A state-space model extraction tool using Modelica-based first principles model[1]

Kaushik Mallick[a], Vinamzi P. Samuel[a], Sarbani B. Belur[b], Madhu N. Belur[a]

[a]*Department of Electrical Engineering, Indian Institute of Technology Bombay, India*
[b]*National Centre for Aerospace Innovation and Research, Indian Institute of Technology Bombay, India*

Abstract

Circuit simulation packages typically provide very accurate and numerically stable methods to simulate complex circuits, but they do not give an explicit state space description of the system. Advantages of an explicit, closed-form analytically obtained state-space model are undebatable. This paper develops and describes an implementation procedure of how a state-space description can be extracted. We describe the building blocks of this implementation and the mathematical theory underlying these techniques. The proposed procedure can be scaled to large dynamical systems using the code developed and made available on github. Such an implementation is currently available in neither proprietary packages nor open-source packages.

We use the Modelica-based modeling and simulation environment OpenModelica as a schematic to interconnect various devices. The text-file created by OpenModelica after the interconnection is scripted using Python to create a polynomial matrix containing the differential equations. This polynomial matrix is then manipulated within Scilab to yield state space matrices A, B, C and D.

We demonstrate this procedure on an RLC circuit.

Keywords: OpenModelica, Scilab, Python, Modelica, FOSS tools, model-extraction

1. Introduction

The demand from automation and technology sectors continues to expand the scope of control systems with an ever growing need for more complex systems to be modelled analytically. The state space approach is a unified method used for modelling, analysing and designing a wide range of dynamical systems. The use of state space approach also facilitates the modelling of nonlinear systems with nonlinearities like saturation and dead zone. This approach also allows handling MIMO (Multi Input Multi Output) systems in a compact way (see [1]). State space techniques are useful for designing controllers meeting advanced control objectives like $\mathcal{H}_2$ and LQR control. They are also very useful for performing eigenvalue/participation analysis.

Though there are plenty of software packages that perform simulation and analysis, most packages do not enable model extraction based on first principles model. For example, Xcos, OrCAD, Ngspice,

[1]Corresponding author: Madhu N. Belur (Email: `belur@ee.iitb.ac.in`). This work was supported in part by SERB, DST, India.

PSpice, OSCAD, Matlab/Simulink, Sequel are software packages that can be used for analysis and simulation of various circuits. Some of them like OrCAD and PSpice are considered to be suitable for simulation and design of PCBs, while Ngspice, Xcos and Simulink are suitable for graphical dynamical system modelling. Model order reduction, controller design, eigenvalue analysis is possible in advanced numerical computation packages like Scilab and/or Matlab only after a first principles model is available.

We elaborate further in Section 2 about how a first principles model is essential for various purposes in analysis and synthesis of dynamical systems and controller design. In this paper we describe a newly developed tool to deduce the state space matrices directly from the model diagram. In order to deduce the state space description, we use OpenModelica 1.7.0 as a front-end to draw/create the circuit. After this circuit is built and compiled, we use the differential equations of the circuit obtained from Modelica and process these equations using the developed Python scripts. The Python scripts yield a polynomial matrix which is further manipulated in Scilab. This finally results in the desired state matrices A, B, C and D: see Figure 3 for a schematic figure. This paper describes this procedure.

There are many benefits of having an explicit state space description of a large system. One can easily interconnect many subsystems (usually from a library of ready-made and user-defined list of systems) in a complex manner and obtain a large system using the OpenModelica package. Once the large system is built, the developed scripts give a state-space description (A, B, C, D) which is useful for eigenvalue analysis, stability checks, controllability/observability checks, controller design for various design objectives, for example. See [2] for an exposition of various uses of the state-space description in control theory and see [3] for an exposition of time-series analysis in econometrics. A detailed study of large-scale dynamical systems described in state-space form can be found in [4]. The method proposed and developed in this paper is scalable to large-sized systems. Further, the method used in this paper is advantageous because the states also have a physical meaning, like capacitor voltages and inductor currents. This property is obviously an advantage of using a first-principles model: unlike an artificial state-space 'fit' of input/output data.

