Designing Data Processing Systems with NumEquaRes

Stepan Orlov and Nikolay Shabrov

Computer Technologies in Endineering dept.
St. Petersburg State Polytechnical University
St. Petersburg, Russia
Email: majorsteve@mail.ru, shabrov@rwwws.ru

***Abstract*—A new Web application for numerical simulations, NumEquaRes, is presented. Its design and architecture are motivated and discussed. Key features of NumEquaRes are the ability to describe data flows in simulations, ease of use, good data processing performance, and extensibility. Simulation building blocks and several examples of application are explained in detail. Technical challenges specific to Web applications for simulations, related to performance and security, are discussed. In conclusion, current results are summarized and future work is outlined.**

Keywords–Simulation; Web application; Ordinary differential equations.

I. Introduction

This work is an extended version of paper [1]. We present a new Web application, NumEquaRes [2] (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 Section XI.

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 [3], or a classical double pendulum [4], — 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 [4], depending on the initial state. To determine which kind of motion corresponds to certain initial state, one needs the Poincaré map [5] — 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 Poincaré map? How many points do we need in the phase trajectory? Maybe 10^7 or more? Obviously, the simplest approach described above would be a 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 a 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 [6] 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 how data flows in a simulation should be organized;
- 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 IX). 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. Therefore, numerical simulations most often perform time integration of equations given by a mathematical model of the system. In this paper, we give the term *simulation* a more general 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, a 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 II); simulation specification (Section III) and workflow semantics (Section IV); software architecture overview (Section V); performance, extensibility, and ease of use (Section VI); simulation building blocks (SectionVII); examples of simulations (Section VIII); comparison with existing tools (Section IX); technical challenges conditioned by system design (Section X). Section XI summarizes current results and presents a roadmap for future work.

II. 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. Our 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 [7] with Node.js, which is not the case with LAMP/WAMP [8].

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 [9] for interactive visualization of simulation schemes, the marked [10] and MathJax [11] libraries to format markdown pages with $\TeX$ formulas. In the future, we are planning to add 3D visualization using WebGL.

III. 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) [12], 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. Figure 1 shows an example of sequences of data frames of different formats.

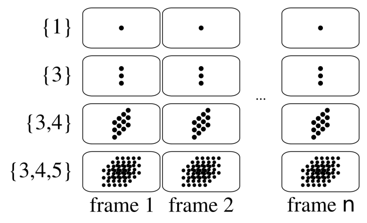


Figure 1. Examples of data frame sequences.

In addition, NumEquaRes supports *element labels* for scalar and one-dimensional data frames. The idea is to give a name to each element of a data frame. Due to labels, user can easily identify parameters specified for `Param` boxes (see Section VII-A), and have better understanding of the data. Notice that labels are not part of frame format, so format compatibility check does not rely on labels.

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 the format of the port connected to it by a link. Thus, data format *propagates along links* (together with element labels, if any). Furthermore, data format can also *propagate through boxes*. This allows to provide a quite flexible design to fit the demands of various simulations.

Figure 2 shows an example of connections between box ports. Each box has a type (e.g., **`Param`**, displayed in bold face) and a name (e.g., `odeParam`), and some input and/or output ports. The figure shows data flow direction along links with solid arrows, and frame format propagation direction with dashed arrows. For ports defining data format, dashed arrows start with a diamond. Notice, e.g., how the `odeInitState` box in this example knows that user should specify values $q, \dot{q}, t$ for pendulum initial state: `odeInitState` receives the format $\{3\}$ and element labels $q, \dot{q}, t$ from the `initState`

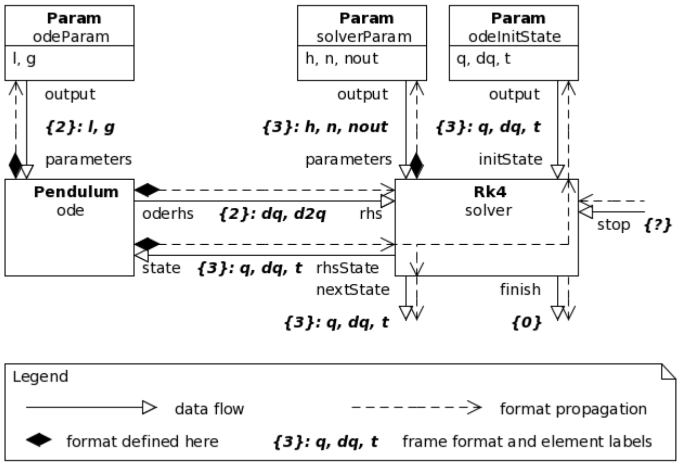


Figure 2. Boxes, ports, links, and frame format propagation.

port of box `solver`. The format of that port is induced by the format of port `rhsState` of the same box (due to format propagation through boxes of type `Rk4`, see Section VII-E4). The `rhsState` port of box `solver` receives the frame format and element labels from port `state` of box `ode`. The `ode` box is the origin of format and labels.

IV. 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*.

A. 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* (Figure 3).

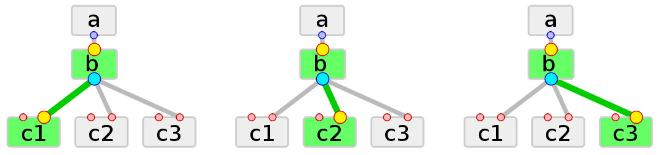


Figure 3. Output port activation (box `b` activates its output port).

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`.

B. 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.

C. 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.

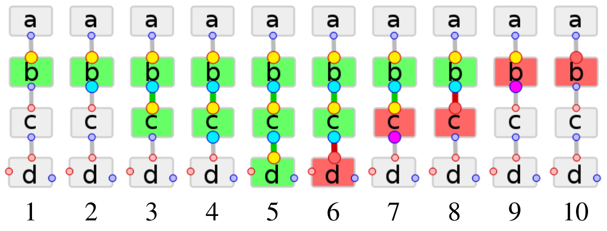


Figure 4. Data processing cancellation.

Figure 4 illustrates the cancellation of data processing: box `a` is the only data source, and its output port is connected to the input port of box `b`. Therefore, the runner activates the input port of `b` (1). Then `b` activates `c` (2, 3), and `c` activates `d` (4, 5). For some reason (e.g., no data on another input port), `d` returns activation failure (6). Callers are obliged to return activation failure as well, therefore the runner finally gets the activation failure (7–10).

D. 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.

E. 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.

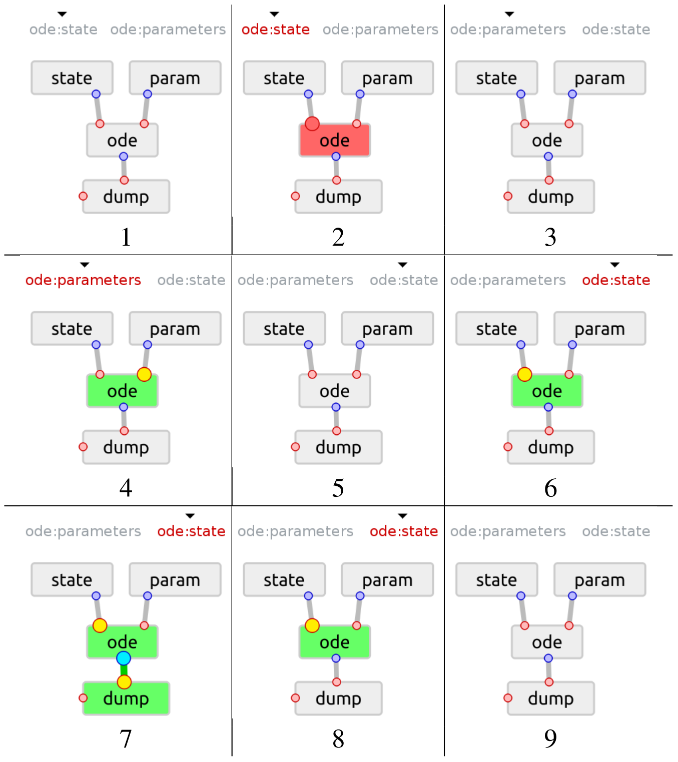


Figure 5. Data processing example.

To illustrate the data processing in a simulation, consider the following example. A box of type `CxxOde` (see Section VII-D10) has two input ports, `state` and `parameters`, and one output port, `rhs`. It ignores activation calls for port `parameters`. On the other hand, the activation of port `state` causes the box to compute ODE right hand side and write it to the output port `rhs`. But the right hand side can only be computed when the parameters are known, i.e., a data frame is available at port `parameters`. Otherwise, the activation of port `state` fails.

Now imagine a simulation with box `ode` of type `CxxOde` and two passive data sources, `state` and `param`, connected to the `state` and `parameters` ports of `ode`, respectively. Besides, the output port of `ode` is connected to the input port of a data storage box.

The runner does not know that `ode` wants parameters before state, so suppose it initializes the initial queue of notifications such that the port `ode:state` is first, and port `ode:parameters` is second. The data processing in this situation is shown in Figure 5 and is as follows.

1) Runner is about to activate port `ode:state`.
2) Runner activates port `ode:state`, and the data processing is canceled by `ode` because there is no data at port `ode:parameters`.
3) Notification for port `ode:state` is moved to the end of the queue. Runner is about to activate port port `ode:parameters`.
4) Runner activates port `ode:parameters`; the activation notification is ignored by `ode`, and control returns back to the runner.
5) Since the activation of `ode:parameters` succeeds, the runner proceeds to next element of the queue, which is `ode:state`.
6) Runner activates port `ode:state`.
7) `ode` computes ODE right hand side and sends it to the output port `ode:rhs`, which leads to the activation of the input port of box `dump`.
8) The `dump` box writes the incoming data frame to a text file and returns control back to `ode`.
9) The box `ode` has nothing more to do in response to the activation of the `state` port, so it returns control back to the runner. There are no more items in the notification queue, and simulation finishes.

F. 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.

G. 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.

- 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.
- 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

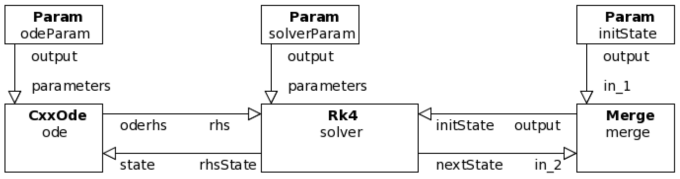


Figure 6. Example of invalid simulation (recursive activation of box `merge`).

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 recursive call counters. Figure 6 shows an example of invalid simulation that will detect recursive activation of box `merge`: First, its port `in_1` is activated by runner, which starts numerical integration in `solver`; once the solver outputs next state, it comes to port `in_2` of box `merge`. At this point, `merge` detects recursive activation, because the activation of port `in_1` is still in progress. As a result, the simulation fails.

V. SOFTWARE ARCHITECTURE OVERVIEW

This section presents an overview of the architecture of software that implements NumEquaRes.

