Proceedings of the 3rd IASME/WSEAS Int. Conf. on HEAT TRANSFER, THERMAL ENGINEERING AND ENVIRONMENT, Corfu, Greece, August 20-22, 2005 (pp326-331)

Object-oriented Language Variables and New Continuous Media Simulation Approach

ALEXANDER SOIGUINE
Tinsley Labs, Inc.
4040 Lakeside Dr., Richmond, CA 94806
USA
alexsoiguine@cox.net http://www.dcmodeling.net

Abstract: - Numerical computer algorithms to model physical processes in continuous media are generally based on one of the schemes of discretezation of PDEs, which are considered mathematical models of phenomena. PDEs, on their own, are relations between values of given functions, unknown functions and their derivatives calculated through a limit process for small media elements interactions when elements' sizes and evolution time steps both approach zero. The limits can typically only be calculated using (many) simplifying assumptions not based on the principles of physics. It follows that traditional numerical schemes simulate oversimplified mathematical models and not real processes. In the Direct Computer Modeling approach, an algorithm exactly reproduces interactions between small volume elements. That paradigm can be mapped onto the object-oriented computer language structure (or class) type variables.

Keywords: - Computational Physics, Continuous Media Simulation.

1 Introduction

A recently published article [1] state "… the new discipline [computational simulation] …is still troublingly immature". One of the reasons for that may be the legacy of mathematical approaches. For roughly the last three centuries, partial differential equations (PDE) have comprised the basis for model phenomena in continuous media. Even the newly introduced computers are used simply as numerical solvers of PDEs. That means that PDEs are still considered basic models of real physical phenomena in continuous media and computers play the role of a powerful resource to numerically solve equations.

Fig.1 shows the logic of the traditional approach. What may look strange is the necessity to model, through a finite numerical scheme, a symbolic model, PDE, as opposed to the physical process itself. PDEs are derived through a discretezation procedure of real physical media, with a subsequent mathematical limit process accompanied generally by many assumptions not based on the principles of physics. Any PDE is just an expression symbolically encoding some unnecessary oversimplifications of a real physical process. A computer, being a finite machine, can run a discrete time/spatial scheme evaluating functions satisfying PDE, so would it not be more practical to run discrete algorithms of interactions/states evolution of small media elements? In other words, it is more consistent to exclude blocks 3 and 4 from the traditional scheme and remove the statement "oversimplified assumptions necessary to derive PDE from" from the block 2.

In terms of the described approach, it is difficult to explain the sense of nonlinearity. When somebody says "nonlinear process", it really means "nonlinear equations" corresponding to the process.

The approach is applicable not just to modeling continuous media but to arbitrary networks (for example, electronic schemes). The first model of soliton like behavior, was

a nonlinear elastic string simulation. Wide spectrum of potential applications expands as far as simulating relativistic charges beams.