We note briefly the role that the Modelica programming language plays in this work. Modelica makes it possible to obtain a first-principles model of a large and complex interconnection of many subsystems. This is the ability of Modelica to deal with the interconnection without an explicit input-output partition. It is noteworthy here that tools like Xcos are dependent on an explicit input/output partition. For example, if one tries to connect an input port of a block to the *input* port of another block, one gets the error message: *"Explicit data output port must be connected to explicit data input port"*. A similar difficulty also arises with Simulink. This handicap forces one to manually classify all the subsystem variables into inputs and outputs before a large system is fully modeled using its subsystem models, moreover with a consistent rule that each variable, if not an external input, can be an output of exactly one subsystem. These drawbacks are eliminated by adoption of the Modelica programming language.

The paper is organized as follows. The next section briefly describes the significance of a first principles model to design a controller. Section 3 contains mathematical preliminaries for the development of the tool, followed by the software tools used to develop it (in Section 4). Section 5 describes the compilation method of the connection model followed by the details of various modules of the Python scripts and the Scilab codes implemented to obtain the state space matrices. Section 6 demonstrates the developed tool on an example of an RLC circuit (with two capacitors, two inductors and four resistors) is taken to give a demonstration of the developed tool. We conclude the paper in Section 7 summarizing the key contributions and suggesting some extensions.

The notation we follow in this paper is standard. The set $\mathbb{R}$ stands for the set of real numbers and $\mathbb{R}[\xi]$ stands for polynomials in the indeterminate ξ. Matrices with q rows and g columns, with entries from $\mathbb{R}[\xi]$ is denoted by $\mathbb{R}^{q\times g}[\xi]$.

2. Benefits of a first principles model

In this section we mention briefly some advantages of a first principles model of a system. We consider the so-called standard control problem in Subsection 2.1, while Subsection 2.2 elaborates on a complex interconnected system comprising of nonlinear devices and a controller.

2.1. Standard control problem

Figure 1 contains variables of a to-be-controlled plant classified as shown. This formulation makes it standard as far as the interface with controller-design packages is concerned. For example both Scilab and Matlab have LQG, $\mathcal{H}_2$ and $\mathcal{H}_\infty$ optimal/suboptimal controller design routines that take the plant in this form. Obviously, having a *first principles model* is helpful when a complex plant is to be controlled with respect to these advanced control specifications: a *simulation* package can at best be of help to a trial and error approach to controller design.

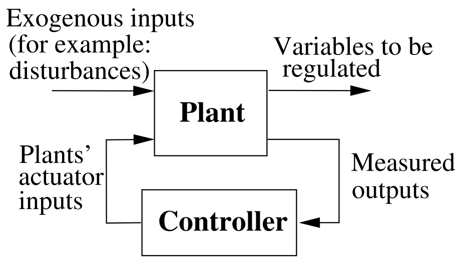
Exogenous inputs (for example: disturbances)
Variables to be regulated
Plant
Plants' actuator inputs
Measured outputs
Controller

Figure 1: The standard control problem

2.2. Interconnection of controller and linear/nonlinear plant

Consider a complex interconnected system in which some of the devices are nonlinear, some variables are accessible to a to-be-designed controller and the rest of the devices are linear. Many such systems allow extracting the nonlinear devices in a feedback form, while the to-be-designed controller also acts in feedback; see Figure 2. In such a formulation, the objective is to design a controller for the linear plant that renders the closed loop system having required properties. More precisely, the controller is designed to make the closed loop system *dissipative* with respect to a suitable notion of power that depends on the nonlinearity aspects of the nonlinear devices. It has been well-investigated how sector nonlinearities and slope restricted monotone nonlinearities

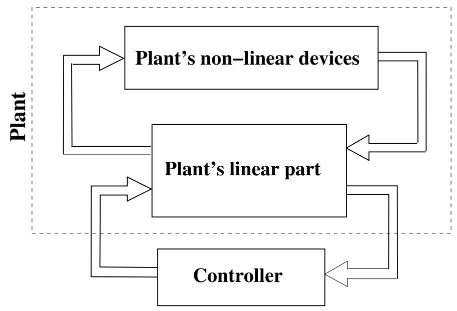

Figure 2: Controller and nonlinear plant

are all easily accommodated as dissipativity with respect to a suitable notion of power (see [5, 6, 7]). Controller design for such a framework obviously is largely benefited by a first principles model.