Essentially, the software consists of the computational core and the web server, and can be deployed by everyone on any server machine running a Linux operating system. Both components are open source, hosted at GitHub (a link to the project is available on the web site [2]).

The computational core is a console application written in C++. Its responsibility is to load simulation specification, run the simulation, and communicate with the controlling process. The communications are necessary for supplying user input and synchronizing with the controlling process.

The web server is written in Node.js and is using several third party packages, most noticeably the express framework [13]. The web server has numerous responsibilities, including the following:

- generating and serving web pages;
- serving files;
- managing user accounts and data;
- managing user sessions;
- handling Ajax [14] requests done by the code running in browser on client machines;
- controlling the computational core.

Notice that web pages sent by the web server to a client contain JavaScript code to be executed by the web browser on the client machine, and that code communicates with the server using the Ajax technology. Therefore, we actually have an application distributed among server and client machines, which is nowadays typical for any web application.

The management of any user data requires a mechanism for persistent data storage. For this purpose, the MongoDB database engine has been chosen.

The interaction between software components is outlined in Figure 7. Large containers represent the server machine, the client machine, and the Internet between them. Rectangular-shaped elements in the containers represent software components that are parts of NumEquaRes or are used by it. Elliptical-shaped elements represent file system folders. Arrows between elements indicate data flow directions; arrow captions explain activities causing the corresponding data transfers.

The detailed discussion of software component architecture is beyond the scope of this paper. However, let us focus on the most important question, namely the interaction between the user, the web server, and the computational core.

User prepares a simulation in the *editor* HTML page. The page is sent by the web server to the client when requested, e.g., through the main menu available in all pages. It contains JavaScript code allowing to design the simulation from scratch or to load an example and further modify it if necessary. When the user prepares a simulation, little interaction with the web server can take place. This is only the case when the user modifies the C++ source code in a box like `CxxOde` (see Section VII-D10) and wants to know if the compilation is successful. Most of the time, no interaction with the server is necessary to prepare a simulation, so this is done locally on the client machine.

Once the user finishes preparing the simulation, the simulation can be saved in the database or run on the server machine. To manage that, the client sends the simulation to the web server as a JSON object within an HTTP request. Handling these requests, the web server either interacts with the database or runs the computational core, depending on the request. In the latter case, the JSON object describing the simulation is passed to the computational core through its standard input stream.

The computational core is a console application that implements a simple text protocol allowing the web server to interact with it. Writing specific lines of text to the standard input causes some commands to be executed, like start or stop the simulation, provide interactive input data, etc. On the other hand, the server reads the standard output of the computational core. Importantly, when the simulation starts, the core writes *annotations* for output files and input controls there — the web server sends these annotations to the client, so the client knows which files need to be requested as simulation results, and which elements need to be created for obtaining user input. Other important things written by the computational core to its standard output are the *synchronization markers*. Once the core writes an output file, it also writes such a marker to the standard output and waits for the corresponding marker on its standard input. At this point, the server reads the marker from the core, informs the client about the update of the output file, and then writes the synchronization marker to the standard input of the computational core. The core reads the marker and resumes the execution of the simulation. Notice that the marker based synchronization is not frequent (e.g., once per second, or other user-specified time period), therefore it does not impact the overall performance.

Notice also that when a simulation is running, the web server sends data from the computational core to the client using HTML5 Server Sent Events [7].

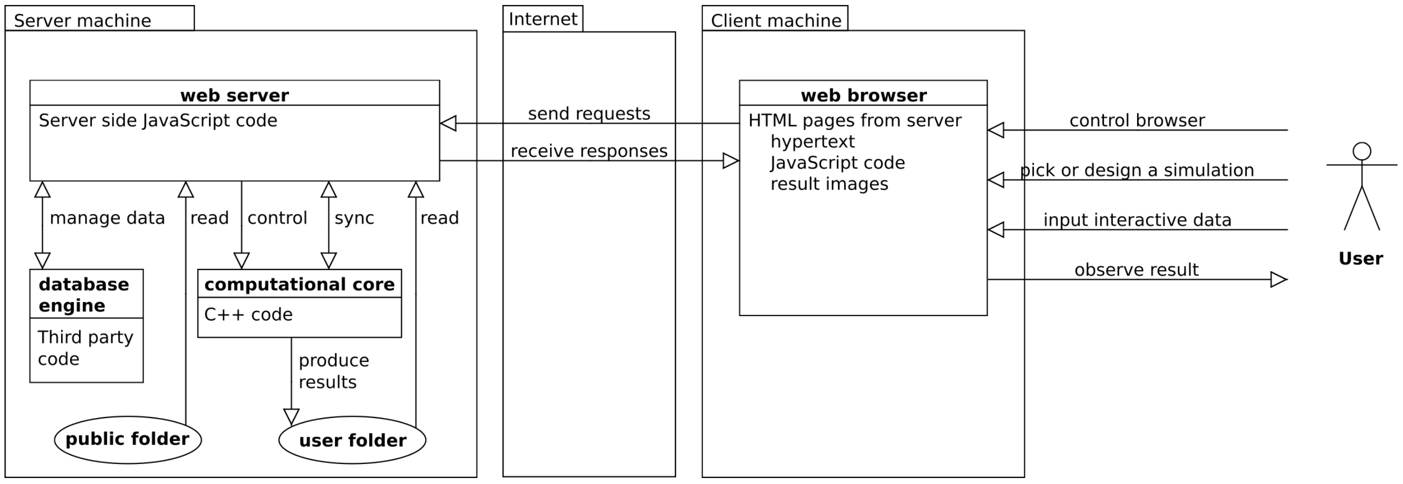


Figure 7. Interaction among software components of NumEquaRes and users.

When the user provides interactive input data for the running simulation, the browser sends requests to the web server; handling them, the server writes corresponding commands to the standard input of the computational core, so the core knows what the user input is. The computational core reads the standard input stream in a separate thread, which is synchronized with the main worker thread not too frequently in order not to impact the performance (see next section).

VI. Performance, extensibility, and ease of use

As stated in Section I, 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.

A. *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 I. 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.

B. *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.

C. *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 be created and reused by everyone.

VII. Simulation building blocks

This section explains the semantics of different boxes from which NumEquaRes simulations are built. There are currently 40 types of boxes; this section categorizes them and describes most important boxes. Knowing how the boxes work gives understanding of NumEquaRes simulations design.

In this section, we introduce the notation `Box:port`, where `Box` is the type of a box, and `port` is a name of one of its ports. For example, `Param:output` means the `output` port of a box of type `Param`; the part `Box:` is omitted when the box type is obvious from context.

A. *Source boxes*

There are three types of boxes that can be used as data sources: `Const`, `Param`, and `ParamArray`. The `Const` and `Param` boxes behave identically once their parameters are specified. A box of type `Const` or `Param` generates just one data frame (see Section III) per simulation run. The format of the frame is a one-dimensional array. The contents of the array is determined by fixed parameters of the box. So what user enters as parameters is turned by the box into a data frame.

The difference between `Param` and `Const` is how they manage their frame format. The `Param` box expects its format to be specified in a port connected to its `output` port. Therefore, before parameters can be specified for a `Param` box, it has to be connected to something providing a data format. For example, if there is a link `Param:output -> Pendulum:parameters`, the format of data frame generated by `Param` will be $\{2\}$ — an array of two elements which are the parameters of a pendulum: the length, l, and the acceleration of gravity, g. The parameters of the `Param` box will be l and g — see Figure 16 (a).

In contrast to `Param`, a box of type `Const` allows user to enter an arbitrary one-dimensional array of parameters. The data format of the box is determined by the user-specified array of parameters.

The `Param` type should typically be preferred to `Const` because its use guarantees format compatibility along the link between `Param:output` and another port. `Const` should only be used when connected port does not provide a frame format. Notice also that if `Param` is connected to more than one port, and these ports provide different sets of element labels, the box cannot be used because it cannot resolve parameter names.

The `ParamArray` box type is similar to `Param`, but it allows user to specify an array of sets of parameters. When user specifies an array of length n, the box generates n output data frames per simulation run. Combining `ParamArray` with an `Interpolator` box (see Section VII-E1 and Figure 18 (b)) allows to generate many data frames in which parameters gradually change between values specified in `ParamArray`. In addition, `ParamArray` box has port `flush`. The box writes an empty data frame to that port after it writes all data frames to `output`.

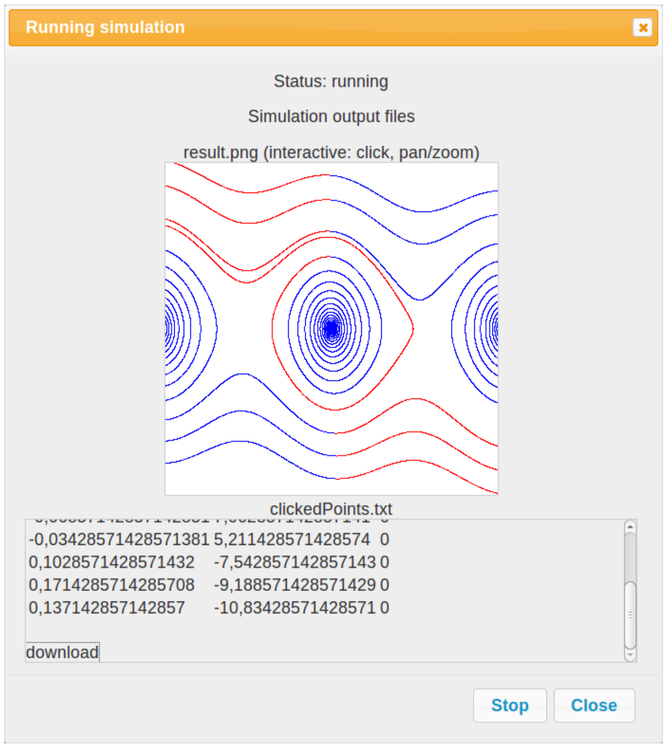
B. *Data storage boxes*

There are two types of boxes that store incoming data frames: `Dump` and `Bitmap`.

Each data storage box in a simulation corresponds to a file generated in user's directory on server. When the simulation runs, user sees these files in browser. Text files appear as tables and can be downloaded as text; image files are seen as images — see Figure 8.

A box of type `Dump` stores data frames of any format coming to its `input` port in a text file. The incoming frames are output to the file as lines of formatted decimal numbers. There is a limitation of 10^6 on total number of scalar values written to the file in order to avoid occasional generation of a large file on server.

