NUMEQUARES — ONLINE SIMULATION TOOL FOR EDUCATION AND ACADEMIC RESEARCH

Stepan Orlov[(a)], Nikolay Shabrov[(b)]

[(a),(b)] Computer Technologies in Endineering dept.,
St. Petersburg State Polytechnical University,
St. Petersburg, Russia

[(a)]majorsteve@mail.ru, [(b)]shabrov@rwwws.ru

ABSTRACT

A new online simulation tool, NumEquaRes, is presented. While other simulation tools focus on helping to formulate model equations and provide a limited number of analysis types, NumEquaRes focuses on the formulation of the analysis algorithm to be applied to the model. The model and the algorithm together make a data processing system that can be run in order to obtain numerical results. We have found that some simple algorithms, e.g., generating Poincare map, require too much programming when implemented using popular tools such as MATLAB, Mathematica, Maple, and others. This motivates the development of yet another simulation tool providing much more flexibility for simulation algorithms without having to code them. The paper presents a set of example simulations that illustrate the ability to perform numerical investigations of dynamic systems. NumEquaRes is an open source system available for use and further extension to everyone. It is hosted at http://equares.ctmech.ru.

Keywords: simulation, web application, numerical research.

1. INTRODUCTION

In this work we present a new Web application, NumEquaRes (the name means “Numerical Equation Research”). It is a general tool for numerical simulations available online. Currently, we are targeting small systems of ordinary differential equations (ODE) or finite difference equations arising in the education process, but that might change in the near future — see Sections 8, 9.

The reasons for developing yet another simulation software have emerged as follows. Students were given tasks to deduce the equations of motions of mechanical systems — for example, a disk rolling on the horizontal plane without slip (Routh 1905), or a classical double pendulum (Meirovitch 1986), — and to try further investigating these equations. While in some cases such an investigation can more or less easily be done with MATLAB, SciLab, or other existing software, in other cases the situation is like there is no (freely available) software that would allow one to formulate the task for numerical investigation in a straightforward and natural way.

For example, the double pendulum system exhibits quasi-periodic or chaotic behavior (Meirovitch 1986), depending on the initial state. To determine which kind of motion corresponds to certain initial state, one needs the Poincaré map (Teschl 2012) — the intersection of phase trajectory with a hyperplane. Of course, there are ODE solvers in MATLAB that produce phase trajectories. We can obtain these trajectories as piecewise-linear functions and then compute intersections with the hyperplane. But what if we want 10^4–10^5 points in the Poincare map? How many points do we need in the phase trajectory? Maybe 10^7 or more? Obviously, the simplest approach described above would be waste of resources. A better approach would look at trajectory points one by one, test for intersections with hyperplane, and forget points that are no longer needed. But there is no straightforward way to have simulation process like this in MATLAB.

Of course, there is software (even free software) that can compute Poincaré maps. For example, the XPP (X-Window PhasePlane) tool (Ermentrout 2002) can do that. But what we have learned from our examples is that we need certain set of features that we could not find in any existing software. These features are as follows:

- ability to explicitly specify simulation algorithm;
- reasonable computational performance;
- ease of use by everyone, at least for certain use cases;
- extensibility by everyone who needs a new feature.

The first of these features is very important, but it is missing in all existing tools we tried (see Section 7). It seems that developers of these tools and authors of this paper have different understanding of what a computer simulation can be. Common understanding is that the goal of any simulation is to reproduce the behavior of system being investigated. Numerical simulations

therefore most often perform time integration of equations given by a mathematical model of the system. In this paper, we give the term *simulation* a broader meaning: it is data processing. Given that meaning, we do not think the term is misused, because time integration of model equations often remains the central part of the entire process. Importantly, researcher might need to organize the execution of that part differently, e.g., run initial value problem many times for different initial states or parameters, do intermediate processing on consecutive system states produced by time integrator, and so on.

Given the above general concept of numerical simulation, our goal is to provide a framework that supports the creation of data processing algorithms in a simple and straightforward manner, avoiding any coding except to specify model equations.

Next sections describe design decisions and technologies chosen for the NumEquaRes system (Section 2); simulation specification (Section 3) and workflow semantics (Section 4); performance, extensibility, and ease of use (Section 5); examples of simulations (Section 6); comparison with existing tools (Section 7); next steps to achieve interoperability with modeling and simulation tools (Section 8). Section 9 summarizes current results and presents a roadmap for future work.

