Programmatic Symbolic Circuit Analysis

Implemented as a MATLAB Toolbox

Nicklas Vraa, Electrical Engineering B.Sc. Aarhus University - nkvraa@gmail.com - Bachelor's Thesis

Abstract: This paper introduces the basics of a new MATLAB toolbox, which is capable of pure and partial symbolic circuit analysis, as well as numerical evaluation. This project was later named ELABorate. The paper explains the approach, I decided upon when designing the software, as well as the actual implementation in code. It also outlines the motivation for the project, and lastly demonstrates some of the basic capabilities of the released software.

To try the project, please visit:
https://github.com/NicklasVraa/ELABorate

1 Introduction

This project attempts to partially automate the process of analyzing electrical circuits by abstracting low-level tasks of the analysis, which are usually done by hand, to high-level functions to be called as a few lines of code in a program. For the implementation, I've chosen MATLAB for it's live-script functionality, that allows automatic LaTeX-style formatted output. MATLAB can also export standalone c-code, and even graphical user interface applications. Implementation in another language, such as Octave or Python using the NumPy and SymPy packages should be fairly simple.

2 Motivation

Today, circuit analysis is done either analytically by hand, or numerically using software like SPICE (LTSpice, PSPICE etc.). When doing symbolic analysis by hand, one gains great insight into not only the circuit, one works on, but also into any numerical variations of that circuit. One gains a much better understanding of electrical engineering by analysing a circuit, than you do by simulation it, as any electrical-engineering professor will tell you. The downside being that the analysis often gets complicated and takes a considerable amount of time and effort, hindering the actual goal of the analysis, which is gaining understanding of the circuit, you're working with. With this project, I intend to answer this question: Is it feasible to combine the advantages of symbolic analysis with the computational power of a computer in an intuitive manner? Currently, there is no software, free or commercial, which accomplishes this. SSPICE [1] is an attempt at this but appears to have been abandoned.

3 Project specification

The overarching goal is to develop an approach for programmatic symbolic circuit analysis, but more concretely, it is to develop a toolbox for MATLAB, which is capable of symbolic circuit analysis. The objective is outlined as follows:

- Develop a program, which constructs a circuit model in code from a simple input. It should handle various common circuit elements, such as:
  - Independent AC- and DC sources.
  - Passives: resistors, capacitors, and inductors.
  - Dep. sources: CCCS, CCVS, VCCS, VCCS.
  - Larger structures: Op-Amps, transformers.
  - Non-linear elements: MOSFETs, BJTs.
  - Arbitrary sub-circuit models.

- Develop and implement methods for symbolically determining:
  - Circuit transfer functions.
  - Input- and output resistances.
  - AC/DC equivalents of a given circuit.
  - Thevenin/Norton equivalents.
  - Stability parameters.

- Develop and implement additional methods for:
  - Connecting the outputs of this toolbox to the rest of MATLAB's system analysis functions.
  - Automate input validation and debugging.
  - After-the-fact manipulation, such as removing or inserting elements and simplifying circuits.

4 Methodology

I take an object-oriented approach, defining each circuit element as its own class to ensure modularity and extendibility. The circuit is itself a class, which contain lists of elements-classes. The element-class only contain information about its own nodal connections. Additional type-specific attributes are implemented by defining sub-classes: As an example, the resistor-class inherits from the passive-class, which inherits from the element-class. Inheriting ensures easy implementation of new circuit elements in the future. I use MATLAB's symbolic toolbox for the symbolic manipulation, and the modified nodal analysis (NMA) approach for relating each circuit element to each other.

4.1 Modified Nodal Analysis

When symbolically analyzing electrical circuits, the electrical engineer usually employs the node voltage method and loop current method. Another similar, but more recent approach is modified nodal analysis [2][3] , which uses linear algebra to speed up the analysis. Modified nodal analysis (MNA) uses the element's branch constitutive equations (BCEs) i.e., their voltage-

and current characteristics and Kirchhoff's current- and voltage laws. The approach is usually broken down into 3 steps.

1. Write the KCL equations of the circuit. At each node of the circuit, write the currents coming into and out of the node. The currents of the independent voltage sources are taken from the positive to negative. Note that the right-hand-side of each equation is always equal to zero, so that the branch currents that come into the node are given a negative sign and those that go out are given a positive sign.