A box of type `Bitmap` stores incoming data frames in an image file in the PNG or JPEG format. Each data frame coming to port `Bitmap:input` is transformed into a colored image as follows. The format of input data frame should be $\{w, h\}$, where w is the width, and h is the height of the image, in pixels. Each scalar element of incoming 2D data frame is transformed into a 32-bit RGB color value using the *color map*. The color map is a mapping from scalar value to color value; it is specified as a parameter of the `Bitmap`



```
-0,03428571428571381 5,211428571428574  0
0,1028571428571432   -7,542857142857143 0
0,1714285714285708   -9,188571428571429 0
0,137142857142857    -10,83428571428571 0
```

Figure 8. Simulation output files generated by storage boxes.

box. There is a limitation of 2000 pixels on w and h in order to avoid occasional generation of large files on server. Notice that `Bitmap` boxes usually receive data frames from `Canvas` boxes — see below.

C. *The `Canvas` box*

The `Canvas` box provides intermediate 2D array to store scalar values, e.g., for further image generation. It is initially filled with zero values. The box has several input ports, one output port, and a set of parameters.

The geometry of canvas is determined by its *range* and *resolution* in each of its two dimensions. The range is a pair of numbers $\{x_{\min}, x_{\max}\}$ for the x dimension and $\{y_{\min}, y_{\max}\}$ for the y dimension; the resolution is the number of array elements, N_x or N_y — see Figure 9 (a). Ranges and resolutions are canvas parameters; ranges can also be supplied through port `range`, such that each frame is $[x_{\min}, x_{\max}, y_{\min}, y_{\max}]$.

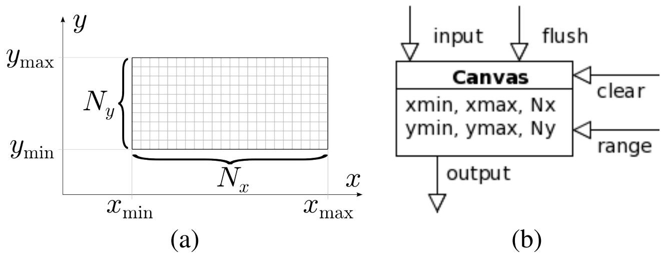


Figure 9. (a) Canvas geometry; (b) `Canvas` box ports and parameters.

The box receives input data at port `input`. The format must be $\{2\}$ or $\{3\}$ — a one-dimensional array of size 2 or 3. First two elements of an incoming data frame are the coordinates x, y of a point. The third element, if present, is a scalar value v; it defaults to 1 if absent. For each input data frame, the box computes canvas coordinates x_c, y_c using the formula

$$x_c = \left\lfloor N_x \frac{x - x_{\min}}{x_{\max} - x_{\min}} \right\rfloor, \qquad y_c = \left\lfloor N_y \frac{y - y_{\min}}{y_{\max} - y_{\min}} \right\rfloor,$$

and writes v into the array using x_c and y_c as indices, if they are valid ($0 \le x_c < N_x$, $0 \le y_c < N_y$).

The output of canvas data occurs either when user-specified timeout is exceeded, or when a data frame comes to port `flush`. Output data frame contains all values currently stored in the 2D array and has format $\{N_x, N_y\}$.

There is also port `clear`; when it receives a data frame, all elements of the internal array assign zero values. All box ports are shown in Figure 9 (b).

D. *Boxes for simple transformations and filters*

Many boxes have quite simple logics, producing one output data frame in response to incoming data frames. They have a primary input port, an output port, and optional input ports; they can also have parameters controlling their behavior. Below we briefly describe some of such simple boxes.

1) CountedFilter: The box passes to port `output` each n-th data frame coming to port `input`; n is the parameter of the box received through input port `count`.

2) Counter: The box counts data frames received in port `input`. Each time the counter value increases, it is sent to port `count`. The counter value can be set to zero by sending a data frame to port `reset`.

3) CrossSection: Data frames received at port `input` are one-dimensional arrays of length n. The elements of i-th data frame are interpreted as coordinates $[x_0^i, x_1^i, \dots x_{n-1}^i]$ of point $\mathbf{x}^i$; the points $\mathbf{x}^i$ are considered to be consecutive points on a piecewise-linear curve in n-dimensional space. The box outputs points of intersection of the curve and the hyperplane $x_k = c$, where k and c are box parameters (Figure 10). Another box parameter allows to count only intersections with x_k increasing along the curve or decreasing along the curve. The `CrossSection` box is crucial for simulations that visualize Poincaré maps.

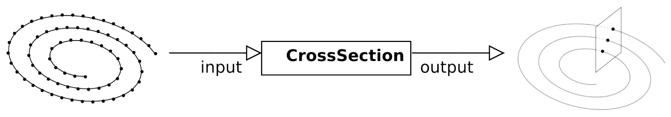


Figure 10. `CrossSection` box input and output.

4) IntervalFilter: The box is similar to `CrossSection`, but instead of one hyperplane $x_k = c$ it considers many hyperplanes, specified by equation $x_k = c + Tm$, where m is an arbitrary integer number, and T is a box parameter. The box considers that x_k increases monotonously in incoming points, since it is usually the time. The box can be used to visualize Poincaré maps in systems with periodic excitation.

5) Differentiate: The box computes differences between consecutive points $\mathbf{x}^i$, $\mathbf{x}^{i+1}$ coming to port `input`:

$$\mathbf{d}^i = \mathbf{x}^{i+1} - \mathbf{x}^i$$

The differences $\mathbf{d}^i$ are written to port `output`.

6) Scalarize: For each data frame $\mathbf{x} = [x_0, x_1, \dots x_{n-1}]$ received at port `input`, the box generates a scalar value v and writes it to port `output`. The method used to compute the scalar is the box parameter; user can choose it among several common norms, minimum, and maximum.

7) Projection: The box accepts input data frames of arbitrary format at its `input` port. Each data frame is interpreted as an array of values, $x_0^{in}, \dots x_{n-1}^{in}$, where n is the total number of elements in the incoming data frame. Once an input data frame is received, an output data frame is generated and written to port `output`. The output data frame contains m elements $x_0^{out}, \dots x_{m-1}^{out}$ and has format $\{m\}$. The elements of the output data frame are picked from input as follows:

$$x_k^{out} = x_{i_k}^{in}, \quad k = 0, \dots m-1.$$

In the above formula, the indices i_k are box parameters. The box is often used, for example, to pick two variables from a vector for plotting on `Canvas` (see Section VII-C and Figure 18 (a)).

8) Eigenvalues: The box expects a square matrix at its input port `matrix`, so the port format is $\{n,n\}$. As soon as a matrix is obtained, its eigenvalues are computed. The real parts of the eigenvalues are then written to output port `eig_real`, and imaginary parts are written to port `eig_imag`.

The implementation of this box uses the ACML library [15].

9) ThresholdDetector: The box receives a scalar-valued data frames, x, at port `input`. For each incoming value, the value v is computed as follows: the logical expression $x * T$ is evaluated; if the result is true, v is set to one; otherwise, v is set to zero. In the above expression, the binary operator $*$ can be one of $<, \leq, \geq, >, =, \neq$ and is determined by box parameter; the threshold value T can be either specified as a box parameter or passed in through port `threshold`.

Once v is computed, it is normally written to port `output`. For more flexibility, the box allows to suppress the output of zero values of v by specifying another box parameter, `quiet`. Values $v = 1$ are always written to `output`.

10) Other transformations: There are a number of other boxes that perform transformations. They all write a data frame $\mathbf{y}(\mathbf{x})$ to the output port as soon as they obtain a data frame $\mathbf{x}$ at the input port. There could be an additional input port for parameters. Here these box types are listed.

- `CxxFde` — a user-defined transformation. The box receives $\mathbf{x}$ at port `state` and writes $\mathbf{y}$ to port `nextState`. Both $\mathbf{x}$ and $\mathbf{y}$ are vectors of length n. The transformation is defined by user in the form of C++ source code. The code can also describe parameters to be obtained at input port `parameters`. The box is designed primarily for use with the `FdeIterator` box (see Section VII-E3) as the source of a system of finite difference equations.
- `CxxOde` — another user-defined transformation. The box receives $\mathbf{x}$ at port `state` and writes $\mathbf{y}$ to port `rhs`. The vector $\mathbf{x}$ contains n state variables and the time: $\mathbf{x} = [x_1, \dots x_n, t]$. The vector $\mathbf{y}$ contains n time derivatives of state variables: $\mathbf{y} = [\dot{x}_1, \dots, \dot{x}_n]$. The transformation and additional parameters to be obtained at input port `parameters` are defined by user in the form of C++ source code. The box is designed primarily for use with the `Rk4` box (see Section VII-E4) or other future solvers as the source of a system of ordinary differential equations.
- `CxxTransform` — yet another user-defined transformation. It gives user freedom to select arbitrary formats of data frames for $\mathbf{x}$ and $\mathbf{y}$. The transformation and optional parameters are also specified in the form of C++ code. The box can be used, e.g., to formulate a linear system of ordinary differential equations to further investigate the dependency of its stability on parameters with the `Eigenvalues` box (see Section VII-D8).
- `Pendulum`, `DoublePendulum`, `Mathieu`, `VibratingPendulum` — these are examples of hard-coded systems of ordinary differential equations; the logics and sets of ports in each of these boxes are the same as in the `CxxOde` box, therefore, they are interchangeable with `CxxOde`.

Notice that since simulations run on server side, C++ source code specified by user for boxes `CxxFde`, `CxxOde`, `CxxTransform`, is compiled and run on server. This creates potential security problem — running malicious code on server. The problem is addressed in Section X-B.

E. Iterators and solvers

Boxes described in this section are essentially iterators. The implementation of such a box contains a loop in its activation handler function, and output data frames are generated inside the body of the loop. Due to this, the activation of an input port can cause the generation of many output data frames.

1) Interpolator: Data frames received at port `input` are one-dimensional arrays of length n. The elements of i-th data frame are interpreted as coordinates $[x_0^i, x_1^i, \dots x_{n-1}^i]$ of point $\mathbf{x}^i$ in n-dimensional space. For each pair of consecutive points $\mathbf{x}^i$, $\mathbf{x}^{i+1}$, the box generates $N-1$ intermediate points $\mathbf{x}^{i,1}, \dots, \mathbf{x}^{i,N-1}$ using the formula

$$\mathbf{x}^{i,k} = (1 - t_k)\mathbf{x}^i + t_k \mathbf{x}^{i+1}, \quad t_k \equiv \frac{k}{N}, \quad k = 1, \dots N-1.$$

All points, including original $\mathbf{x}^i$ and interpolated $\mathbf{x}^{i,k}$ are passed to port `output`. N above is the number of interpolation intervals; it is a parameter of the box. Interpolator input and output are shown in Figure 11.

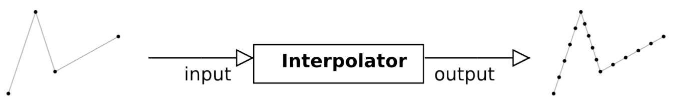


Figure 11. `Interpolator` box input and output.

2) GridGenerator: This box produces a d-dimensional grid of values for each data frame coming to its `input` port. Input data frames must be one-dimensional arrays.

For each input data frame, the grid generator produces, N output frames, $N = N_0 N_1 \dots N_{d-1}$, where N_k is the grid size in k-th dimension, $0 \leq k < d$.

