Virtual Laboratory on Nonlinear Control

Sergey B. Tkachev*, Denis Aldoshin*, Alexey E. Golubev*

* Department of Mathematical Modeling, Bauman Moscow State Technical University (BMSTU), 2-ya Baumanskaya Str., 5, Moscow, 105005, Russia (e-mail: mathmod@bmstu.ru)

Abstract: This paper presents the virtual laboratory software developed at the Department of Mathematical Modeling (BMSTU). The laboratory is designed as an educational tool for the nonlinear control courses. It allows to see 3D demonstrations of control processes for various nonlinear systems. The laboratory software package is developed using the Python programming language. It has open code and architecture. The laboratory includes such models as inverted pendulum on a car, ball and beam system, reaction wheel pendulum, Furuta pendulum and other models of most popular nonlinear systems.

Keywords: Virtual laboratory, Nonlinear control, Control education

1. INTRODUCTION

Design of various control algorithms for nonlinear systems is one of the modern trends in automatic control. These algorithms are based on the differential-geometrical approach, the integrator backstepping tools, passivity-based methods and so on (see Isidori (1995), Krasnoshchekono and Krishchenko (2005), Sira-Ramirez and Agrawal (2004), Krstić et al. (1995), Khalil (2002)). Various models and algorithms are collected by Fantoni and Lozano (2002).

Computer-oriented technologies are now an essential part of the educational process. These technologies allow students to get some practical experience in control of various dynamical systems. A realistic 3D visualization of control processes of some mechanic system is a good tool for understanding control technologies.

Suppose there is a group of students having a task in control theory to be done: each one has to synthesize a control algorithm (e.g. as a feedback) which stabilizes the given dynamic system with inputs or tracks it along the prescribed reference trajectory. Instead of just finding the desired controller, one has to spend time testing and representing the result. That means

(1) programming a modelling environment for the given dynamical system;
(2) implementing a control algorithm to form a closed-loop system;
(3) obtaining graphs of the state vector components;
(4) obtaining graphs of the input variables;
(5) checking given restrictions for the state vector and input values using the graphs acquired or defining these restrictions right in the code;
(6) transferring the graphs to the final report.

In fact, only the second and the fifth activities on this list are undoubtedly useful, but they cannot be done separately from the others. Moreover, the first five steps are to be performed not only by every student in the group, but also by the course professor in case one wants some additional verification of the results represented in the reports. There exist different approaches to automate this routine. One of the ways to ease the process is to use a high-level language of technical computing, e.g. Mathematica or MATLAB. They both support quick visualization. Documents prepared in Mathematica can even be formatted to yield a report which is relatively easy to understand and check.

Another approach is to create a virtual laboratory for these specific needs. That is what the following sections are about.

2. VIRTUAL LABORATORY DESCRIPTION

Virtual laboratories are software packages destined to ease the described procedure by automatically performing steps of numerical modelling, visualization and sometimes even results verification. It is essential to distinguish between virtual laboratories (performing numerical modelling of the processes) and remote laboratories (allowing users to set up and control a real-life experiment remotely, by means of the Internet). A virtual laboratory can still use the Internet for transferring data.

2.1 General requirements

Here are the desired features of a virtual laboratory in application to control theory:

(1) models of the dynamical systems should be easily created and added;
(2) user should be required to give only parameters of the system, initial conditions for simulation, and control algorithm;

* The work was supported by the RFBR (Grants 11-01-00733 and 12-07-329)
(3) laboratory should stay close to WYSIWYG concept, allowing to get the result as soon as user has provided all the necessary information, without recompiling any of the code;

(4) as an extension of (3), the laboratory should give the opportunity not only to plot graphs of various parameters but also to see things in 3D (provided this is appropriate);

(5) a Web-based interface is needed in order to send the synthesized control algorithm to a server, so that the course professor can load it into an instance of the laboratory and verify the results.

2.2 Tools selection

There are special tools needed to develop our virtual laboratory so as to fulfill all the requirements listed above. The toolkit includes a programming language and libraries to be used to dynamically create the interface and visualize the closed-loop system state evolution. In addition, the server part of the laboratory needs some database management system.

