Programmatic Symbolic Circuit Analysis

Implemented as a MATLAB Toolbox

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

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

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

Contents

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.). 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 this 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. With this project, I intend to answer this question: Is it feasible to combine the advantages of symbolic analysis with the computational power of a computer in an intuitive manner? Currently, 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] 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.

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.

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

4.2 Algorithmic MNA

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

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

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

For the intermediate matrices comprising **A**:

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

For the intermediate matrices comprising **x**:

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

For the intermediate matrices comprising **z**:

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

Extending this approach to handle dependent sources is a matter of augmenting these intermediate matrices appropriately. This is slightly complex and as such will be fully explained further into this section, using pseudo-code. This extension is an important step, as extending the algorithm even further to handle more complex elements, such as transistors, will require the program to model these non-linear components using mainly dependent sources. As will become apparent in the implementation section, the *voltage-controlled-current-source* will play a major role in the development of these more high-level features. The following is a high-level, pseudo-code description of the algorithm but extended to include dependent sources and modified to fit 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 $passive$ be either a 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 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 passive is resistor, set $\exp = 1$
  Else if passive is capacitor, set $\exp = 1/s$
  Else if passive is inductor, set $\exp = s$

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

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 $j$ 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
    If first input is not grounded
      Set $c_{p,i} = c_{p,i} + 1$
    If second input is not grounded
      Set $c_{p,j} = c_{p,j} - 1$
  Add parsed op-amps id's to $j$ as $I_{opamp_{id}}$
```

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

```
Let $i,j$ denote anode and cathode
Let $k,l$ denote 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 $p$ 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 $b_{j,p} = b_{j,p} - CCCS_{id}$
```

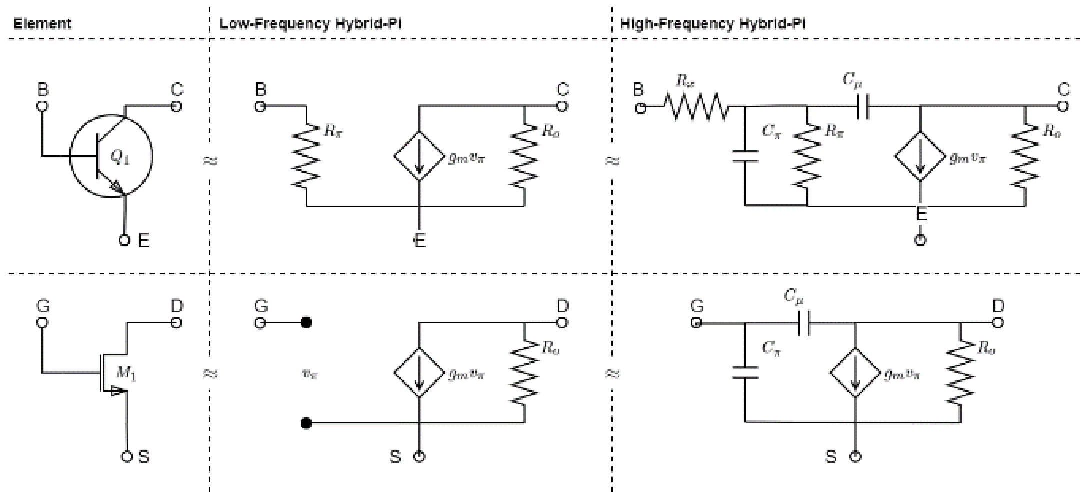
