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 the key features and capabilities of the released software.

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

Contents

1 Introduction 4
2 Motivation 4
3 Project specification 5
4 Methodology 6
4.1 Modified Nodal Analysis 6
4.2 Algorithmic MNA 6
4.3 Modelling Nonlinear Elements 9
4.4 Circuit Simplification 10
4.5 Thevenin/Norton Equivalents 10
5 Implementation 11
5.1 Input to the program 11
5.2 Modelling Circuit Analysis 11
5.3 The Circuit Class 12
5.3.1 Basic Circuit Operations 12
5.4 The ELAB class 14
5.4.1 The Analyzer 14
5.4.2 The Modeller 15
5.4.3 The Transmuter 17
5.4.4 The Visualizer 17
6 Results 18
6.1 Using the Program 18
6.1.1 Simplifying a circuit 19
6.1.2 Adding, removing and Cleaning 21
6.1.3 Thevenin/Norton Equivalents 22
6.1.4 Dependent Sources 23
6.1.5 Operational Amplifiers 25
6.1.6 AC/DC-equivalents 27
6.1.7 Non-linear Modelling 28
6.1.8 Time-domain Analysis 32
6.1.9 Cloning and Exporting 33
7 Conclusion 33
7.1 Project Organization 34
7.2 Unreleased functionality 34
7.3 Future expansions 34
8 References 35
9 Appendices 36
9.1 Appendix I – Systems 36
9.1.1 Base_System.m 36

9.1.2 Circuit.m 36
9.1.3 ODE.m 43
9.2 Appendix II – Engine 44
9.2.1 ELAB.m 44
9.2.2 Analyzer.m 44
9.2.3 Modeller.m 53
9.2.4 Transmuter.m 61
9.2.5 Visualizer.m 66
9.3 Appendix III – Elements 68
9.3.1 Element.m 68
9.3.2 Indep_S.m 68
9.3.3 Impedance.m 70
9.3.4 Transistor.m 73
9.3.5 Amplifier.m 75
9.3.6 Dep_S.m 76

1 Introduction

This project attempts to automate the process of analyzing electrical circuits by abstracting low-level tasks of the analysis, which are traditionally 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.). As is apparent from the acronym - **S**imulation **P**rogram with **I**ntegrated **C**ircuit **E**mphasis – these approaches rely on *simulation*. In other words: these approaches are built around the assumption, that we have already been given numerical values for each and every element in the circuit or have decided numerical values for each component before knowing how the circuit actually functions. Often times, this is an incorrect assumption. Especially and naturally in the process of *designing* a circuit.

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 one does by simulation it, as any adept electrical engineer will tell you.

While symbolic analysis is great in principle, there are drawbacks to the symbolic approach. 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 towards understanding or designing. This becomes immediately apparent to any student of electrical engineering, when we are given a handful of circuits to analyze as a take-home assignment and are expected to use 10+ hours completing the task. The most time-consuming part of this process is often the menial tasks, like simplifying elements or modelling non-linear components.

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? At the start of undertaking this project, there is no software, free or commercial, which accomplishes this task to a satisfactory degree. S-SPICE [1] (Symbolic-SPICE) 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. As MATLAB itself has become an instrumental tool for anyone working in a field concerning itself with linear algebra, like signal processing and control theory, so too, should MATLAB become a valuable tool for circuit analysis. The concrete objectives of this project are outlined here:

- **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, with source signals being defined in multiple domains.
    - Passive elements including a generic impedance element and resistors, capacitors, and inductors.
    - Dep. sources: current-controlled-current-sources, current-controlled-voltage-sources, voltage-controlled-voltage-sources, and voltage-controlled-current-sources.
    - Larger structures: operational amplifiers, transformers, and arbitrary sub-circuit-structures.
    - Non-linear elements: metal–oxide–semiconductor-field-effect-transistors and bipolar-junction-transistors.

- **Develop and implement methods for symbolically and numerically determining:**

    - Circuit transfer functions from any node to another.
    - Input- and output resistances from any node to another.
    - AC/DC equivalents of a given circuit.
    - Thevenin/Norton equivalents of a given circuit, as seen by a specified load element.
    - System 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
    - Simplifying circuits, such as combining series or parallel elements.

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 and functions for manipulating its own structure. 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 impedance-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 within the circuit-class. This constitutes the core of the program. In addition, I will be designing functions for common manipulation and analysis of circuits.

4.1 Modified Nodal Analysis

When symbolically analyzing electrical circuits, the electrical engineer usually employs the node voltage method and/or loop current method. Another similar, but more recent approach is *modified nodal analysis* [2][3], which takes advantage of 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 several steps.

1. Apply Kirchhoff's current law to define current equations for each node in the circuit. At each node, 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. Define an equation for the voltages going through each voltage source. We can now solve the system of $n-1$ unknowns.

This description seems simple enough in principle but becomes slightly complex when implementing it as a program to automate and speed up the process. 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 complex, but becomes easier if one assumes, that 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 [4] but must be modified and extended for additional circuit elements. In essence, 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. $\boldsymbol{A}$ is comprised of several intermediate matrices, as is also the case for the vectors $\boldsymbol{x}$ and $\boldsymbol{z}$.

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

An overview of the method *without* dependent sources will now be presented. Notational note: i and j will now also be used to denote a particular matrix entry when use in subscript.

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 a crucial 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 key 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 the 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 $impedance$ be either a generic, resistor, capacitor, or inductor
Let $exp$ denote a mathematical expression

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

Allocate vectors $i_{n\times 1}$, $e_{m\times 1}$, $j_{m\times 1}$ 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 impedances
  If impedance is generic or resistor, set $exp = 1$
  Else if impedance is capacitor, set $exp = 1/s$
  Else if impedance is inductor, set $exp = s$

  If anode is ground
    Set $g_{j,j} = g_{j,j} + exp$
  Else if cathode is grounded
    Set $g_{i,i} = g_{i,i} + exp$
  Else
    Set $g_{i,i} = g_{i,i} + exp$, $g_{j,j} = g_{j,j} + exp$, $g_{i,j} = g_{i,j} + exp$, $g_{j,i} = g_{j,i} + exp$
```

We then account for the independent voltage- and current sources.

```
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 id's to $e$ as $V_{id}$ and to $j$ 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 turns out to be surprisingly simple and resembles the algorithm for the previous elements.

```
Let $i,j$ denote 1st and 2nd input connections.
  For all op-amps
    Set $b_{output,p} = b_{output,p} + 1$
    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 dependent 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. We start with voltage-controlled sources.

```
Let $i,j$ still denote anode and cathode
Let $k,l$ denote control 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 as the algorithm needs to have knowledge of all voltage sources, including the dependent ones, before processing these.

```
Let $p$ now be the index the controlling voltage
Let $q$ be the index of the 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 was 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.

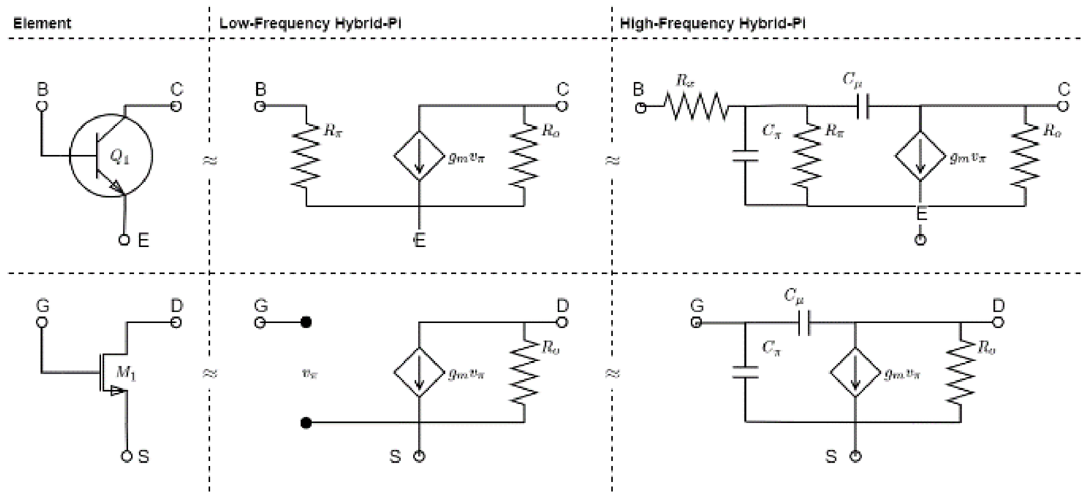


Figure 1 - Variations of the hybrid-pi model.

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 practical 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 dependable 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 elements 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, the event of a series or a 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. Any other elements – or collection of elements - may be connected to these nodes.

The new equivalent element being substituted in must follow these constraints.

- The element being substituted in for a **series** connection must have nodal connections that are not shared between the two elements in the series configuration, i.e., if the two elements together connect to nodes 1, 2 and 3, and they share node 2, but not node 1 and 3, then the new element will connect to node 1 and 3.

- The element being substituted in for a **parallel** connection will simply share its nodal connections with either of the two elements in the parallel configuration.

Using these strict definitions, we can create an algorithm for both detecting and replacing series and parallel elements. Running this algorithm in a loop and keeping track of how many series and parallel are detected in each run, any number of series or parallel can be reduced to its simplest equivalent element.

4.5 Thevenin/Norton Equivalents

Finding a Thevenin or Norton equivalent circuit is a widely used tool for analyzing circuit. Thevenin- and Norton-equivalents share the same initial approach.

1. **Open** the element being considered the load.
2. **Analyze** the circuit, finding the voltage across the open gap.
3. **Open** all voltage-sources and **short** all current-sources.
4. **Simplify** the circuit to a single impedance.

- For the Thevenin equivalent circuit:

  a. **Add** a voltage source with the value found in step 2.
  b. **Add** the impedance found in step 4 in series with the voltage source.
  c. **Add** back the load element.

- For the Norton equivalent circuit.

  a. Apply Ohm’s law: $I_{Norton} = V_{step\ 2}\ /\ Z_{step\ 4}$
  b. **Add** a current source with the value found in step a.
  c. **Add** the impedance found in step 4 in parallel with the current source.
  d. **Add** back the load element.

The words in bold will be standalone functions.

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 best practice 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 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 and files with the ".txt" extension will work equally well. 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. Any numerical value, like the capacitance of a capacitor, must be given in standard SI-units, so a capacitor having a capacitance of $1\mu F$ must be defined as "C[id] … 0.000001" or with a mathematical expression equivalent to this. No units should be given in the netlist file. This is for the sake of simplicity.

Element	Syntax
Resistor	R[id] [+n] [-n] [*value*]
Capacitor	C[id] [+n] [-n] [*value*]
Inductor	L[id] [+n] [-n] [*value*]
Impedance	Z[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 resistances*]
MOSFET	Q[id] [G] [D] [S] [gain] [*internal resistances*]

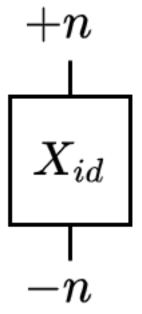


Figure 2 - Generic element.

5.2 Modelling Circuit Analysis

As mentioned in the section on methodology, the program is to be implemented in an object-oriented fashion, leveraging the benefits of modularity and inheritance. In other words: each circuit element is a class. Fig. 1 outlines the class-tree. It is possible to use the program without ever directly interfacing with the element-classes. The Circuit- and the ELAB-class are capable of interfacing with these on their own, expect in the case, where one wishes to insert/add an element into an existing circuit object, where the user may need to specify new nodal connections for surrounding elements. 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 *impedance*-class. All element classes have a "to_string"-function, so they may be converted back to a netlist entry if necessary. 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. Any class, which is outlined by a dashed line, does not have its own constructor and serves only as a superclass for others. Any class in grey is yet to be implemented.

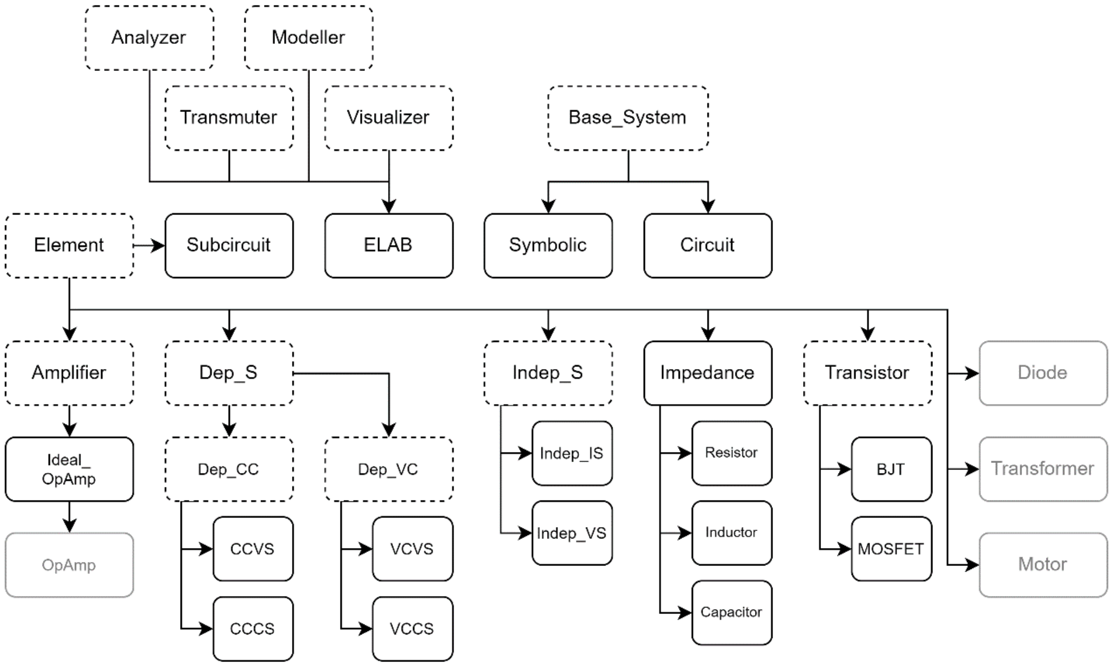


Figure 3 - The class-tree showing how each class inherits from another. Abstract classes are dashed. Unfinished classes are grey.

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.

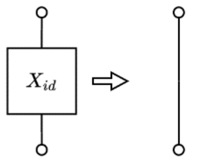
5.3.1 Basic Circuit Operations

Shorting or *opening* an element is standard procedure when conducting circuit-analysis, so naturally these must be available operations inherent to the Circuit-class. Common for both these operations are that they *remove* an element. A Circuit object must therefore be able to search its own element-lists and delete the object from any 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 conduct. Below is the algorithm, I devised to carry out this operation.

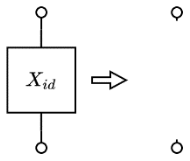
```
Given an element $x$
Find node $n$ connected to $x$ with the lowest number.

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
Update global properties of the circuit
```



Opening

Opening an element is simple and is done by just removing the element from the circuit but *trimming* afterward is another matter. Trimming will be explained shortly.

```
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

If there exist elements in the circuit, which have been rendered inconsequential after manipulating the circuit, they have to be removed, as to not affect any further circuit analysis. I refer to this as trimming. 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. These two steps are shown algorithmically below in separate boxes.

```
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

Update global properties of the circuit
```

Cleaning

Cleaning a circuit netlist is what I will call the attempt to clean up node-labelling to avoid skips as a result of the input-file from the user containing nonsensical node naming, or as a result of having removed or added elements post-circuit-construction. Clean updates the appropriate elements with sensible values for their nodal connections and updates critical properties of the circuit, such as the netlist. In the results section, the utility of the clean function will become more apparent. For now, I simply define the algorithm for carrying it out.

```
For the number of nodes in the circuit
  Let k be the current node value
  Let L be an empty list

  For all elements in the circuit
    Add all element terminal values to L

  Let m be the smallest value in L which is also above k

  For all elements in the circuit
    Let X be the current element

    For all terminals of X
      Let t be the current terminal
      If  t = m
        Set t = k+1

  Update global properties of the circuit
```

Updating global properties of the circuit refers to updating the total number of nodes, the number of each element in their respective arrays, the total number of elements, the number of sources, etc. This will all be done in a separate function, because it is such a generic meta-operation for the circuit-object to carry out on itself, whenever the user – or itself – changes anything about the circuit. Many more such functions are defined in the code but is most easily understood by reading the code itself, which is contained in the appendices of this report.

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 the *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.

Stability Analysis

Apart from the functions mentioned in the previous section - 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 focus primarily on the stability of any given system, and so have broader application than simply electrical circuits. I will be omitting further description of these, as they are not strictly related to circuit analysis, but rather a supplement to circuit analysis when viewing the circuit as a general system. The full code for each can be found in appendix II.

- The *routh* function takes the polynomial coefficients of the characteristic equation of a system and returns the symbolic Routh-array, which can be used to determine the stability of the system.

- The *critical* function can take this Routh-array as an input and returns the interval for the value K in which the system is stable.

- The *breakaway* function takes a MATLAB *system* object and the gain and returns the breakaway point for the system root locus.

- The *static_error_K* function takes a transfer-function and returns the static position error, the static velocity error, and the static acceleration error, as well as printing the steady-state error.

- The *dominant* function takes a transfer-function and determines and returns the system's dominant poles.

- The *damp* function takes a transfer function and returns the natural frequency ω_n and the damping ratio ζ of the system.

5.4.2 The Modeller

The *Modeller*-class handles everything that has to do with modelling and/or modifying circuit objects, including any functions, which aid in accomplishing these tasks. This is where the user, or other functions can find AC- or DC-equivalent circuits, Thevenin- or Norton equivalents and hybrid-pi models as well as biasing. This is also the class that contain all the code for automatic simplification of circuits.

AC/DC-equivalents

These functions, *ac_eq* and *dc_eq*, simply check whether an independent source has been designated as AC or DC in the netlist, then short or open the appropriate elements accordingly, using the lower-level functions, which have been described in a previous section.

- The *dc_eq* function will *short*-circuit any AC voltage-sources and inductors as well as *open*-circuit AC current-sources and capacitors.
- The *ac_eq* function will *short*-circuit any DC voltage-sources and *open*-circuit DC current sources.

Both functions clean up after themselves, using the *trim* function. I want to stress the importance of having implemented these types of functions in a hierarchical manner, because it ensures that any other user of this program, with knowledge of MATLAB coding, can easily extend and contribute their own higher-level operations to the program, without having to re-implement lower-level functions.

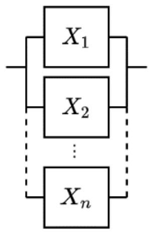
The other functions that were mentioned at the start of this section are best understood by reading the code, which is contained in appendix II. They also build on top of the lower-level functions. The simplification is interesting and underpins a great deal of the other functions, and as such will be the next section.

Automated Recursive Simplification

Simplification, as is described in a previous section, is surprisingly tricky to implement, and as such, I will first describe how the problem of automated simplification can be broken down into smaller problems, which all are given their own function. Simplifying first requires the program to determine what can and cannot be simplified. The function *is_same_type* simply checks if the classes of two circuit-objects are equal. The *is parallel* function and the *is_series* function are described here in pseudo-code:

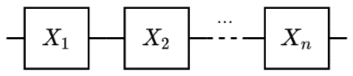
Detecting Parallels

```
Let $x_1$ and $x_2$ be circuit elements

If $z_1$'s anode = $z_2$'s anode and $z_1$'s cathode = $z_2$'s cathode
or $z_1$'s anode = $z_2$'s cathode and $z_1$'s cathode = $z_2$'s anode
  Return true
Else
  Return false
```



Detecting Series

```
If parallel
  Return false
Else
  If $z_1$'s anode = $z_2$'s anode or $z_1$'s anode = $z_2$'s cathode
    Let *shared* be $z_1$'s anode
  Else if $z_1$'s cathode = $z_2$'s anode or $z_1$'s cathode = $z_2$'s cathode
    Let *shared* be $z_1$'s cathode
  Else
    Return false
  If anything else is connected to *shared*
    Return false

Return true
```



ID	Series equivalent	Parallel equivalent
R	R1+R2	(R1*R2)/(R1+R2)
C	(C1*C2)/(C1+C2)	C1+C2
L	L1+L2	(L1*L2)/(L1+L2)
V	V1+v2	V1 (only when v1=v2)
I	Not allowed	I1+I2

Using these detection functions, the program can build 2-by-n lists of parallel and series elements, and then construct an equivalent element. How this element is defined is of course dependent on the type of element in question. Fig. 2 is a short table, describing how the equivalent elements are defined.