3. Theoretical principles underlying the developed tool

In this section we review some behavioral theory which is essential for the development of the routines in Scilab. A detailed exposition can be found in [8]. A key advantage of using behavioral theory that is relevant for this paper is that the system variables are not required to be classified into inputs and outputs while modelling, analysis and controller design of dynamical systems. This advantage is also elaborated in [8].

Consider the mathematical model of a dynamical system described by linear constant coefficient ordinary differential equations $P(\frac{\mathrm{d}}{\mathrm{d}t})v = 0$, where $P(\xi) \in \mathbb{R}^{q\times k}[\xi]$ is a polynomial matrix and $v = (v_1, v_2, \ldots, v_k)$ is vector-valued variable. The matrix P can be split into two blocks as $P = \left[\begin{array}{c|c} R & M \end{array}\right]$ such that $R \in \mathbb{R}^{q\times g}[\xi]$ corresponds to g real-valued manifest variable $w = (w_1, w_2, \ldots, w_g)$ and $M \in \mathbb{R}^{q\times m}[\xi]$ corresponds to m real-valued latent variable $\ell = (\ell_1, \ell_2, \ldots, \ell_m)$ such that $v = (w, \ell)$. Then, the *latent-variable representation* of the system is:

$$R\left(\tfrac{\mathrm{d}}{\mathrm{d}t}\right) w + M\left(\tfrac{\mathrm{d}}{\mathrm{d}t}\right) \ell = 0. \tag{1}$$

Suppose only the w variables are of interest and we are interested in *elimination* of the ℓ-variables. For eliminating ℓ, one uses the fact that there exists a *unimodular*[2] *matrix* $U(\xi) \in \mathbb{R}^{q\times q}[\xi]$ such that

$$U(\xi)M(\xi) = \begin{bmatrix} M_1(\xi) \\ 0 \end{bmatrix}, \quad U(\xi)R(\xi) = \begin{bmatrix} R_1(\xi) \\ R_2(\xi) \end{bmatrix} \tag{2}$$

with $M_1(\xi)$ having full row rank. With this partition of UR, the required differential equation in just the variable w is $R_2(\frac{\mathrm{d}}{\mathrm{d}t})w = 0$. This is known as a *kernel representation* of the system.

The polynomial $R_2(\xi)$ can be further reduced to a full row rank polynomial matrix $R_3(\xi)$ to get a *minimal kernel representation*

$$R_3\left(\tfrac{\mathrm{d}}{\mathrm{d}t}\right) w = 0. \tag{3}$$

Assume the manifest variable w is composed of n number of state variables x, m number of input variables u, and p number of output variables y such that $w = (x, u, y)$. Hence the first n columns of $R_3(\xi)$ corresponds to the states, followed by p columns for the inputs and q columns for the outputs such that $R_3(\xi) = \left[\begin{array}{ccc} R_3^x & R_3^u & R_3^y \end{array}\right]$, where $R_3^x \in \mathbb{R}^{q\times n}[\xi]$, $R_3^u \in \mathbb{R}^{q\times m}$ and $R_3^y \in \mathbb{R}^{q\times p}$. Premultiplying R_3 by a constant nonsingular matrix such that the R_3^y gets the form $\begin{bmatrix} 0 \\ I_p \end{bmatrix}$, with I_p being the identity matrix of size $p \times p$, the minimal kernel representation gets the form:

$$\begin{bmatrix} sI_n - A & -B & 0 \\ -C & -D & I_p \end{bmatrix} \begin{bmatrix} x \\ u \\ y \end{bmatrix} = 0 \tag{4}$$

[2]A square polynomial matrix U is said to be unimodular if its determinant is a nonzero constant.

where I_n is the $n \times n$ identity matrix.