Programming language. The language chosen is Python. This is due to the following reasons:

- it is cross-platform (runs on Windows, Linux / Unix, Mac OS X, has been ported to the Java and .NET virtual machines); it is a standard component in most of Linux distributions and can be run in Mac OS X terminal;
- it is distributed under Python Software Foundation License, which means it is free to use both in open source and commercial products;
- it provides rapid development (the code is usually 2–4 times shorter then code written in C++ or Java);
- its core syntax is quite minimalistic, so it would not take long to become familiar enough with Python to describe a control algorithm.

The main feature is that Python is a dynamic (scripting) language and thus supports precompiling of source code right in the runtime. That gives a way to let a user enter arbitrary control algorithm and provide the corresponding results immediately.

3D graphical API. Though Direct3D may be a better choice for Windows platform, OpenGL is used to preserve cross-platform portability. OpenGL seems a good solution for creating simple scenes without additional effects. Still it remains a low-level, procedural API rather then a descriptive one (e.g. it has no native support for scene graphs so that is a thing to be programmed separately). Access to OpenGL from Python is gained by means of PyOpenGL library.

Database management system. MySQL is used in conjunction with PHP language. Operability has been tested using Apache server with mod_PHP extension though other choices are acceptable.

There is no standard interface builder for Python. Also, as different models require the input of different parameters, some parts of the interface are not predefined and have to be created in the runtime. Therefore wxPython library was chosen to serve the needs of interface creation.

Reasons of the choices we have made are:

- the software is cross-platform;
- it is all free to use and mostly open source;
- it is easily distributable;

The last feature means that user does not have to install any additional software. No records are made in the registry or in the system files; this application can be run right after copying it to the computer (that, of course, does not apply to the server part of our laboratory).

Alternate choices. We can propose another way to implement such a laboratory using any of .NET languages (e.g. C#) instead of Python for the main part of a program. Python can be left just for scripting capability (i.e. for a user to enter a controller algorithm). Python and .NET language can be interconnected through dynamic language runtime (DLR) provided by Microsoft. Instead of CPython language implementation, IronPython should be used. For the application to be cross-platform, it is sufficient to use DLR and IronPython libraries recompiled for Mono framework. Users still do not have to install any additional software except for .NET framework or Mono framework.

2.3 Main concepts

Our application uses the model-view-controller architectural pattern, see Fig. 1. This concept is directly applicable to this sort of problem, as it will be shown further.

![Model-View-Controller Pattern](image)

Fig. 1. The use of the model-view-controller pattern

Model block. The model is a container for all the parameters of a dynamical system. It implements an interface to restore default parameters, set any initial state of the system, and change it as prescribed by the controller block. It also lets the view block to read the current state of the system, so the user interface can be updated.

View block. This one's task is to perform rendering of the graphical user interface (GUI) according to data stored in the model. Though it generates parts of the interface, the view is not meant to be a front end to interact with a user as it does not accept any information from the latter. We use a single view block though more of them are allowed if needed.

Controller block. Here is the core element of the triad. It performs two functions: reading information from the GUI and processing the current state of the dynamical system to predict its change after a certain period of time.
The model must support different types of parameters including integer, floating point, string, and boolean ones. Some parameters may be edited (e.g., desired physical characteristics of the dynamical system, its initial state); others have to be read-only (current time, current full energy of the system etc.) The model also defines, which parameters should be displayed and regularly updated while the simulation is in progress. All the undisplayable constants such as standard gravity acceleration $g_0$ are also defined inside this block.

Once the parameters are updated by the controller, a certain function is called forcing the view block to change properties of on-screen objects. The 3D view is, of course, redrawn not on every change of the parameters but only to yield the desired frame rate. So are the 2D graphs of parameters.

The controller block usually acts like this:

- precompiles the user-defined control algorithm;
- starts the loop with time variable increasing for one time step each iteration;
- in every iteration of the loop
  - checks whether all the additional conditions hold, otherwise breaks the loop;
  - integrates differential equations describing the system for one time step;
  - modifies the model to make it match the new state (this, as it was mentioned above, will cause the view to be refreshed);
- halts the loop by the user’s request.