After having simplified appropriately, new series and/or parallel elements may occur. To account for this, the *simplify* function only has to run in a loop, until no series or parallel elements can be detected. Because of the object-oriented manner in which the individual elements are implemented, this operation is trivial. The reader may have noticed that only the operation of simplifying *pairs* ($n = 2$) of series or parallel elements have been directly implemented, but because of this pseudo-recursive approach, this is all that needs to be strictly defined. For instance, a parallel connection of $n = 5$ elements simply run the *simplify_series* function $n - 1$ times.

5.4.3 The Transmuter

The *Transmuter*-class deals with representing the circuit object in various ways. As any electrical engineer knows, having multiple representations of a circuit in various domains, allows one to understand the circuit on a deeper level.

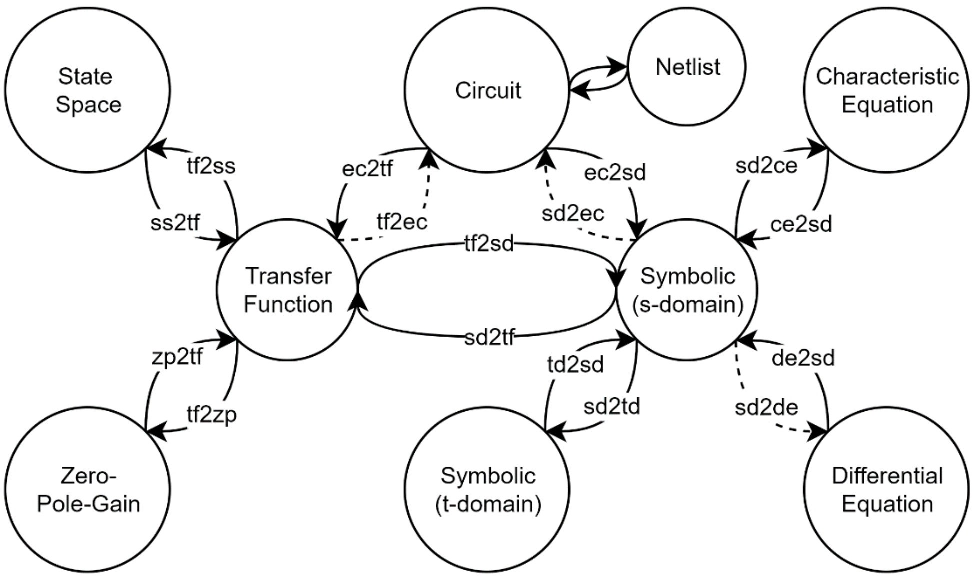


Figure 4 - Map of how the transmuter converts from one domain to another. Arrows correspond to functions.

The spheres represent domains, and the arrows denote which function in the transmuter class will convert between two domains. Note, that dashed arrows represent functions, which have not yet been implemented. Also note that the symbolic representations are at the center, and that every domain is connected, directly or indirectly, to everyone else. The transmuter class also contains a *transmute* function, that can convert any representation into another, by deciding which path of conversion is appropriate. Say, the user has the symbolic representation of the circuit in the time-domain and wants to analyze the system in state-space. The *transmute* function will go through this path, when called with the parameters td and ss. It can also print the process.

$$td2sd \rightarrow sd2tf \rightarrow tf2ss$$

In summation, the current circuit representation may be converted into any other representation and back, if there exists a solid path of conversion between them.

5.4.4 The Visualizer

The *Visualizer*-class contains plotting functions. Given a system-object representing the circuit, this class can plot the response of the circuit, optionally relative to a reference response. Many of the functions within the Visualizer are simple mappings to already existing MATLAB system analysis functionality but customized to the field of electrical circuits. The Visualizer does not provide any significant functionality but is rather for convenience. In the future, it may be able to output circuit diagrams, which will be further explained in the conclusion section.

6 Results

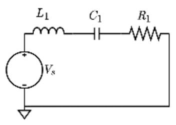
This section presents the stable release of the program. I will be going over the most important features, but not all, as they are most easily understood, when they are presented in the environment, in which they are intended to be used, i.e., MATLAB's LiveScript editor. There are plenty of LiveScript examples and netlist-files on the project's GitHub-page, ready to be tried and tested within MATLAB, and I strongly encourage the reader to do so, as well as adding to the collection of netlists and LiveScripts, via GitHub pull-requests. Any functionality relating more broadly to systems analysis have been omitted in the report for conciseness, but examples can be found in the repository.

6.1 Using the Program

As mentioned, 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).

Introductory Example
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, which we arbitrarily call "rlc_series.circ". The file looks like this:

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



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 the most important properties of the given circuit.

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

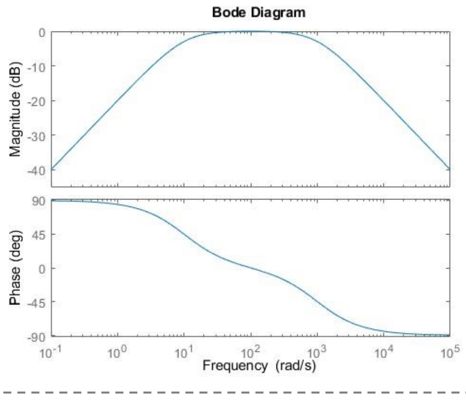
Say we wanted to know every how every node voltage is defined symbolically. We would simply look at the circuit object attributes, after it has been analysed using the analyze function on the ELAB class. From this, one can already deduce how each component in the circuit affects its overall behavior, and the effects of adjusting numerical parameters, such as capacitance of C_1, can be predicted from a few lines of code. From the circuit, one can also easily create a transfer function object, only giving the input and output nodes, which can be of any combination. MATLAB can then be used to visualize the circuit behavior as with any other system. Plotting the Bode diagram, we see that this circuit acts as a band-pass-filter.

```
Circuit.symbolic_node_voltages
```

ans =

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

```
TF = ELAB.ec2tf(circuit, 1, 3)
bode(TF)
```

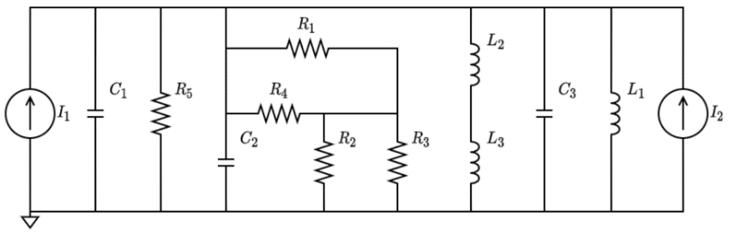


All of MATLAB's built-in functionality for treating systems on the form of a transfer function, a state space, etc. now applies to circuits.

6.1.1 Simplifying a circuit

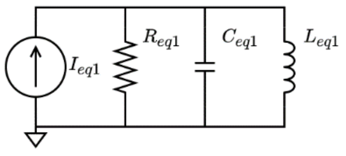
The program is capable of recursively simplifying a circuit of arbitrary complexity in seconds. For the simplify function to work properly, it utilizes the lower-level functions mentioned previously, including the shorting- and opening-functions, as well as the cleaning and trimming functions. Take this example with multiple sources, resistors, capacitors and inductors.

```
circuit = Circuit('circuits/complex.txt');
circuit.list

ans =
    'I1 1 0 AC 10
     I2 1 0 AC 5
     R1 1 2 1000
     R2 2 0 2000
     R3 2 0 2000
     R4 1 2 3000
     R5 1 0 1000
     L1 3 1 0.02
     L2 3 0 0.03
     L3 1 0 0.2
     C1 1 0 0.000002
     C2 0 1 0.000003
     C3 1 0 0.00002'
```



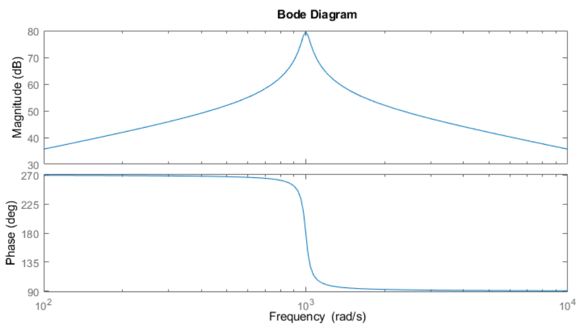
Simply calling the simplify function will reduce the circuit complexity. This circuit was purposely designed to reduce nicely into a single element of each type in a parallel configuration as shown below. Turns out, the original circuit can therefore act as a single frequency isolator when taking the output from node 1, which would not have been apparent at first glance or even after analysis directly on the circuit before simplification.

```
ELAB.simplify(circuit);
circuit.list

ans =
    'I_eq1 1 0 AC 15
     R_eq1 1 0 7000/11
     L_eq1 1 0 0.04
     C_eq1 1 0 0.000025'
```



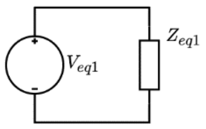
To illustrate that this is indeed a single frequency isolator, we can use the program to find the transfer function and plot it as in the introductory example.

```
ELAB.evaluate(circuit);
TF = ELAB.sd2tf(rhs(circuit.numerical_node_voltages(1)));
bode(TF)
```



Calling the simplify function with its second parameter set to true will treat any resistor, capacitor or inductor as a generic impedance element, and it is therefore possible to simplify the three remaining passive elements to a single impedance element. This functionality also enables input- and output-impedance analysis on applicable circuits, such as amplifiers.

```
ELAB.simplify(circuit, true);
circuit.list

ans =
    'V_eq1 1 0 AC V1+V2
     Z_eq1 1 0 (7000.0*s)/(11.0*s+280.0*s^2+280000000.0)'
```



The simplify function also works symbolically and even ensures appropriate naming of the new equivalent elements, even if the elements cannot be simplified down to a single element as seen in this example.

```
circuit = Circuit('circuits/multiple_eqs.txt');
circuit.list

ans =
    'Vs 1 0 DC Vs
     R1 1 2 R1
     R2 2 3 R2
     R3 4 5 R3
     R4 4 5 R4
     R5 5 0 R5
     C1 3 4 C1'
```

```
ELAB.simplify(circuit);
circuit.list

ans =
    'Vs 1 0 DC Vs
     R_eq1 1 2 R_1+R_2
     R_eq2 3 0 R5+(R3*R4)/(R3+R4)
     C1 2 3 C1'
```

Again, the reader is encouraged to try out the simplify functionality on any applicable circuit, they can think of and of course suggest expansions to the simplification ruleset, on the project's GitHub repository. The simplify function and its lower-level dependencies have been thoroughly tested, but of course edge cases may present bugs, which can also be reported on the GitHub page.

6.1.2 Adding, removing and Cleaning

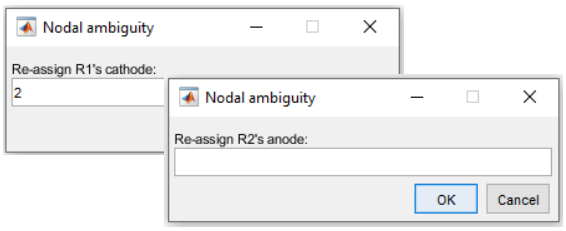
Adding or removing elements are used by several higher-level functions, but are also available to be used directly by the user. Say, we want to add an additional resistor to a simple voltage divider. If the element is defined to be connected to existing nodes, for example when adding an element in parallel with another, the add function works as expected.

```
circuit = Circuit('circuits/voltage_divider.txt');
circuit.list

ans =
    'Vin 1 0 DC 5
     R1 1 2 1000
     R2 2 0 3000'
```

```
R = Resistor('Rx',2,0,2000);
circuit.add(R).list

ans =
    'Vin 1 0 DC 5
     R1 1 2 1000
     R2 2 0 3000
     Rx 2 0 2000'
```

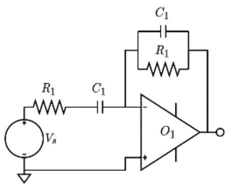
Of course, one may want to add an element “onto a wire”, i.e. onto a single node, the process is slightly more complicated. When -1 is used to denote a nodal connection, it means that the node has yet to be created, so in the next case, the resistor R_x will go onto the wire between R_1 and R_2. However, this will effectively create a new node, and it is not a given how any other elements connected to node “2” will be affected. Therefore, the user is prompted to resolve any connection conflicts manually. In this case, the cathode of R_1 and the anode of R_2 need to be assigned nodal connections. If one inputs “2” and “3” respectively, it will result in the case below.

```
R = Resistor('Rx',2,-1,2000);
circuit.add(R)
circuit.list

ans =
    'Vin 1 0 DC 5
     R1 1 2 1000
     R2 2 0 3000
     Rx 2 3 2000'
```


Nodal ambiguity
Re-assign R1's cathode:
2
Nodal ambiguity
Re-assign R2's anode:

As with the add function, the removal functions – shorting and opening – are also available to the user directly. Say, we are dealing with a op-amp circuit, such as this. We may be interested in its transfer function from the source to the output of the op-amp, so we run the transmuter from circuit to symbolic.

```
circuit = Circuit('circuits/rc_op_amp.txt');
circuit.list

ans =
    'Vs 1 0 AC Vs
     R1 1 2 20000
     R2 3 4 20000
     C1 2 3 C1
     C2 3 4 C2
     O1 0 3 4'
```



```
ELAB.ec2sd(circuit,1,4)

ans =
```

$$\frac{v_4}{v_1} = -\frac{C_1 R_2 s}{(C_1 R_1 s + 1)\,(C_2 R_2 s + 1)}$$

We immediately see two poles at $s = -1/\,(R_1 C_1)$ and $s = -1/\,(R_2 C_2)$. Suppose, we want to know what happens, if we short the first capacitor C_1, or if we open-circuited the second capacitor C_2.

```
circuit.short(circuit.Capacitors(1));
ELAB.ec2sd(circuit,1,3)
ans =
```

$$\frac{v_3}{v_1} = -\frac{R_2}{R_1\,(C_2 R_2 s + 1)}$$

```
circuit.open(circuit.Capacitors(2));
ELAB.ec2sd(circuit,1,4)
ans =
```

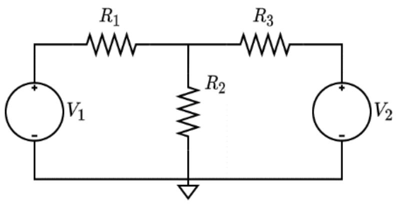
$$\frac{v_4}{v_1} = -\frac{C_1 R_2 s}{C_1 R_1 s + 1}$$

The transfer functions now look like this. Note, that the nodes have been appropriately relabeled by the clean function automatically and therefore the op-amp output node after shorting is node “3” in the first case. We see that shorting C_1 will remove the pole at $s = -1/\,(R_1 C_1)$ as expected, and that opening C_2 will remove the pole at $s = -1/\,(R_2 C_2)$.

6.1.3 Thevenin/Norton Equivalents

Finding Thevenin or Norton Equivalent circuits is often useful for circuit analysis, so of course there exists a high-level function for just this. To illustrate the use of these functions, we load a circuit and decide to find the Thevenin-equivalent. In this case, let us see what the equivalent circuit looks like, if we view R_2 as the load impedance. Again, the choice of load is arbitrary, and the function will work equally well with any other choice of load element in any other applicable circuit.

```
circuit = Circuit('circuits/th_no_equivalents.txt');
circuit.list

ans =
    'V1 1 0 DC 28
     V2 3 0 DC 7
     R1 1 2 4
     R2 2 0 2
     R3 2 3 1'
```

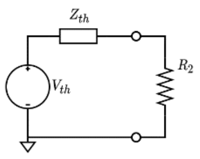


The Thevenin and Norton functions leverages lower-level functions like clone to make a copy of the load element, open to open-circuit the load, simplify to reduce to a single impedance, and then finally the add function to place the clone of the load back into the circuit.

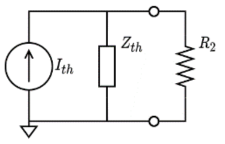
```
circuit = Circuit('circuits/th_no_equivalents_sym.txt');
ELAB.thevenin(circuit, circuit.Resistors(2));
circuit.list

ans =
    'Vth 1 0 DC (R1*V2+R3*V1)/(R1+R3)
     Zth 1 2 (R1*R3)/(R1+R3)
     R2 2 0 R2'
```

Due to them leveraging the simplify function, both functions are of course also capable of handling symbolic netlists - pure or mixed. Here, the same circuit, but without numerical values, is loaded in.

```
ELAB.thevenin(circuit, circuit.Resistors(2));
circuit.list

ans =
    'Vth 1 0 DC 56/5
     Zth 1 2 4/5
     R2 2 0 2'
```



```
ELAB.norton(circuit, circuit.Resistors(2));
circuit.list

ans =
    'Ino 1 0 DC 14
     Zno 1 0 4/5
     R2 1 0 2'
```



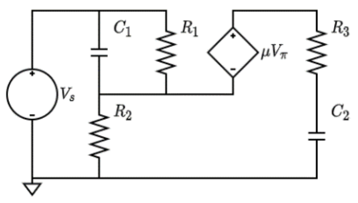
And again, because the simplify function can handle simplifying elements - as if they were generic impedances – down to a single impedance, the Thevenin and Norton functions can handle any type of RLC-circuit as well.

6.1.4 Dependent Sources

Below are arbitrarily constructed circuits wherein each of the dependent-sources are in use. For each circuit, a symbolic expression for the voltage at each node is found and a transfer function is calculated and plotted. These are operations which would require some time to do by hand but may be done very quickly in code. Note, that IDs are used to denote the scaling factor/gain for a dependent-source element when outputting symbolic equations.

Voltage-controlled-voltage-source:

```
circuit = Circuit('circuits/E_src.txt');
circuit.list

ans =
    'V1 1 0 DC 10
     R1 1 2 2000
     R2 2 0 4000
     R3 3 4 20000
     C1 1 2 0.0001
     C2 4 0 0.001
     E1 3 2 1 2 100'
```



```
ELAB.analyze(circuit)
circuit.symbolic_node_voltages
```

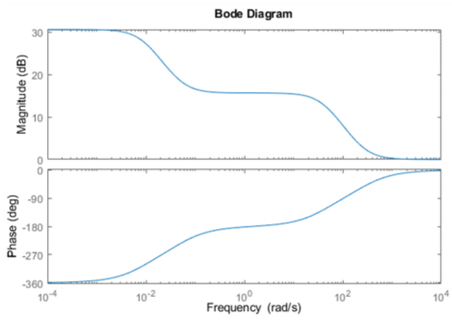
ans =

$$\begin{pmatrix} v_1 = V_1 \\ v_2 = \dfrac{R_2\,V_1\,\left(C_1\,R_1\,s + C_2\,R_3\,s - C_2\,E_1\,R_1\,s + C_1\,C_2\,R_1\,R_3\,s^2 + 1\right)}{\sigma_1} \\ v_3 = \dfrac{V_1\,(C_2\,R_3\,s + 1)\,(R_2 + E_1\,R_1 + C_1\,R_1\,R_2\,s)}{\sigma_1} \\ v_4 = \dfrac{V_1\,(R_2 + E_1\,R_1 + C_1\,R_1\,R_2\,s)}{\sigma_1} \end{pmatrix}$$

where

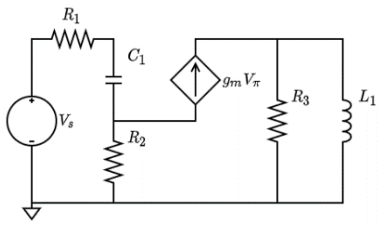
$$\sigma_1 = R_1 + R_2 + C_1\,R_1\,R_2\,s + C_2\,R_1\,R_2\,s + C_2\,R_1\,R_3\,s + C_2\,R_2\,R_3\,s - C_2\,E_1\,R_1\,R_2\,s + C_1\,C_2\,R_1\,R_2\,R_3\,s^2$$

```
TF = ELAB.ec2tf(circuit,1,3);
bode(TF)
```



Voltage-controlled-current-source:

```
circuit = Circuit('circuits/G_src.txt');
circuit.list

ans =
    'V1 1 0 DC 12
     R1 1 2 2000
     R2 3 0 4000
     R3 4 0 8000
     L1 4 0 0.2
     C1 2 3 0.001
     G1 4 3 2 3 50
```



```
ELAB.analyze(circuit)
circuit.symbolic_node_voltages
```