The grid consists of points $\mathbf{x}_I$. Each point of the grid is identified by multi-index $I = i_1, i_2, \dots, i_d$ (indices i_k run from 0 to $N_k - 1$) and has coordinates $x_I^1, x_I^2, \dots, x_I^d$. The coordinates x_I^k are computed by linear interpolation between parameters $x^{k,\min}$, $x^{k,\max}$:

$$x_I^k = \left(1 - t_I^k\right) x^{k,\min} + t_I^k x^{k,\max}, \quad t_I^k \equiv \frac{i_k}{N_k - 1}.$$

Notice that $x^{k,\min}$, $x^{k,\max}$ define *ranges*, just like for Canvas (see Section VII-C); they can either be specified directly as box parameters or supplied through input port range.

For each grid point, a data frame is generated and written to port output (Figure 12). The format of ports input and output is the same. The elements in the output data frame repeat those from the input data frame, except that d elements are replaced by coordinates x_I^k. The indices of replaced elements are box parameters.

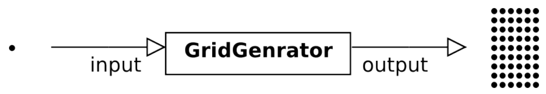


Figure 12. GridGenerator box input and output.

In addition, the box generates empty data frame and sends it to port flush as soon as all output data frames for one input data frame are generated.

The GridGenerator box is very useful when it is necessary to repeat the same operation for parameters varying in certain ranges. One of its applications is the generation of stability diagrams (see Figure 24).

3) FdeIterator: As follows from the box name, it performs iterations of finite difference equations (FDE). The equations are formulated outside the box and should be connected to ports fdeIn, fdeOut.

Suppose that the state of a discrete-time system at k-th time step is described by vector $\mathbf{x}_k$. The explicit form of FDE gives the formula to compute the state of the system at next time step:

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k).$$

To evaluate this formula, the FdeIterator writes $\mathbf{x}_k$ to port fdeOut and afterward expects that $\mathbf{x}_{k+1}$ has come to port fdeIn. The iterations proceed by reading data frames from fdeIn and writing them to fdeOut. In addition, data frames are written to output port nextState — this way we normally make use of the resulting points $\mathbf{x}_k$. For more flexibility, parameters n_o and n_s can be specified for the box, that control which points are written to port nextState: n_s is the number of initial points to skip, and n_o is the number of time steps between consecutive outputs. For example, if $n_s = 10$, $n_o = 2$, the points written to nextState are $\mathbf{x}_{10}, \mathbf{x}_{12}, \mathbf{x}_{14}, \ldots$ Notice also that setting the value of n_o to zero causes only the last system state to be written to nextState.

The iterative process is initiated by sending the initial state of the system, $\mathbf{x}_0$, to port initState. The process is controlled by parameters, supplied through port parameters, which are n_s and n_o described above, and the total number of iterations, n. If n is zero, the iterations never end. However, sending any data frame to port stop causes the iteration loop to terminate.

When iterations finish, an empty data frame is sent to port finish.

Ports of the FdeIterator box are shown in Figure 13 (a).

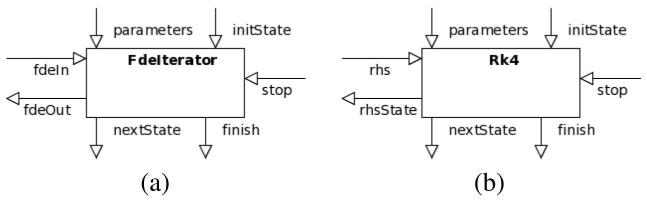


Figure 13. (a) FdeIterator box ports; (b) Rk4 box ports.

4) Rk4: This box has logics very similar to that of FdeIterator, but the box is designed to obtain numerical solution of a system of ordinary differential equations (ODE). The ODE system in the normal form is specified by formula

$$\dot{\mathbf{x}} = f(\mathbf{x}, t),$$

where $\mathbf{x}$ is the state vector, t is the time, and $(\ldots)^{\cdot}$ is the time derivative.

The box implements the well known Runge — Kutta explicit 4-th order numerical integration scheme [16], hence its name. It has ports similar to ports of FdeIterator, but the ports for interaction with the ODE system are named rhsState and rhs, and they are slightly different from FdeIterator:fdeOut, FdeIterator:fdeIn. To evaluate ODE right hand side, the Rk4 box writes a data frame containing system state vector $\mathbf{x}$ and the time t to port rhsState; it then expects the vector $f(\mathbf{x}, t)$ in port rhs.

Parameters of the Rk4 box supplied through port parameters are the time integration step, the number of steps to perform, and n_o (see Section VII-E3).

All ports of the Rk4 box are shown in Figure 13 (b).

5) LinOdeStabChecker: The box is designed to analyze the stability of linear ODE systems with periodic coefficients and zero right hand side:

$$\dot{\mathbf{y}} = \mathbf{A}(t)\mathbf{y}, \quad \mathbf{A}(t + T) = \mathbf{A}(t),$$

where $\mathbf{y} = [y_1, \ldots, y_n]^T$ is the vector of n state variables, t is the time, and $\mathbf{A}$ is an $n \times n$ matrix of coefficients, which are considered to be periodic functions of time, with period T.

The stability analysis is done as follows [5]. Take n initial states at $t = 0$, $\mathbf{y}^1(0), \ldots \mathbf{y}^n(0)$ such that

$$\mathbf{y}^k(0) = \left[y_1^k(0), \ldots y_n^k(0)\right]^T, \quad y_k^s(0) = \delta_k^s \equiv \left[\begin{array}{ll} 1, & \text{if } s = k \\ 0, & \text{if } s \neq k \end{array}\right.$$

In other words, vectors $\mathbf{y}^k(0)$ make up the $n \times n$ identity matrix: $\mathbf{Y}(0) \equiv \left[\mathbf{y}^1(0), \ldots \mathbf{y}^n(0)\right] = I$, $I_{ks} = \delta_{ks}$. For each initial state $\mathbf{y}^k(0)$, the initial value problem is solved and $\mathbf{y}^k(T)$ are obtained. Then the stability is determined by characteristic multipliers ρ_k — the eigenvalues of the monodromy matrix (system fundamental matrix computed at period T):

$$\mathbf{M}\mathbf{z}_k = \rho_k \mathbf{z}_k, \quad \mathbf{M} = \mathbf{Y}(T) \equiv \left[\mathbf{y}^1(T), \ldots \mathbf{y}^n(T)\right]$$

The solution is stable if the absolute value of each multiplier does not exceed 1, and is unstable if there is at least one k such that $|\rho_k| > 1$.

Practically, in many cases, for multipliers we have $|\rho_k| = 1$ (for all k) if the system is stable, and $|\rho_k| > 1$ (for one or more k) if the system is unstable. For such systems, numerical solution will most likely always give $1 < |\rho_k| < 1 + \varepsilon$ if the system is stable, where $\varepsilon \ll 1$ is a small value. Therefore, the stability detection is based on checking inequality $|\rho_k| < 1+\varepsilon$ rather than $|\rho_k| \le 1$, and $\varepsilon = 10^{-5}$ is a hardcoded constant.

The box does not need to know the period T; however, the solver connected to the box should return the state $\mathbf{y}(T)$ when given initial state $\mathbf{y}(0)$.

The box is connected to an ODE solver by output port `initState`, to pass initial state $\mathbf{y}(0)$ to it, and by input port `solution`, to obtain the system state $\mathbf{y}(T)$. Since solver implementation typically involves the use of `Rk4` box, data frames at these ports actually include the time as well, so a data frame contains values $y_1, \dots, y_n, t$.

The stability analysis is performed as soon as any data frame comes to input port `activator`. After that, the result is written to output port `result`. The output value is 1 if the system is stable, and 0 if unstable.

F. Boxes that have specific logics

This section describes some boxes that implement specific logics. Attempts to design certain simulations have led us to the invention of these boxes. We are not sure that the presented set of such logical boxes is complete in some sense, and that there is no better way to design them. Still the logic boxes are extremely useful in some simulations.

1) Join: The box has two input ports, `in_1` and `in_2` with formats $\{n_1\}$ and $\{n_2\}$, respectively, and one output port, `out`, with format $\{n_1 + n_2\}$. In short, the box glues together each two data frames coming to the input ports and writes the result to the output port.

Suppose that the input data frame at port `in_1` has elements $x_0^{in,1}, \dots x_{n_1-1}^{in,1}$, and the input data frame at port `in_2` has elements $x_0^{in,2}, \dots x_{n_2-1}^{in,2}$. Then the output data frame consists of all elements of data frame at port `in_1`, followed by all elements of data frame at port `in_2`: $x_0^{in,1}, \dots x_{n_1-1}^{in,1}, x_0^{in,2}, \dots x_{n_2-1}^{in,2}$.

The box has two internal boolean state variables, s_1 and s_2, indicating that an unprocessed data frame is pending at input ports `in_1` and `in_2`, respectively. The value of true means that an input data frame has been received but has not been processed so far. The value of false means that there were no data frames at all, or the last received input data frame has already been processed.

When an input data frame comes to port `in_1` or `in_2`, the value of state variable s_1 or s_2, respectively *must be* false (otherwise, simulation stops with the error message saying "Join box overflow"). Then, the state variable is set to true. After that, if the other state variable (s_2 or s_1, respectively) is false, the processing finishes — the box will be waiting for a data frame at the other input port. If both state variables s_1 and s_2 are true, the box resets them to false, generates one output frame, and writes it to port `out`.

The logics of the box ensures that k-th output data frame at port `out` is generated from k-th data frame at input port `in_1` and k-th data frame at input port `in_2`.

Notice that to satisfy the requirements of the `Join` box on the order of input data frames and ensure no overflow error, it is often used in combination with the `Replicator` box (see Section VII-F3 below).

Figure 14 (a) illustrates data processing by a `Join` box.

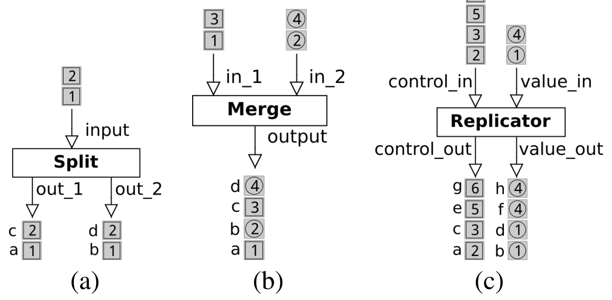


Figure 14. Input and output in (a) `Join`, (b) `Merge`, and (c) `Replicator` boxes. Numbers denote the order of input data frames and identify them; letters denote the order of output data frames.

2) Merge: The box has several input ports, `in_1`, `in_2`, etc. The number of input ports is a box parameter. There is one output port, `output`. All ports have the same format, which can be arbitrary. Once the box obtains a data frame at any of its input ports, it writes it to the output port immediately; see Figure 14 (b).