Here the matrices $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{p\times n}$ and $D \in \mathbb{R}^{p\times m}$ are respectively the state transition matrix, input matrix, output matrix and the feed-through matrix. The state space representation of the system is given by

$$\dot{x} = Ax + Bu, \quad y = Cx + Du. \tag{5}$$

It is well-known that a regular state-space description as above with input u and output y exists if and only if the transfer matrix from variable u to variable y exists and is proper. In the case of an RLC circuit with p ports and no sources within, properness of the impedance transfer matrix from input i, the vector of currents at the ports, to output v, the vector of corresponding voltages at the ports, follows if no set of capacitors together with one or more port form a loop, and further, no set of inductors together with one or more ports form a cut-set. We assume this throughout this paper. The proposed method has been demonstrated to work on this situation: codes for this case have been made available at [9].

4. Softwares and tools

The tool developed and described in this paper requires the following Free and Open Source Software and corresponding version numbers: OpenModelica 1.7.0, Scilab 5.3.3 and Python 2.7.3 programming language. These are briefly described below.

4.1. OpenModelica

"OpenModelica is an open-source Modelica-based modeling, simulation and analysis tool intended for industrial and academic usage": (see [10]). OpenModelica is supported by the non-profit organization *Open Source Modelica Consortium (OSMC)*. The tool has an easy-to-use graphical user interface based environment to create a model, edit connections and simulate the model. Various subsystems are integrated in OpenModelica (version 1.7.0): for example, an interactive session handler, a Modelica compiler *omc*, a graphical model editor/browser *OMEdit*, an interactive shell *OMShell*, an optimization module *OMOptim*, a 2-D plotting module *OMPlot*, a Modelica development environment *OMDev*, an execution and run-time module, Eclipse plugin (MDT) editor/browser, a DrModelica notebook model editor *OMNotebook* and a Modelica debugger; see [11].

4.2. Python

Python is a cross-platform general-purpose high-level programming language, often used as a scripting language; see [12]. The language supports multiple programming paradigms including object-oriented, imperative and functional programming styles. It features very high level dynamic data types, automatic memory management and also has an extensive standard library. Python implementation is freely usable and distributable under an open source license administered by the *Python Software Foundation*; see [13].

4.3. Scilab

Scilab is a free and open source cross-platform software for numerical computation [14] and is released under the CeCILL license (see [15]). It uses matrix based computation for signal processing, statistical analysis, image enhancement, control system design and analysis, modelling and simulation of dynamical systems and numerical optimization: see [14] and [15].

5. Extraction procedure implementation

In this section we elaborate the development of the model extraction tool. Figure 3 is a schematic of the execution flow. The following subsections elaborate the individual functions in Python and Scilab. The overall procedure may be summarized as follows. OpenModelica provides a graphical tool to interconnect various subsystems and also builds an integrated system equations. The full set of equations (together with more text generated by the *omc* compiler) is scripted in Python to generate a polynomial matrix. The polynomial matrix is manipulated in Scilab using techniques from behavioral theory (see [8]) to finally obtain a state-space description. Analysis and controller design can be easily performed in Scilab using the obtained state-space description.

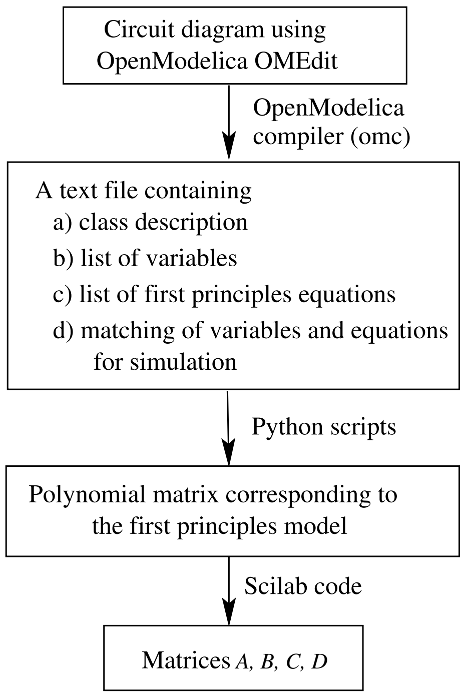
Circuit diagram using OpenModelica OMEdit
OpenModelica compiler (omc)
A text file containing
a) class description
b) list of variables
c) list of first principles equations
d) matching of variables and equations for simulation
Python scripts
Polynomial matrix corresponding to the first principles model
Scilab code
Matrices *A, B, C, D*