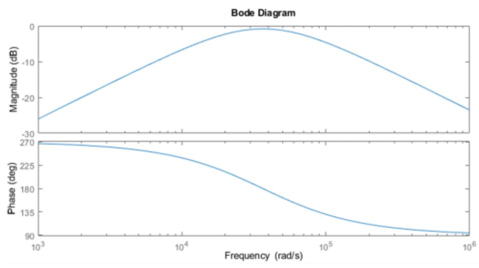
ans =

$$\begin{pmatrix} v_1 = V_1 \\ v_2 = \dfrac{V_1\,(G_1\,R_2 + C_1\,R_2\,s + 1)}{\sigma_1} \\ v_3 = \dfrac{R_2\,V_1\,(G_1 + C_1\,s)}{\sigma_1} \\ v_4 = -\dfrac{G_1\,L_1\,R_3\,V_1\,s}{(R_3 + L_1\,s)\,\sigma_1} \end{pmatrix}$$

where

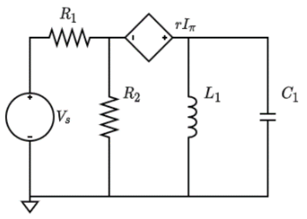
$$\sigma_1 = G_1\,R_2 + C_1\,R_1\,s + C_1\,R_2\,s + 1$$

```
TF = ELAB.ec2tf(circuit,1,4);
bode(TF)
```



Current-controlled-voltage-source:

```
circuit = Circuit('circuits/H_src.txt');
circuit.list

ans =
    'V1 1 0 DC 15
     R1 1 2 2000
     R2 2 0 4000
     L1 3 0 0.02
     C1 3 0 0.0002
     H1 3 2 V1 10'
```



```
ELAB.analyze(circuit)
circuit.symbolic_node_voltages
```

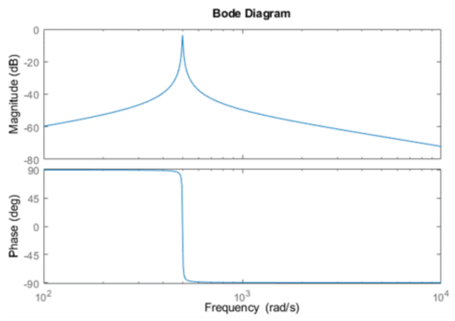
ans =

$$\begin{pmatrix} v_1 = V_1 \\ v_2 = \dfrac{R_2\,V_1\,\left(C_1\,H_1\,L_1\,s^2 + L_1\,s + H_1\right)}{\sigma_1} \\ v_3 = -\dfrac{L_1\,V_1\,s\,\left(H_1 - R_2\right)}{\sigma_1} \end{pmatrix}$$

where

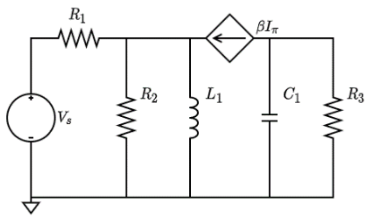
$$\sigma_1 = R_1\,R_2 + H_1\,R_2 + L_1\,R_1\,s + L_1\,R_2\,s + C_1\,H_1\,L_1\,R_2\,s^2 + C_1\,L_1\,R_1\,R_2\,s^2$$

```
TF = ELAB.ec2tf(circuit,1,3);
bode(TF)
```



Current-controlled-current-source:

```
circuit = Circuit('circuits/F_src.txt');
circuit.list

ans =
    'V1 1 0 DC 9
     R1 1 2 2000
     R2 2 0 4000
     R3 3 0 8000
     L1 2 0 0.001
     C1 3 0 0.000001
     F1 3 2 V1 200'
```

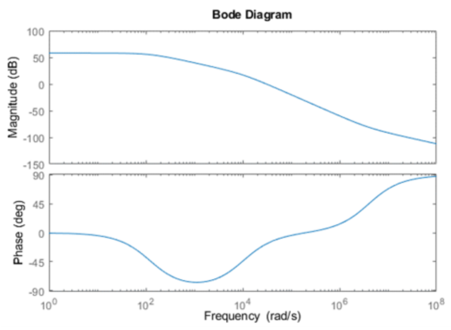


```
ELAB.analyze(circuit)
circuit.symbolic_node_voltages
```

ans =

$$\begin{pmatrix} v_1 = V_1 \\ v_2 = -\dfrac{L_1\,R_2\,V_1\,s\,\left(F_1 - 1\right)}{R_1\,R_2 + L_1\,R_1\,s + L_1\,R_2\,s - F_1\,L_1\,R_2\,s} \\ v_3 = \dfrac{F_1\,R_3\,V_1\,\left(R_2 + L_1\,s\right)}{\left(C_1\,R_3\,s + 1\right)\,\left(R_1\,R_2 + L_1\,R_1\,s + L_1\,R_2\,s - F_1\,L_1\,R_2\,s\right)} \end{pmatrix}$$

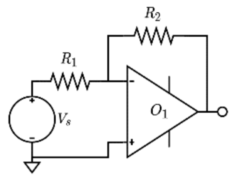
```
TF = ELAB.ec2tf(circuit,1,3);
bode(TF)
```



6.1.5 Operational Amplifiers

To view how the program handles circuit with operational amplifiers, let us first load in a simple inverting amplifier circuit built around an ideal operational amplifier. We then analyze as we would any other circuit.

```
circuit = Circuit('circuits/r_op_amp.txt');
circuit.list

ans =
    'Vs 1 0 AC Vs
     R1 1 2 R1
     R2 2 3 R2
     O1 0 2 3'
```

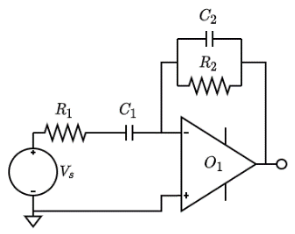


```
ELAB.analyze(circuit)
circuit.symbolic_node_voltages
ans =
```

$$\begin{pmatrix} v_1 = \mathrm{Vs} \\ v_2 = 0 \\ v_3 = -\dfrac{R_2\,\mathrm{Vs}}{R_1} \end{pmatrix}$$

Let us look at a more complex example: We imagine inserting reactive elements, in this case capacitors to end up with the circuit shown loaded here. We could also use the add function to create this circuit from the previous.

```
circuit = Circuit('circuits/rc_op_amp.txt');
circuit.list

ans =
    'Vs 1 0 AC Vs
     R1 1 2 20000
     R2 3 4 20000
     C1 2 3 C1
     C2 3 4 C2
     O1 0 3 4'
```



Suppose we want to know what values for the capacitors, we need to choose, to achieve poles at $p_1 = -1000\ rad/s$ and $p_2 = -5000\ rad/s$.

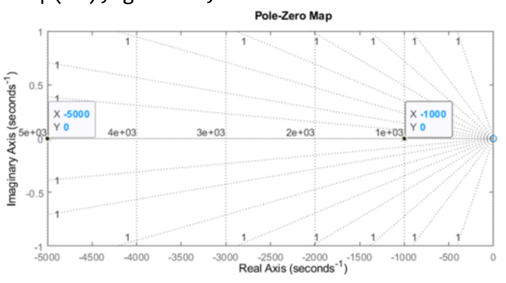
```
ELAB.ec2sd(circuit,1,4)
Symbolic analysis successful (0.381535 sec).
Symbolic transfer function calculated successfully (3.900998e-01 sec).
ans =
```

$$\frac{v_4}{v_1} = -\frac{C_1 R_2 s}{(C_1 R_1 s + 1)\,(C_2 R_2 s + 1)}$$

From this transfer function, it is immediately apparent, that if $R_1 = R_2 = 20\ k\Omega$, then $C_1 = 1/1000R_1 = 0.05\mu F$ and $C_2 = 1/5000R_1 = 0.01\mu F$. Applying these values to their respective element objects and numerically evaluating and creating a transfer-function object, we can produce a pole-zero plot confirming our results.

```
circuit.Capacitors(1).capacitance = 0.05e-6;
circuit.Capacitors(2).capacitance = 0.01e-6;
TF = ELAB.ec2tf(circuit,1,4)
TF =

        -5000 s
   ------------------
   s^2 + 6000 s + 5e06

Continuous-time transfer function.
```

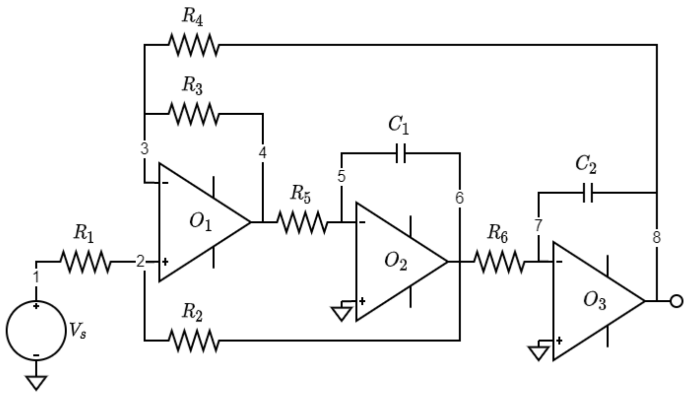
```
pzmap(TF); grid on;
```



The program can handle circuits with an arbitrary number of ideal operational amplifiers, but only if they obey the contraint of each having a negative feedback loop, since this is a requirement for the op-amp being assumed to be ideal.

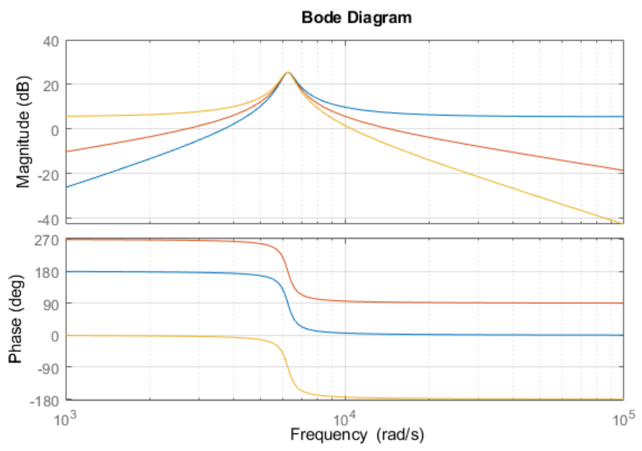
To show that the program is capable of handling more complex op-amp circuitry, we load in a state-variable-filter designed around three ideal operational amplifiers.

```
ELAB.ec2sd(circuit,1,4)
circuit.list

ans =
    'Vs 1 0 AC Vs
     R1 1 2 1000
     R2 2 6 19000
     R3 3 4 10000
     R4 3 8 10000
     R5 4 5 16000
     R6 6 7 16000
     C1 5 6 0.00000001
     C2 7 8 0.00000001
     O1 2 3 4
     O2 0 5 6
     O3 0 7 8'
```



Say we wanted to see how the transfer function looks at each stage of the filter.

```
grid on; hold on;
bode(ELAB.ec2tf(circuit,1,4));
bode(ELAB.ec2tf(circuit,1,6));
bode(ELAB.ec2tf(circuit,1,8));
hold off;
 Symbolic analysis successful (1.08841 sec).

 Numerical evaluation successful (0.245662 sec).
 Transfer function object created successfully (2.703424e-01 sec).
```



As expected with the state-variable-filter, if we take the output at each stage separately, we can simultaneously use this circuit as a low-pass filter after the first stage (in yellow), a band-pass filter after the second stage (in red), and finally as a high-pass filter after the third stage (in blue).

Getting a symbolic transfer function for this circuit would be a tedious task but is done with a single line of code and only takes about 5 milliseconds.

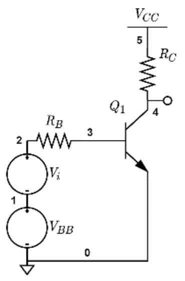
```
ELAB.ec2sd(circuit,1,8)
 Symbolic transfer function calculated successfully (5.072700e-03 sec).
 ans =
```

$$\frac{v_8}{v_1} = \frac{R_2\ (R_3 + R_4)}{R_1\,R_3 + R_2\,R_3 + C_2\,R_1\,R_3\,R_6\,s + C_2\,R_1\,R_4\,R_6\,s + C_1\,C_2\,R_1\,R_4\,R_5\,R_6\,s^2 + C_1\,C_2\,R_2\,R_4\,R_5\,R_6\,s^2}$$

6.1.6 AC/DC-equivalents

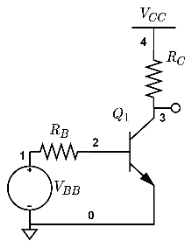
It is often useful when performing circuit analysis to view the circuit as operating in AC and DC separately. This is the case when performing analysis on non-linear circuits, such as a transistor amplifier. We start by loading in a circuit. In this case, it's a simple common-source amplifier with biasing. The circuit is taken from "Microelectronic Circuits", 7th edition, page 410.

```
circuit = Circuit('circuits/bjt_cs_amp.txt');
circuit.list

ans =
    'V_BB 1 0 DC V_BB
     V_CC 5 0 DC V_CC
     V_i 2 1 AC V_i
     R_B 2 3 R_B
     R_C 4 5 R_C
     Q_1 3 4 0 beta_Q_1'
```

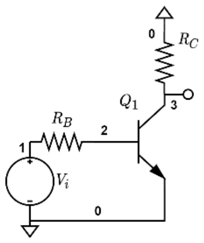


Typically, when analyzing such circuits, we split up the analysis into a DC-part and an AC-part. To find the DC-equivalent of the circuit above, one can simply call the dc_eq function (short for direct-current-equivalent). This function uses lower-level functions such as short, open and clean to modify the given circuit into its DC-equivalent. Similarly, the program can convert the given circuit into its AC-equivalent.

```
ELAB.dc_eq(circuit);
circuit.list

ans =
    'V_BB 1 0 DC V_BB
     V_CC 4 0 DC V_CC
     R_B 1 2 R_B
     R_C 3 4 R_C
     Q_1 2 3 0 beta_Q_1'
```



```
ELAB.ac_eq(circuit);
circuit.list

ans =
    'V_i 1 0 AC V_i
     R_B 1 2 R_B
     R_C 3 0 R_C
     Q_1 2 3 0 beta_Q_1'
```

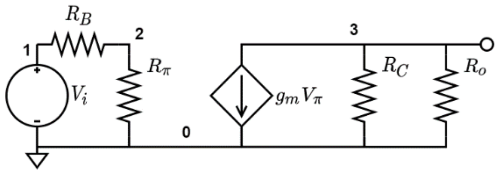


The AC- and DC-equivalent functions work for any type of circuit with any combination of elements in any configuration, because they utilize the previously defined lower-level functions to accomplish their task.

6.1.7 Non-linear Modelling

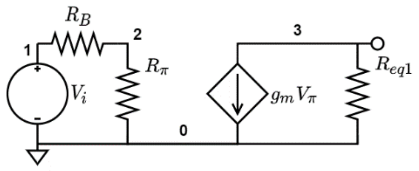
Extending from the previous example, we may use the AC-equivalent circuit to gain further insight. From the AC-equivalent, you may model the transistor as linear elements, using the *hybrid_pi* function, which can either assume the circuit will operate in low-frequency "lf" or high-frequency "hf". We may use the clone function to preserve the AC-equivalent, removing having to load in the circuit again. This is of course up the user. The *hybrid_pi* function will call the AC-equivalent modelling function automatically before proceeding.

```
circuit = Circuit('circuits/bjt_cs_amp.txt');
ELAB.hybrid_pi(circuit,'lf');
lf_circuit.list

ans =
    'V_i 1 0 AC V_i
     R_B 1 2 R_B
     R_C 3 0 R_C
     R_pi_Q_1 2 0 R_pi_Q_1
     R_o_Q_1 3 0 R_o_Q_1
     G_Q_1 3 0 0 2 G_Q_1
```



Of course, we may also simplify this circuit. The simplifier takes into account that a dependent source relies on the voltage across R_π and so does not simplify it, despite it being in series with R_B.

```
ELAB.simplify(circuit);
circuit.list

ans =
    'V_i 1 0 AC V_i
     R_B 1 2 R_B
     R_pi_Q_1 2 0 R_pi_Q_1
     R_eq1 3 0 (R_C*R_o_Q_1)/(R_C+R_o_Q_1)
     G_Q_1 3 0 0 2 G_Q_1
```



Now, it is simply a circuit containing basic linear elements and it can therefore be analyzed by the analyzer function, much like it is shown in the section on dependent sources.

```
ELAB.analyze(circuit)
circuit.symbolic_node_voltages
```

ans =

$$\begin{pmatrix} v_1 = V_i \\ v_2 = \dfrac{R_{\pi,Q,1}\,V_i}{R_B + R_{\pi,Q,1}} \\ v_3 = \dfrac{G_{Q,1}\,R_{\text{eq1}}\,R_{\pi,Q,1}\,V_i}{R_B + R_{\pi,Q,1}} \end{pmatrix}$$

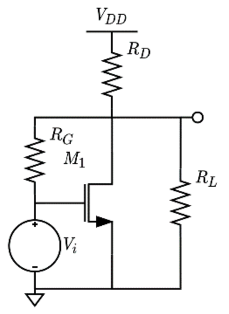
```
ELAB.ec2sd(circuit,1,3)
```

ans =

$$\frac{v_3}{v_1} = \frac{G_{Q,1}\,R_{\text{eq1}}\,R_{\pi,Q,1}}{R_B + R_{\pi,Q,1}}$$

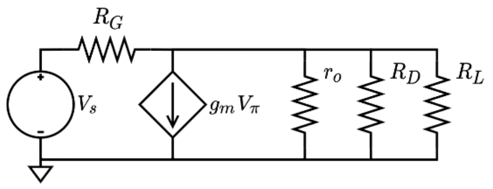
Of course, all of the functionalities shown on BJT's also works for circuits with MOSFETs or mix of the two. Let us load in a similar circuit, but this time with a MOSFET.

```
circuit = Circuit('circuits/mos_cs_amp.txt');
circuit.list

ans =
    'V_i 1 0 AC V_i
     V_DD 3 0 DC V_DD
     R_D 2 3 R_D
     R_G 1 2 R_G
     R_L 2 0 R_L
     M_1 1 2 0 100'
```



We skip the AC- and DC-equivalents this time and go straight to calling the hybrid_pi function.

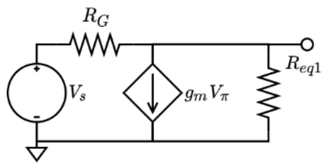
```
ELAB.hybrid_pi(circuit,'lf');
circuit.list

ans =
    'V_i 1 0 AC V_i
     R_D 2 0 R_D
     R_G 1 2 R_G
     R_L 2 0 R_L
     R_o_M_1 2 0 R_o_M_1
     G_M_1 2 0 0 1 G_M_1'
```



Once again, we simplify and decide to find the transfer function from the input to the load element.

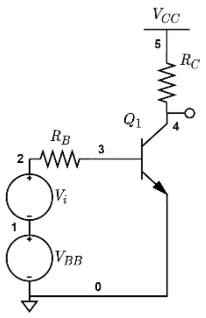
```
ELAB.simplify(circuit);
circuit.list

ans =
    'V_i 1 0 AC V_i
     R_G 1 2 R_G
     R_eq1 2 0 (R_D*R_L*R_o_M_1)/(R_D*R_L+R_D*R_o_M_1+R_L*R_o_M_1)
     G_M_1 2 0 0 1 G_M_1'

ELAB.analyze(circuit)
ELAB.ec2sd(circuit,1,2)

ans =
```

$$\frac{v_2}{v_1} = \frac{R_{\text{eq1}}\ (G_{M,1}\,R_G + 1)}{R_G + R_{\text{eq1}}}$$



What if we want to evaluate such circuits numerically? If there are numerical values available, the program is also able to incorporate these into the modelling of the transistor. We reload the common-source amplifier with biasing, but this time with numerical values.

```
circuit = Circuit('circuits/bjt_cs_amp_num.txt');
circuit.list

ans =
    'V_BB 1 0 DC 3
     V_CC 5 0 DC 10
     V_i 2 1 AC V_i
     R_B 2 3 100000
     R_C 4 5 3000
     Q_1 3 4 0 100'
```



We may call the function called *biasing*, which automatically finds the DC-equivalent and performs calculations to find the biasing current. This also happens to work symbolically.

```
circuit = ELAB.biasing(circuit);
 Symbolic analysis successful (0.280537 sec).

 Numerical evaluation successful (0.0484789 sec).
```

$I_{V,\mathrm{BE},Q,1} = 2.3\text{e-}5$

In this case, the biasing/base current is $I_B = 23\ \mu A$, so the collector current is $I_C = \beta I_B = 2.3\ mA$, because the beta parameter of the transistor was set to 100 in the netlist. After calculation, these values are stored in their corresponding transistors, and can be utilized by the hybrid-pi function. This time, we neglect the early effect.