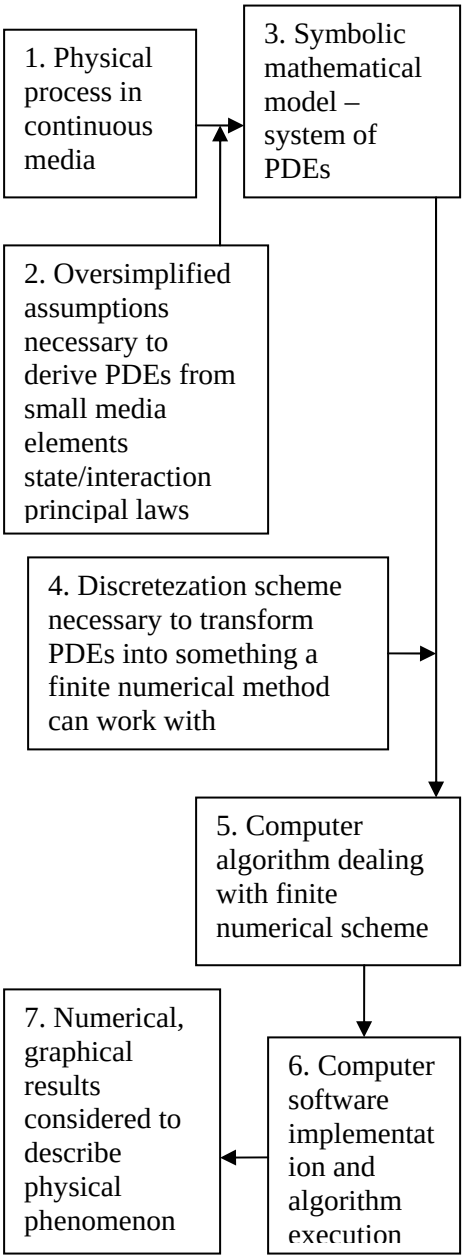


Fig. 1

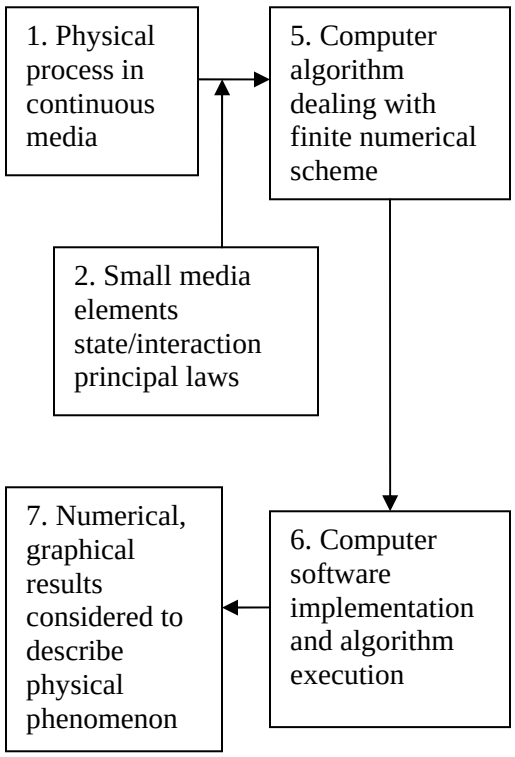


Fig. 2

2 Quasi Analog Dream

An option existed to use a huge array of microprocessors, corresponding to elements in a physical medium [2]. Each microprocessor would save all numerical parameters defining the state of an element, implementation of all functions necessary to transform the element state parameters, and sufficient number of ports to communicate with processors within the network. In that case, the array should behave exactly like a real physical medium. To some extent, it would resemble an analog computer but have much greater capabilities. Something similar but much more simple was implemented in the “cellular automata” approach [3].

It was not obvious then which operating system could work with such multiprocessor systems, though some good projects existed. Fortunately, common computer languages such as C++ (and object-oriented PASCAL) had special types of variables that could support interacting media elements on a software (program) level.

Let me remind the reader the definition of the structure type variable in the C++ language.

In plain "C" language, the *struct* type variable can be thought of as a list of "standard" type variables (integers, doubles, strings, etc.), for example:

```
struct sample
{
    int i;
    double d;
    char c[30];
} st_one ,st_ two;
```

In C++, structure fields may be functions:

```
struct cl
{
    int i;
    int get_i(void);
    void put_i(int j);
};
```

Functions declared in the structure must have implementations:

```
int cl::get_i(void)
{
    return i;
}

void cl::put_i(int j)
{
i = j;
}
```

The *class* type variables are more flexible in the sense that you can keep class members, for example, *private*, accessible just to functions from the class. In *struct* all members are by default *public*. Any function from a program can get access to all of them.

It is evident that such types of variables ideally match the requirements for modeling small media elements interactions. Each single media element can be uniquely identified, at a particular instant of time, by some number of its state parameters that are "regular" type variables of the class. The class member functions derived from laws of physics then define the element behavior (its evolution in time).

All that exactly resembles what is going on in real physical media when we want to simulate it through partitioning and small element interactions. In that sense, we have analog simulation scheme implemented on software level.

3 Elements Behaviors and C++ Classes

It is more convenient to use C++ class type variables to represent elements' states and transformations, mainly because in C++ (or any other object-oriented language) it is definitely stressed that a class is an object type variable comprising of both elements' state variables and their transformation functions. Inheritance also makes the class approach more flexible.

Base abstract class representing a medium element can be defined as:

```
class MediumElement   {
        double current_t, delta_t;
        double* state_parameters;
        int nmb_parameters = 0;

public:
        virtual void
transform_parameters(void (func*)(double*
param, double* m_param,

        MediumElement** neighbors))=0;

// some interface functions

// etc.
```

}

The pointer to doubles, *state_parameters*, is a dynamically allocated array of numerical parameters, uniquely identifying (physical) element state at a particular instant in time.

Virtual (specifically implemented for each considered physical process) function *transform_parameters* should model arbitrary element behavior, depending on its current state (pointer *param*), global media parameters (pointer *m_param*) and an array of pointers to other elements interacting with the current element.

A derived class corresponding to some special type of a medium and specific model then can be defined as:

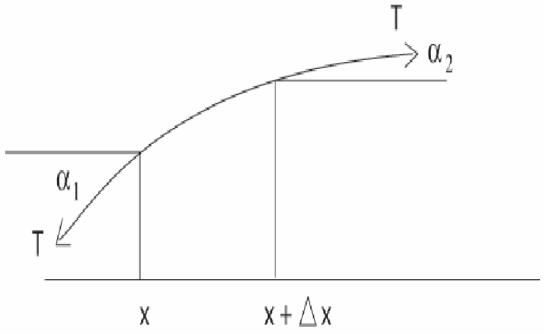
```
class SpecMediumElement: public
MediumElement        {
        double* model_parameters;
        void Model(double* param, double*
m_param, SpecMediumElement**
neighbors);

// and necessary interface functions
}
```

The function *Model(…)* is used to input variable for the *transform_parameters.* Actual implementation of it should be written in accordance with physical laws applied to a particular model. Unquestionably, additional interface functions to initialize elements' state parameters, model parameters and external inputs should be defined.