2. Use the BCEs in terms of the node voltages of the circuit to eliminate as many branch currents as possible. Writing the BCEs in terms of the node voltages saves one step. If the BCEs were written in terms of the branch voltages, one more step, i.e., replacing the branches voltages for the node ones, would be necessary.

3. Write down any unused equations.

Exactly how this approach will be handled by a computer, will be described in the next section.

4.2 Algorithmic MNA

Converting MNA to an algorithm, that can be performed by a computer is relatively straight-forward, assuming you already have an abstract circuit-object, which contain all the information needed for complete analysis. Much of the functional approach has already been described in the literature but must be modified and extended for additional circuit elements [4]. The goal is to define the circuit as a linear time-invariant system: $\boldsymbol{Ax} = \boldsymbol{z}$. For a circuit of n nodes and m independent sources, $\boldsymbol{A}$ is the matrix of known values, having $n + m$ rows and columns. The $\boldsymbol{x}$ vector holds all the n unknown voltages and m unknown currents. The $\boldsymbol{z}$ vector holds known quantities related to the m independent current sources and n independent voltage sources. The exact relation will be described shortly. A is comprised of several intermediate matrices, as is also the case for the vectors x and z.

$$A = \begin{bmatrix} G & B \\ C & D \end{bmatrix} \quad x = \begin{bmatrix} v \\ j \end{bmatrix} \quad z = \begin{bmatrix} i \\ e \end{bmatrix}$$

An overview of the method *without* dependent sources will now be presented. Notational note: i and j will now be used to denote a particular matrix entry, as each intermediate matrix and vector.

For the intermediate matrices comprising **A**:

- **G** is a diagonal matrix holding the sum of conductances of each circuit element connected between nodes i and j.
- **B** is an $n \times m$ matrix denoting the location of each voltage source. For entry $b_{i,j}$ a 0 means no voltage source, whereas a ± 1 means a voltage source. The sign indicates its orientation.
- **C** is an $m \times n$ matrix and simply the transpose of **B** so long as there are no dependent sources.
- **D** is an $m \times m$ matrix of zeros.

For the intermediate matrices comprising **x**:

- **v** is a vector holding the n unknown node voltages, excluding the ground voltage assumed to be 0.
- **j** is a vector holding the m unknown currents running through each voltage source in the circuit.

For the intermediate matrices comprising **z**:

- **i** is a vector holding the n sum of all current sources going into any given node.
- **e** is a vector simply holding the values of all independent current sources.

Extending this approach to handle dependent sources is a matter of augmenting these intermediate matrices appropriately. This is slightly complex and as such will be fully explained further into this section using pseudo-code. This extension is an important step, as extending the algorithm even further to handle more complex elements, such as transistors, will require the program to model these non-linear components using mainly dependent sources. As will become apparent in the implementation section, the *voltage-controlled-current-source* will play a major role in the development of these more high-level features. The following is a high-level pseudo-code description of the algorithm but extended to include dependent sources and modified to fit my project specification. The reader should view *setting* a variable not as a numeric evaluation, but as defining a symbolic expression, like appending to an equation. Firstly, I define specific notation to shorten the description.

```
Let n be the number of nodes
Let m be the number of voltage sources
Let p be the number of processed voltage sources

Let passive be either a resistor, capacitor, or inductor
Let exp denote a mathematical expression

Allocate matrices G_{n×n}, B_{n×m}, C_{m×n}, D_{m×m} and fill with 0
Let g,b,c,d denote element within matrices

Allocate vectors i_{n×1}, e_{m×1}, j_{m×1} and fill with 0
Let i,j denote anode and cathode connections
```