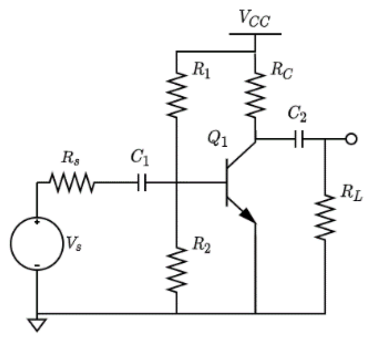
```
ELAB.hybrid_pi(circuit,'lf', false);
circuit.list

ans =
    'V_i 1 0 AC V_i
     R_B 1 2 100000
     R_C 3 0 3000
     R_pi_Q_1 2 0 1130.4347826086958178137709958459
     G_Q_1 3 0 2 0 0.088461538461538448576407565561762'
```

As can be seen from the netlist, the elements used for the hybrid-pi model has been given numerical values. The voltage-controlled-current-source transconductance is $88.5\ mA/V$, and the resistance over which the controlling voltage is, has been calculated to be $R_{pi} = 1130\Omega$. We can now treat is as any other linear circuit, like finding the symbolic transfer function, or evaluating the transfer function to check the gain.

```
ELAB.ec2tf(circuit,1,3)


 Numerical evaluation successful (0.0740902 sec).
 Transfer function object created successfully (1.042769e-01 sec).

 ans =

   -2.966

 Static gain.
```

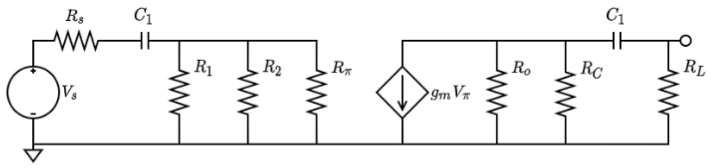
Here is a last example of how the program can be used to model non-linear circuits. We load in a circuit that happen to be a common-emitter, single-source-biased BJT-amplifier with coupling capacitors.

```
circuit = Circuit('circuits/bjt_complex_amp.txt');
circuit.list

ans =
    'V_cc 4 0 DC V_cc
     V_s 1 0 AC V_s
     R_s 1 2 R_s
     R_1 3 4 R_1
     R_2 3 0 R_2
     R_C 4 5 R_C
     R_L 6 0 R_L
     C_1 2 3 C_1
     C_2 5 6 C_2
     Q_1 3 5 0 beta_Q_1'
```



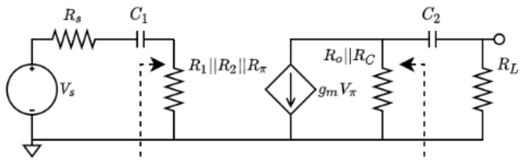
Say we want to know the input- and output-resistances for the transistor, assuming the coupling capacitors to act as shorts for all frequencies of interest. We model the circuit using the hybrid-pi model.

```
ELAB.hybrid_pi(circuit, 'lf', true);
circuit.list

ans =
    'V_s 1 0 AC V_s
     R_s 1 2 R_s
     R_1 3 0 R_1
     R_2 3 0 R_2
     R_C 0 4 R_C
     R_L 5 0 R_L
     R_pi_Q_1 3 0 R_pi_Q_1
     R_o_Q_1 4 0 R_o_Q_1
     C_1 2 3 C_1
     C_2 4 5 C_2
     G_Q_1 4 0 3 0 G_Q_1'
```



And simplify to get the circuit from which we can derive our desired quantities.

```
ELAB.simplify(circuit);
circuit.list

ans =
    'V_s 1 0 AC V_s
     R_s 1 2 R_s
     R_L 5 0 R_L
     R_eq1 0 4 (R_C*R_o_Q_1)/(R_C+R_o_Q_1)
     R_eq2 3 0 (R_1*R_2*R_pi_Q_1)/(R_1*R_2+R_1*R_pi_Q_1+R_2*R_pi_Q_1)
     C_1 2 3 C_1
     C_2 4 5 C_2
     G_Q_1 4 0 3 0 G_Q_1'
```



```
circuit.Resistors(3).resistance
```

```
ans =
```

$$\frac{R_C\, R_{o,Q,1}}{R_C + R_{o,Q,1}}$$

```
circuit.Resistors(4).resistance
```

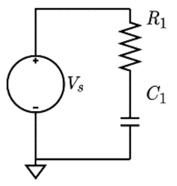
```
ans =
```

$$\frac{R_1\, R_2\, R_{\pi,Q,1}}{R_1\, R_2 + R_1\, R_{\pi,Q,1} + R_2\, R_{\pi,Q,1}}$$

6.1.8 Time-domain Analysis

Say, the user wants not only to examine circuits in the s-domain, but also in the more intuitive time-domain, as is so often the case. ELABorate supports this, and this example applies to all other examples in this report. Let's start by loading in a simple 1st-order-series-rc-circuit, otherwise known as a low-pass filter, when treating the voltage across the capacitor as the output. This has an AC voltage-source with the numerical value defined as a function of time.

```
circuit = Circuit('circuits/ac_source.txt');
circuit.list

ans =
    'Vs 1 0 AC 10-10*exp(-5000*t)
     R1 1 2 10000
     C1 2 0 0.00000001'
```



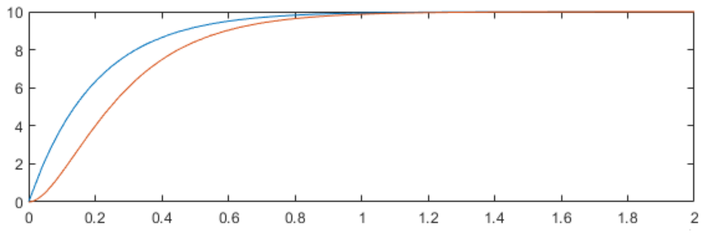
We analyze the circuit as usual to obtain the circuit equations. If evaluate detects that the circuit has not been analyzed, it will call the analyze function automatically. In this case, we would like to know the voltage across the capacitor. Evaluate has saved the results in the circuit object. We transmute the result into the time-domain. We only need the right-hand-side of the equation.

```
ELAB.evaluate(circuit)
sd = circuit.numerical_element_voltages(2)
td = ELAB.sd2td(rhs(sd))
```

td = $10\,\mathrm{e}^{-10000\,t} - 20\,\mathrm{e}^{-5000\,t} + 10$

We plot the input voltage (blue) and the voltage across the capacitor (orange) over time.

```
fplot(circuit.Indep_VSs(1).voltage); hold on;
fplot(td); hold off;
```



6.1.9 Cloning and Exporting

Cloning a circuit is a straight-forward way of preserving the original circuit by manipulating a copy of the circuit. We load an arbitrary circuit, clone it and then print the netlist of the clone. As is seen in the second box, the original circuit is preserved. This functionality simply exists for the program to adhere to best practices of the object-oriented methodology. It saves the program the computational effort of creating a circuit model multiple times in the event that the user may want to try different things on the same circuit.

```
c1 = Circuit('circuits/voltage_divider.txt');
c2 = c1.clone;
ELAB.simplify(c2);
c2.list

ans =
    'Vin 1 0 DC 5
     Req1 1 0 4000'
```

```
c1.list

ans =
    'Vin 1 0 DC 5
     R1 1 2 1000
     R2 2 0 3000'
```

Cloning also applies to individual elements. MATLAB usually treats objects via references, so making a standalone copy is not as simple as setting R2 = R1.

7 Conclusion

At the start of this project, the goal was to support as many types of circuit elements as possible, as is partly evident by the project specification, but I ended up shifting priority to the core of the program, which is the lower-level functionality that actually allows for programmatic manipulation and symbolic analysis of circuits.

I wish for the program, or toolbox as it is referred to by MATLAB, to be a complete replacement for pen-and-paper analysis, and so it must encompass as many elements as possible. In any case, it is not possible for me to envision and implement all the variations of circuit elements there exist today, so I will let that be up to anyone using the toolbox. This is also why I have stressed the importance of the object-oriented methodology, and it is also the reason why the source code is fully available on GitHub for anyone that may be interested in extending the program.

The other overarching goal was to find out if it was feasible to combine the advantages of symbolic analysis with the computational power of a computer, and have it be useful. I started writing the code for this project during my first class on electrical circuits at the start of my bachelor's degree in 2019 out of sheer frustration that such a thing did not already exist at a commercial level. Since then, I have shelved the project many times due to time-constraints, but after having put the toolbox on GitHub, a handful of people across the electrical engineering community has shown interest and appreciation, so there must be some benefit to be gained from this approach to circuit analysis – This approach being *programmatic symbolic circuit analysis* – hence the title.

As the intention for the project is to encourage development of this approach, I will be discussing how the project is structured and what functionality I would like to see implemented in the future.

7.1 Project Organization

The structure of the finished project is as shown in the figure to the right. It is split into code and examples which showcase the capabilities of the code. The layout encourages further contribution from third parties. If one wishes to add their own class of element, they simply create a sub-directory in the appropriate location, and have the new class inherit from one of the other classes. Say, a user wanted to implement a variable resistor. They would simply have to create a "resistors" directory within the "impedance" folder and place the new class "Variable_resistor.m" or maybe named "Trimmer.m" within the new directory. The custom class must implement any abstract properties or methods for it to be a valid class. If one wishes to share their custom element classes, they may submit a pull-request on the project's GitHub page.

```
ELABorate/
├── code/
│   ├── elements/
│   │   ├── amplifiers/
│   │   │   ├── Ideal_OpAmp.m
│   │   │   └── ...
│   │   ├── dep_sources/
│   │   │   ├── dep_cc_sources/
│   │   │   │   ├── CCCS.m
│   │   │   │   └── CCVS.m
│   │   │   ├── dep_vc_sources/
│   │   │   │   ├── VCCS.m
│   │   │   │   └── VCVS.m
│   │   │   ├── Dep_CC.m
│   │   │   └── Dep_VC.m
│   │   ├── impedances/
│   │   │   ├── Capacitor.m
│   │   │   ├── Inductor.m
│   │   │   └── Resistor.m
│   │   ├── indep_sources/
│   │   │   ├── Indep_IS.m
│   │   │   └── Indep_VS.m
│   │   ├── transistors/
│   │   │   ├── BJT.m
│   │   │   └── MOSFET.m
│   │   ├── Amplifier.m
│   │   ├── Dep_S.m
│   │   ├── Element.m
│   │   ├── Impedance.m
│   │   ├── Indep_S.m
│   │   ├── Transistor.m
│   │   └── ...
│   ├── engine/
│   │   ├── Analyzer.m
│   │   ├── Modeller.m
│   │   ├── Transmuter.m
│   │   └── Visualizer.m
│   ├── notes/
│   │   └── ...
│   ├── signals/
│   │   └── ...
│   ├── systems/
│   │   ├── Base_System.m
│   │   ├── Circuit.m
│   │   ├── ODE.m
│   │   └── ...
│   └── ELAB.m
├── examples/
│   ├── circuits/
│   ├── pdfs/
│   └── scripts/
├── gui/
│   └── ...
├── resource/
│   └── ...
├── readme.md
└── plan.md
```

7.2 Unreleased functionality

In the overview to the side, any "…" denote additional unfinished classes not included in this report. The "gui" directory contains efforts to create a graphical user interface for a standalone export of the program. Some of the core functionality of the program has been connected to this GUI but has not been finished. The reason being that the symbolic math toolbox, which this program leverages, is not yet supported by MATLAB's compiler, meaning it is not yet possible to create a fully standalone version of the program.

7.3 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.

8 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

9 Appendices

The appendices contain the entire codebase of the project, except for the code that constitutes unfinished features and element classes.

9.1 Appendix I – Systems

This appendix contains the code for the systems classes of this project. The reader is encouraged to inspect the code on GitHub instead of in this report, as it is much easier to read and understand when formatted appropriately and contained within separate files.

9.1.1 Base_System.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef (Abstract) Base_System < handle % & matlab.mixin.Heterogeneous
% The basis of all ELABorate's system classes.

    properties
        % Classifications.
        id; order; type;

        % Representations.
        s_domain; t_domain; transfer; state_space;
        diff_equation; zero_pole_gain; char_equation;

        % List of assumption about the system to decrease
        % complexity of the calculations.
        Assumptions;
    end
end
```

9.1.2 Circuit.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Circuit < Base_System
% Circuit model, utilizing the Element subclasses.

    properties
    % Information about this circuit object, available to the user.

        % Globals.
        file_name;  num_nodes;  num_elements; list;

        % Arrays of elements.
        Indep_VSs = Indep_VS.empty;    Indep_ISs = Indep_IS.empty;
        Resistors = Resistor.empty;    Inductors = Inductor.empty;
        Capacitors = Capacitor.empty;  Ideal_OpAmps = Ideal_OpAmp.empty;
        Generic_zs = Impedance.empty;

        VCVSs = VCVS.empty;            CCCSs = CCCS.empty;
        VCCSs = VCCS.empty;            CCVSs = CCVS.empty;
        BJTs = BJT.empty;              MOSFETs = MOSFET.empty;

        % Results.
        equations;
        symbolic_transfer_function;    numerical_transfer_function;
        symbolic_node_voltages;        numerical_node_voltages;
        symbolic_element_voltages;     numerical_element_voltages;
        symbolic_element_currents;     numerical_element_currents;
        symbolic_source_currents;      numerical_source_currents;
    end

    properties (Access = {?ELAB, ?Analyzer, ?Modeller, ?Transmuter, ?Visualizer})
    % Information only available to ELABorate modules.

        % Flags.
        symbolically_analyzed = false;     numerically_analyzed = false;
        symbolic_transfer_found = false;   numerical_transfer_found = false;

        % Lengths of arrays.
        num_impedances;  num_transistors;  num_VSs;        num_ISs;
        num_Indep_VSs;   num_Indep_ISs;    num_resistors;  num_inductors;
        num_capacitors;  num_VCVSs;        num_VCCSs;      num_CCVSs;
        num_CCCSs;       num_op_amps;      num_BJTs;       num_MOSFETs;
        num_Indep_Ss;    num_Dep_Ss;
        num_generic_zs;

        % Combined element arrays.
        Elements = Element.empty;          Indep_Ss = Indep_S.empty;
        Transistors = Transistor.empty;    Dep_Ss = Dep_S.empty;
        Impedances = Impedance.empty;

        % For calculations.
        of_interest; expressions; netlist;
    end

    methods
    % Methods pertaining to this circuit.

        function obj = Circuit(file_name)
        % Constructor of the circuit class.

            % Parse the text file at the given path (file_name).
            obj.file_name = file_name;
            file = fopen(file_name);

            % Convert to cell matrix, based on line number and spaces.
            netlist = textscan(file,'%s %s %s %s %s %s %s %s', 'CollectOutput', 1);
            obj.netlist = {netlist{1}(:,1) netlist{1}(:,2) netlist{1}(:,3) ...
                           netlist{1}(:,4) netlist{1}(:,5) netlist{1}(:,6) ...
                           netlist{1}(:,7) netlist{1}(:,8)};
            fclose(file);

            % Split cell matrix into vectors.
            [Name, N1, N2, arg3, arg4, arg5, arg6, arg7] = obj.netlist{:};

            % Convert node values from string-type to number-type.
            N1 = str2double(N1); N2 = str2double(N2);

            % Find number of elements by number of entries in the vectors.
            obj.num_elements = length(Name);
            obj.num_nodes = length(unique([N1; N2])) - 1;

            % Creating element objects based the parsed netlist information.
            for i = 1:obj.num_elements

                % Element id is always in the same place.
                id = Name{i}(1:end);

                % First letter in ID tells the type of element.
                switch id(1)
                    case {'V'}
                        obj.Indep_VSs(end+1) = Indep_VS(id, N1(i), N2(i), arg3{i}, arg4{i});
                    case {'I'}
                        obj.Indep_ISs(end+1) = Indep_IS(id, N1(i), N2(i), arg3{i}, arg4{i});
                    case {'R'}
                        obj.Resistors(end+1) = Resistor(id, N1(i), N2(i), arg3{i});
                    case {'L'}
                        obj.Inductors(end+1) = Inductor(id, N1(i), N2(i), arg3{i});
                    case {'C'}
                        obj.Capacitors(end+1) = Capacitor(id, N1(i), N2(i), arg3{i});
                    case {'Z'}
                        obj.Impedances(end+1) = Impedance(id, N1(i), N2(i), arg3{i});
                    case {'E'}
                        obj.VCVSs(end+1) = VCVS(id, N1(i), N2(i), str2double(arg3{i}), ...
                                                str2double(arg4{i}), arg5{i});
                    case {'G'}
                        obj.VCCSs(end+1) = VCCS(id, N1(i), N2(i), str2double(arg3{i}), ...
                                                str2double(arg4{i}), arg5{i});
                    case {'H'}
                        obj.CCVSs(end+1) = CCVS(id, N1(i), N2(i), arg3{i}, arg4{i});
                    case {'F'}
                        obj.CCCSs(end+1) = CCCS(id, N1(i), N2(i), arg3{i}, arg4{i});
                    case {'O'}
                        obj.Ideal_OpAmps(end+1) = Ideal_OpAmp(id, N1(i), N2(i), ...
                                                              str2double(arg3{i}));
                    case {'Q'}
                        obj.BJTs(end+1) = BJT(id, N1(i), N2(i), str2double(arg3{i}), ...
                                              arg4{i}, arg5{i}, arg6{i}, arg7{i});
                    case {'M'}
                        obj.MOSFETs(end+1) = MOSFET(id, N1(i), N2(i), str2double(arg3{i}), ...
                                                    arg4{i}, arg5{i}, arg6{i}, arg7{i});
                end
            end

            % This creates compound element arrays and handles setup.
            obj.update();
        end

        function export(obj, name)
        % Create a circuit file from the circuit object
        % netlist in the current working directory.

            file = fopen(name,'w');
            fprintf(file,'%s %s %s %s %s %s %s %s', obj.list);
            fclose(file);
        end

        function copy = clone(obj)
        % Creates a standalone object, that is a clone of the given circuit.
        % i.e. not a reference to the original object.

            obj.export('tmp.txt');
            copy = Circuit('tmp.txt');
            delete('tmp.txt');
        end

        function status(obj)
        % Prints the status of the circuit object.

            fprintf('<strong>%Status:</strong>\n', obj.id);
            fprintf('- Sym. analyzed: %s\n', mat2str(obj.symbolically_analyzed));
            fprintf('- Num. analyzed: %s\n', mat2str(obj.numerically_analyzed));
            fprintf('- Sym. transfer function found: %s\n', ...
                mat2str(obj.symbolic_transfer_found));
            fprintf('- Num. transfer function found: %s\n\n', ...
                mat2str(obj.numerical_transfer_found));
        end

        function update_netlist(obj)
        % Returns a netlist-string which describes this circuit object.

            L = [];
            for index = 1:obj.num_elements
```

```
                L = [L, obj.Elements(index).to_net];
            end

            obj.list = L;
            obj.netlist = textscan(L,'%s %s %s %s %s %s %s %s');
        end

        function short(obj, X)
        % Update circuit object, after effectively shorting element X.

            % After shorting, use the node with the lowest number.
            new_node = min(X.terminals);

            for index = 1:X.num_terminals
                node = X.terminals(index);

                Ys = get_connected(obj, X, node);
                for index_1 = 1:length(Ys)
                    Y = Ys{index_1};

                    for index_2 = 1:Y.num_terminals
                        if Y.terminals(index_2) == node
                            Y.update_terminals(index_2, new_node);
                        end
                    end
                end
            end

            remove(obj, X);
        end

        function open(obj, X)
        % Update circuit object, after effectively open-circuiting element X.

            remove(obj, X);
            trim(obj);
        end

        function clean(obj)
        % Update node numbering to avoid skips in indexing.

            for k = 0:obj.num_nodes
                % Make list of all terminals.
                nodes = [];
                for index = 1:obj.num_elements
                    X = obj.Elements(index);
                    nodes = [nodes, X.terminals];
                end
                % Find smallest number above k
                m = min(nodes(nodes > k));

                % Replace any terminal value matching m with k+1.
                for index = 1:obj.num_elements
                    X = obj.Elements(index);
                    for t = 1:X.num_terminals
                        if X.terminals(t) == m
                            X.update_terminals(t, k+1);
                        end
                    end
                end
            end
            obj.update;
        end

        function add(obj, X)
        % Add element to circuit and update appropriately.