Figure 3: Execution flow of the tool

5.1. Extraction of full system equations using OpenModelica

We use the OpenModelica 1.7.0 connection editor OMEdit 1.7.0 to browse and create the models of the dynamical systems (saved in the file `input.mo`), followed by compiling the models using the command `omc`. This command compiles the model by linking with the `Modelica` libraries. The flags `+s +d=bltdump` performs the BLT transformation and generates the simulation codes of the model along with the circuit equations. We save this output to the file `out.txt`.

```
omc +s +d=bltdump input.mo Modelica > out.txt
```

The sections of our interest in the file out.txt are the initial *class* description, *Variables*, *AliasVariables*, *Equations* and *Transpose Incidence Matrix*.

5.2. Construction of the polynomial matrix using Python scripts

The Python scripts use out.txt, the output of OpenModelica compilation, to generate a polynomial matrix. The various procedures to get the polynomial matrix are described below.

1. Procedure `extract_var`: gets the list of variables
Input: `out.txt` (the output of compilation by *omc*)
Output: `var_num`:= number of variables, `var_list`:= list of variables, `var_to_remove`:= variables to remove, `states_list`:= list of indices of state variables

1: From the *bltdump* section, the subsection 'Variables' gives the number of variables.
2: The next few rows contain the list of variables. Only the names of the variables are taken. In case the variable is a *dummy variable* and a *dummy* of any another variable in the list, the index of the dummy variable is stored in the list of variables to remove.
3: If a variable is a state, its index is listed in a list of state variables.

2. Procedure `extract_var_inout`: gets the input and output variables
Input: `out.txt`, `var_list`
Output: `in_var`:= input variable, `out_var`:= output variable

1: From the initial definition of various parameters of the devices, the input variable is determined.
2: The output variable is chosen by the user. The default is all the variables: this means the matrices C and D will have many rows corresponding to the variables.

3. Procedure `extract_eqn`: gives the list of equations
Input: `out.txt`, `var_num`, `var_list`, `var_out`
Output: `eqn_list`:= list of equations, `var_list`:= list of updated variables (if any), `newvar_index`:= a list of indices of variables (if any), `eqn_to_remove`:= a list of indices of equations which are to be removed from original list of equations (default: remove none)

1: From the *bltdump* section, the subsection 'Equations' gives the list of equations.
2: If an equation contains the *dummy derivative* variables, these variables are replaced with derivatives of actual variables.
3: The equation for source variable (equating the source value) and the equations containing the *Loss Power* terms are not required to find the state space representation. Hence they are put in a list of equations to be removed. Their removal speeds up the computation.
4: If the chosen output variable is a combination of different variables, a new output equation is created, and a new output variable is inserted in the original list of variables.

4. Procedure `eqnvar_index`: gives the list of indices of equations for different variables
Input: `file`, `newvar_index`, `var_to_remove`, `eqn_to_remove`
Output: `eqn_index`:= a list of indices of equations for different variables appearing in them

1: From the *bltdump* section, the subsection 'Transpose Incidence Matrix' gives the list of equations. Each list contains the indices of equations where a particular variable is present.

5. Procedure `reorder_var_eqn`: gives the list of variables reordered such that first few variables are state variables, followed by the input and output variables and the other variables

Input: `eqn_list`, `eqn_to_remove`, `var_list`, `var_to_remove`, `state_list`, `var_in`, `var_out`, `eqn_index`
Output: `var_list_final`:= list of variables after removing unnecessary variables,
`eqn_list_final`:= list of equations after removing unnecessary equations,
`eqn_index_final`:= final list of indices of equations for different variables

1: All the variables are reordered such that first few variables are state variables, then the input and output variables, followed by other variables. All unnecessary variables are removed from the list.
2: All the equations are reordered according to the order of the variables and unnecessary equations are removed.
3: An updated list for indices of equations for different variables are created.

6. Procedure `poly_matrix`: creates the polynomial matrix
Input: `eqn_list_final`, `var_list_final`, `eqn_index_final`
Output: `polymat`:= polynomial matrix, `const`:= constant terms

1: A polynomial matrix of size *(# of equations)* $\times$ *(# of variables)* is created, where the rows are indexed by the equations and the columns are indexed by the variables. Referring to the indices of equations for each variable, each row of the polynomial matrix is filled with the corresponding coefficients of the variables. The coefficients are either '1' or '-1' or a $constant$ indicating circuit parameters and all others are '0'. All the derivative terms (containing 'der') are replaced with 's'.