2. DESIGN DECISIONS AND CHOICE OF TECHNOLOGIES

Keeping in mind the primary goals formulated above, we started our work.

Traditionally, simulation software have been designed as desktop applications or high performance computing (HPC) applications with desktop front-ends. Nowadays, there are strong reasons to consider Web applications instead of desktop ones, because on the one hand, main limitations for doing so in the past are now vanishing, and, on the other hand, there are many well-known advantages of Web apps.

For example, our “ease of use” goal benefits if we have a Web app, because this means “no need for user to install any additional software”.

Thus we have decided that our software has to be a Web application, available directly in user’s Web browser.

Now, the “extensibility by everyone” goal means that our project must be free software, so the GNU Affero GPL v3 license has been chosen. That should enforce the usefulness of software for anyone who could potentially extend it.

The “Reasonable performance” goal has determined the choice of programming language for software core components.

Preliminary measurements have shown that for a typical simulation, native code compiled from C++ runs approx. 100 times faster than similar code in MATLAB, SciLab, or JavaScript (as of JavaScript, we tested QtScript from Qt4; with other implementations, results might be different). Therefore, we decided that the simulation core has to be written in C++. The core is a console application that runs on the server and interacts with the outer world through its command line parameters and standard input and output streams. It can also generate files (e.g., text or images).

JavaScript has been chosen as the language for simulation description and controlling the core application. However, this does not mean that any part of running simulation is executing JavaScript code.

The decision to use the Qt library has been made, because it provides a rich set of platform-independent abstractions for working with operating system resources, and also because it supports JavaScript (QtScript) out of the box.

Other parts of the applications are the Web server, the database engine, and components running on the client side. For the server, we preferred Node.js over other technologies because we believe its design is really suitable for Web applications — first of all, due to the asynchronous request processing. For example, it is easy to use HTML5 Server Sent Events (W3C 2014) with Node.js, which is not the case with LAMP/WAMP (Wikipedia 2015).

The MongoDB database engine has been picked among others, because, on the one hand, its concept of storing JSON-like documents in collections is suitable for us, and, on the other hand, we do not really need SQL, and, finally, it is a popular choice for Node.js applications.

As of the client code running in the browser, the components used so far are jQuery and jQueryUI (which is no surprise), the d3 library (Bostock 2015) for interactive visualization of simulation schemes, the marked (Jeffrey 2015) and MathJax (Krautzberger 2014) libraries to format markdown pages with TEX formulas.

In the future, we are planning to add 3D visualization using WebGL.

3. SIMULATION SPECIFICATION

The very primary requirement for NumEquaRes is to provide user with the ability to explicitly specify how data flows are organized in a simulation. This determines how simulations are described. This is done similarly to, e.g., the description of a scheme in the Visualization Toolkit (VTK) (Kitware 2010), employing the “pipes and filters” design pattern. The basic idea is that simulation is a *data processing system* defined by a scheme consisting of *boxes* (filters) with *input ports* and *output ports* that can be connected by *links* (pipes). Output ports may have many connections; input ports are allowed to have at most one connection. Simulation data travels from output ports to input ports along the links, and from input ports to output ports inside boxes. Inside each box, the data undergoes certain transformation determined by the box type.

Typically boxes have input and output ports, so they are *data transformers*. Boxes without input ports are *data sources*, and boxes without output ports are *data storage*.

Simulation data is considered to be a sequence of *frames*. Each frame can consist of a scalar real value or

one-dimensional or multi-dimensional array of scalar real values. The list of sizes of that array in all its dimensions is called *frame format*. For example, format {1} describes frames of scalar values, and format {500,400} describes frames of two-dimensional arrays, each having size 500×400. The format of each port is assumed to be fixed during simulation.

Links between box ports are logical data channels, they cannot modify data frames in any way. This means that data format has to be the same at ports connected by a link. Some ports define data format, while some do not; instead, such a port takes format of port connected with it by a link. Thus, data format *propagates along links*. Furthermore, data format can also *propagate through boxes*. This allows to provide quite flexible design to fit the demands of various simulations.

4. SIMULATION WORKFLOW

This section explains how simulation runs, i.e., how the core application processes data frames generated by boxes.

Further, the main routine that controls the data processing is called *runner*.

4.1. Activation notifications

When a box generates a data frame and sends it to an output port, it actually does two things:

- makes the new data frame available in its output port;
- *activates* all links connected to the output port. This step can also be called *output port activation*.