```

```
            if X.num_terminals == 2

                % If it is necessary to create a new node.
                if X.anode == -1 || X.cathode == -1
                    new_node = obj.num_nodes + 1;

                    if X.anode == -1
                        node = X.cathode;
                        X.anode = new_node;
                        X.update_terminals(1,new_node);
                    elseif X.cathode == -1
                        node = X.anode;
                        X.cathode = new_node;
                        X.update_terminals(2,new_node);
                    end

                    connected = get_connected(obj, X, node);

                    for index = 1:length(connected)
                        Y = connected{index};
                        if Y.anode == node
                            prompt = sprintf('Re-assign %s''s anode:', Y.id);
                            new_connection = inputdlg({prompt},'Nodal ambiguity',[1,50]);
                            Y.anode = str2double(new_connection);
                        elseif Y.cathode == node
                            prompt = sprintf('Re-assign %s''s cathode:', Y.id);
                            new_connection = inputdlg({prompt},'Nodal ambiguity',[1,50]);
                            Y.cathode = str2double(new_connection);
                        end
                    end
                end
            end

            switch X.id(1)
                case {'V'}
                    obj.Indep_VSs(end+1) = X;
                case {'I'}
                    obj.Indep_ISs(end+1) = X;
                case {'R'}
                    obj.Resistors(end+1) = X;
                case {'L'}
                    obj.Inductors(end+1) = X;
                case {'C'}
                    obj.Capacitors(end+1) = X;
                case {'Z'}
                    obj.Generic_zs(end+1) = X;
            end

            obj.update;
            obj.reset;
        end
    end

    methods(Access = {?ELAB, ?Analyzer, ?Modeller, ?Transmuter, ?Visualizer})
    % Methods only available to ELABorate modules.

        function connected = get_connected(obj, X, node)
        % Get all the elements connected to given element X at given node.

            connected = [];
            connected = [connected, Circuit.check_elem_array(obj.Indep_VSs, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.Indep_ISs, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.Resistors, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.Inductors, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.Capacitors, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.VCVSs, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.VCCSs, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.CCVSs, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.CCCSs, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.Ideal_OpAmps, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.BJTs, X, node)];
            connected = [connected, Circuit.check_elem_array(obj.MOSFETs, X, node)];
        end

        function remove(obj, X)
        % Remove X from circuit's element-arrays, without updating connections.

            if     isa(X, 'Resistor'),  Xs = obj.Resistors;  obj.Resistors  = Xs(Xs ~= X);
            elseif isa(X, 'Inductor'),  Xs = obj.Inductors;  obj.Inductors  = Xs(Xs ~= X);
            elseif isa(X, 'Capacitor'), Xs = obj.Capacitors; obj.Capacitors = Xs(Xs ~= X);
            elseif isa(X, 'Impedance'), Xs = obj.Generic_zs; obj.Generic_zs = Xs(Xs ~= X);
            elseif isa(X, 'Indep_VS'), Xs = obj.Indep_VSs; obj.Indep_VSs = Xs(Xs ~= X);
            elseif isa(X, 'Indep_IS'), Xs = obj.Indep_ISs; obj.Indep_ISs = Xs(Xs ~= X);
            elseif isa(X, 'VCVS'), Xs = obj.VCVSs; obj.VCVSs = Xs(Xs ~= X);
            elseif isa(X, 'VCCS'), Xs = obj.VCCSs; obj.VCCSs = Xs(Xs ~= X);
            elseif isa(X, 'CCVS'), Xs = obj.CCVSs; obj.CCVSs = Xs(Xs ~= X);
            elseif isa(X, 'CCCS'), Xs = obj.CCCSs; obj.CCCSs = Xs(Xs ~= X);
            elseif isa(X, 'Ideal_OpAmp'), Xs = obj.Ideal_OpAmps; obj.Ideal_OpAmps = Xs(Xs ~= X);
            elseif isa(X, 'BJT'), Xs = obj.BJTs; obj.BJTs = Xs(Xs ~= X);
            elseif isa(X, 'MOSFET'), Xs = obj.MOSFETs; obj.MOSFETs = Xs(Xs ~= X);
            end

            delete(X);
            obj.update;
            obj.clean;
            obj.reset;
        end

        function update(obj)
        % Updates all circuit properties.

            obj.update_nums;       obj.update_arrays;
            obj.update_num_nodes;  obj.update_netlist;
        end

        function reset(obj)
        % Reset flags and arrays, for example when the structure of the
        % circuit is altered.

            obj.symbolic_node_voltages = [];      obj.symbolic_source_currents = [];
            obj.symbolic_element_voltages = [];   obj.symbolic_element_currents = [];
            obj.symbolic_transfer_function = [];  obj.numerical_node_voltages = [];
            obj.numerical_source_currents = [];   obj.numerical_transfer_function = [];
            obj.numerical_element_voltages = [];  obj.numerical_element_currents = [];
            obj.equations = [];

            obj.symbolically_analyzed = false;    obj.numerically_analyzed = false;
            obj.symbolic_transfer_found = false;  obj.numerical_transfer_found = false;
        end

        function trim(obj)
        % Remove elements not affecting circuit.

            to_be_removed = [];

            % Find anything not connected to anything else.
            for index = 1:obj.num_elements
                X = obj.Elements(index);
                sum = 0;

                for index_1 = 1:X.num_terminals
                    node = X.terminals(index_1);

                    if node ~= 0
                    sum = sum + length(get_connected(obj, X, node));
                end
            end

            if sum == 0
                to_be_removed = [to_be_removed, X];
            end
        end

        % Find anything connected only to itself.
        for index = 1:obj.num_elements
            found = false;
            X = obj.Elements(index);

            for index_1 = 1:X.num_terminals
                for index_2 = (1 + index_1):X.num_terminals
                    if X.terminals(index_1) ~= X.terminals(index_2)
                        found = true; break;
                    end
                end
            end

            if ~found
                disp(['Removing ', X.id, ' (self connected).']);
                to_be_removed = [to_be_removed, X];
            end
        end

        % Remove found elements.
        for index = 1:length(to_be_removed)
            remove(obj, to_be_removed(index));
        end
    end
end

methods(Access = private)
% Methods only available to this circuit object.

    function update_num_nodes(obj)
    % Check and update the number of nodes in the circuit.

        n = [];
        for index = 1:obj.num_elements
            X = obj.Elements(index);
            n = [n, X.terminals];
        end
        % Find how many unique nodes, minus ground.
        obj.num_nodes = length(unique(n)) - 1;
    end

    function update_nums(obj)
    % Check and update the number of each element in the circuit.

        % Elements
        obj.num_Indep_VSs = length(obj.Indep_VSs);   obj.num_Indep_ISs = length(obj.Indep_ISs);
        obj.num_resistors = length(obj.Resistors);   obj.num_capacitors = length(obj.Capacitors);
        obj.num_inductors = length(obj.Inductors);   obj.num_generic_zs = length(obj.Generic_zs);
        obj.num_VCVSs = length(obj.VCVSs);           obj.num_VCCSs = length(obj.VCCSs);
        obj.num_CCVSs = length(obj.CCVSs);           obj.num_CCCSs = length(obj.CCCSs);
        obj.num_BJTs = length(obj.BJTs);             obj.num_MOSFETs = length(obj.MOSFETs);
        obj.num_op_amps = length(obj.Ideal_OpAmps);

        % Super-classes
        obj.num_Indep_Ss = obj.num_Indep_VSs + obj.num_Indep_ISs;
        obj.num_Dep_Ss = obj.num_VCVSs + obj.num_CCVSs + obj.num_VCCSs + obj.num_CCCSs;
        obj.num_VSs = obj.num_Indep_VSs + obj.num_op_amps + obj.num_VCVSs + obj.num_CCVSs;
        obj.num_ISs = obj.num_Indep_ISs + obj.num_VCCSs + obj.num_CCCSs;
```

```
            obj.num_transistors = obj.num_BJTs + obj.num_MOSFETs;
            obj.num_impedances = obj.num_generic_zs + obj.num_resistors ...
                               + obj.num_inductors + obj.num_capacitors;

            % All
            obj.num_elements = obj.num_VSs + obj.num_ISs + obj.num_impedances + obj.num_transistors;
        end

        function update_arrays(obj)
        % Update heterogeneous arrays.

            obj.Indep_Ss = [obj.Indep_VSs, obj.Indep_ISs];
            obj.Impedances = [obj.Generic_zs, obj.Resistors, obj.Inductors, obj.Capacitors];
            obj.Dep_Ss = [obj.VCVSs, obj.VCCSs, obj.CCVSs, obj.CCCSs];
            obj.Transistors = [obj.BJTs, obj.MOSFETs];
            obj.Elements = [obj.Indep_Ss, obj.Impedances, obj.Dep_Ss, ...
                            obj.Ideal_OpAmps, obj.Transistors];
        end
    end

    methods(Static)
    % Methods shared among objects of this class.

        function connected = check_elem_array(Xs, Y, node)
        % Return all elements in given array, connected to the given node.

            connected = cell.empty;
            for index_1 = 1:length(Xs)
                X = Xs(index_1);
                if X ~= Y
                    bools = X.is_connected(node);
                    for index_2 = 1:length(bools)
                        if bools(index_2)
                            connected{end+1} = X;
                        end
                    end
                end
            end
        end
    end
end
```

9.1.3 ODE.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef ODE
% Simply a container to make transmuting cleaner.

    properties
        eq;
        cond;
    end

    methods
        function obj = ODE(eq, cond)
            obj.eq = eq;
            obj.cond = cond;
        end
    end
end
```

9.2 Appendix II – Engine

This appendix contains the core code of the program, which I have called the engine. The reader is encouraged to inspect the code on GitHub instead of in this report, as it is much easier to read and understand when formatted appropriately and contained within separate files.

9.2.1 ELAB.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef ELAB < Analyzer & Modeller & Visualizer & Transmuter
% The combination of the main classes of the ELABorate project.

    properties
        notes;
    end

    methods(Static)

        function help()
            fprintf('See README.md and Manual.md');
        end

        function credits()
            fprintf('Built by Nicklas Vraa');
        end
    end
end
```

9.2.2 Analyzer.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Analyzer
% A collection of functions designed to facilitate symbolic
% and numerical analysis of a given circuit object.

    methods(Static)

        function analyze(obj)
        % Symbolic analysis of given circuit object.

            if obj.symbolically_analyzed
                return
            end

            tic;

            % Array allocation.
            G = cell(obj.num_nodes, obj.num_nodes);  [G{:}] = deal('0');
            B = cell(obj.num_nodes, obj.num_VSs);    [B{:}] = deal('0');
            C = cell(obj.num_VSs,   obj.num_nodes);  [C{:}] = deal('0');
            D = cell(obj.num_VSs,   obj.num_VSs);    [D{:}] = deal('0');
            i = cell(obj.num_nodes, 1);              [i{:}] = deal('0');
            e = cell(obj.num_VSs,   1);              [e{:}] = deal('0');
            j = cell(obj.num_VSs,   1);              [j{:}] = deal('0');
            v = compose('v_%d', (1:obj.num_nodes)');

            % Building circuit equations:
            num_vs_parsed = 0;

            % Impedances (R, L, C).
            for index = 1:obj.num_impedances
```

```
                X = obj.Impedances(index);
                if     isa(X, 'Resistor'), g = ['1/' X.id];
                elseif isa(X, 'Inductor'), g = ['1/s/' X.id];
                elseif isa(X, 'Capacitor'), g = ['s*' X.id];
                elseif isa(X, 'Impedance'), g = ['1/' X.id];
                end

                if X.anode == 0
                    G{X.cathode, X.cathode} = sprintf('%s + %s', G{X.cathode, X.cathode}, g);
                elseif X.cathode == 0
                    G{X.anode,   X.anode}   = sprintf('%s + %s', G{X.anode,   X.anode}, g);
                else
                    G{X.anode,   X.anode}   = sprintf('%s + %s', G{X.anode,   X.anode}, g);
                    G{X.cathode, X.cathode} = sprintf('%s + %s', G{X.cathode, X.cathode}, g);
                    G{X.anode,   X.cathode} = sprintf('%s - %s', G{X.anode,   X.cathode}, g);
                    G{X.cathode, X.anode}   = sprintf('%s - %s', G{X.cathode, X.anode}, g);
                end
            end

            % Independent voltage sources (V).
            for index = 1:obj.num_Indep_VSs
                V = obj.Indep_VSs(index);
                num_vs_parsed = num_vs_parsed + 1;

                if V.anode ~= 0
                    B{V.anode, num_vs_parsed} = [B{V.anode, num_vs_parsed} ' + 1'];
                    C{num_vs_parsed, V.anode} = [C{num_vs_parsed, V.anode} ' + 1'];
                end
                if V.cathode ~= 0
                    B{V.cathode, num_vs_parsed} = [B{V.cathode, num_vs_parsed} ' - 1'];
                    C{num_vs_parsed, V.cathode} = [C{num_vs_parsed, V.cathode} ' - 1'];
                end
                e{num_vs_parsed} = V.id;
                j{num_vs_parsed} = ['I_' V.id];
            end

            % Independent current sources (I).
            for index = 1:obj.num_Indep_ISs
                I = obj.Indep_ISs(index);

                if I.anode ~= 0
                    i{I.anode} = [i{I.anode} ' - ' I.id];
                end
                if I.cathode ~= 0
                    i{I.cathode} = [i{I.cathode} ' + ' I.id];
                end
            end

            % Op-amps (O).
            for index = 1:obj.num_op_amps
                O = obj.Ideal_OpAmps(index);
                num_vs_parsed = num_vs_parsed + 1;

                B{O.output, num_vs_parsed} = [B{O.output, num_vs_parsed} ' + 1'];

                if O.input_1 ~= 0
                    C{num_vs_parsed, O.input_1} = [C{num_vs_parsed, O.input_1} ' + 1'];
                end
                if O.input_2 ~= 0
                    C{num_vs_parsed, O.input_2} = [C{num_vs_parsed, O.input_2} ' - 1'];
                end
                j{num_vs_parsed} = ['I_' O.id];
            end

            % Voltage Controlled Voltage Sources, VCVS (E).
            for index = 1:obj.num_VCVSs
                VCVS = obj.VCVSs(index);
                num_vs_parsed = num_vs_parsed + 1;
                ctrl_node_1 = VCVS.ctrl_anode;
                ctrl_node_2 = VCVS.ctrl_cathode;

                if VCVS.anode ~= 0
                    B{VCVS.anode, num_vs_parsed} = [B{VCVS.anode, num_vs_parsed} ' + 1'];
                    C{num_vs_parsed, VCVS.anode} = [C{num_vs_parsed, VCVS.anode} ' + 1'];
                end
                if VCVS.cathode ~= 0
                    B{VCVS.cathode, num_vs_parsed} = [B{VCVS.cathode, num_vs_parsed} ' - 1'];
                    C{num_vs_parsed, VCVS.cathode} = [C{num_vs_parsed, VCVS.cathode} ' - 1'];
                end
                if ctrl_node_1 ~= 0
                    C{num_vs_parsed, ctrl_node_1} = [C{num_vs_parsed, ctrl_node_1} ' - ' VCVS.id];
                end
                if ctrl_node_2 ~= 0
                    C{num_vs_parsed, ctrl_node_2} = [C{num_vs_parsed, ctrl_node_2} ' + ' VCVS.id];
                end

                j{num_vs_parsed} = ['I_' VCVS.id];
            end

            % Voltage Controlled Current Sources, VCCS (G).
            for index = 1:obj.num_VCCSs
                VCCS = obj.VCCSs(index);

                pattern = num2str([VCCS.anode ~= 0, VCCS.cathode ~= 0, ...
                    VCCS.ctrl_anode ~= 0, VCCS.ctrl_cathode ~= 0]);
                pattern = pattern(~isspace(pattern));

                switch pattern
                    case {'0000', '0001',  '0010', '0011', '0100', '1000', '1100'}
                        error('Invalid VCCS configuration.');
                    case '1111' % If nothing is grounded.
                        G{VCCS.anode,   VCCS.ctrl_anode} ...
                            = [G{VCCS.anode,   VCCS.ctrl_anode}   ' + ' VCCS.id];
                        G{VCCS.anode,   VCCS.ctrl_cathode} ...
                            = [G{VCCS.anode,   VCCS.ctrl_cathode} ' - ' VCCS.id];
                        G{VCCS.cathode, VCCS.ctrl_anode} ...
                            = [G{VCCS.cathode, VCCS.ctrl_anode}   ' - ' VCCS.id];
                        G{VCCS.cathode, VCCS.ctrl_cathode} ...
                            = [G{VCCS.cathode, VCCS.ctrl_cathode} ' + ' VCCS.id];
                    case '0111' % If only anode is grounded.
                        G{VCCS.cathode, VCCS.ctrl_anode} ...
                            = [G{VCCS.cathode, VCCS.ctrl_anode}   ' - ' VCCS.id];
                        G{VCCS.cathode, VCCS.ctrl_cathode} ...
                            = [G{VCCS.cathode, VCCS.ctrl_cathode} ' + ' VCCS.id];
                    case '0101' % If only anodes and control anode are grounded.
                        G{VCCS.cathode, VCCS.ctrl_cathode} ...
                            = [G{VCCS.cathode, VCCS.ctrl_cathode} ' + ' VCCS.id];
                    case '0110' % If only anode and control cathode are grounded.
                        G{VCCS.cathode, VCCS.ctrl_anode} ...
                            = [G{VCCS.cathode, VCCS.ctrl_anode}   ' - ' VCCS.id];
                    case '1011' % If only cathode is grounded.
                        G{VCCS.anode,   VCCS.ctrl_anode} ...
                            = [G{VCCS.anode,   VCCS.ctrl_anode}   ' + ' VCCS.id];
                        G{VCCS.anode,   VCCS.ctrl_cathode} ...
                            = [G{VCCS.anode,   VCCS.ctrl_cathode} ' - ' VCCS.id];
                    case '1001' % If only cathode and control anode are grounded.
                        G{VCCS.anode,   VCCS.ctrl_cathode} ...
                            = [G{VCCS.anode,   VCCS.ctrl_cathode} ' - ' VCCS.id];
                    case '1010' % If only both cathodes are grounded.
                        G{VCCS.anode,   VCCS.ctrl_anode} ...
                            = [G{VCCS.anode,   VCCS.ctrl_anode}   ' + ' VCCS.id];
                    case '1101' % If only control anode is grounded.
                        G{VCCS.anode,   VCCS.ctrl_cathode} ...
                            = [G{VCCS.anode,   VCCS.ctrl_cathode} ' - ' VCCS.id];
                        G{VCCS.cathode, VCCS.ctrl_cathode} ...
                            = [G{VCCS.cathode, VCCS.ctrl_cathode} ' + ' VCCS.id];
                    case '1110' % If only control cathode is grounded.
                        G{VCCS.anode,   VCCS.ctrl_anode} ...
                            = [G{VCCS.cathode, VCCS.ctrl_anode}   ' + ' VCCS.id];
                        G{VCCS.cathode, VCCS.ctrl_anode} ...
                            = [G{VCCS.cathode, VCCS.ctrl_anode}   ' - ' VCCS.id];
                end
            end

            % Current Controlled Voltage Sources, CCVS (H).
            for index = 1:obj.num_CCVSs
                CCVS = obj.CCVSs(index);
                num_vs_parsed = num_vs_parsed + 1;

                if CCVS.anode ~= 0
                    B{CCVS.anode, num_vs_parsed} = [B{CCVS.anode, num_vs_parsed} ' + 1'];
                    C{num_vs_parsed, CCVS.anode} = [C{num_vs_parsed, CCVS.anode} ' + 1'];
                end
                if CCVS.cathode ~= 0
                    B{CCVS.cathode, num_vs_parsed} = [B{CCVS.cathode, num_vs_parsed} ' - 1'];
                    C{num_vs_parsed, CCVS.cathode} = [C{num_vs_parsed, CCVS.cathode} ' - 1'];
                end

                j{num_vs_parsed} = ['I_' CCVS.id]; % Edit this to fit the others.