The `Merge` box is used to collect data frames from different ports.

3) Replicator: It is not obvious that there is a need for this box at all, but it is often really needed in simulations. The box has two input ports, `control_in` and `value_in`, and two output ports, `control_out` and `value_out`. It passes *control data frames* from `control_in` to `control_out` and *value data frames* from `value_in` to `value_out`.

When a value data frame comes to `value_in`, nothing happens. In contrast to that, when a control data frame comes to `control_in`, the box writes the incoming control data frame to `control_out`, and then it writes the previously received value data frame to port `value_out`, as shown in Figure 14 (c). If no value data frame has been received before control data frame, the processing is canceled.

As already mentioned, the `Replicator` box can be combined with the `Join` box to synchronize data frames; but it has many more different applications.

4) Split: The box has one input port, `input`, and several output ports, `out_1`, `out_2`, etc. The number of output ports is a box parameter. All ports have the same format, which can be arbitrary. Once the box receives a data frame at port `input`, it writes it to output ports `out_1`, `out_2`, etc. Importantly, the order of output port activation is guaranteed — first `out_1`, then `out_2`, and so on. Figure 15 (a) illustrates the data processing by the `Split` box.

Simulations in NumEquaRes allow to connect an output port of a box to any number of input ports of other boxes. However, this introduces uncertainty into simulation, because when such an output port is activated, the order of activation of connected input ports is undefined (it is actually the order of link creation, but it cannot currently be seen or easily modified). The `Split` box potentially allows to design simulations that have no multiple connections of output ports at all: each multiple connection can be replaced by single connection to `Split:input` and several single connections to `Split:out_1`, `Split:out_2`, etc.

Practically, the order of activation of input ports connected to the same output port is not always important. When it is, the `Split` box should be used.

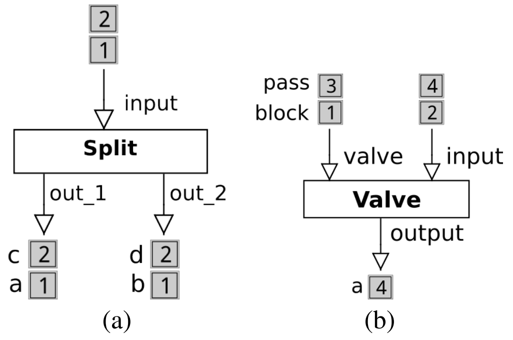


Figure 15. Input and output in (a) `Split` and (b) `Valve` boxes.

5) Valve: The box contains two input ports, `valve` and `input`, and one output port, `output`. The `valve` port accepts scalar *controlling values*; other two ports should have the same format, which can be arbitrary. In short, the box passes a data frame coming to its `input` port to the `output` port only if it has a nonzero value in the `valve` port; otherwise, the input data frame is not passed — see Figure 15 (b).

The logics of this box is similar to that of `Join` (see Section VII-F1) but slightly differs from it. Internally, the box holds two boolean state variables, s^i and s^v, indicating if there are unprocessed data frames at ports `input` and `valve`, respectively. Initially, they are both false.

When a controlling data frame comes to port `valve`, s^v is set to true (note: in contrast to the `Join` box, there is no requirement that s^v should be false at this moment). Current controlling value v is set to true if the controlling data frame element is nonzero, and to false otherwise. Then, if s^i is true, further processing is done: last data frame received at port `input` is written to port `output` if v is true and not written if v is false. Then s^v and s^i are both set to false.

When an input data frame comes to port `input`, s^i is set to true (note: in contrast to the `Join` box, there is no requirement that s^i should be false at this moment). Then, if s^v is also true, further processing is done exactly the same way as explained above: last data frame received at port `input` is written to port `output` if v is true and not written if v is false. Then s^v and s^i are both set to false.

G. Input boxes

NumEquaRes provides several box types dedicated to interactive input of data.

1) General behavior of input boxes: All user input is non-blocking, which means that input boxes never wait for user input. On the other hand, an input box can only check if there is an input event when one of its input ports is activated. Most of input boxes have the `activator` port specifically for this.

Another way to activate an input box is to use the special `Pause` box (see Section IV-F) — in that case, all input boxes are activated when data processing finishes.

Once an input box receives user input data, it takes an action that depends on box type. For example, it can write a data frame to the port `output`, or it can restart simulation.

Among input box types, three of them (`SimpleInput`, `RangeInput`, and `PointInput` — see below) allow user to interactively input *vector data*. They all have the same logics, and only differ in how user inputs the data.

All vector data input boxes remember the data user entered within the last input event (or, at the beginning of simulation, they know that no input events have taken place).

Vector data input boxes have input port `input`. The format of this port is a one-dimensional array of arbitrary size. There is also the `output` port with the same format. Besides, vector input data boxes have the `activator` port.

When a data frame comes to port `input`, a data frame is written to port `output`. The output data is the same as the input data if no user input has taken place on this box yet. If, however, there was user input, the box changes part of the input data frame before writing it to `output`: it replaces some elements of input data frame with values user entered last time. Which elements are replaced depends on box parameters.

When a vector data input box is activated (either by sending a data frame to port `activator` or due to the activity of the `Pause` box), it first checks if any data is available at port `input`. If no data has been received on that port, nothing happens. User input data, if any, will be waiting for further processing, till the box is activated next time.

If some data is available at port `input`, the box checks for user input. If there is no unprocessed user input, nothing happens. If user input has taken place, the box reads the user input data and replaces part of last data frame obtained from `input` with new user input data. The resulting data frame will be available at port `data`, but the exact behavior of the box now depends on two boolean parameters, `restartOnInput` and `activateBeforeRestart`.

- If `restartOnInput` is false, the simulation data processing loop is exited and entered again, starting from the input box. The input box then writes the prepared output data frame to port `output`, and simulation continues.
- If `restartOnInput` is true, then
  - if `activateBeforeRestart` is true, the prepared output data frame is sent to port `output`; otherwise, it is not sent.
  - Then simulation data processing loop is exited and entered again, starting from data sources, as it happens when simulation is started (see Section IV).

Notice that the combination `restartOnInput`=true and `activateBeforeRestart`=true implies that there will be no extensive data processing when the data frame is sent to port `output` before restarting (otherwise, there probably will be no restart at all). This combination can be used, for example, to specify ODE solver parameters: when solver receives them, it does nothing. More often both `restartOnInput` and `activateBeforeRestart` are false.

2) SimpleInput: Boxes of this type allow user to enter numeric values. Box parameters specify the display names for these values and the indices in the output data frame where these values are written to.

When a simulation having boxes of this type is running, user sees a set of named input fields. Entering a value into such a field causes user input event, which is processed as described above.

3) RangeInput: The box is similar to `SimpleInput`, but instead of entering numeric values user moves sliders. For each input value, it is necessary to specify value range and resolution in addition to the display name and index.

4) PointInput: The box allows user to enter coordinates x, y of points in plane by clicking on an image that corresponds to the `Bitmap` box (see Section VII-B) associated with `PointInput`. The coordinates x^{img}, y^{img} of pixel clicked on the image (notice that the point $x^{img} = y^{img} = 0$ is at the top-left corner of the image) are mapped to x, y using linear interpolation:

$$\begin{array}{rcll} x & = & x_{\min}+ & \frac{x^{img}}{N_x} \quad (x_{\max}-x_{\min}), \\ y & = & y_{\min}+ & \frac{N_y - y^{img}}{N_y} \quad (y_{\max}-y_{\min}), \end{array}$$

where N_x and N_y are image pixel width and height, respectively.

Parameters $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$ determine the rectangle that the entire image maps onto. They can be specified as box parameters or supplied through additional input port `range`.

Other parameters of the box are the name of image file and the indices of elements in output data frames where x and y are written to.

5) RectInput: The box allows user to enter two data ranges that determine translation and scaling of a plane. These ranges are specified by parameters $x_{\min}, x_{\max}, y_{\min}, y_{\max}$. When user input occurs, the box computes new ranges and writes them to port `output` as one data frame. Several boxes described above (`GridGenerator`, `Canvas`, `PointInput`) have port `range` compatible with `RectInput:output` and can be connected to it.

Similarly to `PointInput`, a `RectInput` box must be associated with a `Bitmap` box by providing the name of image file. The input comes to `RectInput` when user rotates the mouse wheel on the corresponding image (this causes scaling) and drags across that image (this causes panning).

The `RectInput` box is used when basic pan/zoom functionality is desired for a generated image, e.g., to explore fractals (see Figure 26).

6) SignalInput: This is the simplest input box. A button is displayed for each box of this type at simulation run time. Pressing the button causes an empty data frame to be written to port `output`.

The box can be used to trigger some actions, for example, to clear `Canvas` (see Section VII-C) by connecting `SignalInput:output` to `Canvas:clear`.

H. Common box connections

In this section we provide a number of examples showing typical connections between boxes. These examples aim to ease the understanding of examples presented in Section VIII.

Figure 16 (a) shows an example of connecting the `output` port of the `Param` box to an input port of another box. This can always be done when the input port format is known and is $\{N\}$, i.e., one-dimensional array or scalar. The `Param` box extracts port format from its connection and exhibits corresponding values as its own parameters. User enters parameter values, and at simulation startup they are sent to receiver(s) in just one data frame. These parameters remain constant during the simulation. Using `Param` to specify constant parameters is very common in simulations.

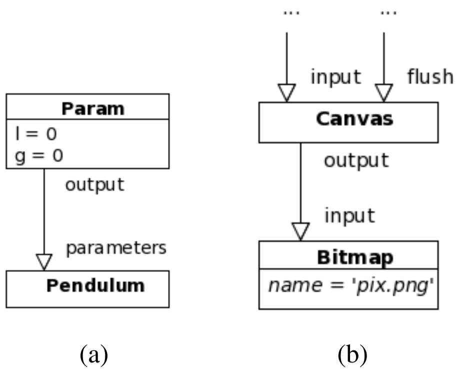


Figure 16.
(a) Using `Param` box to specify parameters.
(b) Attaching `Canvas` to `Bitmap`.

Figure 16 (b) shows the connection of `Canvas` box to `Bitmap` box. It is typical that the data for visualization is first accumulated in the canvas, and is written to image rarely (either when a data frame comes to `Canvas:flush` or automatically with user-specified time interval).

Figure 17 (a) explains how to make it possible to interactively modify a parameter during simulation. To do so, it is necessary to cut an existing link and place a `RangeInput` box (or other vector input box — see Section VII-G) in between, so that instead connection `a->b` we have two connections, `a->RangeInput:input` and `RangeInput:output->b`. Parameters of the input box determine which elements of data frames must be user-editable during simulation. In this example, the `RangeInput` box causes the slider bar shown in Figure 17 (b) to appear in running simulation, and the value of parameter a can be changed from 0 to 10 with step 0.01. Notice also that it is necessary to perform explicit activation through port `RangeInput:activator` frequently enough, because otherwise the box will not have any chance to process user input before the data processing finishes (see Section IV).

