied the Bridge and Strategy patterns. The Bridge pattern puts the essence of the numerical method, CsODE, and its implementation, IntODE, in separate class. It avoids a permanent binding between an abstraction and its implementation. These patterns avoid class explosion.

The implementation has the classical four Runge-Kutta methods and the Runge-Kutta Fehlberg method. It facilitates the incorporation of others Runge-Kutta algorithms or new integration methods. In order to incorporate other Runge-Kutta scheme into the design of the class structure, it is only necessary to give the number of stages and the coefficients for the integration algorithm. The code was written and tested using Borland C++ Builder 6 and Visual C++ 6.

A similar design can be found in [10], but they do not use scientific patterns to design the ODE-Solver. Furthermore, they do not decouple the concrete and the abstract part of the numerical method. It is possible that they have a class explosion when adding new integration methods.

5 Test Cases and Numerical Results

To illustrate how easy it is to use the basic solver engine, we solve the Euler’s equations and the mathematical model of a vehicle of mass $M$, suspended by a lumped spring-dashpot [11].

The Euler’s equations are given by

\[
\begin{align*}
\dot{y}_1 &= y_2 y_3, \\
\dot{y}_2 &= -y_1 y_3, \\
\dot{y}_3 &= -0.51 y_1 y_2.
\end{align*}
\]

The initial conditions are $y_1(0) = 0$, $y_2(0) = 1$, $y_3(0) = 1$. We need to give the following simulation parameters:

- Initial time $= 0$
- End time $= 12$
- Integration step $= 0.01$
- Number of differential equations $= 3$
- Number of parameters $= 0$.

The simulation results are shown in Figure 3 and it matches with the expected results.

The mathematical model of the vehicle is represented by a set of differential–algebraic equations given by

\[
\begin{align*}
\dot{r} &= r_0 (1 + c |\dot{x} - \dot{x}_0|), \\
x_0(t) &= A (1 - \cos \omega t), \\
M \ddot{x} &= -k(x - x_0) - r(\dot{x} - \dot{x}_0).
\end{align*}
\]

The initial conditions are $\dot{x} = 0$ and $\ddot{x} = 0$ and the simulation parameters are given by

- Initial time $= 0$
- End time $= 5$
- Integration step $= 0.001$
- Number of differential equations $= 2$
- Number of parameters $= 6$.

The vehicle model parameters are the following

- $M = 10$, $A = 2$, $\omega = 7$, $k = 640$, $\dot{r}_0 = 80$, $c = 0$.

The simulation results are shown in Figure 4 and it matches with the expected results.
The interface between the models and the basic solver engine is through the Facade pattern as shown in Figure 2. According with the analysis patterns Model-Solver and System Modularization, once we have entered the initial conditions, simulation parameters and model parameters, we create the Simulation Block by creating the Model and its corresponding ODE-Solver.

The user can select the following integration methods: Runge-Kutta 2, Runge-Kutta 3, Runge-Kutta 4, Runge-Kutta-Fehlberg 45 and Runge-Kutta 78, where the Strategy pattern is used to select the proper integration algorithm. Finally, we implement the Visualization module to see the simulation results.

6 Conclusions

This work has proposed a strategy for the development of software scientific through scientific computing patterns. A successful application of conceptual and design software patterns has been applied to firstly, establish two new scientific computing patterns and, secondly, using those patterns, a design solver engine is proposed that deals with the numerical solution of algebraic–differential equations, found in many applications in Engineering. The underlying paradigm used in the implementation is OOP. Runge–Kutta methods are redesigned in order to prove the application of these concepts and simulation results are presented where these benefits are shown. The structure of the design of the solving machine demonstrates how the OOP and the patterns can make implementations of numerical methods easier to understand and modify than traditional procedural approaches. It is easy to add new numerical algorithms not previously considered according to developer needs.

Although the present work has focused more at the issues of flexibility, reusability, scalability, extensibility and easiness to maintain than on computational efficiency, this topic becomes important when the problem size increases, or in the development of a project. Abstraction in OOP, in particular C++, has been accompanied by a performance loss called “abstraction penalty”. Nevertheless, it is possible to gain computational efficiency through a special de-
sign oriented towards a given application.

7 Acknowledgements

This work was supported in part by the Consejo Nacional de Ciencia Tecnología of México(CONACYT) under the project CONACYT–2002–C01–40022. The authors wish to thank the support provide from INAOE, UAA and TM to develop this project.

References


[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object–Oriented Software. Addison Wesley, 1995.