2.4 On realization

In order to implement the model-view-controller architecture a special namespace is defined. It contains four modules: model, view, controller, and parameters.

Each of the first three modules defines a base class for the respective block of the model-view-controller pattern.

A base class for the model implements procedures which interact with configuration file and set parameter’s value by its name given. This class depends heavily on the parameters module where all types of parameters are described with corresponding editors as subclasses. These editors are used by the main module when building the user interface.

The module view defines a base class for 3D visualization. It holds references to the model object and an active OpenGL engine. All the functionality must be implemented separately depending on the system being modeled (see further description).

It must be mentioned that other functions of the view block are distributed between the main module and the module which plots parameters’ graphs. That is because they do not depend on a dynamical system being simulated and thus there is no need in defining the corresponding base classes.

2.5 Enveloping OpenGL

OpenGL does not support objects as such, so it is necessary to implement quite a lot of new classes manually. As a result, OpenGL is enveloped so as not to call its low-level procedures directly. The classes mentioned stand for camera, OpenGL canvas, and graphical primitives. Amongst primitives there are defined the box, the sphere and the cylinder (or the conic frustum). That is enough for simple scenes. Still, more complex primitives can be programmed additionally. One more construct defined is the triangulated surface which is described by an enumerated set of vertices and a list of triplets. Each triplet contains numbers of vertices to form one triangle.

A key feature to be included in 3D engine is the support of scene graphs. That means, any object may have a “parent” such that every transform applied to the “parent” object is immediately applied to all of its “children”. That makes creating complex scenes way easier and quicker. The code of scene construction becomes more readable; it can be modified locally without rewriting big blocks. Generally, there are two ways to build a scene: to create all of its components and then unite them using a scene graph, or to add each component to the graph as soon as it is created. A special class representing a node of such a graph is implemented in the source code. Its only method (not taking into account the class constructor) is recursively rendering the graph tree, starting from the current node and proceeding along each branch as far as possible before backtracking.

The transforms available are translation, rotation, and non-uniform scaling. A pivot point is specified for each object. It becomes the fixed point for rotation and scaling transforms of this object and all of its “children”.

3. ADDING NEW PLUG-INS

Ability to add new plug-ins representing various dynamical systems is a must for an application like ours. There are at least two methods allowing to provide such functionality.

The first one relies on dynamic-link libraries. Following this way, programmer should create a library including all the necessary classes for model, view, and controller. Each class has to implement certain interfaces in order to be successfully handled by the main program. This method could be used while implementing the virtual laboratory by means of the .NET language + DLR + IronPython conjunction.

The second method is somewhat more natural to Python programmers. A Python script is reinterpreted every time it is run. Thus all it takes to create a new plug-in is to put its source files into the certain directory and modify the configuration file so the main script can detect the new plug-in. It is also possible to omit the last step because Python has powerful tools for code introspection, so the main script can search for plug-ins automatically.

3.1 Needed source files

Every new plug-in is defined in its own namespace. In Python namespaces correspond to directories of the file system so at first one should create a new subdirectory where all the files will be stored. (For a directory to behave like a namespace it is necessary to place __init__.py file there. It may be void though.) There are generally four
source code files to be put into this directory. Three of them represent blocks of the model-view-controller triad; one more is needed to provide information about plug-in (e.g. name, author, description etc.)

**Model source file.** Here the descendant class of the base `model` class is implemented. It includes at least two methods: its constructor `__init__` and a method restoring the default state of parameters. Constructor assigns values to the variables which represent a table of parameters and a list of pairs of the parameters to plot by default. The table of parameters is itself a list. Each entry on the list corresponds to one parameter; the entry is a tuple containing two elements: parameter's name and an instance of one of the parameters classes. Such an instance can be created in-place by calling its constructor (e.g. `FloatParameter(hint = "initial position", default = 1, minValue= -1)`). Such a structure is easily processed by Python as it has native support of iterable types including lists and tuples.

**View source file.** In this module one must define a descendant of the base `view` class. It includes methods which initialize the scene and modify it as the parameters change. The first method creates objects of the scene and forms the scene graph. The second method is called every time the view needs to be redrawn. It sets new positions for moving objects and updates displayable values such as modelling time.