We then fill the intermediate matrices. This is the crux of this extended, programmatic, modified nodal analysis approach, which allows for fast and efficient symbolic computation. We start with the most basic circuit elements. Mind the notation, where 'i' and 'j' refer to both vectors and, when subscripted, matrix-indices.

```
For all passives
  If passive is resistor, set exp = 1
  Else if passive is capacitor, set exp = 1/s
  Else if passive is inductor, set exp = s

  If anode is ground
    Set g_{j,j} = g_{j,j} + 1
```

```
  Else if cathode is grounded
    Set g_{i,i} = g_{i,i} + 1
  Else
    Set g_{i,i} = g_{i,i} + 1, g_{j,j} = g_{j,j} + 1, g_{i,j} = g_{i,j} + 1, g_{j,i} = g_{j,i} + 1

For all independent voltage sources
  If anode is not grounded
    Set b_{i,p} = b_{i,p} + 1, c_{p,i} = c_{p,i} + 1
  If cathode is not grounded
    Set b_{j,p} = b_{j,p} − 1, c_{p,j} = c_{p,j} − 1
Add parsed voltage source id's to e as V_{id} and j as I_{v_id}

For all independent current sources
  If anode is not grounded
    Set i_i = i_i − I_{id}
  If cathode is not grounded
    Set i_i = i_i + I_{id}
```

For operational amplifiers, the approach is surprisingly simple and resembles the algorithm for the previous parts.

```
Let i,j denote 1st and 2nd input connections.
For all op-amps
  If first input is not grounded
    Set c_{p,i} = c_{p,i} + 1
  If second input is not grounded
    Set c_{p,j} = c_{p,j} − 1
Add parsed op-amps id's to j as I_{opamp_id}
```

Now for the active sources. The approach is very similar, but with some additional complexity, especially for VCCS's. When talking about control nodes, I am referring to the nodes, on which the source's output depend. I start with voltage-controlled sources.

```
Let i,j denote anode and cathode
Let k,l denote controlled anode and cathode.

For all VCVS's
  If anode is not grounded
    Set b_{i,p} = b_{i,p} + 1, c_{p,i} = c_{p,i} + 1
  If cathode is not grounded
    Set b_{j,p} = b_{j,p} − 1, c_{p,j} = c_{p,j} − 1
  If control anode is not grounded
    Set c_{p,k} = c_{p,k} − VCVS_{id}
  If control cathode is not grounded
    Set c_{p,k} = c_{p,k} + VCVS_{id}
Add parsed VCVS id's to j as I_{VCVS_id}

For all VCCS's
  If nothing is grounded
    Set g_{i,k} = g_{i,k} + VCCS_{id}, g_{i,l} = g_{i,l} − VCCS_{id}
    Set g_{j,k} = g_{j,k} − VCCS_{id}, g_{j,l} = g_{j,l} + VCCS_{id}
  If only anode is grounded
    Set g_{j,k} = g_{j,k} − VCCS_{id}, g_{j,l} = g_{j,l} + VCCS_{id}
  If only anode and control anode are grounded
    Set g_{j,l} = g_{j,l} + VCCS_{id}
  If only anode and control cathode are grounded
    Set g_{j,k} = g_{j,k} − VCCS_{id}
  If only cathode is grounded
    Set g_{i,k} = g_{i,k} + VCCS_{id}, g_{i,l} = g_{i,l} − VCCS_{id}
  If only cathode and control anode are grounded
    Set g_{i,l} = g_{i,l} − VCCS_{id}
  If only both cathodes are grounded
    Set g_{i,k} = g_{i,k} + VCCS_{id}
  If only control anode is grounded
    Set g_{i,l} = g_{i,l} − VCCS_{id}, g_{j,l} = g_{j,l} + VCCS_{id}
  If only control cathode is grounded
    Set g_{i,k} = g_{j,k} + VCCS_{id}, g_{j,k} = g_{j,k} − VCCS_{id},
```

Now for the current-controlled sources. These must be done last and in this order.

```
Let p denote the index the controlling voltage
Let q denote the index of the current dependent source

For all CCVS's
  If anode is not grounded
    Set b_{i,p} = b_{i,p} + 1, c_{p,i} = c_{p,i} + 1
  If cathode is not grounded
    Set b_{j,p} = b_{j,p} − 1, c_{p,j} = c_{p,j} − 1
Add parsed CCVS id's to j as I_{CCVS_id}
Find index p of controlling node in j
Set d_{q,p} = −CCVS_{id}

For all CCCS's
  Find index p of controlling node in j
  If anode is not grounded
    Set b_{i,p} = b_{i,p} + CCCS_{id}
  If cathode is not grounded
    Set b_{j,p} = b_{j,p} − CCCS_{id}
```