Figure 17 (c) shows the typical connection between the `Rk4` solver box (see Section VII-E4) and the `CxxOde` box (see Section VII-D10) providing the formulation of a system of ordinary differential equations. The `Rk4` box writes ODE

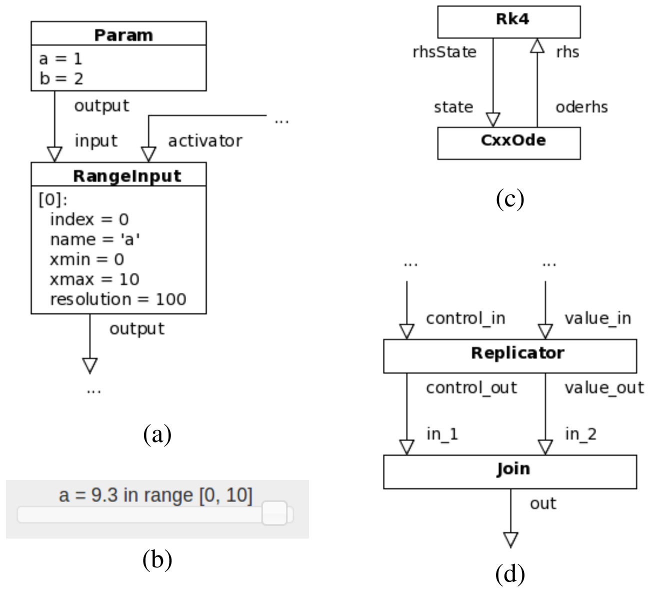


Figure 17.
(a) Using `RangeInput` to interactively modify parameters.
(b) Slider element for interactive input in running simulation.
(c) Coupling the `Rk4` solver and an ODE system.
(d) Using `Replicator` to feed `Join`.

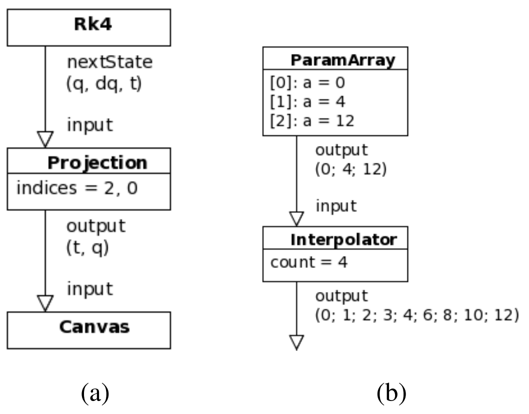


Figure 18.
(a) Obtaining 2D projection of points for plotting to `Canvas`.
(b) Example of usage for `Interpolator` box.

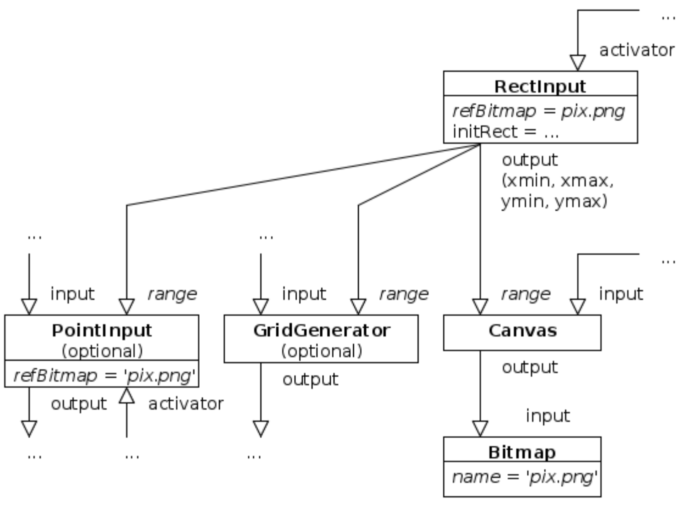


Figure 19. Implementation of panning, zooming, and point input.

system state to port `Rk4:rhsState` and reads ODE right hand side from port `Rk4:rhs`. The `CxxOde` box computes the right hand side, as soon as it receives state at port `CxxOde:state`, and writes it to port `CxxOde:oderhs`.

Figure 17 (d) illustrates one of many possible applications of the `Replicator` box (see Section VII-F3). It is used here to make sure that the `Join` box (see Section VII-F1) receives equal number of data frames in ports `in_1` and `in_2`. When a data frame comes to `Replicator:value_in`, nothing happens. When a data frame comes to `Replicator:control_in`, it is written to `Replicator:control_out` and hence to `Join:in_1`. After that, the last data frame received at `Replicator:value_in` is written to `Replicator:value_out` and hence to `Join:in_2`. As a result, overflow never happens in the `Join` box.

Figure 18 (a) shows how a point can be projected onto 2D canvas. This is done using the `Projection` box. In this example, the box generates output data frames with elements t, q from input frames with elements q, $\dot{q}$, t: first element is t because it is the element with index 2 in the input frame; second element is q because it has index 0 in the input frame. Indices 2 and 0 are parameters of the `Projection` box.

Figure 18 (b) shows a combination of `ParamArray` (see Section VII-A) and `Interpolator` (see Section VII-E1) boxes. The interpolator in this example splits each span between two consecutive input data frames into 4 pieces and writes interpolated data to port `output`. Annotations near `output` ports of the boxes contain scalar data that is generated in this example.

Figure 19 shows a typical way to organize panning and zooming of image generated in a simulation. The image corresponds to a `Bitmap` box and is identified by image file name. Interactive user input for panning and zooming actions is provided by the `RectInput` box (see Section VII-G5). The box needs to be associated with image by specifying image file name as box parameter. The image is considered to cover the rectangle $x_{\min} \leq x \leq x_{\max}$, $y_{\min} \leq y \leq y_{\max}$ in plane x, y. Initial rectangle is specified by `RectInput` box parameters. Whenever user scrolls mouse wheel or drags across image, the box modifies rectangle parameters $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$, and writes them to port `output`. This port is connected to the `range` port of the `Canvas` box (see Section VII-C) supplying image data. It can also be connected to the `range` port of some other boxes, in accordance with simulation logics. For example, if each pixel on an image corresponds to a point of a grid, it is connected to port `GridGenerator:range` of box that generates the grid. If there is a `PointInput` box for the same image, its `range` port should also be connected to `RectInput:output`. Notice that all input boxes must be activated frequently enough through port `activator` in order to be able to process user input when simulation is running. This is currently the responsibility of simulation designer.

VIII. Examples of Simulations

This section lists several examples of simulations. We will often use the notation `box:port`, where `box` is a name (not a type) of a box, and `port` is the name of its port.

A. *Single phase trajectory of a simple pendulum*

Figure 20 shows one of the simplest simulations — it plots a single phase trajectory for a simple pendulum. The ODE system is provided by the `ode` box (type `Pendulum`, see Section VII-D10). The box computes the right hand side $[\dot\varphi, \ddot\varphi]$ according to the pendulum equation

$$l\ddot\varphi + g\sin\varphi = 0,$$

where l is the pendulum length and g is the acceleration of gravity. The ODE right hand side depends on the state variables $[\varphi, \dot\varphi]$ and the vector of parameters $[l, g]$. They are supplied through input ports. Parameters are specified in the `odeParam` box. State variables come from the `solver` box (type `Rk4`, see Section VII-E4). The solver performs numerical integration of the initial value problem, starting from the user-specified initial state (the `initState` box). The solver is configured to perform a fixed number of time steps (the corresponding parameters come to the solver from the `solverParam` 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 of type `Projection`, see Section VII-D7) and then rasterized by the `canvas` box (type `Canvas`, see Section VII-C). Finally, the data comes to the `bitmap` box (type `Bitmap`, see Section VII-B) that generates the output image file. Notice that this simulation has three data sources, `odeParam`, `solverParam`, and `initState`, of type `Param` — see Section VII-A.

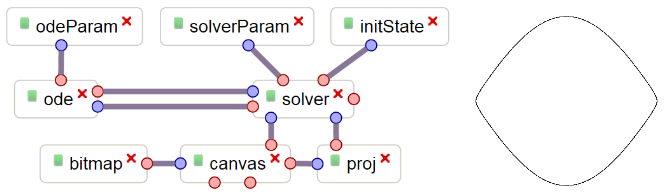


Figure 20. Single phase trajectory

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.

B. *Interactive phase portrait*

An important aspect of a simulation is its ability to *interact with the user*. This can be achieved using input boxes (see Section VII-G). Figure 21 shows an example of interactive simulation: it generates phase trajectories passing through points clicked by the user on the phase plane. The box `isInput` has type `PointInput` and is responsible for that kind of input. Another available kind of user input is panning and zooming of the phase plane; it is handled by the `pan-zoom` box of type `RectInput`. Notice that there is no need to activate

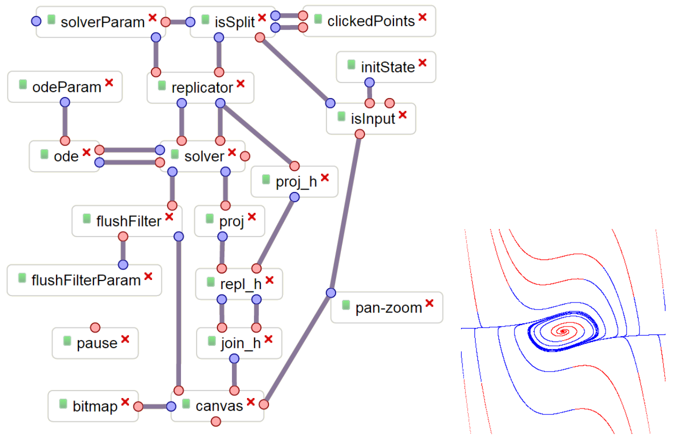


Figure 21. Interactive phase portrait

input boxes during data processing. It finishes very fast, and the input processing occurs after all data processing finishes, due to the presence of box `pause` of type `Pause`.

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. Many of the remaining boxes serve to achieve this behavior. The `solverParam` box has type `ParamArray`. It specifies two sets of solver parameters, one with positive value of time integration step h, and the other one with negative time step $-h$. Each data frame coming from `solverParam` causes an initial value problem to be solved, with a specific value of the time step. The data flow initiating the initial value problem solution is as follows. First, initial state travels the route `initState -> isInput -> isSplit -> replicator:value_in`. Then `replicator` returns control to `isSplit`, which then activates `solverParam`. That causes two sets of solver parameters to be generated in two data frames. When each of them reaches `replicator:control_in`, the `replicator` box first writes solver parameters to `solver:parameters`. At this point, the `solver` box (type `Rk4`) already knows which parameters to use, but it doesn't start the integration (it only does so when it receives an initial state). Therefore, the control is returned back to `replicator`. It then writes the initial state to `solver:initState`, which finally starts numerical integration.