                % Add -CCVS_id to matrix D.
                ctrl_voltage = CCVS.ctrl_anode;
                ctrl_voltage_index = find(contains(j, ctrl_voltage));
                h_index = find(contains(j, CCVS.id));
                D{h_index, ctrl_voltage_index} = ['-' CCVS.id];
            end

            % Current Controlled Current Sources, CCCS (F).
            for index = 1:obj.num_CCCSs
                CCCS = obj.CCCSs(index);

                ctrl_voltage = obj.CCCSs(index).ctrl_anode;
                ctrl_voltage_index = find(contains(j, ctrl_voltage));

                if (CCCS.anode ~= 0)
                    B{CCCS.anode, ctrl_voltage_index} ...
                        = [B{CCCS.anode, ctrl_voltage_index} ' + ' CCCS.id];
                end
                if (CCCS.cathode ~= 0)
                    B{CCCS.cathode, ctrl_voltage_index} ...
                        = [B{CCCS.cathode, ctrl_voltage_index} ' - ' CCCS.id];
                end
            end

            % Matrix formation.
            A = str2sym([G B; C D]); x = str2sym([v;j]); z = str2sym([i;e]);
            syms([symvar(A), symvar(x), symvar(z)]);

            % Solving circuit equations.
            obj.equations = A * x == z;
            a = simplify(A \ z);
            obj.of_interest = x;
            obj.expressions = a;
            solutions = x == eval(a);
            obj.symbolic_node_voltages = solutions(1:obj.num_nodes,:);
            obj.symbolic_source_currents = solutions(obj.num_nodes + 1:end,:);
            obj.symbolically_analyzed = true;
            Analyzer.find_v_and_i(obj);
            fprintf('Symbolic analysis successful (%g sec).\n', toc);
        end

        function evaluate(obj)
        % Analyze circuit, using numerical values, if any exist.

            obj.numerical_element_voltages = [];
            obj.numerical_element_currents = [];

            Analyzer.analyze(obj)

            if obj.numerically_analyzed
                return
            end

            tic; syms('s');
            [Name, ~, ~, arg3, arg4, arg5] = obj.netlist{:};

            for index = 1:obj.num_elements
                switch Name{index}(1)
                    case {'V', 'I'}
                        if isempty(arg4{index})
                            syms(Name{index});
                        else
                            [num, status] = str2num(arg4{index});
                            if status
                                eval(sprintf('%s = %g;', Name{index}, num));
                            else
                                if strcmp(arg3{index}, 'AC')
                                    syms t;
                                    exp = laplace(str2sym(arg4{index}));
                                    eval(sprintf('%s = %s;', Name{index}, exp));
                                else
                                    syms(Name{index});
                                end
                            end
                        end
                    case {'R', 'L', 'C'}
                        if isempty(arg3{index})
                            syms(Name{index});
                        else
                            [num, status] = str2num(arg3{index});
                            if status
                                eval(sprintf('%s = %g;', Name{index}, num));
                            else
                                syms(Name{index});
                            end
                        end
                    case {'H', 'F'}
                        if isempty(arg4{index})
                            syms(Name{index});
                        else
                            [num, status] = str2num(arg4{index});
                            if status
                                eval(sprintf('%s = %g;', Name{index}, num));
                            else
                                syms(Name{index});
                            end
                        end
                    case {'E', 'G'}
                        if isempty(arg5{index})
                            syms(Name{index});
                        else
                            [num, status] = str2num(arg5{index});
                            if status
                                eval(sprintf('%s = %g;', Name{index}, num));
                            else
                                syms(Name{index});
                            end
                        end
                end
            end

            for index = 1:length(obj.of_interest)
                eval(sprintf('%s = %s;',obj.of_interest(index), obj.expressions(index)));
            end

            solutions = obj.of_interest == eval(obj.of_interest);
            obj.numerical_node_voltages = solutions(1:obj.num_nodes,:);
            obj.numerical_source_currents = solutions(obj.num_nodes + 1:end,:);

            obj.numerical_element_voltages = ...
                lhs(obj.symbolic_element_voltages) == eval(rhs(obj.symbolic_element_voltages));
            obj.numerical_element_currents = ...
                lhs(obj.symbolic_element_currents) == eval(rhs(obj.symbolic_element_currents));

            obj.numerically_analyzed = true;
            fprintf('\nNumerical evaluation successful (%g sec).\n', toc);
        end

        function out = ec2sd(obj, node_in, node_out)
        % Returns symbolic transfer function of circuit, given input and output node.

            tic;
            Analyzer.analyze(obj);

            v_in = obj.symbolic_node_voltages(node_in);
            v_out = obj.symbolic_node_voltages(node_out);
            out = collect(v_out/v_in);
            obj.symbolic_transfer_function = out;
            obj.symbolic_transfer_found = true;
            fprintf('Symbolic transfer function calculated successfully (%s sec).\n', toc);
        end

        function out = ec2tf(obj, node_in, node_out)
        % Returns transfer function object of circuit, given input and output node.

            tic;
            Analyzer.evaluate(obj);

            v_in = obj.numerical_node_voltages(node_in);
            v_out = obj.numerical_node_voltages(node_out);
            H = collect(v_out/v_in);
            [numerator, denominator] = numden(rhs(H));
            out = minreal(tf(sym2poly(numerator), sym2poly(denominator)));
            obj.numerical_transfer_function = out;
            obj.numerical_transfer_found = true;
            fprintf('Transfer function object created successfully (%s sec).\n', toc);
        end

        function out = apply(TF, input)
        % Apply a signal to a circuit transfer function.

            out_s = TF * ELAB.transmute(input, 'td', 'tf');
            syms t;
            out = ELAB.transmute(out_s, 'tf', 'td') * heaviside(t);
        end

        function RA = routh(poly_coeffs, show, epsilon)
        % Takes the coefficients of a characteristic equation of a system
        % and returns the Routh array for stability analysis.
            if nargin < 2; show = false; end
            if nargin < 3; syms epsilon; end

            dim = size(poly_coeffs);              % Get size of poly_coeffs.
            coeff = dim(2);                       % Get number of coefficients.
            RA = sym(zeros(coeff,ceil(coeff/2))); % Initialize symbolic Routh array.

            for i = 1:coeff
                RA(2-rem(i,2),ceil(i/2)) = poly_coeffs(i); % Assemble 1st and 2nd rows.
            end

            rows = coeff-2;                       % Number of rows that need determinants.
            index = zeros(rows,1);                % Inits columns-per-row index vector.

            for i = 1:rows
                index(rows-i+1) = ceil(i/2);      % Form index vector from bottom to top.
            end

            for i = 3:coeff                       % Go from 3rd row to last.
                if all(RA(i-1,:) == 0)            % Row of zeros.
                    disp('Special Case: Row of zeros.');
                    a = coeff-i+2;                % Order of auxiliary equation.
                    b = ceil(a/2)-rem(a,2)+1;     % Number of auxiliary coeffs.
                    tmp1 = RA(i-2,1:b);           % Get auxiliary polynomial.
                    tmp2 = a:-2:0;                % Auxiliry polynomial powers.
                    RA(i-1,1:b) = tmp1.*tmp2;            % Derivative of auxiliary.
                elseif RA(i-1,1) == 0             % First element in row is zero.
                    disp('Special Case: First element is zero.');
                    RA(i-1,1) = epsilon;          % Replace by epsilon.
                end

                for j = 1:index(i-2) % Compute the Routh array elements.
                    RA(i,j) = -det([RA(i-2,1) RA(i-2,j+1); RA(i-1,1) RA(i-1,j+1)])/RA(i-1,1);
                end
            end

            RA = simplify(RA);
            if show; disp('Routh Array'); disp(RA); end
        end

        function cr = critical(RA, show)
        % Takes Routh-array and finds interval of symbolic, where system is stable.

            if nargin < 2; show = false; end

            try
                % Solve for K.
                roi = size(RA,1); % Row of interest.
                eq1 = RA(roi - 1) == 0; eq2 = RA(roi) == 0;
                k1 = solve(eq1); k2 = solve(eq2);
                K = [k1, k2];

                % Solve for s, given K.
                syms s;
                eq3 = RA(roi - 2)*s^2 + K == 0;

                S = [];
                for i = 1:length(eq3)
                    S = [S, solve(eq3(i))];
                end

                if show
                    disp('Solving equations for param:');
                    disp(eq1); disp(eq2);
                    fprintf('For stability, param (K) = ]%d;%d[\n\n', K(1), K(2));
                    disp('From s^2 auxiliary equation:'); disp(eq3);
                    disp('Solved for s:'); disp(S);
                end
            catch
                disp('No critical points. Parameter does not affect stability.');
                cr = 0;
            end
        end
```

```
        function bp = breakaway(sys, gain, show)
        % Returns breakaway points for root locus.

            if nargin < 3; show = false; end

            eq1 = solve(sys, gain);
            eq2 = collect(eq1);
            eq3 = diff(eq2);
            eq4 = solve(eq3 == 0);
            s = double(eq4);
            K = double(subs(eq2));
            bp = [s K];

            if show
                disp('Solving closed-loop for gain:'); disp(gain == eq1);
                disp('Expanding expression:'); disp(gain == eq2);
                disp('Derivative of gain of s:'); disp(eq3);
                disp('Solving diff(gain) = 0 for s:'); disp('s' == eq4');
                disp('Sub s in 1st equation, solving for gain.');
                disp('Breakaway points [s, gain]:');
                disp(bp);
            end
        end

        function [ord, type] = order_type(G, show)
        % Returns the order and type of the given system.

            if nargin < 2; show = false; end

            ord = order(G);
            type = sum(pole(G) == 0);

            if show
                fprintf('Order: %d\n', ord);
                fprintf('Type : %d\n', type);
            end
        end

        function [Kp, Kv, Ka] = static_error_K(G, show)
        % Returns static position error, static velocity error and static
        % acceleration error and steady-state error of the given system.

            if nargin < 2; show = false; end

            g = ELAB.tf2sd(G); syms s;
            Kp = double(limit(g, s, 0, 'right'));
            s = tf('s'); g = ELAB.tf2sd(s*G); syms s;
            Kv = double(limit(g, s, 0, 'right'));
            s = tf('s'); g = ELAB.tf2sd(s^2*G); syms s;
            Ka = double(limit(g, s, 0, 'right'));

            if show
                disp('Static error constants:');
                disp('  Position:');
                fprintf('    As s -> 0, G(s) -> K_p = %.2f\n', Kp);
                disp('  Velocity:');
                fprintf('    As s -> 0, s*G(s) -> K_v = %.2f\n', Kv);
                disp('  Acceleration:');
                fprintf('    As s -> 0, s^2*G(s) -> K_a = %.2f\n', Ka);
                disp('Steady-state error e_ss:');

                [~, type] = ELAB.order_type(G);
                switch type
                    case 0; disp('  Step: 1/(1+K)  Ramp: Inf  Accel: Inf');
                    case 1; disp('  Step: 0  Ramp: 1/K  Accel: Inf');
                    case 2; disp('  Step: 0  Ramp: 0  Accel: 1/K');
                end
            end
        end

        function dp = dominant(G)
        % Returns dominant poles, i.e. closest to imaginary axis.

            poles = pole(G);
            dom = max(real(nonzeros(poles)));
            dp = poles(real(poles) == dom);
        end

        function [wn, zeta] = damp(G, show)
        % Returns the natural frequency and damping ratio of a given system.

            if nargin < 2; show = false; end

            p = ELAB.dominant(G);
            wn = abs(p(1));
            zeta = -cos(angle(p(1)));

            if show
                fprintf('\nNatural frequency  = |%f%+fj|\n', real(p(1)), imag(p(1)));
                fprintf('Damping ratio zeta = -cos(angle(%f%+fj))\n', real(p(1)), imag(p(1)));
            end
        end

        function N = period(x)
        % Find fundamental period of function.
            try
                n = 0:1:100;
                x = double(subs(x));
                N = seqperiod(x);
                if N == length(n)
                    disp('Function is aperiodic.'); N = 0;
                end
            catch
                disp('Function is aperiodic.'); N = 0;
            end
        end

        function out = observable(in)
            out = canon(in, 'companion');
        end

        function out = controllable(in)
            out = ELAB.observable(in).';
        end

        function out = jordan(in)
            out = canon(in, 'modal');
        end
    end

    methods(Static, Access = private)

        function find_v_and_i(obj)
         % Find voltages and currents in single array of specific element-type.

            obj.symbolic_element_voltages = sym.empty;
            obj.symbolic_element_currents = sym.empty;

            for index = 1:obj.num_impedances
                X = obj.Impedances(index); from_node = 0; to_node = 0;
                if X.anode > 0, from_node = obj.expressions(X.anode); end
                if X.cathode > 0, to_node = obj.expressions(X.cathode); end
                X.v_across = simplify(from_node - to_node);
                obj.symbolic_element_voltages(end+1) = sym(sprintf('v_%s', X.id)) == X.v_across;
                X.i_through = simplify(X.v_across / X.id);
                obj.symbolic_element_currents(end+1) = sym(sprintf('i_%s', X.id)) == X.i_through;
            end

            obj.symbolic_element_voltages = obj.symbolic_element_voltages(:);
            obj.symbolic_element_currents = obj.symbolic_element_currents(:);
        end
    end
end
```

9.2.3 Modeller.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Modeller
% A collection of functions, specifically for safely altering the
% circuit object or even converting the circuit to another form.

    methods(Static)

        function obj = simplify(obj, z_eq)
        % Simplifies the given circuit object to equivalent elements,
        % when pairs of equal class in series or parallels are found.
        % If z_eq is true, any element with the superclass 'impedance'
        % will be reduced to a generic impedance object.

            if nargin < 2; z_eq = false; end

            obj.reset;

            while true
                series_done = Modeller.simplify_series(obj, z_eq);
                parallel_done = Modeller.simplify_parallel(obj, z_eq);
                if series_done && parallel_done
                    break;
                end
            end

            Modeller.rename(obj);
            obj.update;
            obj.reset;
        end

        function obj = dc_eq(obj)
        % Short-circuit AC voltage sources and inductors.
        % Open-circuit AC current sources and capacitors.

            to_be_shorted = [];  to_be_opened = [];

            % Find everything that needs to be shorted
            % or opened, and store it in arrays.
            for index = 1:obj.num_Indep_VSs
                if obj.Indep_VSs(index).is_AC
                    to_be_shorted = [to_be_shorted, obj.Indep_VSs(index)];
                end
            end

            for index = 1:obj.num_Indep_ISs
                if obj.Indep_ISs(index).is_AC
                    to_be_opened = [to_be_opened, obj.Indep_ISs(index)];
                end
            end

            for index = 1:obj.num_inductors
```

```
                to_be_shorted = [to_be_shorted, obj.Inductors(index)];
            end

            for index = 1:obj.num_capacitors
                to_be_opened = [to_be_opened, obj.Capacitors(index)];
            end

            % Carry out the shorting and opening.
            for index = 1:length(to_be_shorted)
                obj.short(to_be_shorted(index));
            end

            for index = 1:length(to_be_opened)
                obj.open(to_be_opened(index));
            end

            obj.trim; % Remove redundancy.
        end

        function obj = ac_eq(obj)
        % Short-circuit DC voltage sources.
        % Open-circuit DC current sources.

            to_be_shorted = [];  to_be_opened = [];

            % Find everything that needs to be shorted
            % or opened, and store it in arrays.
            for index = 1:obj.num_Indep_VSs
                if ~obj.Indep_VSs(index).is_AC
                    to_be_shorted = [to_be_shorted, obj.Indep_VSs(index)];
                end
            end

            for index = 1:obj.num_Indep_ISs
                if ~obj.Indep_ISs(index).is_AC
                    to_be_opened = [to_be_opened, obj.Indep_ISs(index)];
                end
            end

            % Carry out the shorting and opening.
            for index = 1:length(to_be_shorted)
                obj.short(to_be_shorted(index));
            end

            for index = 1:length(to_be_opened)
                obj.open(to_be_opened(index));
            end

            obj.trim; % Remove redundancy.
        end

        function obj = remove_sources(obj)
        % Short voltage-sources and open current-sources.

            to_be_shorted = [];  to_be_opened = [];

            for index = 1:obj.num_Indep_VSs
                to_be_shorted = [to_be_shorted, obj.Indep_VSs(index)];
            end

            for index = 1:obj.num_VCVSs
                to_be_shorted = [to_be_shorted, obj.VCVSs(index)];
            end

            for index = 1:obj.num_CCVSs
                to_be_shorted = [to_be_shorted, obj.CCVSs(index)];
            end

            for index = 1:obj.num_Indep_ISs
                to_be_opened = [to_be_opened, obj.Indep_ISs(index)];
            end

            for index = 1:obj.num_VCCSs
                to_be_opened = [to_be_opened, obj.VCCSs(index)];
                obj.open(obj.VCCSs(index));
            end

            for index = 1:obj.num_CCCSs
                to_be_opened = [to_be_opened, obj.CCCSs(index)];
            end

            for index = 1:length(to_be_shorted)
                obj.short(to_be_shorted(index));
            end

            for index = 1:length(to_be_opened)
                obj.open(to_be_opened(index));
            end

            obj.trim; % Remove redundancy.
        end

        function obj = thevenin(obj, load)
        % Returns the Thevenin-equivalent of the given circuit,
        % as seen by the given load.

            Z_L = load.clone;

            [v_th, Z] = Modeller.equivalent(obj, load);
            z_th = Z.impedance;

            obj.add(Indep_VS('V_th', 1, 0, 'DC', v_th));
            obj.add(Impedance('Z_th', 1, 2, z_th));
            obj.remove(Z);

            Z_L.update_terminals(1, 2);
            Z_L.update_terminals(2, 0);
            obj.add(Z_L); % Add load back.
        end

        function obj = norton(obj, load)
        % Returns the Norton-equivalent of the given circuit,
        % as seen by the given load.

            Z_L = load.clone;

            [v_th, Z] = Modeller.equivalent(obj, load);
            z_th = Z.impedance;
            i_no = v_th / z_th;

            obj.add(Indep_IS('I_no', 1, 0, 'DC', i_no));
            obj.add(Impedance('Z_no', 1, 0, z_th));
            obj.remove(Z);

            Z_L.update_terminals(1, 1);
            Z_L.update_terminals(2, 0);
            obj.add(Z_L); % Add load back.
        end

        function orig = biasing(obj)
        % Determine the biasing current for a transistor.

            orig = obj.clone;
            obj = Modeller.dc_eq(obj);

            % Replace all BJTs with a negative voltage source,
            % representing a voltage drop, and store references.
            V_BEs = Indep_VS.empty;
            for index = 1:obj.num_BJTs
                X = obj.BJTs(index);

                id = sprintf('V_BE_%s', X.id);
                V_BE = Indep_VS(id, X.base_node, X.emitter_node, 'DC', X.V_BE);
                V_BEs = [V_BEs, V_BE];
                obj.Indep_VSs(end+1) = V_BE;

                obj.remove(obj.BJTs(index))
            end

            ELAB.evaluate(obj)

            % Display all currents through the new sources.
            biasing_currents = obj.numerical_source_currents(end-length(V_BEs)+1:end);
            disp(vpa(biasing_currents, 2));

            for index = 1:length(V_BEs)
                X = orig.BJTs(index);
                X.biasing = rhs(biasing_currents(index));
            end
        end

        function obj = hybrid_pi(obj, freq, early)
        % Replace all transistors with their low/high-frequency-hybrid-pi model.

            % Short DC-VS's and open DC-CS's.
            obj = Modeller.ac_eq(obj);

            % Replace MOSFETS.
            for index = 1:obj.num_MOSFETs
                M = obj.MOSFETs(index);

                % Add new elements.
                id = sprintf('R_o_%s', M.id);
                R_o = Resistor(id, M.drain_node, M.source_node, id);
                obj.Resistors(end+1) = R_o;

                id = sprintf('G_%s', M.id);
                G = VCCS(id, M.drain_node, M.source_node, M.source_node, M.gate_node, id);
                obj.VCCSs(end+1) = G;

                if freq == "hf" % High frequency model.
                    id = sprintf('C_pi_%s', M.id);
                    C_pi = Capacitor(id, M.gate_node, M.source_node, id);
                    obj.Capacitors(end+1) = C_pi;

                    id = sprintf('C_mu_%s', M.id);
                    C_mu = Capacitor(id, M.gate_node, M.drain_node, id);
                    obj.Capacitors(end+1) = C_mu;
                end

                obj.remove(M); obj.update;
            end

            for index = 1:obj.num_BJTs
                Q = obj.BJTs(index);

                % Create id's for new elements.
                G_id = sprintf('G_%s', Q.id);
                R_pi_id = sprintf('R_pi_%s', Q.id);

                % Get biasing information, if available.
                if ~isempty(Q.biasing)
                    I_C = Q.beta * Q.biasing;
                    gm = vpa(I_C / Q.V_T);
                    r_pi = vpa(Q.beta / gm);
                else
                    gm = G_id;
                    r_pi = R_pi_id;
                end

                % Add new elements.
                G = VCCS(G_id, Q.collector_node, Q.emitter_node, Q.base_node, Q.emitter_node, gm);
                obj.VCCSs(end+1) = G;

                R_pi = Resistor(R_pi_id, Q.base_node, Q.emitter_node, r_pi);
                obj.Resistors(end+1) = R_pi;

                if freq == "hf" % High frequency model.
                    id = sprintf('C_pi_%s', Q.id);
                    C_pi = Capacitor(id, Q.base_node, Q.emitter_node, id);
                    obj.Capacitors(end+1) = C_pi;

                    id = sprintf('R_mu_%s', Q.id);
                    R_mu = Resistor(id, Q.base_node, Q.collector_node, id);
                    obj.Resistors(end+1) = R_mu;

                    id = sprintf('C_mu_%s', Q.id);
                    C_mu = Capacitor(id, Q.base_node, Q.collector_node, id);
                    obj.Capacitors(end+1) = C_mu;
                end

                if early
                    R_o_id = sprintf('R_o_%s', Q.id);
                    R_o = Resistor(R_o_id, Q.collector_node, Q.emitter_node, Q.r_o);
                    obj.Resistors(end+1) = R_o;
                end

                obj.remove(Q); obj.update;
            end
        end

        function bool = is_same_type(X1, X2)
        % Checks if elements, X1 and X2, are of the same class.

            bool = isequal(class(X1), class(X2));
        end

        function bool = is_parallel(X1, X2)
        % Checks if 2-terminal-elements, X1 and X2, are parallel.

            if X1 == X2, error('Comparing an object to itself'); end

            if X1.num_terminals == 2 && X2.num_terminals == 2
                bool = (X1.anode == X2.anode && X1.cathode == X2.cathode)...
                    || (X1.anode == X2.cathode && X1.cathode == X2.anode);
            else
                error('Given elements are not 2-terminal-components');
            end
        end

        function bool = is_series(obj, X1, X2)
        % Checks if 2-terminal-elements, X1 and X2, are parallel.

            bool = true;
            if X1.num_terminals > 2 || X2.num_terminals > 2
                error('Given elements are not 2-terminal-components');
            end

            if Modeller.is_parallel(X1, X2), bool = false; return; end

            if X1.anode == X2.anode || X1.anode == X2.cathode
                shared = X1.anode;
            elseif X1.cathode == X2.anode || X1.cathode == X2.cathode
                shared = X1.cathode;
            else
                bool = false;
                return;
            end

            if  Modeller.check_shared(obj.Indep_VSs, X1, X2, shared) || ...
                Modeller.check_shared(obj.Indep_ISs, X1, X2, shared) || ...
                Modeller.check_shared(obj.Resistors, X1, X2, shared) || ...
                Modeller.check_shared(obj.Inductors, X1, X2, shared) || ...
                Modeller.check_shared(obj.Capacitors, X1, X2, shared) || ...
                Modeller.check_shared(obj.Generic_zs, X1, X2, shared) || ...
                Modeller.check_shared(obj.VCVSs, X1, X2, shared) || ...
                Modeller.check_shared(obj.VCCSs, X1, X2, shared) || ...
                Modeller.check_shared(obj.CCVSs, X1, X2, shared) || ...
                Modeller.check_shared(obj.CCCSs, X1, X2, shared) || ...
                Modeller.check_shared(obj.Ideal_OpAmps, X1, X2, shared) || ...
                Modeller.check_shared(obj.BJTs, X1, X2, shared) || ...
                Modeller.check_shared(obj.MOSFETs, X1, X2, shared)
                bool = false;
            end
        end

    end

    methods(Static, Access = private)

        function done = simplify_parallel(obj, z_eq)
        % Simplifies parallel elements of same type and returns how many pairs were found.

            done = true; % Set to false if any parallels are found.

            R = Modeller.find_parallel(obj, obj.Resistors);
            if ~isempty(R)
                eq = simplify(R(1).resistance * R(2).resistance / (R(1).resistance + R(2).resistance));
                obj.Resistors(end+1) = Resistor('R_eq', R(1).anode, R(1).cathode, eq);
                obj.remove(R(1)); obj.remove(R(2)); done = false;
            end

            L = Modeller.find_parallel(obj, obj.Inductors);
            if ~isempty(L)
                eq = simplify(L(1).inductance * L(2).inductance / (L(1).inductance + L(2).inductance));
                obj.Inductors(end+1) = Inductor('L_eq', L(1).anode, L(1).cathode, eq);
                obj.remove(L(1)); obj.remove(L(2)); done = false;
            end

            C = Modeller.find_parallel(obj, obj.Capacitors);
            if ~isempty(C)
                eq = simplify(C(1).capacitance + C(2).capacitance);
                obj.Capacitors(end+1) = Capacitor('C_eq', C(1).anode, C(1).cathode, eq);
                obj.remove(C(1)); obj.remove(C(2)); done = false;
            end

            V = Modeller.find_parallel(obj, obj.Indep_VSs);
            if ~isempty(V)
                if V(1).voltage == V(2).voltage && V(1).is_AC == V(2).is_AC % Same type.
                    eq = V(1).voltage;
                    if V(1).is_AC, type = 'AC'; else, type = 'DC'; end
                    obj.Indep_VSs(end+1) = Indep_VS('V_eq', V(1).anode, V(1).cathode, type, eq);
                    obj.remove(V(1)); obj.remove(V(2)); done = false;
                end
            end

            I = Modeller.find_parallel(obj, obj.Indep_ISs);
            if ~isempty(I)
                if I(1).is_AC == I(2).is_AC
                    eq = I(1).current + I(2).current;
                    if I(1).is_AC, type = 'AC'; else, type = 'DC'; end
                    obj.Indep_ISs(end+1) = Indep_IS('I_eq', I(1).anode, I(1).cathode, type, eq);
                    obj.remove(I(1)); obj.remove(I(2)); done = false;
                end
            end

            if z_eq
                Z = Modeller.find_parallel(obj, obj.Impedances);
                if ~isempty(Z)
                    %disp(['Parallel found: ', Z(1).id, '||', Z(2).id]);
                    eq = simplify(Z(1).impedance * Z(2).impedance / (Z(1).impedance + Z(2).impedance));
                    obj.Generic_zs(end+1) = Impedance('Z_eq', Z(1).anode, Z(1).cathode, eq);
                    obj.remove(Z(1)); obj.remove(Z(2)); done = false;
                end
            end
        end

        function done = simplify_series(obj, z_eq)
        % Simplifies series elements of same type and returns how many pairs were found.

            done = true; % Set to false if any series are found.

            R = Modeller.find_series(obj, obj.Resistors);
            if ~isempty(R)
                eq = simplify(R(1).resistance + R(2).resistance);
                n = Modeller.unique_nodes(R(1), R(2));
                obj.Resistors(end+1) = Resistor('R_eq', n(2), n(1), eq);
                obj.remove(R(1)); obj.short(R(2)); done = false;
            end

            L = Modeller.find_series(obj, obj.Inductors);
            if ~isempty(L)
                eq = simplify(L(1).inductance + L(2).inductance);
                n = Modeller.unique_nodes(L(1), L(2));
                obj.Inductors(end+1) = Inductor('L_eq', n(2), n(1), eq);
                obj.remove(L(1)); obj.short(L(2)); done = false;
            end

            C = Modeller.find_series(obj, obj.Capacitors);
            if ~isempty(C)
                eq = simplify(C(1).capacitance * C(2).capacitance / ...
                    (C(1).capacitance + C(2).capacitance));
                n = Modeller.unique_nodes(C(1), C(2));
                obj.Capacitors(end+1) = Capacitor('C_eq', n(2), n(1), eq);
                obj.remove(C(1)); obj.short(C(2)); done = false;
            end

            V = Modeller.find_series(obj, obj.Indep_VSs);
            if ~isempty(V)
                if V(1).is_AC == V(2).is_AC
                    eq = V(1).voltage + V(2).voltage;
                    if V(1).is_AC, type = 'AC'; else, type = 'DC'; end
                    n = Modeller.unique_nodes(V(1), V(2));
                    obj.Indep_VSs(end+1) = Indep_VS('V_eq', n(2), n(1), type, eq);
                    obj.remove(V(1)); obj.short(V(2)); done = false;
                end
            end

            if z_eq
                Z = Modeller.find_series(obj, obj.Impedances);
                if ~isempty(Z)
                    %disp(['Series found: ', Z(1).id, '+', Z(2).id]);
                    eq = simplify(Z(1).impedance + Z(2).impedance);
                    n = Modeller.unique_nodes(Z(1), Z(2));
                    obj.Generic_zs(end+1) = Impedance('Z_eq', n(2), n(1), eq);
                    obj.remove(Z(1)); obj.short(Z(2)); done = false;
                end
            end
        end

        function pair = find_parallel(~, Xs)
        % Returns only parallel elements from Xs.

            pair = [];
            for index_1 = 1:length(Xs)
                X1 = Xs(index_1);
                for index_2 = 1 + index_1:length(Xs)
                    X2 = Xs(index_2);
                    if Modeller.is_parallel(X1, X2)
                        pair = [X1, X2];
                        return;
                    end
                end
            end
        end

        function pair = find_series(obj, Xs)
        % Returns only series elements from Xs.

            pair = [];
            for index_1 = 1:length(Xs)
                X1 = Xs(index_1);
                for index_2 = 1 + index_1:length(Xs)
                    X2 = Xs(index_2);
                    if Modeller.is_series(obj, X1, X2)
                        pair = [X1, X2];
                        return;
                    end
                end
            end
        end

        function rename(obj)
        % Rename elements to maintain netlist order.

            Modeller.rename_eqs(obj.Resistors);
            Modeller.rename_eqs(obj.Inductors);
            Modeller.rename_eqs(obj.Capacitors);
            Modeller.rename_eqs(obj.Generic_zs);
            Modeller.rename_eqs(obj.Indep_VSs);
            Modeller.rename_eqs(obj.Indep_ISs);
        end

        function rename_eqs(Xs)
        % Rename any element in a given list, which is an equivalent of others.

            eqs = 0;
            for index = 1:length(Xs)
                X = Xs(index);
                if strcmp(X.id(2:end), '_eq')
                    eqs = eqs + 1;
                    X.id = sprintf('%s_eq%d', X.id(1), eqs);
                end
            end
        end

        function bool = check_shared(Xs, X1, X2, node)
        % Check if any element is connected to given node. X1 and X2 are not checked.