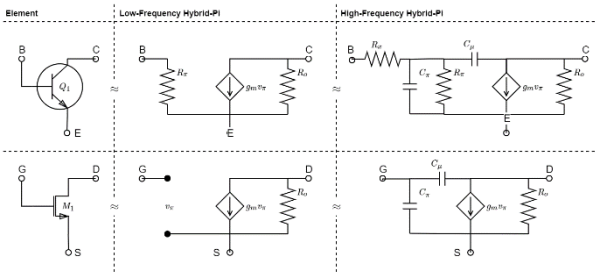
As I previously mentioned, updating the entries in the matrices, should be viewed as appending to symbolic equations. When all matrices and vectors are filled appropriately, we may compact these equations through algebraic simplification. When done, we can define A, x and z, representing the entire system as previously discussed. To solve the system for the unknown node voltages and source currents, we simply solve the system $\boldsymbol{Ax} = \boldsymbol{z}$ for $\boldsymbol{x}$.

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{z}$$

When having obtained these unknowns, the matter of defining objects-of-interest such as the circuit transfer function across any two nodes becomes trivial, since the entire circuit is now symbolically described. Evaluating the circuit numerically is simply done by inserting the desired numerical values for any component into their corresponding variable and evaluating the circuit equations.

4.3 Modelling Nonlinear Elements

The transistor is arguably the most important nonlinear circuit element, if not the most important element, period. When modelling a transistor, one may choose between various models, depending on the application, in which the transistor will operate. If the AC signal portion of a signal is small when compared to the DC biasing, as is often the case in modern electronics, we favor the small-signal model, the most popular of which is the hybrid-pi model. Using this model, we may substitute a transistor for an appropriately selected collection of linear elements. There are several versions of the hybrid-pi model, but I've chosen to focus on two versions for each of the supported transistor types. One for low-frequency, which retains simplicity, and one for high-frequency, which is slightly more complex.



As is apparent, when a transistor is to be replaced by its linear model, the program needs a way to update the entire circuit object appropriately. The number of nodes change, as do the elements to which they connect. Implementing this higher-level functionality will first require implementing low-level functions, such as removing, shortening and/or opening an element. These must therefore be functions inherent to the circuit class. Note, that the selected models are not the only possible options, and the program could easily be extended to include so-called *full*-hybrid-pi models. Especially for the MOSFET, which is inherently a 4-terminal element, when including the body-terminal, there exists far more complex linear models. When one has the previously mentioned low-level functions available, extending the software by implementing these additional linear models should not pose a significant challenge. As a general rule-of-thumb for modelling nonlinear elements in this software: if there exists a reliable linear model of the element in question, and that model is comprised of basic linear elements, it should be fairly straight-forward to provide support for said element. This is ensured by the object-oriented approach, with which the program is built.

4.4 Circuit Simplification

As any electrical engineer knows, when having two or more of the same element in series or in parallel, it is often beneficial to replace these with a single equivalent element. For an engineer, the act of spotting these cases, is done visually without giving it much thought, but for a computer to do the same, a series and parallel connection must be strictly defined. If the simplification is done recursively until no series or parallel connections are left, one only needs to consider the most basic case: When the series or parallel connection consists of only *two* elements.

- A **series** connection has one and *only* one node shared among the two elements. No other element may be connected to the shared node.
- A **parallel** connection has two nodes shared among the two elements.

Using these definitions, one can create an algorithm for both detecting and replacing series and parallel elements.

5 Implementation

This section details the implementation of the program in code. Only the larger and most important structures of the program will be described. See the code on the GitHub repository for more detail. The code is extensively commented and attempts to adhere to the guidelines set by MATLAB for its users and employs good-practices of object-oriented programming in general.