7. Procedure `output`: creates a file containing the polynomial matrix
Input: `polymat`, `var_list_final`, `const`, `num_state`:= number of states, `num_invar`:= number of input variables, `num_outvar`:= number of output variables
Output: `output_file`:= a file containing polynomial matrix and a file containing the list of variables

1: A file is created containing the list of variables. All other output data viz. constants initialised to value 1, number of states, input and output variables and the polynomial matrix are stored in a file readable by Scilab.

5.3. Generation of A, B, C and D matrices using Scilab

Using the file generated by Python scripts, we generate the A, B, C and D matrices. We initialise the constant terms to the default value of one. Note that the first few columns of the polynomial matrix correspond to the state variables, then the input variables and finally the output variables. The algorithm is briefly described below. The output of the Python program is a Scilab readable file (.sce) which contains the polynomial matrix named $P(s)$ and two more variables *num_state* which gives us the information of the number of state variables and *num_source* which tells us the number of port variables. The algorithm to extract the state matrices is as follows.

Step 1: Extract the polynomial matrix, *num_state* and *num_source* into Scilab environment and also define the output variable if only specific outputs are required.

Step 2: The function `ReOrderStateEqns` is called by the main program to rearrange the polynomial matrix whose first few columns correspond to the state variable in such a way that the derivative variables 's' will appear as the diagonal elements of the principal minor.

Step 3: The function `findRindex1` is called by the main program to remove all rows with single entry.

Step 4: The function `MinKer` is called by the main program to remove all redundant rows that do not add to the rank of the matrix $P(s) \in \mathbb{R}^{m\times n}$ where $m < n$. The fact that m is at most $n-1$ is ensured by removing the row corresponding to the source definition while scripting in Python. This happens naturally during the elimination because it is a redundant row where the right hand side terms of the equation are not a part of the variable list, hence does not add to the rank of the matrix and is removed.

Step 5: The reduced polynomial matrix $P(s)$ is split into two blocks $R(s)$ and $M(s)$

$$\begin{aligned} P(s) &= C_k(s) && \text{for } k \in \{1,2,\dots,n\} \\ R(s) &= C_i(s) && \text{for } i \in S_1 \qquad\qquad \text{such that } \{1,2,\dots,n\} = S_1 \sqcup S_2. \\ M &= C_j && \text{for } j \in S_2 \end{aligned}$$

The polynomial matrix $R(s)$ contains all the columns corresponding to the state variables and port variables $P(s) \in \mathbb{R}^{m\times n}[s]$ and $R(s) \in \mathbb{R}^{m\times w}[s]$. The number of components in the variable w is $num\_state{+}num\_source{+}num\_output$. The rest of the columns of $P(s)$ make up $M(s)$. It is important to note that all columns that contain *'s'*, which represents the derivative variable, are absorbed into $R(s)$ while the rest of the columns constitute the *real* matrix $M \in \mathbb{R}^{m\times(n-w)}$. Variables corresponding to ***R*** are called observable variables and those corresponding to ***M*** are called latent variables.

Step 6: Find a left annihilator $V(s)$ of $M(s)$ where $M(s) \in \mathbb{R}^{m\times(n-w)}[s]$. Since $m > (n-w)$, there exists a nonzero left annihilator. We modify the left annihilator by using the function `LeftBlockI0`: this function ensures that only the top n rows are of degree one (or more) in the variable s.

Using the left annihilator, eliminate all the latent variables from ***P*** to obtain the matrix R_1

$$V(s)P(s) = \left(\begin{array}{c|c} R_1(s) & 0 \end{array} \right) \text{ with } R_1 \in \mathbb{R}^{p\times q}[s]$$

where *p = number of states+number of outputs*, and
q = number of states+number of inputs+number of outputs.

Step 7: The polynomial matrix $R_1(s)$ is brought to the standard form by performing simple row operations to ensure that the last few columns corresponding to the output variable form $\begin{bmatrix} 0 \\ I \end{bmatrix}$.