The whole medium volume of interest may be represented as an array, *SpecMediumElement**, (with one index per element as assumed in previous sections) or as *SpecMediumElement****, with three indexes for each single element, which may be more convenient.

4 String Elements and C++ Class Type Variables

Let me demonstrate how all that looks for the well-known elastic string physical and mathematical model.



Consider an element of a string. Let's create a class, describing the element in the same way as in deriving the string wave equation. The element state is fully defined by:

- moment in time, t
- element's position on x-axis, x
- element's length projection on x-axis, Δx
- element's left and right ends' tangent directions, α_1 and α_2, or their trigonometric functions
- element's current vertical displacement from x-axis, u
- element's velocity in the direction perpendicular to x-axis, v
- two global parameters, stretch tension force value, T and linear density, ρ

We have to consider string element's state evolution in time. Let's take the simplest rule of time parameter transformation:

$$t \to t + \Delta t ,$$

Δt is constant.

Parameters x and Δx, for a single element, do not change values.

For now, we can express u and v as:

$$v(t+\Delta t, x) = V(t, x, \Delta x, v(t), u(t), \alpha_1, \alpha_2)$$

$$u(t+\Delta t, x) = U(t, x, \Delta x, v(t), u(t), \alpha_1, \alpha_2),$$

where V and U are some (integration) functions (schemes).

An interesting issue is how we can define parameters α_1 and α_2. I do not want to consider unnecessary complicated schemes involving predefined α_1, and have to average element's curvature, etc. I will consider a much more relevant scheme with explicitly involved elements' interactions (that do not take place in a traditional scheme). In that scheme, the element representing class should contain two pointers to (left and right) neighbors:

```
class_element_{
.....
element*_left;
element*_right;
.....
}
```

Then, we have:

$$\tan\alpha_1 = \frac{u-(left \to u)}{\Delta x}$$

$$\tan\alpha_2 = \frac{u-(right \to u)}{\Delta x}$$

At this point, we already have:

```
class_element_{
.....
double_t;
double_Δt;
double_x;
double_Δx;
double_α_left;
double_α_right;
double_u;
double_v;
element*_left;
element*_right;
public:
double_U(t,Δt,x,Δx,u*,v*,α_left,α_right);
double_V(t,Δt,x,Δx,u*,v*,α_left,α_right);
double_get_alpha(element*_elmnt);
.....
}
```

Implementation of the last function is already known:

```
get_alpha(element*_elmnt)
{
return atan^-1(u-(elmnt→u))/Δx;
},
```

that gives both α_left and α_right, depending on the $elmnt$ argument.

5 Conclusion

I have tried to briefly describe an approach to computer modeling of physical phenomena in continuous media. The Direct Computer Modeling approach is an attempt to use computers and software more adequately, instead of using an outdated method of symbolically representing real processes flows. This approach (or something similar) may become a good alternative to numerical schemes, based on hieroglyphic PDE descriptions.

The approach was initially outlined [4] and described putting more stress on object-oriented implementation [5] in earlier works. A general scheme for continuous media dynamics has also been addressed [6]. As I mentioned before, the first computer program modeling an elastic string dynamics with Direct Computer Modeling demonstrated soliton type string excitation behaviors and showed that split stresses, not the stretch ones, actually defined the whole picture. I mention this example to emphasize that with Direct Computer Modeling we can make further discoveries even in well-known situations [7].

References:

[1] D. E. Post, L. G. Votta, Computational Science Demands a New Paradigm, *Physics Today*, January, 2005, pp.35-41.

[2] V. V. Alexandrov, A. Soiguine, The method of direct computer modeling, *Institute of Informatics of Russian Academy of Science*, preprint #102, 1989.

[3] S. Wolfram, *Cellular Automata and Complexity: Collected Papers*. Reading, MA, Addison-Wesley, 1994.

[4] A. Soiguine, Direct Computer Modeling vs. Traditional Methods of Mathematical Modeling of Physical Processes, *Proceedings of the 1st International Workshop on Human-Computer Interactions, Moscow, Russia, 3-7 August,* pp.707-718, 1991.

[5] A. Soiguine, User Interface in Direct Computer Modeling, *Proceedings of the 2nd International Workshop on Human-Computer Interactions, St. Petersburg, Russia,12-15 August,* 1992, pp.622-626.

[6] A. Soiguine, The Direct Computer Modeling Approach To Continuous Media Dynamics: C++ Implementation, *1st International Conference "From Scientific Computing to Computational Engineering", Athens, Greece.* September 1994.

[7] A. Soiguine, The Direct Computer Modeling Paradigm applied To String Simulation: An Example of Intellectual Medium, *1st International Conference on Experiments/Process/System Modeling/Optimization, Patras, Greece.* July 2005.