5.1 Input to the program

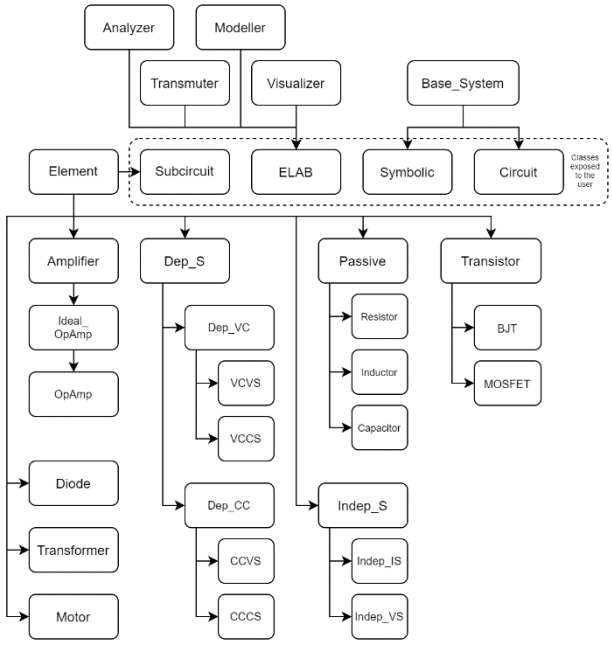
The industry standard for defining circuits is using the netlist format. Simply a text file, where each line is a component. Each component is defined by a symbol identifying the type, a name, its nodal connections, and some additional information specific to the type of component. For the MNA algorithm to work properly, the reference node must be ground. Each node must also obey the sign convention. The syntax used by this program is SPICE-like with a few simplifications. I have chosen the custom file extension to be '.circ' for easier distinction between circuit-files and regular text files, but it is still a raw text file. Below is an overview of the syntax. Anything enclosed by square brackets is a variable set by the user, like the name or number given to a resistor. Anything in italic is an optional parameter, like the numerical value of a voltage source. If not specified, a symbolic is automatically assigned for instead.

An "n" is for "node", and "cn" is for a "controlling node". "mode" must be either "AC" or "DC". For MOSFETs, G, D and S denote where its gate, drain and source connect. For BJTs, B, C and E denotes where its body, collector and emitter connect.

Element	**Syntax**
Resistor	R[id] [+n] [-n] [*value*]
Capacitor	C[id] [+n] [-n] [*value*]
Inductor	L[id] [+n] [-n] [*value*]
V-source	V[id] [+n] [-n] [mode] [*value*]
I-source	I[id] [+n] [-n] [mode] [*value*]
VCVS	E[id] [+n] [-n] [+cn] [-cn] [*gain*]
VCCS	G[id] [+n] [-n] [+cn] [-cn] [*gain*]
CCVS	H[id] [+n] [-n] [cn] [*gain*]
CCCS	F[id] [+n] [-n] [cn] [*gain*]
Op-Amp	O[id] [+n] [-n] [output-node]
BJT	M[id] [B] [C] [E] [gain] [*internal_Rs*]
MOSFET	Q[id] [G] [D] [S] [gain] [*internal_Rs*]

5.2 Modelling Circuit Analysis

Fig. 1 outlines the class-tree. Only the fenced classes are exposed to the user to maintain user-friendliness. All the element-classes are only interfaced with by the circuit- and ELAB-class.



The classes are laid out this manner to reduce redundancy using the object-oriented principle of inheritance. For instance: The resistor, capacitor and inductor all have several common characteristics, and as such, may as well share the code that implements them, which in this instance comes from the *passive*-class. All element classes have a "to_string"-function, so they may be converted back to a netlist entry if need be. This methodology repeats for most of the classes, the exception being the *ELAB*-class. This class collects functionality from the four classes: *analyzer*, *modeller*, *transmuter*, and *visualizer*. This is for the sake of the user, making the program simpler to use, while retaining some level of abstraction for anyone wishing to contribute the program. As the class names suggest, these four classes are responsible for a separate part of the task that constitutes circuit analysis. They interact directly with any given circuit object.

5.3 The Circuit Class

The circuit class is at the heart of the program. It handles the parsing of a given netlist and creates a list of each element type. All analytical results obtained from a circuit are stored within the circuit-object itself as to reduce the amount of repeat calculation. The circuit class also contain low-level functions, that allows changing the circuit, post-constructor. The following sections detail will describe the idea of these functions. Any *emphasized* word corresponds to a real MATLAB function in the circuit class.