Each link connects an output port to an input port, and its activation means sending notification to input port owner box. The notification just says that a new data frame is available at that input port.

When a box receives such a notification, it is free to do whatever it wants to. In some cases, these notifications are ignored; in other cases, they cause box to start processing data and generate output data frames, which leads to link activation again, and the data processing goes one level deeper. For example, the `Pendulum` box has two input ports, `parameters` and `state`. When a data frame comes to `parameters`, the activation notification is ignored (but next time the box will be able to read parameters from that port). When a data frame comes to `state`, the activation is not ignored. Instead, the box computes ODE right hand side and sends it to the output port `oderhs`.

4.2. Cancellation of data processing

Link activation notification is actually a function call, and the box being notified returns a value indicating success or failure. If link activation fails, the data processing is *canceled*. This can happen when some box cannot obtain all data it needs from input ports. For example, the `Pendulum` box can process the activation of link connected to port `state` only if there are some parameters available in port `parameters`. If it is so, the activation succeeds. Otherwise, the activation fails, and the processing is canceled.

If a box sends a data frame to its output port, and the activation of that output port fails, the box always cancels the data processing. Notice that this is always done by returning a value indicating activation failure, because the box can only do something within an activation notification.

4.3. Data source box activation

Each simulation must have at least one *data source* box — a box having output ports but no input ports. There can be more than one data source in a simulation.

Data sources can be *passive sources* or *generators*. A generator is a box that can be notified just as a link can be. A passive data source cannot be notified.

A passive data source produces one data frame (per output port) during the entire simulation. The data frame is available on its output port from the very beginning of the simulation.

4.4. Initialization of the queue of notifications

When the runner starts data processing, it first considers all data sources and builds the initial state of the *queue of notifications*. For each generator, its notification is enqueued. For each passive data source, the notification of each of its links is enqueued.

4.5. Processing of the queue of notifications

Then the queue is processed by sending the activation notifications (i.e., calling notification functions) one by one, from the beginning to the end. If a notification call succeeds, the notification is removed from the queue. Otherwise, if the notification call fails (i.e., the data processing gets canceled), the notification is moved to the end of the queue, and the process continues.

The runner processes its queue of notifications until it becomes empty, or maximum number of activation notification failures (currently 100) is exceeded. In the latter case, the entire simulation fails.

4.6. Post-processing

When the queue of notifications becomes empty, the runner can enqueue *post-processors* before it stops the data processing. The only example of a post-processor is the `Pause` box. Post-processors, like generators, are boxes that can receive activation notifications.

4.7. User input events

The above process normally takes place during the simulation. In addition, there could be events that break the processing of the queue of notifications. These events are caused by *interactive user input*. Once a user input event occurs, an exception is thrown, which leads to the unwinding of any nested link activation calls and the change of the queue of notifications. Besides, each box gets notified about simulation restart.

The queue of notifications is changed as follows when user input occurs. First, the queue is cleared. Then one of two things happens.

- \item If the box that threw the exception specifies which box should be activated after restart, the notifications for that box are enqueued (if the box is a generator, its activation notification is enqueued; otherwise, the activation notifications of all links connected to its output ports are enqueued). An input box can only specify itself as the next box to activate, or specify nothing.
- \item If the box that threw the exception specifies no box to be activated after restart, the standard initialization of the notification queue is done.

After that, the processing of notification queue continues.

There is an important issue that must be taken care of. Simulation can potentially be defined in such a way that its execution leads to an infinite loop of recursive invocation of activation notifications. This normally causes program to crash due to stack overflow. In our system, however, some boxes (not all, but only those activating outputs in response to more than one input notification) are required to implement counters for recursive call depth. When such a counter reaches 2, simulation is considered to be invalid and is terminated. This allows to do some kind of runtime validation against recursion at the cost of managing call depth counters.

It should be noticed that theoretically, simulations that we are dealing with here are a subclass of discrete event systems (Zeigler, Kim, and Praehofer 2000) with discrete time; one time step corresponds to the returning from an activation call.

5. PERFORMANCE, EXTENSIBILITY, AND EASE OF USE

As stated in Section 1, computational performance and functional extensibility are considered important design features of the NumEquaRes system. This section provides technical details on what has been done to achieve performance and support extensibility. Last subsection highlights design features that make system easier to use.