**Controller source file.** As in previous cases, a class defined here descends from the base `controller` class and specifies actions to be done in the very beginning of the simulation, and on every time step of it. There is only one other method to be added. It calculates new values for changing parameters of the model and sends a signal that the changes were made, so the scene can be refreshed.

3.2 Handling the control algorithm.

As most of other dynamic languages do, Python has an instruction `exec` (it is a function in Python 3.x) which executes an expression given in string form as if it was written in the source code. All one has to do is pass the multiline text parameter which contains the user-defined control algorithm to this instruction. The problem is, this action takes place on every time step so our application is likely to consume excessively much computational resources. Luckily, Python supports precompilation not only for source files but for code arising in run-time as well. Thus the control algorithm an be compiled once using `compile` function and then the result is passed to `exec` on every time step.

*A simple debugger.* If a user makes a mistake while programming the controlling algorithm, an error will halt the execution of the whole program. This, of course, is undesirable situation. It can be handled by try-except-finally block and by turning on the built-in debugger.

There are two types of errors: compile-time (usually syntactic) and runtime ones. If there is a syntactic error in the control algorithm, precompilation will be unsuccessful. In such case the compilation statement is wrapped in try-except block to cancel the simulation and display the error message. Function `sys.exc_info()` is used to obtain information about the location of syntactic error.

Runtime errors are also caught by try-except block, but only to mark them as handled. The main role here is played by the debugger. Before using `exec` it is turned on by calling `sys.settrace(tracer)` procedure (`tracer` is a callback function which actually handles an error). If the latter occurs, `tracer` procedure will stop the simulation and display an error message. Otherwise, the debugger is turned off until next time step, so it will not handle any errors outside the user-defined algorithm.

**Using templates.** It turns out that the files of different plug-ins are very much alike in their structure. In order to make adding of new plug-ins quicker a template is created which contains common code and remarks on creating the plug-in. Due to this it became easier to understand and modify source files created by others. Still, further automation is possible.

4. INTERFACE

Our application provides simple single-window interface, the overall view shown in Fig. 2. It is divided into three vertical areas and also includes an actions bar. The latter provides buttons for establishing a connection with the server, adding and deleting of simulation sessions, adding and deleting of graphs, and controlling the process of simulation.

4.1 Plug-in selector

The leftmost, see Fig. 3, area allows to select a plug-in (i.e. a dynamical system) to work with.

![Fig. 2. The overall view of the user interface](image)

![Fig. 3. Dynamical system selection panel](image)
At first user should select a dynamical system to work with by picking the corresponding bold item. Then one might want to create a new simulation or modify and run an existing one. In the first case, Add button on the actions bar should be used. Otherwise, user should just choose the existing simulation from the unfolding sublist. In order to delete a simulation one should select it and press the Delete button on the actions bar. All the simulation sessions are automatically saved to the configuration file on exit.

4.2 Visualization area

The central part of the window is oriented for visualization. In the upper part of it the 3D view is placed. The lower part is occupied by graphs. Since there may be not enough room for all the graphs, a scrollbar is added. Naturally, to create or delete a graph, one should click the corresponding button on the actions bar. Any pair of displayable parameters may be chosen to create a graph (none of them is necessarily current time).

4.3 Monitoring the state of the system

The rightmost, see Fig. 4, area of the window is used to display and modify the parameters of the current simulation session, including the desired control algorithm. Uneditable parameters usually change through time while editable ones are set by the user before the simulation begins.

Fig. 4. The pane of parameters (some of them omitted)

4.4 Existing plug-ins

At present, the virtual laboratory includes five developed plug-ins which correspond to the following dynamical systems:

1. reaction wheel pendulum;
2. Furuta pendulum (inverted rotation pendulum);
3. cart pendulum (cart and pole system);
4. one-link flexible robot;
5. beam and ball system.


5. EXAMPLE

The beam and ball system is chosen to describe how the application works. The system is stabilized by the backstepping procedure (see Olfati-Saber and Megretski (1998)).