5.3.1 Basic Circuit Operations

Shorting or *opening* an element is standard procedure when conducting circuit-analysis, so naturally these must be available operations tied to the circuit object. Common for both these operations are that they *remove* an element. The Circuit element must therefore be able to search its own element-lists and delete the object from the appropriate list. After removing an element, everything about the circuit may change, and so the circuit object must be *updated* to reflect this. The netlist must be updated, as well as the number of nodes and other critical information about the circuit. The update function must also *clean* the circuit by assigning new numbers to nodes and *trimming* any elements that may be rendered inconsequential for example removing another element which may only be connected to the rest of the circuit though the element, which was just removed by open-circuiting. Below are descriptions of the short- and open-circuiting algorithms. An uppercase variable denotes a list. A lowercase variable denotes an entry in said list.

Shorting

Shorting an element is a surprisingly complex operation for a computer to carry out.

```
Given an element x
Find the node n connected to x with the lowest identifier.

For all terminals T_1 of x
  Find all elements Y connected to t_1
  For all y in Y
    For all terminals T_2 of y
      If t_2 is equal to t_1
        Change t_2 to n

Remove x from circuit
```

Opening

Opening an element is simple and simply removing the element from the circuit but *trimming* afterward is another matter.

```
Given an element x
Remove x from circuit
Let Y be all elements in the circuit
Initialize a list L of items to be removed
```

Trimming involves finding any elements not connected to anything else in the circuit and storing these elements in a list for later removal. Afterwards, we need to find any element, which is only connected to itself, as these are also inconsequential for the circuit. Naturally, these will also be added to the removal list.

```
For all y in Y
  Let sum = 0
  For all terminals t of y
    If node n at t is not ground
      Let Z be y's connected to n
      sum += length of Z
  If sum = 0
    Append Z to L

For all y in Y
  Let found = false
  For all terminals t_1 of y
    For all terminals t_2 of y excluding t_1 itself
      If t_2 is not equal t_1
        Set found = true
        Break loop

  If not found
```

```
Append y to L

Remove all elements in L from circuit
```

5.4 The ELAB class

The ELAB class itself does not contain any functionality, but inherits everything from the four classes, which are presented below.

5.4.1 The Analyzer

The *Analyzer*-class implements modified nodal analysis in it's *analyze* function. Exactly how this implementation is done was explained previously in its own section. The function takes a circuit object and sets the circuit's own properties appropriately. These properties include the circuit equations, symbolic expressions for the voltage at every node, and symbolic expressions for the current through every element. The *evaluate* function simply evaluates every symbolic expression with any given numerical values, if any was given in the circuit netlist. The *ec2sd* function, which is short for *electrical-circuit-to-s-domain*, takes the circuit object, as well as any two nodes within, and computes the symbolic transfer function between them, in the Laplace-domain. It will both return the result, and assign the result to the appropriate circuit property, so one doesn't have to run the function again to get the result a second time. When the voltage at every node has been symbolically described, one only needs to divide the expression for the output node with the expression for the input node. If the *analyze* function has not been run before running *ec2sd*, the program will do it for you, automatically. The numerical counterpart to this function is *ec2tf*, which is short for *electrical-circuit-to-transfer-function*. This function will create a MATLAB Transfer-Function object, which can be used in conjunction with MATLAB's extensive systems analysis functionality.

Apart from these functions, which are very specific to the field of electrical engineering, a range of high-level systems analysis functions are also implemented in the *Analyzer*-class. At the time of implementation, I found these to be missing from the standard MATLAB ecosystem. These functions mostly concern themselves with the stability of any given system, and so have broader application than simply electrical circuits.

5.4.2 The Modeller

5.4.3 The Transmuter

5.4.4 The Visualizer

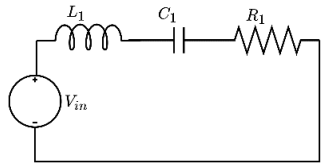
6 Results

This sections presents the stable release of the program. I will be going over some of the features, but not all, as they are most easily understood, when they are presented in the environment, in which they are intended to function, i.e. MATLAB's LiveScript editor. There are plenty of LiveScript examples and netlist-files on the project's GitHub-page, ready to be tried and tested within MATLAB.