5.1. Performance

To achieve reasonable performance, it is not enough to just use C++. Some additional design decisions should be made. Most important of them are already described above. The ability to organize simulation workflow arbitrarily allows to achieve efficient memory usage, which is illustrated by an example in Section 1. A number of specific decisions made in the design of NumEquaRes core are targeted to high throughput. They are driven by the following rules.

- Perform simulation in a single thread. While this is a serious performance limitation for a single simulation, we have made this decision because the simulation runs on the Web server, and parallelization inside a single simulation is likely to impact the performance of server, as it might run multiple simulations simultaneously. And, on the other hand, single thread means no synchronization overhead.
- No frequent operations involving interaction with operating system. Each box is responsible for that. For example, data storage boxes should not write output data to files or check for user input frequently. The performance might drop even if the time is measured using `QTime::elapsed()` too frequently.
- No memory management for data frames within activation calls. In fact, almost 100\% of simulation time is spent in just one activation call made by runner (during that call, in turn, other activation calls are made). Therefore, memory management outside activation calls (e.g., the allocation of an element of the queue of notifications) is not a problem. Still some memory allocation happens when a box writes its output data, but this is not a problem as well, since such operations are not frequent.
- No movement of data frames in memory. If a box produces an output frame and makes it available in its output port, all connected boxes read the data directly from memory it was originally written to. This item and the previous one both imply that there are nothing like queues of data frames, and each frame is processed immediately after it is produced.
- No virtual function calls within activation calls. Instead, calls by function pointer are preferred.

A simple architecture of classes has been developed to comply with the rules listed above and, in the same time, to encapsulate the concepts of box, port, link, and others. These classes are split into ones for use at the initialization stage, when simulation is loaded, and others for use at simulation run time. First set of classes may rely on Qt object management system to support their lifetime and the exposure of parameters as JavaScript object properties. Classes of the second set are more lightweight; their implementations are inlined whenever possible and appropriate, in order to reduce function call overhead.

Although NumEquaRes core performance has been optimized in many aspects, it seems impossible to combine speed and flexibility. Our experience with some examples indicates that hand-coded algorithms run several times faster than those prepared in our system.

5.2. Extensibility

The functionality of NumEquaRes mostly resides in boxes. To add a new feature, one thus can write code for a new box. Boxes are completely independent. Therefore, adding a new one to the core simply boils down to adding one header file and one source file and recompiling. The core will be aware of the presence of the new box through its box factory mechanism. Next

steps are to support the new box on server by adding some meta-information related to it (including user documentation page) and some client code reproducing the semantics of port format propagation through the box. The checklist can be found in the online documentation.

Some extensions, however, cannot be done by adding boxes. For example, to add 3D visualization, one needs to change the client-side JavaScript code. We are planning to simplify extensions of this kind; however, this requires refactoring of current client code.

5.3. Ease of use

First of all, NumEquaRes is an online system, so user does not have to download and install any software, provided user already has a Web browser. All user interaction with the system is done through the browser.

To formulate a simulation as a data processing algorithm, user composes a scheme consisting of boxes and links, and there is no need to code.

Online help system contains a detailed documentation page for each box; it also explains simulation workflow, user interface, and other things; there is one step-by-step tutorial.

To prepare a simulation, user can find a similar one in the database, then clone it and modify. User can decide to make his/her simulation public or private; public simulations can be viewed, run, and cloned by everyone. To share a simulation with a colleague, one shares a hyperlink to it; besides, simulations can be downloaded and uploaded.

Currently, user might have to specify part of simulation, such as ODE right hand side evaluation, in the form of C++ code. We understand this might be difficult for people not familiar with C++. To mitigate this problem, there are two features. Firstly, each box that needs C++ code input provides a simple working example that can be copied and modified. Secondly, NumEquaRes supports the concept of *code snippets*. Each piece of C++ input can be given a documentation page and added to the list of code snippets. These snippets can easily be reused by everyone. See also Section 8.

Another feature that plays a role similar to debugger's is the visualization of simulation data flows. The feature is currently under development and not available through the Web interface.

6. EXAMPLES OF SIMULATIONS

This section lists several examples of simulations.