When this structure has been obtained, we extract the state matrices from the coefficients of matrix $R_1(s)$, whose standard form is

$$\begin{bmatrix} sI - A & -B & 0 \\ -C & -D & I \end{bmatrix} \begin{bmatrix} x \\ u \\ y \end{bmatrix} = 0.$$

The variables u, y and x are respectively the input, output and state variables. Further, A, B, C and D are the required state-space matrices and *I* is the identity matrix.

6. Implementation of the tool on an RLC circuit

The circuit given in Figure 4 is used as an illustration to implement the developed tool. The output file of OpenModelica compilation is not reproduced below. The outputs of the Python script and the Scilab code are shown below in the following subsections.

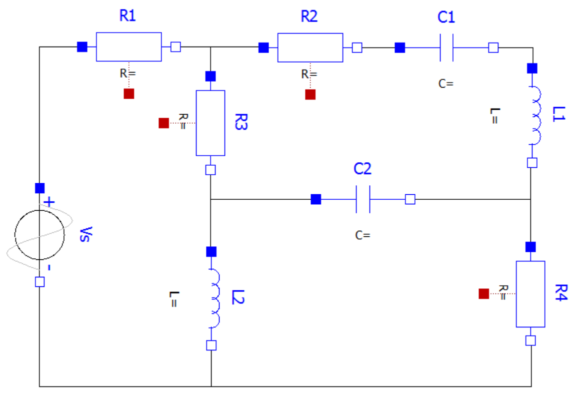

Figure 4: RLC circuit schematic from OpenModelica

6.1. *Output of the Python script*

The Python script creates two files - one containing the list of variables, and the other is a Scilab readable file containing the polynomial matrix and initialization of variables. We have chosen R4.n.i, the current through the resister R4 as the output variable.

6.1.1. *Content of the variables file generated by Python script*

1.L2_i, 2.C1_v, 3.C2_v, 4.L1_i, 5.Vs_signalSource_y, 6.R4_n_i, 7.Vs_n_v, 8.Vs_p_v, 9.L2_v, 10.R2_n_v, 11.R2_v, 12.R3_n_i, 13.R3_n_v, 14.R3_p_v, 15.R3_v, 16.C2_p_i, 17.R4_p_v, 18.R4_v, 19.L1_p_v, 20.L1_v, 21.R1_n_i, 22.R1_v

6.1.2. *Content of the Scilab file generated by Python script*

```
L2_L = 1; C1_C = 1; C2_C = 1; L1_L = 1;
R2_R = 1; R4_R = 1; R3_R = 1; R1_R = 1;
num_state = 4; num_invar = 1;
num_outvar = 1; s = poly(0,‘s’);
polymat = ...
[ L2_L*s,0,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0;
-1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,0;
1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0;
0,-C1_C*s,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,1,0,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,1,0,0,0;
0,0,-C2_C*s,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0;
0,0,1,0,0,0,0,0,0,0,0,0,-1,0,0,0,1,0,0,0,0,0;
0,0,0,L1_L*s,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,0,0;
0,0,0,1,0,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,1,0;
0,0,0,-1,0,-1,0,0,0,0,0,0,0,0,0,-1,0,0,0,0,0,0;
0,0,0,-R2_R,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,1,0,1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,R4_R,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0;
0,0,0,0,0,0,1,0,1,0,0,0,-1,0,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,-1,1,0,0,0,0;
0,0,0,0,0,0,0,-1,0,0,0,0,0,1,0,0,0,0,0,0,0,1;
0,0,0,0,0,0,0,0,0,1,1,0,0,-1,0,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,R3_R,0,0,1,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,1,-1,1,0,0,0,0,0,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,-1,1,0,0;
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,R1_R,1 ];
```

Note that the system parameters have been set to value 1. These can be changed easily within Scilab to suitable values. Alternatively (and less usefully) these parameters can be specified in OpenModelica itself.