6.1 Using the Program

The program is designed to be used in conjunction with MATLAB's LiveScript, as it neatly outputs the results of the program. The reader is encouraged to experiment with the LiveScripts, which are available on the project repository (see abstract). For this simple example, we will analyse a classic RLC circuit using the program. The numerical values have been chosen at random. We begin by defining a netlist in a text file called "rlc.circ":

```
Vin 1 0 DC 5
R1 3 0 1000
L1 1 2 1
C1 2 3 0.0001
```



The netlist is then fed to the circuit object *constructor* via its file path. Passing the circuit object to the *analyze* function from ELAB is all that is needed for the program to learn everything about the circuit.

```
circuit = Circuit('rlc_series.txt')
ELAB.analyze(circuit)
```

Say you wanted to know every how every node voltage is defined, you would simply look at the circuit object attributes, after it has been analysed.

```
circuit.symbolic_node_voltages
ans =
```

$$\begin{pmatrix} v_1 = \mathrm{Vin} \\ v_2 = \dfrac{\mathrm{Vin}\ (C_1 R_1 s + 1)}{C_1 L_1 s^2 + C_1 R_1 s + 1} \\ v_3 = \dfrac{C_1 R_1\, \mathrm{Vin}\, s}{C_1 L_1 s^2 + C_1 R_1 s + 1} \end{pmatrix}$$

6.2 Future expansions

As the input is simply a netlist, it would make sense to implement an integrated way to create said netlist, either graphically with a drag-and-drop UI, or with a simple drop-down list, where the user may choose their components and interconnections. Another possibility, which I personally find very intriguing, is to use computer vision to create a netlist from a hand-drawn circuit. This would of course require some type of artificial intelligence which would need to be fed a heap of labeled hand-drawn circuits. MATLAB already has an extensive computer vision library and the task of building the system is a realistic one. Acquiring the training, validation and testing data presents the greatest challenge. This addition to the program would be worthwhile, since the most time-consuming process of symbolic circuit analysis is now defining the netlist. Another way to improve user-friendliness would be to automatically

generate circuit drawings when having modified the netlist through the program. This could be done using the excellent LaTeX package CircuiTikZ [5], which takes *LaTeX/netlist-like* input and draws a circuit figure. MATLAB's LiveScript already supports automatic rendered LaTeX output, so the challenge here would be to put the netlist onto the form, which LaTeX understands. Another possible addition to the program would be to write an implementation in Python. This is entirely possible using the packages NumPy [6] and SymPy [7]. Python does not have native support for system analysis at the level MATLAB. Nor is it as seamless to generate formatted output. These features, however, are not the core of this piece of software. The speed differences would be negligible, especially if one implements the core functionality in Cython [8] or using the C-code API for Python. The most important difference is that this would free any non-professional users from the MATLAB licensing fee.

7 Conclusion

References

[1] Willow Electronics. (2021, December 13). Symbolic SPICE®. Willow Electronics, Inc. Retrieved March 10, 2022, from https://willowelectronics.com/symbolic-spice/

[2] Chung-Wen, H., Ruehli, A., & Brennan, P. (1975, June). The modified nodal approach to network analysis. IEEE Transactions on Circuits and Systems, 22(6), 504-509. doi:10.1109/TCS.1975.1084079

[3] Decarlo, R.A., & Lin, P.M. (1995). Linear Circuit Analysis: Time Domain, Phasor, and Laplace Transform Approaches.

[4] Stojadinovic, N. (1998). VLSI circuit simulation and optimization: V. Litovski and M. Zwolinski, Chapman and Hall, London, UK, 1996, 368 pp., ISBN: 0-412-63890-6, Microelectronics Journal, 29, 359.

[5] Redaelli, M., Erhardt, S., Lindner, S., & Romano Giannetti, R. (2022, February 4). CTAN: Package CircuiTikZ. CircuiTikZ. Retrieved April 18, 2022, from https://ctan.org/pkg/circuitikz

[6] Numpy Development Team. (2022). NumPy Reference. Numpy.Org. Retrieved April 18, 2022, from https://numpy.org/doc/stable/reference/index.html

[7] SymPy Development Team. (2022, March 19). Reference Documentation — SymPy 1.10.1 documentation. SymPy.Org. Retrieved April 18, 2022, from https://docs.sympy.org/latest/reference/index.html

[8] Behnel, S., Bradshaw, R., Dalcín, L., Florisson, M., Makarov, V., & Seljebotn, D. S. (2022). Cython: C-Extensions for Python. Cython.Org. Retrieved April 18, 2022, from https://cython.org/#about