Figure 1 shows one of the simplest simulations — the one that plots a single phase trajectory for a simple pendulum. The ODE system is provided by the `ode` box. NumEquaRes has a number of options for user to supply equations. In particular, it is possible to provide C++ code that computes ODE right hand side. Such code compiles and runs on the server, if it passes a security check. The ODE right hand side depends on the state variables and the vector of parameters. They are supplied through input ports. Parameters are specified in the `odeParam` box. State variables come from the `solver` box. The solver performs numerical integration of the initial value problem, starting from the user-specified initial state (the `initState` box). The solver can be configured to perform a fixed number of time steps or to run until interrupted by a data frame at its `stop` port. Each time the solver obtains a new system state vector, it sends the vector to its `nextState` port. Once the solver finishes, it activates the `finish` port to let others know about it. In this simulation, consecutive system states are projected to the phase plane (the `proj` box) and then rasterized by the `canvas` box. Finally, the data comes to the `bitmap` box that generates the output image file. Notice that this simulation has three data sources, `odeParam`, `solverParam`, and `initState`.

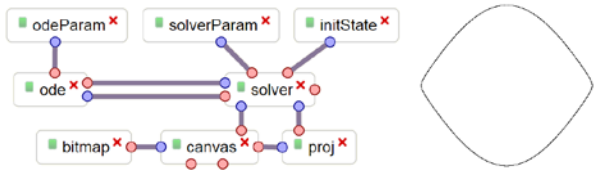

Figure 1: Single phase trajectory

From this simplest example one can see how to construct simulation scheme from boxes and links that computes what user needs. Other examples are more complex, but they basically contain boxes of the same types, plus probably some more. So far, there are 40 different box types in NumEquaRes, and it is beyond the scope of this article to describe them all. Further, we will just focus on some of them to show how simulations work.

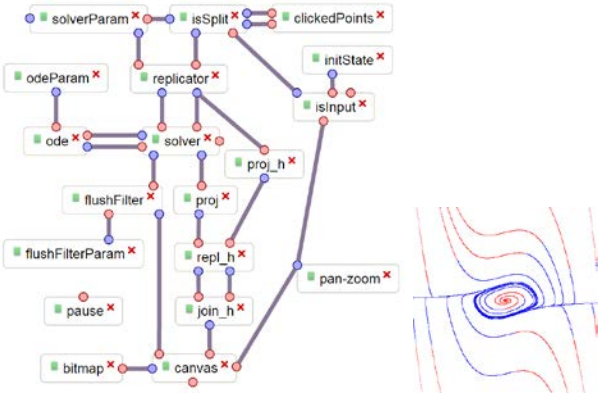

Figure 2: Interactive phase portrait

An important aspect of a simulation is its ability to *interact with the user*. There are a few boxes that transform various kinds of interactive user input (clicking, moving sliders, rotating mouse wheel, etc.) into numerical values. These boxes usually act as simple filters of data frames; they replace some components of data frames with values obtained from user. Figure 2 shows an example of interactive simulation: it generates phase trajectories going through points on plane — the ones user has clicked with the mouse. The box `isInput` is responsible for that kind

of input. Each generated phase curve has two parts: blue in the time-positive direction (with resp. to the clicked point) and red in the time-negative direction.

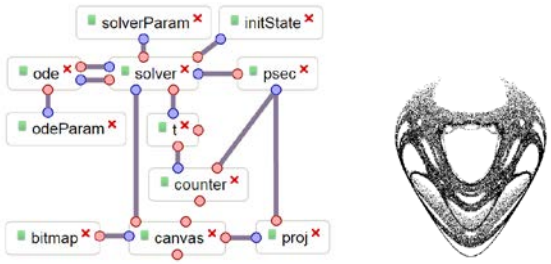

Figure 3: Double pendulum, Poincaré map (50000 points, 28.5 s)

Figure 3 shows the Poincaré map for the classical double pendulum system. Importantly, there is no need to store phase trajectory or individual points of intersection of the trajectory with the plane during simulation. The entire processing cycle (test for intersection; projection; rasterization) is done as soon as a new point of the trajectory is obtained. After that, we need to store just one last point from the trajectory. Simulations like this are what we could not do easily in MATLAB or SciLab, and they have inspired us to develop NumEquaRes.