When the solver produces a point of phase curve, $[x, \dot x, t]$, it is transformed into $[x, \dot x, \pm h]$ by boxes `proj` ($[x, \dot x, t] \to [x, \dot x]$), `proj_h` (solver parameters $\to \pm h$) of type `Projection`, `repl_h` (type `Replicator`), and `join_h` (type `Join`). Last two boxes glue data frames $[x, \dot x]$ and $\pm h$ together. Data frames $[x, \dot x, \pm h]$ come to the `canvas` box, so that points of time-positive part of phase curve are assigned value h, and time-negative parts are assigned value $-h$. The `bitmap` box has a color map that maps 0 to white, h to blue, and $-h$ to red, which finally gives us the desired look of the output image.

The remaining two boxes (`flushFilter` of type `CountedFilter` and `flushFilterParam` of type `Param`) are here in order to pass each second data frame from `solver:finish` to `canvas:flush`. As a result, the image user sees in the browser is updated only when both branches of phase curve are computed.

Importantly, there is no need to modify this scheme to replace the ODE system: it is sufficient to provide the system formulation as a parameter of box `ode` (type `CxxOde`) and specify fixed system parameters in box `odeParam` (type `Param`).

C. *Poincare map for double pendulum*

The classical double pendulum system is a model of two pendulums moving in plane, with motionless support of the first pendulum and the second pendulum attached at the end of the first one, as shown in Figure 22. The parameters of the system are two masses, two lengths, and the acceleration of gravity. The configuration is determined by two angles, φ (rotation of the upper part) and ϑ (rotation of the lower part).

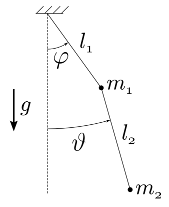
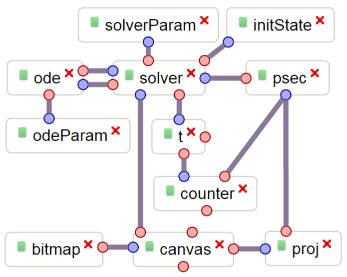


Figure 22. Double pendulum system

The equations of motion in the Lagrange form are as follows.

$$(m_1+m_2)l_1^2\ddot{\varphi}+m_2 l_1 l_2\left[\cos(\vartheta-\varphi)\ddot{\vartheta}-\sin(\vartheta-\varphi)\dot{\vartheta}^2\right]+ g(m_1+m_2)\sin\varphi=0,$$
$$m_2 l_2^2\ddot{\vartheta}+m_2 l_1 l_2\left[\cos(\vartheta-\varphi)\ddot{\varphi}-\sin(\vartheta-\varphi)\dot{\varphi}^2\right]+ g m_2\sin\vartheta=0.$$

This ODE system is non-integrable, and its phase trajectories can be quasi-periodic or chaotic, depending on the initial state. An easy way to reveal the type of behavior of a given trajectory is to look at its Poincaré map. This is done in the following simulation, shown in Figure 23.



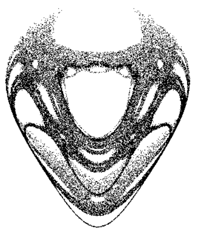


Figure 23. Double pendulum, Poincaré map (50000 points, 28.5 s)

Essentially, the scheme is very close to the one shown in Figure 20. But rather than to pass each next point of the phase curve from `solver:nextState` directly to `proj` and then to the `canvas`, we check for intersection with a hyperplane first. The `psec` box has type `CrossSection` (see Section VII-D3), hence `proj` receives points on the hyperplane; the rest of processing is same as for the simplest example in Section VIII-A.

Two boxes, `counter` of type `Counter` and `t` of type `ThresholdDetector`, are introduced in order to stop the integration as soon as 50000 points of the Poincaré map are obtained.

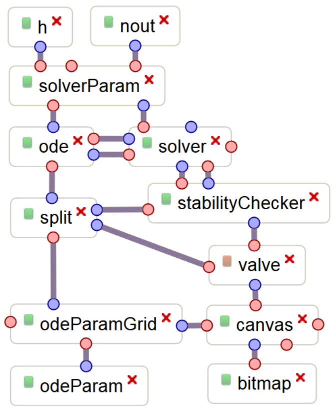
Importantly, there is no need to store phase trajectory or individual points of intersection of the trajectory with the plane during simulation. The entire processing cycle (test for intersection; projection; rasterization) is done as soon as a new point of the trajectory is obtained. After that, we 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.

D. *Ince–Strutt diagram*

Figure 24 shows a simple simulation that allows one to obtain a stability diagram for 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 [17]:

$$\ddot{q}+[\lambda-2\gamma\cos(2q)]\,q=0,$$

where λ and γ are parameters.



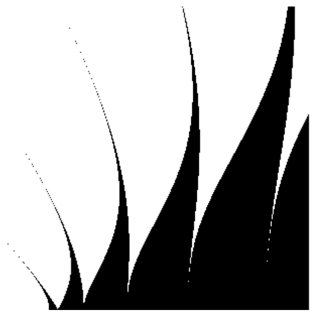


Figure 24. Ince-Strutt stability diagram (500 × 500 points, 6.3 s)

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 $\lambda_1 \le \lambda \le \lambda_2$, $\gamma_1 \le \gamma \le \gamma_2$ into pixels and analyze the stability in the bottom-left corner of each pixel (by computing eigenvalues of the monodromy matrix [5]), then assign pixel color to black or white depending on the result. In this simulation, important new boxes are `odeParamGrid` (type `GridGenerator`, see Section VII-E2) and `stabilityChecker` (type `LinOdeStabChecker`, see Section VII-E5). The former one provides a way to generate points $[\lambda, \gamma]$ on a multi-dimensional grid, and the latter one analyzes the stability of a linear ODE system with periodic coefficients.

The simulation works as follows. When a new point $[\lambda, \gamma]$ is generated by `odeParamGrid`, it is sent to the `split` box;

then `split` sends it to `ode:parameters`, `valve:input`, and finally to `stabilityChecker:activator`. The `stabilityChecker` box analyzes the stability of ODE system with given values of λ, γ, and sends the result to `valve:valve`. At this point, the `valve` box has received data frames at both of its input ports, so it decides whether to pass data frame from its port `input` to port `output`. The decision is determined by the result of stability analysis, since it comes to port `valve`. If the ODE system is stable, the data frame is passed, otherwise it is blocked. Therefore, the `canvas` box receives points $[\lambda, \gamma]$ for which the ODE system is stable, and the corresponding pixel in the `bitmap` box is drawn in black color. The `canvas` flushes its data to `bitmap` at the end of simulation due to the link `odeParamGrid:flush -> canvas:flush`, and also each second when simulation is running.

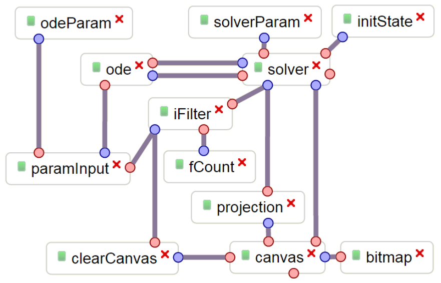
The box `solverParam` has type `Rk4ParamAdjust` not described in this paper; it is used here to compute the necessary number of numerical integration steps when period and time integration steps are known.

E. Strange attractor in forced Duffing equation

Figure 25 shows another application of Poincaré map, now in the visualization of the strange attractor arising in the forced Duffing equation [18]:

$$\ddot{x} + \delta\dot{x} + \alpha x + \beta x^3 = \gamma\cos(\omega t)$$

with parameters α, β, γ, δ, ω. User can change parameters interactively and see how the picture changes. This simulation is simpler than the one shown in Figure 23, because to obtain a new point on canvas, one just needs to apply time integration over known time period $T = 2\pi/\omega$ of system excitation. The solver is configured such that data frames $[x, \dot{x}, t]$ generated at port `solver:nextState` are in plane $t = kT$, with an integer k. Then the `projection` box throws t away, and `canvas` receives points $[x, \dot{x}]$.



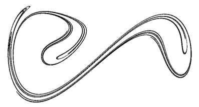
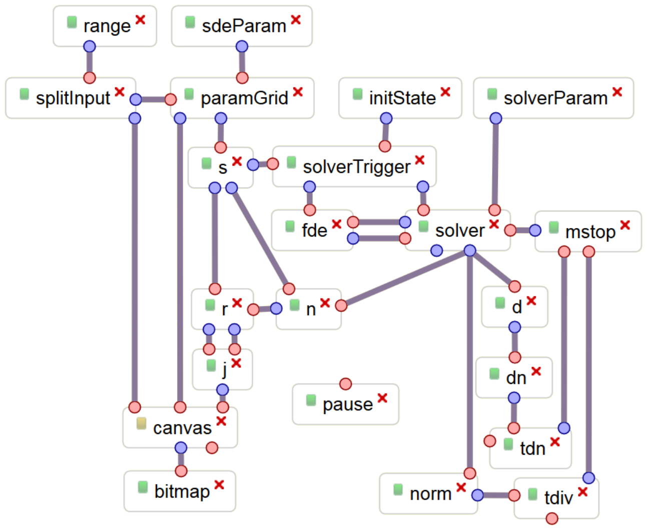


Figure 25. Strange attractor for forced Duffing equation (interactive simulation)

Boxes `paramInput` and `clearCanvas` have types `RangeInput` and `SignalInput`, respectively. The box `paramInput` allows user to modify parameters α, β, γ, δ by moving sliders. The box `clearCanvas` allows user to clear canvas by clicking a button. Notice that the user input is processed together with the data processing. Therefore, both input boxes have to be activated from time to time. This is done through box `iFilter` of type `CountedFilter`: as soon as a new point is generated at port `solver:nextState`, it comes to `iFilter:input`, and each 10-th point reaches `iFilter:output` and `paramInput:activate`, `clearCanvas:activate` connected to it. The counted filter box is used in order to activate the input boxes not too often, otherwise simulation performance could suffer due to the input processing.

F. The Mandelbrot set

Figure 26 shows an interactive simulation of the Mandelbrot set [19], which is defined as the set of complex numbers c for which the sequence $z_0, z_1, z_2, \ldots : z_0 = 0, z_{k+1} = z_k^2 + c$ is bounded.



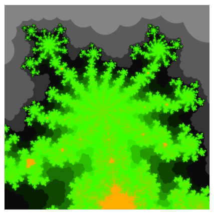


Figure 26. Colored Mandelbrot set (interactive simulation)

Since the Mandelbrot set is a fractal, it is important for the user to be able to pan and zoom the picture using the mouse. This is achieved by using the `range` box of type `RectInput` (see Section VII-G5) that feeds the ranges for real and imaginary parts of c to `canvas:range` and `paramGrid:range` through the `splitInput` box of type `Split`.