```

```
            bool = false;
            for index_1 = 1:length(Xs)
                X = Xs(index_1);
                if X ~= X1 && X ~= X2
                    bools = X.is_connected(node);
                    for index_2 = 1:length(bools)
                        if bools(index_2)
                            bool = true;
                            return;
                        end
                    end
                end
            end
        end

        function nodes = unique_nodes(X1, X2)
        % Return the nodes, which will connect a new equivalent element
        % to the rest of the circuit, i.e. the unique nodes.

            L = [X1.terminals, X2.terminals];
            uL = unique(L);
            nodes = uL(histcounts(L,[uL,inf])==1);
        end

        function [v_th, Z_th] = equivalent(obj, load)
        % Load object is deleted upon envoking open().

            ports = load.terminals;
            obj.open(load);
            ELAB.evaluate(obj);

            if ports(1) == 0
                v1 = 0;
            else
                v1 = rhs(obj.numerical_node_voltages(ports(1)));
            end

            if ports(2) == 0
                v2 = 0;
            else
                v2 = rhs(obj.numerical_node_voltages(ports(2)));
            end

            Modeller.remove_sources(obj);
            Modeller.simplify(obj, true);

            v_th = v1 - v2;
            Z_th = obj.Impedances(1);
        end
    end
end
```

9.2.4 Transmuter.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Transmuter
% A collection of functions designed to transmute a system into
% another form, that may be more helpful for analysis or design.

    methods(Static)

        function out = transmute(in, type_in, type_out, show)
        % Convert any representation of a system into another representation
        % of the type, denoted by target. Act as a switcher for sub-functions.

        % 'sd' = Symbolic s-domain       'td' = Symbolic t-domain.
        % 'tf' = Transfer function obj.  'ss' = State space.
        % 'zp' = Zero-pole-gain.         'de' = Differential equation.
        % 'ce' = Characteristic eq.      'ec' = Electrical circuit.

            if nargin < 4; show = false; end

            switch type_in
                case 'sd'
                    switch type_out
                        case 'td'
                            out = ELAB.sd2td(in, show);
                        case 'tf'
                            out = ELAB.sd2tf(in, show);
                        case 'ss'
                            in  = ELAB.sd2tf(in, show);
                            out = ELAB.tf2ss(in, show);
                        case 'zp'
                            in  = ELAB.sd2tf(in, show);
                            out = ELAB.tf2zp(in, show);
                        case 'de'
                            out = ELAB.sd2de(in, show);
                        case 'ce'
                            out = ELAB.sd2ce(in, show);
                        otherwise; error('Invalid type_out');
                    end
                case 'tf'
                    switch type_out
                        case 'sd'
                            out = ELAB.tf2sd(in, show);
                        case 'td'
                            in  = ELAB.tf2sd(in, show);
                            out = ELAB.sd2td(in, show);
                        case 'ss'
                            out = ELAB.tf2ss(in, show);
                        case 'zp'
                            out = ELAB.tf2zp(in, show);
                        case 'de'
                            in  = ELAB.tf2sd(in, show);
                            out = ELAB.sd2de(in, show);
                        case 'ce'
                            in  = ELAB.tf2sd(in, show);
                            out = ELAB.sd2ce(in, show);
                        otherwise; error('Invalid type_out');
                    end
                case 'td'
                    in = ELAB.td2sd(in, show);
                    switch type_out
                        case 'sd'
                            out = in;
                        case 'tf'
                            out = ELAB.sd2tf(in, show);
                        case 'ss'
                            in  = ELAB.sd2tf(in, show);
                            out = ELAB.tf2ss(in, show);
                        case 'zp'
                            in  = ELAB.sd2tf(in, show);
                            out = ELAB.tf2zp(in, show);
                        case 'de'
                            out = ELAB.sd2de(in, show);
                        case 'ce'
                            out = ELAB.sd2ce(in, show);
                        otherwise; error('Invalid type_out');
                    end
                case 'ss'
                    in = ELAB.ss2tf(in, show);
```

```
                    switch type_out
                        case 'sd'
                            out = ELAB.tf2sd(in, show);
                        case 'td'
                            in  = ELAB.tf2sd(in, show);
                            out = ELAB.sd2td(in, show);
                        case 'tf'
                            out = in;
                        case 'zp'
                            out = ELAB.tf2zp(in, show);
                        case 'de'
                            in  = ELAB.tf2sd(in, show);
                            out = ELAB.sd2de(in, show);
                        case 'ce'
                            in  = ELAB.tf2sd(in, show);
                            out = ELAB.sd2ce(in, show);
                        otherwise; error('Invalid type_out');
                    end
                case 'zp'
                    in = ELAB.zp2tf(in, show);
                    switch type_out
                        case 'sd'
                            out = ELAB.tf2sd(in, show);
                        case 'td'
                            in  = ELAB.tf2sd(in, show);
                            out = ELAB.sd2td(in, show);
                        case 'tf'
                            out = in;
                        case 'ss'
                            out = ELAB.tf2ss(in, show);
                        case 'de'
                            in  = ELAB.tf2sd(in, show);
                            out = ELAB.sd2de(in, show);
                        case 'ce'
                            in  = ELAB.tf2sd(in, show);
                            out = ELAB.sd2ce(in, show);
                        otherwise; error('Invalid type_out');
                    end
                case 'de'
                    in = ELAB.de2sd(in, show);
                    switch type_out
                        case 'sd'
                            out = in;
                        case 'td'
                            out = ELAB.sd2td(in, show);
                        case 'tf'
                            out = ELAB.sd2tf(in, show);
                        case 'ss'
                            in  = ELAB.sd2tf(in, show);
                            out = ELAB.tf2ss(in, show);
                        case 'zp'
                            in  = ELAB.sd2tf(in, show);
                            out = ELAB.tf2zp(in, show);
                        case 'ce'
                            out = ELAB.sd2ce(in, show);
                        otherwise; error('Invalid type_out');
                    end
                case 'ce'
                    in = ELAB.ce2sd(in, show);
                    switch type_out
                        case 'sd'
                            out = in;
                        case 'td'
                            out = ELAB.sd2td(in, show);
                        case 'tf'
                            out = ELAB.sd2tf(in, show);
                        case 'ss'
```

```
                            in  = ELAB.sd2tf(in, show);
                            out = ELAB.tf2ss(in, show);
                        case 'zp'
                            in  = ELAB.sd2tf(in, show);
                            out = ELAB.tf2zp(in, show);
                        case 'de'
                            out = ELAB.sd2de(in, show);
                        otherwise; error('Invalid type_out');
                    end
                otherwise
                    error('Invalid type_in');
            end
        end

        function out = sd2tf(in, show)
        % Symbolic expression to transfer function.

            if nargin < 2; show = false; end
            eval("s = tf('s');");
            eval(['T = ', char(in), ';']);

            out = minreal(T);
        end

        function out = zd2tf(in, show)
        % Symbolic expression to transfer function.

            if nargin < 2; show = false; end
            eval("z = tf('z');");
            eval(['T = ', char(in), ';']);
            out = minreal(T);
        end

        function out = tf2sd(in, show)
        % Transfer function to symbolic expression.

            if nargin < 2; show = false; end
            syms s;
            out = poly2sym(cell2mat(in.num),s)/poly2sym(cell2mat(in.den),s);
        end

        function out = tf2ss(in, show)
        % Transfer function to state space.

            if nargin < 2; show = false; end
            out = ss(in);
        end

        function out = ss2tf(in, show)
        % State space to transfer function.

            if nargin < 2; show = false; end
            [num, den] = ss2tf(in.A, in.B, in.C, in.D);
            out = tf(num,den);
        end

        function out = tf2zp(in, show)
        % Transfer function to zero-pole-gain function.

            if nargin < 2; show = false; end
            out = zpk(in);
        end

        function out = zp2tf(in, show)
        % Zero-pole-gain function to transfer function.

            if nargin < 2; show = false; end
            out = tf(in);
        end

        function out = de2sd(in, show)
        % Differential equation to symbolic expression.

            if nargin < 2; show = false; end
            syms y(t) s Y;
            eq = laplace(in.eq, t, s);
            eq = subs(eq, laplace(y, t, s), Y);

            if ~isempty(in.cond)
                eq = subs(eq, y(0), in.cond(1));
            end
            if length(in.cond) > 1
                eq = subs(eq, subs(diff(y(t), t), t, 0), in.cond(2));
            end
            warning('off','all');
            out = solve(eq,Y); warning('on','all');


            if show
                disp('Diff. equation'); disp(in.eq);
                disp('Laplace transform:'); disp(eq);
                disp('Solve for tf:'); disp(out);
            end
        end

        function out = sd2de(in, show)
        % Symbolic expression to differential equation.

            if nargin < 2; show = false; end
            out = in;
            error('Not yet implemented');
        end

        function out = sd2ce(in, show)
        % Symbolic expression to characteristic equation.

            if nargin < 2; show = false; end
            syms s;
            eq1 = expand(in);
            eq2 = numden(simplify(eq1)); % Extract numerator.
            char_eq = collect(eq2);
            out = flip(coeffs(char_eq, s));

            if show
                disp('Original expression:'); disp(in);
                disp('Expanded form:'); disp(eq1);
                disp('Set equal to 0, then simplify:'); disp(eq2);
                disp('Collect terms of equal power:'); disp(char_eq);
            end
        end

        function out = ce2sd(in, show)
        % Characteristic equation to symbolic expression.

            if nargin < 2; show = false; end
            syms s;
            out = poly2sym(in,s);
        end

        function out = sd2td(in, show)
        % Symbolic in s-domain to t-domain.

            if nargin < 2; show = false; end
            syms s t;
```

```
            pf = partfrac(in);
            out = ilaplace(pf, s, t);

            if show
                disp('In s-domain'); disp(in);
                disp('Partial fraction decomp:'); disp(pf);
                disp('Inverse Laplace:'); disp(out);
            end
        end

        function out = td2sd(in, show)
        % Symbolic in t-domain to s-domain.

            if nargin < 2; show = false; end
            syms s t;
            ex = expand(in);
            tr = laplace(ex, t, s);
            out = simplify(tr);
            out = collect(out);

            if show
                disp('In t-domain'); disp(in);
                disp('Expanded:'); disp(ex);
                disp('Laplace transform:'); disp(tr);
                disp('Collected and simplified:'); disp(out);
            end
        end

        function out = nd2zd(in, show)
        % Symbolic in n-domain to z-domain.

            if nargin < 2; show = false; end
            syms n z;
            ex = expand(in);
            tr = ztrans(ex, n, z);
            out = simplify(tr);
            out = collect(out);

            if show
                disp('In n-domain'); disp(in);
                disp('Expanded:'); disp(ex);
                disp('Z-transform:'); disp(tr);
                disp('Collected and simplified:'); disp(out);
            end
        end
    end
End
```

9.2.5 Visualizer.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Visualizer
% A collection of visualization- and graphing-functionality, specifically
% designed to further the user's understanding of a given system.

    properties(Constant, Access = private)
        M = containers.Map({'impulse','step','ramp', 'sin', 'cos'}, ...
            {tf(1,1), tf(1,[1,0]), tf(1,[1,0,0]), tf(1,[1,0,1]), tf([1,0],[1,0,1])})
    end

    methods(Static)

        function res = response(sys, ref)
        % Plots the response of the given system, using the given reference.
```

```
            name = '';

            % Input validity check.
            if ~isa(sys, 'tf')
                if isa(sys, 'sym')
                    disp('Converting symbolic to tf-object.');
                    ELAB.transmute(sys, 'td', 'tf', false);
                else
                    error('Sys must be system object, like ''tf''.');
                end
            end

            % If ref is a string, map it to a function.
            if ~isa(ref,'tf')
                if isa(ref, 'char')
                    name = ref;
                    ref = Visualizer.M(lower(ref));
                elseif isa(ref, 'sym')
                    disp('Converting symbolic to tf-object.');
                    ref = ELAB.transmute(ref, 'td', 'tf', false);
                else
                    error('Reference signal type invalid. Try using its name.');
                end
            end

            impulse(ref); hold on;
            impulse(sys * ref);

            title([char(name), ' response']);
            legend('Reference', 'response'); hold off; grid off;
            res = sys * ref;
        end

        function lat = plot_eq(sym, fontsize)
        % Displays an equation in a plot-window. For future GUI.

            if ~isa(sym,'sym'); error('Input must be a symbolic expression'); end
            if nargin == 1; fontsize = 18; end

            T = { 'alpha', 'beta', 'gamma', 'Gamma', 'delta', 'Delta',   ...
                  'epsilon', 'zeta', 'eta', 'theta','Theta', 'iota',     ...
                  'kappa', 'lamba', 'Lambda', 'mu', 'nu', 'xi', 'Xi',    ...
                  'pi', 'Pi', 'rho', 'sigma', 'Sigma', 'tau', 'upsilon', ...
                  'Upsilon', 'phi', 'Phi', 'chi', 'Chi', 'psi', 'Psi',   ...
                  'omega', 'Omega'};

            C  = {'b\eta', 'th\eta', 'Th\eta', 'Th\eta','u\psilon', 'U\psilon' ;
                  'beta',  'theta',  'Theta',  'Theta', 'upsilon',  'Upsilon'  };

            lat = ['$$',latex(sym),'$$'];            % Math typesetting environment
            for k = 1:numel(T)                       % Add '\' before greek.
                lat = strrep(lat, T{k}, ['\',T{k}]);
            end
            lat = strrep(lat,'\\','\');              % Some function e.g. Gamma already has '\'.
            for k = 1:numel(C)/2                     % Correct false '\', e.g. 'th\eta'.
                lat = strrep(lat, C{1,k}, C{2,k});
            end

            figure('Color','white','Menu','none')    % Change to 'figure' for menu bar.
            text(0.5, 0.5, lat, 'FontSize',fontsize, 'Color','k', ...
                'HorizontalAlignment','Center', 'VerticalAlignment','Middle', ...
                'Interpreter','Latex');
            axis off;
        end
    end
end
```

9.3 Appendix III – Elements

This appendix contains the code for the various elements used by the engine (core) of the program. The reader is encouraged to inspect the code on GitHub instead of in this report, as it is much easier to read and understand when formatted appropriately and contained within separate files.