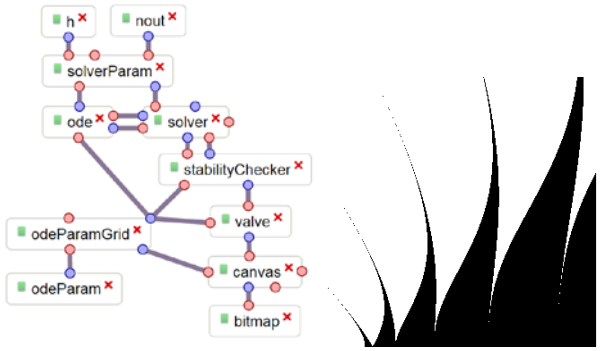

Figure 4: Ince-Strutt stability diagram (500×500 points, 6.3 s)

Figure 4 shows a simple simulation that allows one to obtain a stability diagram of a linear ODE system with periodic coefficients on the plane of parameters. Here the picture on the right is the Ince–Strutt diagram for the Mathieu equation (Abramowitz and Stegun 1972). People who have experience with it know how difficult it is to build such kind of diagrams analytically, even to find the boundaries of stability region near the horizontal axis. What we suggest here is the brute force approach — it is fast enough, general enough, and it is done easily. The idea is to split the rectangle of parameters into pixels and analyze the stability in the bottom-left corner of each pixel (by computing eigenvalues of the monodromy matrix (Teschl 2012)), then assign pixel color to black or white depending on the result. In this simulation, important new boxes are `odeParamGrid` and `stabilityChecker`. The former one provides a way to generate points on a multi-dimensional grid, and the latter one analyzes the stability of a linear ODE system with periodic coefficients.

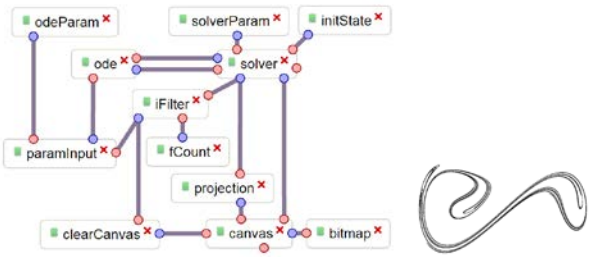

Figure 5: Strange attractor for forced Duffing equation (interactive simulation)

Figure 5 shows another application of the Poincaré map, now in the visualization of the strange attractor arising in the forced Duffing equation (Bender and Orszag 1999). User can change parameters interactively and see how the picture changes. This simulation is simpler than the one shown in Figure 3, because to obtain a new point on canvas, one just needs to apply time integration over known time period of system excitation.

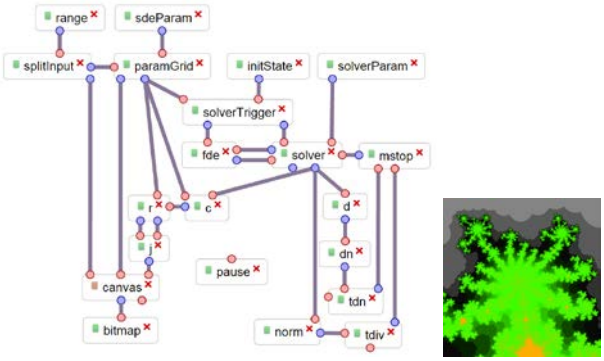

Figure 6: Colored Mandelbrot set (interactive simulation)

Figure 6 shows an interactive simulation of the Mandelbrot set (Mandelbrot 2004). User can pan and zoom the picture using the mouse. Importantly, we did not have to develop any new box types in order to describe the logic of convergence analysis for sequences of complex numbers generated by the system. This is done with general purpose boxes `d` (computes differences of subsequent data frames), `dn` (computes vector norm), and `tdn` (detects if a scalar value exceeds some threshold). Pixel colors depend on how many iterations passed (box `c` is a counter, its output value is joined with pixel coordinates at box `j` and sent to `canvas`).

7. COMPARISON WITH OTHER TOOLS

Direct comparison between NumEquaRes and other existing tools is problematic because all of them (at least, those that we have found) do not provide an easy way for user to describe the data processing algorithm. In some systems, the algorithm can be available as a predefined analysis type; in others, user would have to

code the algorithm; also, there are systems that need to be complemented with external analysis algorithms.
Let us consider example simulations shown in Figures 3, 4, 5, and try to solve those using different free tools; for commercial software, try to find out how to do it from the documentation. Further in this section, figure number refers to the example problem.