Fig. 5. The generalized coordinates of the system

The Euler-Lagrange equations for this system (with damping excluded) have the following form:

\[ m\ddot{q}_1 - m_1q_2^2 + mg\sin q_2 = 0, \]
\[ (I + m_1q_1^2)\ddot{q}_2 + 2m_1q_1\dot{q}_1\dot{q}_2 + mgq_1\cos q_2 = \tau. \]  

Here \( q_1 \) and \( q_2 \) are the generalized coordinates shown in Fig. 5, \( m \) denotes the mass of the ball, \( g \approx 9.81 \text{ m/s}^2 \) is the gravity constant, \( J \) stands for the inertia of the beam, and \( \tau \) is the torque applied to the beam. There are also constraints defining the domain of operability. Firstly, the generalized coordinates should be bounded:

\[ -\frac{L}{2} < q_1(t) < \frac{L}{2}, \]
\[ -\frac{\pi}{2} < q_2(t) < \frac{\pi}{2}. \]  

Secondly, this system of ODEs becomes completely inadequate in case the contact between the ball and the beam is lost. Thus another condition must be checked for all \( t \geq 0 \):

\[ N(t) = m(q_1(t)\dot{q}_2(t) + 2\dot{q}_1(t)\dot{q}_2(t) + g\cos q_2(t)) > 0. \]

The goal is to make this system’s equilibrium point corresponding to \( q_1 = 0 \) asymptotically stable. Local stabilization will be enough because of the requirements mentioned above.

Following the backstepping procedure, we conclude that the closed-loop system becomes locally asymptotically stable if the control applied is

\[ \tau = \left(\frac{ML^2}{12} + m\dot{q}_1\right)v + mgq_1\cos q_2 + 2m_1\dot{q}_1\dot{q}_2, \]

where

\[ v = c_4(c_4 + c_5)\dot{q}_2 - c_1c_4c_5\tanh(c_2(c_3q_1 + \dot{q}_1)) + c_1c_2(c_3(-c_4 - c_5)q_1 - (c_3 + c_4 + c_5)q_1 + \dot{q}_1)\dot{q}_2^2 + g\dot{q}_2\cos q_2 +
\]

\[ (c_3 + c_4 + c_5)\sin q_2 + 2c_2(-c_3\dot{q}_1 - q_1\dot{q}_2^2 + g\sin q_2)^2 \times 
\]

\[ \tanh(c_2(c_3q_1 + \dot{q}_1)))/(2c_1c_2q_1\dot{q}_2 - \cosh^2(c_2(c_3q_1 + \dot{q}_1))) \]

with \( c_i \) being constant values:

\[ c_1 = 0.1, \quad c_2 = 5/4, \quad c_3 = 1, \quad c_4 = 20, \quad c_5 = 20. \]

In the above expression \( M \) denotes the mass of the beam.

To implement this control in our laboratory we just put the corresponding Python code into the multiline editor.
Fig. 6. The control algorithm written in Python provided by the plug-in, see Fig. 6. The modelling is run immediately after all the initial values are set. The 3D model can be seen in the Fig. 2. It contains the system itself and three coordinate axes for better orientation in space (because users are allowed to rotate the scene as well as zoom in an out).

In this session of modeling two graphs are chosen for plotting. The first is a graph of the counterpressure applied to the ball, see Fig. 7. It shows that the condition (3) holds. The second graph is a projection of the trajectory in the state space on the \((q_1, \dot{q}_1)\) plane, see Fig. 8. It should be noted that conditions (2), (3) are checked automatically during the simulation so that it stops once any of them is not met; then a message describing the problem is shown in Fig. 9.

CONCLUSIONS

The developed virtual laboratory software allows 3D visualization of control processes for various nonlinear systems. It includes nonlinear models of such popular mechanical systems as inverted pendulum on a cart, ball and beam system, reaction wheel pendulum, and Furuta pendulum. There is also the possibility to add new 3D virtual objects. The laboratory contains a set of built-in control laws and allows to add user-defined control algorithms. It could be helpful for the nonlinear control education and is used within the undergraduate and postgraduate training at BMSTU.

REFERENCES