As was previously mentioned, updating the entries in the matrices, should be viewed as appending to symbolic equations. When all matrices and vectors are filled appropriately, we may compact these equations through algebraic simplification. When done, we can define A, x and z, representing the entire 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.



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

4.4 Circuit Simplification

As any electrical engineer knows, when having two or more of the same 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, a series and parallel connection must be strictly defined. If the simplification is done recursively until no series or parallel connections are left, one only needs to consider the most basic case: When the series or parallel connection consists of only *two* elements.

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

- A **parallel** connection has two nodes shared among the two elements. Any other elements – or collection of elements - may be connected to these nodes.

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

5 Implementation

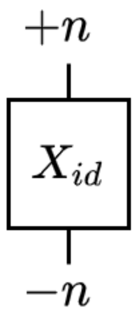
This section details the implementation of the program in code. Only the larger and most important structures of the program will be described. See the code on the GitHub repository for more detail. The code is extensively commented and attempts to adhere to the guidelines set by MATLAB for its users and employs 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 some additional information specific to the type of component. For the MNA algorithm to work properly, the reference node must be ground. Each node must also obey the sign convention. The syntax used by this program is SPICE-like with a few simplifications. I have chosen the custom file extension to be '.circ' for easier distinction between circuit-files and regular text files, but it is still a raw text file. Below is an overview of the syntax. Anything enclosed by square brackets is a variable set by the user, like the name or number given to a resistor. Anything in italic is an optional parameter, like the numerical value of a voltage source. If not specified, a symbolic is automatically assigned for instead.

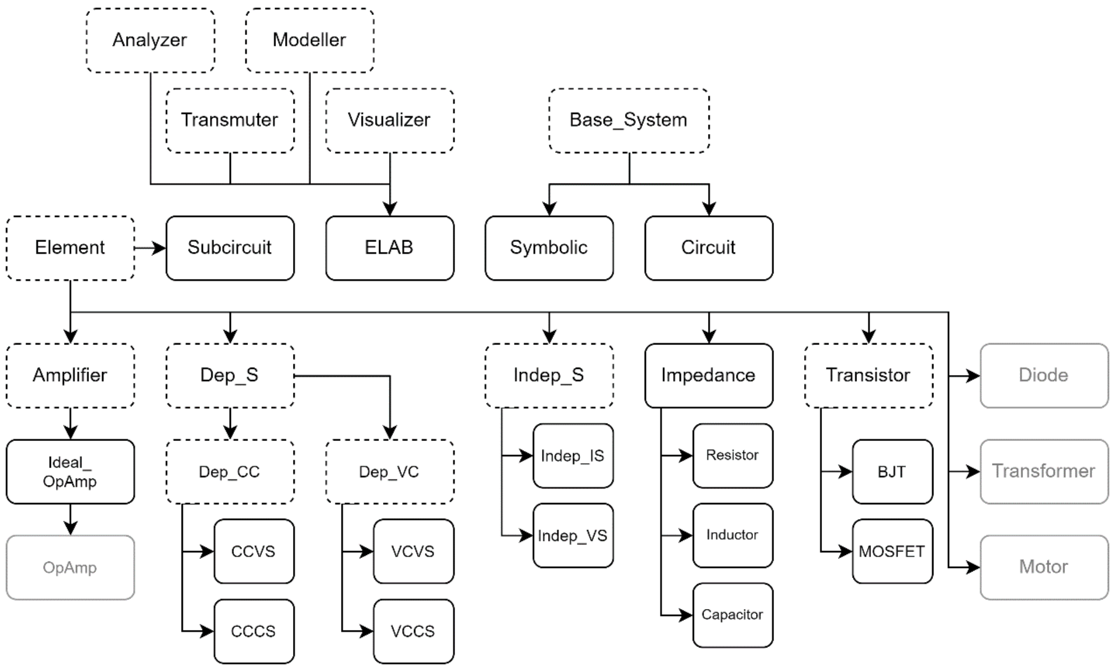
An "n" is for "node", and "cn" is for a "controlling node". "mode" must be either "AC" or "DC". For MOSFETs, G, D and S denote where its gate, drain and source connect. For BJTs, B, C and E denotes where its body, collector and emitter connect. Any numerical value, like the capacitance of a capacitor, 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*]



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 need be. This methodology repeats for most of the classes, the exception being the *ELAB*-class. This class collects functionality from the four classes: *analyzer*, *modeller*, *transmuter*, and *visualizer*. This is for the sake of the user, making the program simpler to use, while retaining some level of abstraction for anyone wishing to contribute the program. As the class names suggest, these four classes are responsible for a separate part of the task that constitutes circuit analysis. They interact directly with any given circuit object. 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.



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 carry out.