Table 1: Comparison of NumEquaRes with other tools

Name	Free	Web	Can solve	Fast
Mathematica	no	yes	3, 4, 5; needs coding	n/a
Maple	no	no	3, 4, 5; needs coding	n/a
MATLAB	no	no	3, 4, 5; needs even more coding	no
SciLab	yes	no		no
OpenModelica	yes	no	none	could be
XPP	yes	no	3, 5	yes
InsightMaker	yes	yes	none	n/a

In Table 1, commercial proprietary software is limited to most popular tools — Mathematica, Maple, and MATLAB. In many cases, purchasing a tool might be not what a user (e.g., a student) is likely to do.
All of the three example simulations are solvable with commercial tools Mathematica, Maple, and MATLAB.
In Mathematica, it is possible to solve problems like 3, 5 using standard time-stepping algorithms since version 9 (released 24 years later than version 1) due to the `WhenEvent` functionality. Problem 4 can also be solved. All algorithms have to be coded. Notice that Wolfram Alpha (Wolfram, 2015) (freely available Web interface to Mathematica) cannot be used for these problems.
Maple has the `DEtools[Poincare]` subpackage that makes it possible to solve problem 3 and others with Hamiltonian equations; problems 4, 5 can be solved by coding their algorithms.
With MATLAB or SciLab, one can code algorithms for problems 4, 5 using standard time-stepping algorithms. For problem 3, one needs either to implement time-stepping algorithm separately or to obtain Poincaré map points by finding intersections of long parts of phase trajectory with the hyperplane. Both approaches are more difficult than those in Mathematica and Maple. And, even if implemented, simulations are much slower than with NumEquaRes.
OpenModelica (Fritzson 2004) is a tool that helps user formulate the equations for a system to be simulated; however, it is currently limited to only one type of analysis — the solution of initial value problem. Therefore, to solve problems like 3, 4, 5, one has to code their algorithms (e.g., in C or C++, because the code for evaluating equations can be exported as C code).
XPP (Ermentrout 2002) provides all functionality necessary to solve problems 3, 5. It contains many algorithms for solving equations (while NumEquaRes does not) and is a powerful research tool. Yet it does not allow user to define a simulation algorithm, and we have no idea how to use it for solving problem 4.
Among other simulation tools we would like to mention InsightMaker (Fortmann-Roe 2014). It is a free Web application for simulations. It has many common points with NumEquaRes, although its set of algorithms is fixed and limited. Therefore, problems 3, 4, 5 cannot be solved with InsightMaker.
Concluding this section, we have to state that other tools either provide a fixed set of data processing algorithms or require them to be coded by users.

8. STEPS TO ACHIEVE INTEROPERABILITY WITH MODELING AND SIMULATION TOOLS

8.1. Importing model equations

NumEquaRes focuses on the development of simulation algorithm and the execution of it. Simulations are dedicated to investigation of certain systems that are described by some equations. Currently, these equations have to be coded in C++ by hand as parameters of some boxes, such as `CxxOde`. We realize that it's easy only for very simple systems. We also realize that there are tools, such as OpenModelica (Fritzson 2004), capable of generating C-code plus some meta-data that describe models specified by user in a more simple and natural way.
To utilize the power of modeling tools in NumEquaRes, it is not necessary to create a monolith system integrating all pieces together. Instead, one could export model equations from a modeling tool and use them in a NumEquaRes simulation. Before year 2010, it was questionable which data format to use for the export of model equations, and how to enable the export feature in a modeling tool. In 2010, the Functional Mock-up Interface (FMI) (Modelica 2015) has been proposed, and since then has been adopted as the model exchange format by many modeling and simulation tools. The format is suitable for describing a model of dynamical system or its part; the system can be described by differential, algebraic, and discrete-time equations with possible state transitions due to events. Therefore, FMI is perfectly suitable for use to import model equations in NumEquaRes.
Using the FMI, a modeling tool exports the equations of a model in the form of Functional Mock-up Unit (FMU). It is quite easy to develop a box in NumEquaRes that wraps an arbitrary FMU and exhibits all necessary parameters and variables through its input and output ports.
Next step is to have a time integrator box in NumEquaRes that could be used for solving the initial value problem for an FMU (currently, we only have explicit Runge — Kutta scheme of fourth order with fixed step-size, which is insufficient). To accomplish this step, we are planning to use the CVODE (Woodward, Reynolds, Hindmarsh, and Banks 2015)

solver, providing its functionality through a box. Other existing freely available time integrators can be used as well in a similar manner.