6.2. Output of the Scilab code (generated by Scilab)

When the Scilab code is executed, one obtains the following state-space matrices:

$$A=\begin{bmatrix}-0.667 & 0 & 0.667 & 0.333\\ 0 & 0 & 0 & 1\\ -0.667 & 0 & -0.333 & -0.667\\ 0.333 & -1 & 0.667 & -1.667\end{bmatrix},\ B=\begin{bmatrix}0.333\\ 0\\ 0.333\\ 0.333\end{bmatrix},\ C=\begin{bmatrix}-0.667 & 0 & -0.333 & 0.333\end{bmatrix}$$

and $D = 0.333$.

7. Conclusion

We described the implementation of a procedure to obtain a state space description from a first principles model. For large complex systems this is a tedious task when done manually and circuit simulator packages give just a simulation for different circuit parameters. Thus, having a closed form analytical expression for the state space systems is beneficial for various purposes: for example, eigenvalue analysis, controllability/observability studies and controller design for various basic and advanced control objectives. This paper uses the OpenModelica tool to build a system from a library of simpler subsystems. The interconnection is made using OpenModelica package and the file that gets ‘dumped’ using the *omc-compiler* is scripted in Python to obtain a Scilab readable polynomial matrix. Further manipulation is performed in Scilab to obtain a state-space description of the system.

In this way the full capability of OpenModelica and the Modelica programming language is utilized. It is noteworthy that a key feature of the Modelica programming language is that it deal with just *interconnection* of subsystems and does not require an input/output classification of variables. This strength of Modelica gets combined with the same feature of behavioral theory of modeling and analysis of dynamical systems. This paper exploits this combined strength to develop this tool. While we demonstrated this concept on an RLC circuit using the code made freely available at [9]: the method requires just a small modification before handling interconnections of other model-based systems to automatically generate the state space model.

This paper dealt with numerical values specified for the various system parameters: like the resistances and capacitances. An easy extension is to allow symbolic manipulation so that the system matrices A, B, C and D are obtained with the system parameters directly. This can be pursued using Scilab 5.4.0 and/or the Python library *SymPy*.

Acknowledgement: We thank Jens Frenkel and Srikant Patnaik for their help during the process of this tool development.

References

[1] N. S. Nise, Control Systems Engineering, John Wiley & Sons, 2004.

[2] D. Hinrichsen, A. J. Pritchard, Mathematical Systems Theory, Modelling, State Space Analysis, Stability and Robustness, Springer, 2005.

[3] J. Durbin, S. Koopman, Time Series Analysis by State Space Methods, Oxford University Press, 2001.

[4] D. D. Šiljak, Large-Scale Dynamical Systems: Stability and Structure, North Holland, 1978.

[5] A. Megretski, A. Rantzer, System Analysis via Integral Quadratic Constraints, IEEE Transactions on Automatic Control 42 (6) (1997) 819–830.

[6] J. C. Willems, H. L. Trentelman, Synthesis of dissipative systems using quadratic differential forms: Part I, IEEE Transactions on Automatic Control 47 (1) (2002) 53–69.

[7] H. K. Khalil, Nonlinear Systems, Prentice Hall, 1996.

[8] J. W. Polderman, J. C. Willems, Introduction to Mathematical Systems Theory: a Behavioral Approach, Springer-Verlag, 1998.

[9] Kaushik Mallick, Model Extraction Python/Scilab Codes, http://github.com/kaushikmallick/, Updated: 11.03.2013.

[10] OpenModelica, An Introduction to OpenModelica, https://openmodelica.org, Accessed: 04.03.2013.

[11] P. Fritzson, P. Aronsson, H. Lundvall, K. Nyström, A. Pop, L. Saldamli, D. Broman, OpenModelica - a Free Open-Source Environment for System Modeling, Simulation and Teaching, in: IEEE International Conference on Control Applications, 2006, pp. 1588 –1595.

[12] Wikipedia, Python Programming Language,
http://en.wikipedia.org/wiki/Python_(programming_language), Accessed: 04.03.2013.

[13] Python Software Foundation, Python Programming Language, http://www.python.org/about/, Accessed: 04.03.2013.

[14] Wikipedia, Scilab, http://en.wikipedia.org/wiki/Scilab, Accessed: 04.03.2013.

[15] Scilab, Introduction to Scilab, http://www.scilab.org/scilab/about, Accessed: 04.03.2013.