The `paramGrid` box (type `GridGenerator`) generates c on a grid covering the specified rectangle on complex plane. For each c from that grid, a part of the sequence z_k is evaluated, and its boundness is checked. Iterations stop as soon as sequence convergence or divergence is detected. The number of iterations, n, done for each c is stored in box `n` of type `Counter`; it determines the color of pixel corresponding to c in the final image.

The simulation works as follows. The initial state $z_0 = 0$ comes from `initState` to `solverTrigger:value_in` (the box `solverTrigger` is of type `Replicator`). Then `paramGrid` comes into play, activated by dummy value $c = 0$ from `sdeParam`. Each value of c

generated by `paramGrid` comes through box `s` (type `Splitter`) to `n:reset` (to reset iteration counter n), and then to `solverTrigger:control_in`, which causes the latter to write c to `fde:parameters` and z_0 to `solver:initState`.

The boxes `fde` and `solver` are of types `CxxFde` and `FdeIterator`, respectively. When z_0 comes to `solver:initState`, the `solver` box starts generating elements of sequence z_k. It generates at most 500 elements, but is stopped if sequence convergence or divergence is detected. The divergence analysis is performed by boxes boxes `norm` (type `Scalarizer`, computes $|z_k|$) and `tdiv` (type `ThresholdDetector`, evaluates $|z_k| > 3$). The convergence analysis is performed by boxes `d` (type `Differentiate`, computes $z_k - z_{k-1}$), `dn` (type `Scalarizer`, computes $|z_k - z_{k-1}|$), and `tdn` (type `ThresholdDetector`, evaluates $|z_k - z_{k-1}| < 10^{-5}$). When the expression in either `tdiv` or `tdn` evaluates to true, the solver is stopped (the box `mstop` is of type `Merge`, its port `output` is connected to `solver:stop`).

When the iterations of z_k stop, the control returns to the box `s`, and it writes c to `r:control_in` by activating its third output port. At this point, `r` (box of type `Replicator`) forwards c to its port `r:control_out` and the number of iterations done, n (obtained earlier from box `n`) to port `r:value_out`. Then the box `j` of type `Join` glues the coordinates of c together with n, and the data frame $[\operatorname{Re} c, \operatorname{Im} c, n]$ comes to `canvas`.

Importantly, we did not have to develop any new box types in order to describe the logics of convergence analysis for sequences of complex numbers generated by the system, but used standard general-purpose boxes instead.

IX. 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 23, 24, 25, and try to solve them 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 I. COMPARISON OF NUMEQUARES WITH OTHER TOOLS

Name	*Free*	*Web*	*Can solve*	*Fast*
Mathematica	no	yes	23, 24, 25; needs coding	n/a
Maple	no	no	23, 24, 25; needs coding	n/a
MATLAB	no	no	23, 24, 25; needs even more coding	no
SciLab	yes	no		no
OpenModelica	yes	no	none	could be
XPP	yes	no	23, 25	yes
InsightMaker	yes	yes	none	n/a

In Table I, 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 23, 25 using standard time-stepping algorithms since version 9 (released 24 years later than version 1) due to the `WhenEvent` functionality. Problem 24 can also be solved. All algorithms have to be coded. Notice that Wolfram Alpha [20] (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 23 and others with Hamiltonian equations; problems 24, 25 can be solved by coding their algorithms.

With MATLAB or SciLab, one can code algorithms for problems 24, 25 using standard time-stepping algorithms. For problem 23, 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 [21] 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 23, 24, 25, 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 [6] provides all functionality necessary to solve problems 23, 25. 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 24.

Among other simulation tools we would like to mention InsightMaker [22]. 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 23, 24, 25 cannot be solved with InsightMaker.

X. TECHNICAL CHALLENGES

The design of NumEquaRes governs technical challenges specific to Web applications for simulations. They are related to performance and security, and are discussed in this section.

A. Server CPU resources

Currently, all simulations run on the server side. Some of them can be computationally intensive and consume considerable amount of CPU time. For example, there are simulations that consume 100% of single CPU core time for as long as user wishes. This is a problem if the number of users grows. Of course, we do not expect millions of users simultaneously running their simulations, but still there is a scalability problem.

The problem can be addressed in a number of ways. Firstly, the server can be an SMP computer, so it will be able to run as many simulations as the number of CPU cores, without any loss of performance. Secondly, it is technically possible to have a cluster of such computers and map its nodes to user sessions. Obviously, this approach requires the growth of server hardware to provide sufficient server performance.

A different approach is to move running simulations to the client side. In this case, the server loading problem will disappear. But how is it possible to offer user's browser to run something? Actually, today the only choice seems to be JavaScript. We will have to compile simulations into it, or to the asm.js subset of JavaScript. This approach is quite possible for some simulations, but is problematic for other ones that can make use of some large libraries like LAPACK.

B. User code security

NumEquaRes web server accepts C++ code as part of simulation description provided by user. This is the direct consequence of our wish to provide good computational performance of simulations. Such pieces of code typically describe how to compute the right hand side of an ODE system, or how to compute another transformation of input data frames into the output data frames. The server compiles that code into dynamic library to be loaded and executed by core application that performs the simulation. Potentially, we have serious risk of direct execution of malicious code.

Currently, this problem is solved as follows. Once user code is compiled into a library (shared object on UNIX or dynamically linked library on Windows), it is checked for the absence of any external or weak symbols that are not found in a predefined white list (the list contains symbol names for mathematical functions and a few more). Due to this, user code is not able to make any system calls. For example, it cannot open file `/etc/passwd` and send it to the user because it cannot open files at all. If the security check on the compiled library fails, no attempt to load it is done, and the user gets notified about the reason of check failure.

On the other hand, malicious code could potentially exploit such things as buffer overrun and inline assembly. It is an open problem now how to ensure nothing harmful will happen to the server due to that. However, the ban on any non-white-listed calls seems to be strong enough. Probably, one more level of protection could be achieved with a utility like chroot.

A better approach to provide security is to disallow any C++ code provided by user. But this would imply giving the user a good alternative to C++ allowing to describe his/her algorithms equally efficiently. For example, there could be a compiler of formulas into C++ code. Nothing like this is implemented at the moment, but can be done in the future. In this case, the user code security problem will vanish.

XI. 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. Simulations run on server; besides, they may contain C++ code provided by user. This creates two challenges — potential problems of server performance and security. The security problem has been addressed in our work; the performance problem is not currently taken into account.

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. Another option is to eliminate internal uncertainty in simulation specification: only allow one connection per output port and require the initial order of source box activation to be explicitly specified by the user.

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. Currently, the most serious bottleneck for user is having to supply equations in the form of C++ code. This problem can be addressed by implementing interoperability between NumEquaRes and other tools. For example, many simulation tools are able to formulate problem equation using the Functional Mock-up Interface (FMI) standard format [23]. It is well possible to develop a new box type with interface similar to `CxxOde` but taking its input from an FMI model exported from another tool. It is important to notice, however, that more advanced numerical time-stepping solvers (e.g., CVODE from the Sundials library [24]) have to be used to simulate these models.

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.

Last but not least, an important usability improvement can be achieved with a feature that visualizes simulation data flows; its role is similar to debugger's.

References

[1] S. Orlov and N. Shabrov, "Numequares — web application for numerical analysis of equations," in SOFTENG 2015: The First International Conference on Advances and Trends in Software Engineering, S. F. Felipe and H. Jameleddine, Eds. IARIA XPS Press, 2015, pp. 41–47.

[2] "Numequares — an online system for numerical analysis of equations," URL: http://equares.ctmech.ru/ [accessed: 2015-11-25].

[3] E. J. Routh, The Advanced Part of a Treatise on the Dynamics of a System of Rigid Bodies, 6th ed. Macmillan, London, 1905, reprinted by Dover Publications, New York, 1955.

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

[5] G. Teschl, Ordinary Differential Equations and Dynamical Systems, ser. Graduate studies in mathematics. American Mathematical Soc., URL: http://books.google.ru/books?id=FSObYfuWceMC [accessed: 2015-11-25].

[6] B. Ermentrout, Simulating, Analyzing, and Animating Dynamical Systems: A Guide to XPPAUT for Researchers and Students, ser. Software, Environments and Tools. Society for Industrial and Applied Mathematics, 2002, URL: http://books.google.ru/books?id=Qg8ubxrA060C [accessed: 2015-11-25].

[7] "Using server-sent events," URL: https://developer.mozilla.org/en-US/docs/Server-sent_events [accessed: 2015-11-25].

[8] "Lamp (software bundle)," URL: http://en.wikipedia.org/wiki/LAMP_(software_bundle) [accessed: 2015-11-25].

[9] "D3.js — data-driven documents," URL: http://d3js.org/ [accessed: 2015-11-25].

[10] "A full-featured markdown parser and compiler, written in javascript," URL: https://github.com/chjj/marked [accessed: 2015-11-25].

[11] "Mathjax — beautiful math in all browsers," URL: http://www.mathjax.org/ [accessed: 2015-11-25].

[12] VTK user's guide. Kitware, Inc., 2010, 11th ed.

[13] E. Brown, Web Development with Node and Express. Sebastopol, CA: O'Reilly Media, 2014.

[14] "Ajax (programming)," URL: https://en.wikipedia.org/wiki/Ajax_(programming) [accessed: 2015-11-25].

[15] "Acml — amd core math library," URL: http://developer.amd.com/tools-and-sdks/archive/amd-core-math-library-acml/ [accessed: 2015-11-25].

[16] J. C. Butcher, Numerical Methods for Ordinary Differential Equations. New York: John Wiley & Sons, 2008.

[17] M. Abramowitz and I. Stegun, Mathieu Functions, 10th ed. Dover Publications, 1972, chapter 20, pp. 721–750, in Abramowitz, M. and Stegun, I., Handbook of Mathematical Functions, URL: http://www.nr.com/aands [accessed: 2015-11-25].

[18] C. M. Bender and S. A. Orszag, Advanced Mathematical Methods for Scientists and Engineers I: Asymptotic Methods and Perturbation Theory. Springer, 1999, pp. 545–551.

[19] J. W. Milnor, Dynamics in One Complex Variable, 3rd ed., ser. Annals of Mathematics Studies. Princeton University Press, 2006, vol. 160.

[20] "Wolframalpha — computational knowledge engine," URL: http://www.wolframalpha.com/ [accessed: 2015-11-25].

[21] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Computer Society Pr, 2003.

[22] S. Fortmann-Roe, "Insight maker: A general-purpose tool for web-based modeling & simulation," Simulation Modelling Practice and Theory, vol. 47, no. 0, 2014, pp. 28 – 45, URL: http://www.sciencedirect.com/science/article/pii/S1569190X14000513 [accessed: 2015-11-25].

[23] "Functional mock-up interface," URL: https://www.fmi-standard.org/ [accessed: 2015-11-25].

[24] "Sundials — suite of nonlinear and differential/algebraic equation solvers," URL: https://computation.llnl.gov/casc/sundials/main.html[accessed: 2015-11-25].