9.3.1 Element.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef (Abstract) Element < handle & matlab.mixin.Heterogeneous
% The abstract basis for all circuit elements. All sub-classes need
% to implement the abstract properties and methods.

    properties
    % Shared by all sub-classes inheriting from this class.

        id; terminals;
    end

    properties(Abstract)
    % Properties must be implemented by sub-classes.

        num_terminals;
    end

    methods(Abstract)
    % Methods must be implemented by sub-classes.

        % Return string representation of object. Used to implement
        % netlist functionality, like updating the netlist.
        str = to_net(obj)

        % Check if this element is connected to the given node.
        bools = is_connected(obj, node)

        % Update terminal at index with new node value in given
        % object (usually a circuit)
        update_terminals(obj, index, value)

        % Create clone of this object, which is an object itself,
        % and not a reference to the original object, as is usually
        % the case for Matlab objects.
        cloned = clone(obj)
    end
end
```

9.3.2 Indep_S.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef (Abstract) Indep_S < Element
% The abstract basis for all independent sources. Should never
% be constructed directly, but through its sub-classes.

    properties
        is_AC;
        anode; cathode;
        num_terminals = 2;
    end

    methods
        function obj = Indep_S(id, anode, cathode, type)
        % Independant-voltage-source object constructor.

            obj.id = id;
            obj.anode = anode;
            obj.cathode = cathode;
            obj.terminals = [obj.anode, obj.cathode];

            if strcmpi(type, 'AC')
                obj.is_AC = 1;
            elseif strcmpi(type, 'DC')
                obj.is_AC = 0;
            else
                error("Invalid type. Use 'AC' or 'DC'.");
            end
        end

        function bools = is_connected(obj, node)
            bools = [obj.anode == node, obj.cathode == node];
        end

        function update_terminals(obj, index, value)
            obj.terminals(index) = value;
            obj.anode = obj.terminals(1);
            obj.cathode = obj.terminals(2);
        end
    end
end
```

Indep_IS.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Indep_IS < Indep_S
% Independent-current-source class implementing
% its abstract super-class: independent source.
% May be extended to implement non-ideal versions.

    properties
        current;
    end

    methods
        function obj = Indep_IS(id, anode, cathode, type, current)
        % Independant-current-source object constructor.

            if isempty(current)
                i = sym(id);
            else
                i = str2sym(string(current));
            end

            obj = obj@Indep_S(id, anode, cathode, type);
            obj.current = i;
        end

        function str = to_net(obj)
        % Override of the super-class function.

            if obj.is_AC, type = 'AC';
            else, type = 'DC'; end

            str = sprintf('%s %s %s %s %s\n', ...
                obj.id, num2str(obj.anode), num2str(obj.cathode), ...
                type, strrep(string(obj.current),' ',''));
        end

        function cloned = clone(obj)
            cloned = Indep_IS(obj.id, obj.anode, obj.cathode, ...
                obj.type, obj.current);
        end
    end
end
```

Indep_VS.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Indep_VS < Indep_S
% Independent-voltage-source class implementing
% its abstract super-class: independent source.
% May be extended to implement non-ideal versions.

    properties
        voltage;
    end

    methods
        function obj = Indep_VS(id, anode, cathode, type, voltage)
        % Independant-voltage-source object constructor.

            if isempty(voltage)
                v = sym(id);
            else
                v = str2sym(string(voltage));
            end

            obj = obj@Indep_S(id, anode, cathode, type);
            obj.voltage = v;
        end

        function str = to_net(obj)
        % Override of the super-class function.

            if obj.is_AC, type = 'AC';
            else, type = 'DC'; end

            str = sprintf('%s %s %s %s %s\n', ...
                obj.id, num2str(obj.anode), num2str(obj.cathode), ...
                type, strrep(string(obj.voltage),' ',''));
        end

        function cloned = clone(obj)
            cloned = Indep_VS(obj.id, obj.anode, obj.cathode, ...
                obj.type, obj.voltage);
        end
    end
end
```

9.3.3 Impedance.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Impedance < Element
% A generic impedance class from which the resistor, capacitor
% and inductor inherits, but may also be constructed directly.

    properties
        impedance;
        anode; cathode;
        v_across; i_through;
        num_terminals = 2;
    end

    methods
        function obj = Impedance(id, anode, cathode, impedance)
        % Impedance object constructor. Impedance is optional.

            obj.id = id;
            obj.anode = anode;
            obj.cathode = cathode;
            obj.terminals = [obj.anode, obj.cathode];

            if isempty(impedance)
                obj.impedance = sym(id);
            else
                obj.impedance = sym(impedance);
            end
        end

        function bools = is_connected(obj, node)
            bools = [obj.anode == node, obj.cathode == node];
        end

        function update_terminals(obj, index, value)
            obj.terminals(index) = value;
            obj.anode = obj.terminals(1);
            obj.cathode = obj.terminals(2);
        end

        function str = to_net(obj)
            str = sprintf('%s %s %s %s\n', ...
                obj.id, num2str(obj.anode), num2str(obj.cathode), ...
                strrep(string(obj.impedance),' ',''));
        end

        function cloned = clone(obj)
            cloned = Impedance(obj.id, obj.anode, obj.cathode, obj.impedance);
        end
    end
end
```

Resistor.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Resistor < Impedance
% A generic resistor class extending the impedance class.
% May be the basis for more specific resistors with their
% own properties and methods.

    properties
        resistance;
    end

    methods
        function obj = Resistor(id, anode, cathode, resistance)
        % Resistor object constructor. Resistance is optional.

            if isempty(resistance)
                r = sym(id);
            else
                r = str2sym(string(resistance));
            end

            obj = obj@Impedance(id, anode, cathode, r);
            obj.resistance = r;
        end

        function str = to_net(obj)
        % Override of the super-class function.

            str = sprintf('%s %s %s %s\n', ...
                obj.id, num2str(obj.anode), num2str(obj.cathode), ...
                strrep(string(obj.resistance),' ',''));
        end

        function cloned = clone(obj)
            cloned = Resistor(obj.id, obj.anode, obj.cathode, obj.resistance);
        end
    end
end
```

Capacitor.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Capacitor < Impedance
% A generic capacitor class extending the impedance class.
% May be the basis for more specific capacitors with their
% own properties and methods.

    properties
        capacitance;
    end

    methods
        function obj = Capacitor(id, anode, cathode, capacitance)
        % Capacitor object constructor. Capacitance is optional.

            if isempty(capacitance)
                c = sym(id);
            else
                c = str2sym(string(capacitance));
            end

            obj = obj@Impedance(id, anode, cathode, sym('s')*c);
            obj.capacitance = c;
        end

        function str = to_net(obj)
        % Override of the super-class function.

            str = sprintf('%s %s %s %s\n', ...
                obj.id, num2str(obj.anode), num2str(obj.cathode), ...
                strrep(string(obj.capacitance),' ',''));
        end

        function cloned = clone(obj)
            cloned = Capacitor(obj.id, obj.anode, obj.cathode, obj.capacitance);
        end
    end
end
```

Inductor.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

```

```
classdef Inductor < Impedance
% A generic inductor class extending the impedance class.
% May be the basis for more specific inductors with their
% own properties and methods.

    properties
        inductance;
    end

    methods
        function obj = Inductor(id, anode, cathode, inductance)
        % Inductor object constructor. Inductance is optional.

            if isempty(inductance)
                l = sym(id);
            else
                l = str2sym(string(inductance));
            end

            obj = obj@Impedance(id, anode, cathode, 1/(sym('s')*l));
            obj.inductance = l;
        end

        function str = to_net(obj)
        % Override of the super-class function.

            str = sprintf('%s %s %s %s\n', ...
                obj.id, num2str(obj.anode), num2str(obj.cathode), ...
                strrep(string(obj.inductance),' ',''));
        end

        function cloned = clone(obj)
            cloned = Inductor(obj.id, obj.anode, obj.cathode, obj.inductance);
        end
    end
end
```

9.3.4 Transistor.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef (Abstract) Transistor < Element
% The abstract basis for all transistor variations.

    properties
        beta; % Amplification factor.
        biasing;
        num_terminals = 3;
    end

    methods
        function obj = Transistor(id, beta)
        % Transistor object constructor.

            obj.id = id;

            if isempty(beta)
                obj.beta = sym(sprintf('beta_%s', id));
            else
                obj.beta = sym(beta);
            end
        end
    end
end
```

BJT.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef BJT < Transistor
% Bipolar Junction Transistor.

    properties
        base_node; collector_node; emitter_node;

        % May be changed directly by the user.
        V_T = 0.026;  % Thermal voltage.
        V_BE = 0.7;   % Voltage drop across base-emitter junction.
        r_o;
    end

    methods
        function obj = BJT(id, B, C, E, beta)
            obj = obj@Transistor(id, beta);
            obj.base_node = B;
            obj.collector_node = C;
            obj.emitter_node = E;
            obj.r_o = sprintf('R_o_%s', id);
            obj.terminals = [obj.base_node, obj.collector_node, obj.emitter_node];
        end

        function str = to_net(obj)
            str = sprintf('%s %s %s %s %s\n', ...
                obj.id, num2str(obj.base_node), num2str(obj.collector_node), ...
                num2str(obj.emitter_node), obj.beta);
        end

        function bools = is_connected(obj, node)
            bools = [obj.base_node == node, obj.collector_node == node, ...
                     obj.emitter_node == node];
        end

        function update_terminals(obj, index, value)
            obj.terminals(index) = value;
            obj.base_node = obj.terminals(1);
            obj.collector_node = obj.terminals(2);
            obj.emitter_node = obj.terminals(3);
        end

        function cloned = clone(obj)
            cloned = BJT(obj.id, obj.base_node, obj.collector_node, ...
                obj.emitter_node, obj.beta);
        end
    end
end
```

MOSFET.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef MOSFET < Transistor
% Metal Oxide Semiconductor Field Effect Transistor (CMOS).

    properties
        gate_node; drain_node; source_node;

        % May be changed directly by the user.
        V_GS = 0.8; % Threshold voltage.
        K_c = 5;    % mA/V^2.
    end

    methods
        function obj = MOSFET(id, G, D, S, beta)
            obj = obj@Transistor(id, beta);
            obj.gate_node = G;
            obj.drain_node = D;
            obj.source_node = S;
            obj.terminals = [obj.gate_node, obj.drain_node, obj.source_node];
        end

        function str = to_net(obj)
            str = sprintf('%s %s %s %s %s\n', ...
                obj.id, num2str(obj.gate_node), num2str(obj.drain_node), ...
                num2str(obj.source_node), obj.beta);
        end

        function bools = is_connected(obj, node)
            bools = [obj.gate_node == node, obj.drain_node == node, ...
                     obj.source_node == node];
        end

        function update_terminals(obj, index, value)
            obj.terminals(index) = value;
            obj.gate_node = obj.terminals(1);
            obj.drain_node = obj.terminals(2);
            obj.source_node = obj.terminals(3);
        end

        function cloned = clone(obj)
            cloned = MOSFET(obj.id, obj.gate_node, obj.drain_node, ...
                obj.source_node, obj.beta);
        end
    end
end
```

9.3.5 Amplifier.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef (Abstract) Amplifier < Element
% The basis for all transistor variations (BJT, MOSFET).

    properties
        gain; gain_val;
        num_terminals = 3;
    end
end
```

Ideal_OpAmp.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef Ideal_OpAmp < Amplifier
% Model of the ideal operational amplifier.
% May be used as a basis for non-ideal op-amps.

    properties
        input_1; input_2; output;
    end

    methods
        function obj = Ideal_OpAmp(id, input_1, input_2, output)
        % Operational-amplifier object constructor.

            obj.id = id;
            obj.input_1 = input_1;
            obj.input_2 = input_2;
            obj.output = output;
            obj.terminals = [obj.input_1, obj.input_2, obj.output];
        end

        function str = to_net(obj)
            str = sprintf('%s %s %s %s\n', ...
                obj.id, num2str(obj.input_1), num2str(obj.input_2), num2str(obj.output));
        end

        function bools = is_connected(obj, node)
            bools = [obj.input_1 == node, obj.input_2 == node, obj.output == node];
        end

        function update_terminals(obj, index, value)
            obj.terminals(index) = value;
            obj.input_1 = obj.terminals(1);
            obj.input_2 = obj.terminals(2);
            obj.output  = obj.terminals(3);
        end

        function cloned = clone(obj)
            cloned = Ideal_OpAmp(obj.id, obj.input_1, obj.input_2, obj.output);
        end
    end
end
```

9.3.6 Dep_S.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef (Abstract) Dep_S < Element
% The abstract basis for all dependent sources.

    properties
        anode; cathode;
        v_across; i_through;
    end
end
```

Dep_CC.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef (Abstract) Dep_CC < Dep_S
% The abstract basis for all dependent current-controlled sources.

    properties
        num_terminals = 2;
        ctrl_anode; ctrl_cathode;
    end

    methods
        function obj = Dep_CC(id, anode, cathode, ctrl_anode)
        % Dependent-current-controlled-source object constructor.

            obj.id = id;
            obj.anode = anode;
            obj.cathode = cathode;
```

```matlab
            obj.ctrl_anode = ctrl_anode;
            obj.terminals = [obj.anode, obj.cathode];
        end

        function bools = is_connected(obj, node)
            bools = [obj.anode == node, obj.cathode == node];
        end

        function update_terminals(obj, index, value)
            obj.terminals(index) = value;
            obj.anode = obj.terminals(1);
            obj.cathode = obj.terminals(2);
        end
    end
end
```

CCCS.m

```matlab
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef CCCS < Dep_CC
% Current Controlled Current Source (F).

    properties
        beta_gain;
    end

    methods
        function obj = CCCS(id, anode, cathode, ctrl_anode, beta_gain)
        % CCCS object constructor.

            if exist('beta_gain', 'var')
                beta = sym(beta_gain);
            end

            obj = obj@Dep_CC(id, anode, cathode, ctrl_anode);
            obj.beta_gain = beta;
        end

        function str = to_net(obj)
            str = sprintf('%s %s %s %s %s\n', ...
                obj.id, num2str(obj.anode), num2str(obj.cathode), ...
                num2str(obj.ctrl_anode), obj.beta_gain);
        end

        function cloned = clone(obj)
            cloned = CCCS(obj.id, obj.anode, obj.cathode, ...
                obj.ctrl_anode, obj.beta_gain);
        end
    end
end
```

CCVS.m

```matlab
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef CCVS < Dep_CC
% Current Controlled Voltage Source (H).

    properties
        r_gain;
    end

    methods
```

```
        function obj = CCVS(id, anode, cathode, ctrl_anode, r_gain)
        % CCVS object constructor.

            if exist('r_gain', 'var')
                r = sym(r_gain);
            end

            obj = obj@Dep_CC(id, anode, cathode, ctrl_anode);
            obj.r_gain = r;
        end

        function str = to_net(obj)
            str = sprintf('%s %s %s %s %s\n', ...
                obj.id, num2str(obj.anode), num2str(obj.cathode), ...
                num2str(obj.ctrl_anode), obj.r_gain);
        end

        function cloned = clone(obj)
            cloned = CCVS(obj.id, obj.anode, obj.cathode, ...
                obj.ctrl_anode, obj.r_gain);
        end
    end
end
```

Dep_VC.m

```
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef (Abstract) Dep_VC < Dep_S
% The abstract basis for all dependent voltage-controlled sources.

    properties
        num_terminals = 4;
        ctrl_anode; ctrl_cathode;
    end

    methods
        function obj = Dep_VC(id, anode, cathode, ctrl_anode, ctrl_cathode)
        % Dependent-voltage-controlled-source object constructor.

            obj.id = id;
            obj.anode = anode;
            obj.cathode = cathode;
            obj.ctrl_anode = ctrl_anode;
            obj.ctrl_cathode = ctrl_cathode;
            obj.terminals = [obj.anode, obj.cathode, obj.ctrl_anode, obj.ctrl_cathode];
        end

        function bools = is_connected(obj, node)
            bools = [obj.anode == node, obj.cathode == node, ...
                     obj.ctrl_anode == node, obj.ctrl_cathode == node];
        end

        function update_terminals(obj, index, value)
            obj.terminals(index) = value;
            obj.anode = obj.terminals(1);
            obj.cathode = obj.terminals(2);
            obj.ctrl_anode = obj.terminals(3);
            obj.ctrl_cathode = obj.terminals(4);
        end
    end
end
```

VCCS.m

```matlab
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef VCCS < Dep_VC
% Voltage Controlled Current Source (G).

    properties
        gm_gain;
    end

    methods
        function obj = VCCS(id, anode, cathode, ctrl_anode, ctrl_cathode, gm_gain)
        % VCCS object constructor.

            if exist('gm_gain', 'var')
                gm = sym(gm_gain);
            end

            obj = obj@Dep_VC(id, anode, cathode, ctrl_anode, ctrl_cathode);
            obj.gm_gain = gm;
        end

        function str = to_net(obj)
            str = sprintf('%s %s %s %s %s %s\n', ...
                obj.id, num2str(obj.anode), num2str(obj.cathode), ...
                num2str(obj.ctrl_anode), num2str(obj.ctrl_cathode), obj.gm_gain);
        end

        function cloned = clone(obj)
            cloned = VCCS(obj.id, obj.anode, obj.cathode, ...
                obj.ctrl_anode, obj.gm_gain);
        end
    end
end
```

VCVS.m

```matlab
% Part of ELABorate, all rights reserved.
% Auth: Nicklas Vraa

classdef VCVS < Dep_VC
% Voltage Controlled Voltage Source (E).

    properties
        mu_gain;
    end

    methods
        function obj = VCVS(id, anode, cathode, ctrl_anode, ctrl_cathode, mu_gain)
        % VCVS object constructor.

            if exist('mu_gain', 'var')
                mu = sym(mu_gain);
            end

            obj = obj@Dep_VC(id, anode, cathode, ctrl_anode, ctrl_cathode);
            obj.mu_gain = mu;
        end

        function str = to_net(obj)
            str = sprintf('%s %s %s %s %s %s\n', ...
                obj.id, num2str(obj.anode), num2str(obj.cathode), ...
                num2str(obj.ctrl_anode), num2str(obj.ctrl_cathode), obj.mu_gain);
        end

        function cloned = clone(obj)
            cloned = VCVS(obj.id, obj.anode, obj.cathode, ...
                obj.ctrl_anode, obj.mu_gain);
        end
    end
end
```