8.2. Using NumEquaRes core library

The other way to interoperate with NumEquaRes is to use its core library in a simulation tool. The developer of the tool can supply model equations to NumEquaRes through the `CxxOde` box or develop a new one that better integrates with the tool. It might also be necessary to wrap some additional functionality in new boxes. Any tool can use NumEquaRes, provided that the usage conforms to the GNU Affero General Public License, version 3.

9. CONCLUSION AND FUTURE WORK

A new tool for numerical simulations, NumEquaRes, has been developed and implemented as a Web application. The core of the system is implemented in C++ in order to deliver good computational performance. It is free software and thus everyone can contribute into its development. The tool already provides functionality suitable for solving many numerical problems, including the visualization of Poincaré maps, stability diagrams, fractals, and more. NumEquaRes lacks any modeling capabilities, since model equations have to be coded by hand. However, there are ways to interoperate with modeling tools using FMI. Besides, NumEquaRes core library can be used in a simulation tool.

The algorithm of simulation runner implies that the order of activation calls it makes is not important, i.e., does not affect simulation results. While this is true for typical simulations, counter-examples can be invented. Further work is to make it possible to distinguish such simulations from regular ones and render them invalid.

NumEquaRes is a new project, and the current state of its source code corresponds more to the proof-of-concept stage than the production-ready stage, because human resources assigned to the project are very limited. To improve the source code, it is necessary to add developer documentation, add unit tests, and deeply refactor both client and server parts of the Web interface.

Further plans of NumEquaRes development include new features that would significantly extend its field of application. One of them is a box wrapping arbitrary FMUs containing model equations; the other one is a box wrapping the functionality of the CVODE solver.

Another set of planned features aims to enhance the level of presentation of simulation results (currently, it is quite modest). Among them is 3D visualization and animation.

REFERENCES

Abramowitz M., Stegun I., 1972. Handbook of mathematical functions, pp. 721–750. Washington DC: Dover Publications Inc.

Bender C.M., Orszag S.A., 1999. Advanced mathematical methods for scientists and engineers I: Asymptotic methods and perturbation theory, pp. 545–551. New York: Springer-Verlag.

Bostock M. 2015. Data-driven documents. Available from: http://d3js.org [Accessed June 2015].

Ermentrout B., 2002. Simulating, analyzing, and animating dynamical systems: A guide to XPPAUT for researchers and students. SIAM.

Fortmann-Roe S., 2014. Insight Maker: A general-purpose tool for web-based modeling & simulation. Simulation modelling practice and theory, 47:28–45.

Fritzson P., 2004. Principles of object-oriented modeling and simulation with Modelica 2.1. Wiley-IEEE Press, New Jersey.

Jeffrey C., 2015. A markdown parser and compiler. Built for speed. Available from: https://github.com/chjj/marked [Accessed June 2015].

Kitware, Inc., 2010. VTK user's guide. Kitware.

Krautzberger P., 2014. MathJax — beautiful mathematics on the web. Available from: http://pkra.github.io/slides-mathjax [Accessed June 2015].

Mandelbrot B.B., 2004. Fractals and chaos: the Mandelbrot set and beyond. New York, Berlin, Paris: Springer-Verlag.

Meirovitch, L., 1986. Elements of vibration analysis. New York: McGraw-Hill.

Modelica Association, 2015. Functional mock-up interface. Available at https://www.fmi-standard.org [Accessed June 2015].

Routh E.J., 1905. The advanced part of a treatise on the dynamics of a system of rigid bodies. London: Macmillan.

Teschl G., 2012. Ordinary differential equations and dynamical systems. Available from: http://www4.ncsu.edu/~schecter/ma_732_sp13/teschl_ode.pdf [Accessed June 2015].

W3C, 2014. Server-sent events (second edition). Available from: https://w3c.github.io/eventsource [Accessed June 2015].

Wikipedia, 2015. LAMP (software bundle). Available from: https://en.wikipedia.org/wiki/LAMP_(software_bundle) [Accessed June 2015].

Wolfram Research, 2015. WolframAlpha — computational knowledge engine. Available at http://www.wolframalpha.com [Accessed June 2015].

Woodward C.S., Reynolds D.R., Hindmarsh A.C., Banks L.E., 2015. SUNDIALS — suite of nonlinear and dIfferential/algebraic equation solvers. Available at https://computation.llnl.gov/casc/sundials/main.html [Accessed June 2015].

Zeigler B.P., Kim, T.G., Praehofer H., 2000. Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems. Amsterdam, San Diego (Calif.), London: Academic Press.