```
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
```

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
```

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.

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

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

    If not $found$
      Append $y$ to $L$

  Remove all elements in $L$ from circuit
```

5.4 The ELAB class

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

5.4.1 The Analyzer

The *Analyzer*-class implements modified nodal analysis in 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 these functions, which are very specific to the field of electrical engineering, a range of high-level systems analysis functions are also implemented in the *Analyzer*-class. At the time of implementation, I found these to be missing from the standard MATLAB ecosystem. These functions focus primarily on the stability of any given system, and so have broader application than simply electrical circuits. 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 access 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, using the *ac_eq* and *dc_eq* functions. These functions simply check whether an independent source has been designated as AC or DC in the netlist, then shorts the elements accordingly, using the lower-level functions, which have been described in a previous section. The *dc_eq* function will *short* any inductors and *open* any capacitors.

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
```

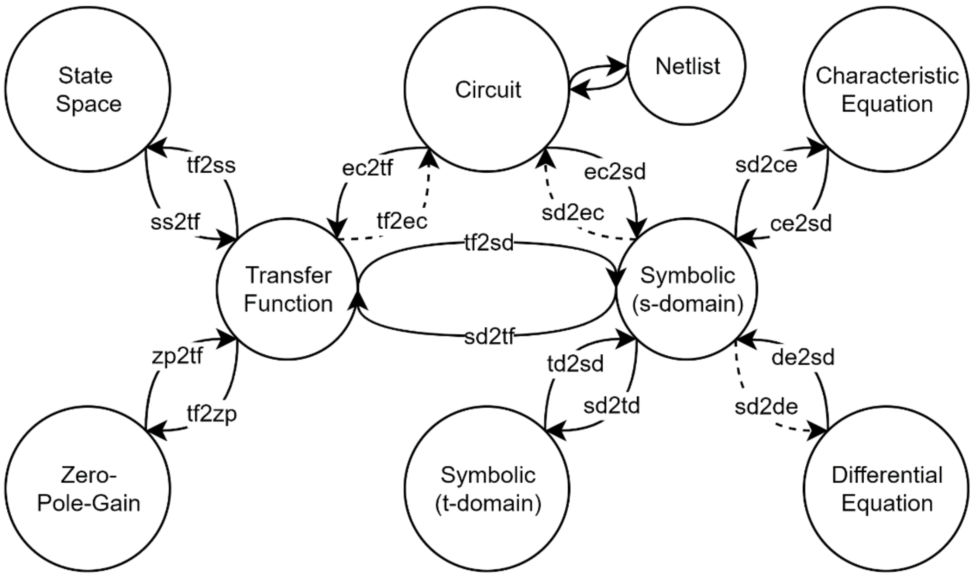
ID	Series equivalent	Parallel equivalent
R	R1+R2	(R1*R2)/(R1+R2)
C	(C1*C2)/(C1+C2)	C1+C2
L	L1+L2	(L1*L2)/(L1+L2)
V	V1+v2	V1 (only when v1=v2)
I	Not allowed	I1+I2

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.



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.

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.

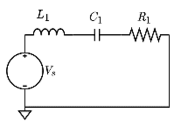
6.1 Using the Program

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).

6.1.1 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 facts about the given circuit.

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

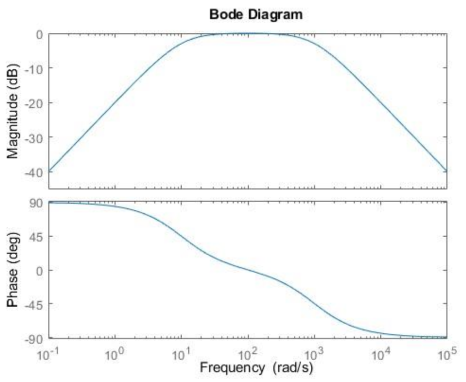
Say we wanted to know every how every node voltage is defined symbolically. We would simply look at the circuit object attributes, after it has been analysed using the analyze function on the ELAB class. From this, one can already deduce how each component in the circuit affects its overall behavior, 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)
```

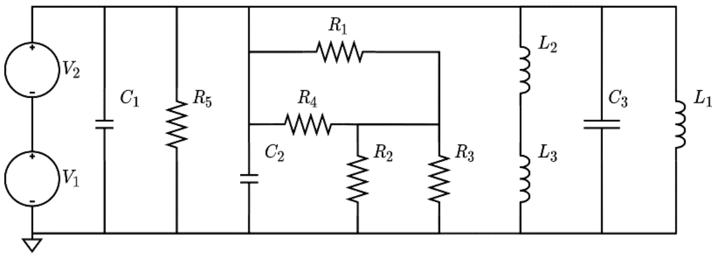


All of Matlab's built-in functionality for treating systems on the form of a transfer function, a state space, etc. now applies to circuits.

6.1.2 Simplifying a circuit

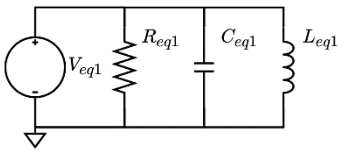
The program is capable of recursively simplifying a circuit of arbitrary complexity in seconds. For the simplify function to work properly, it utilizes the lower-level funtions mentioned previously, including the shorting- and opening-functions, as well as the cleaning and trimming functions. Take this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

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

ans =
    'V1 1 0 AC 10
     V2 2 1 AC 5
     R1 2 3 1000
     R2 3 0 2000
     R3 3 0 2000
     R4 2 3 3000
     R5 2 0 1000
     L1 4 2 0.02
     L2 4 0 0.03
     L3 2 0 0.2
     C1 2 0 0.000002
     C2 0 2 0.000003
     C3 2 0 0.00002'
```



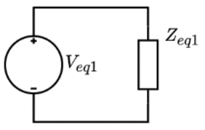
Simply calling the simplify function will reduce the circuit complexity. This circuit was purposely designed to reduce nicely into a single element of each type in a parallel configuration as shown below. Turns out, the original circuit can therefore act as a frequency isolator, which would not have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

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

ans =
    'Veq1 1 0 AC 15
     Req1 1 0 7000/11
     Leq1 1 0 0.04
     Ceq1 1 0 0.000025'
```



Calling the simplify function with its second parameter set to true will the function 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 =
    'Veq1 1 0 AC V1+V2
     Zeq1 1 0 (7000.0*s)/(11.0*s+280.0*s^2+280000000.0)'
```



The simplify function also works symbolically and even ensures approriate 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
     Req1 1 2 R1+R2
     Req2 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 rule-set, on the project's GitHub repository. The simplify function and its lower-level dependencies have been thoughouly tested, but of course edge cases may present bugs, which can also be reported on the GitHub page.

6.1.3 Adding, removing and Cleaning

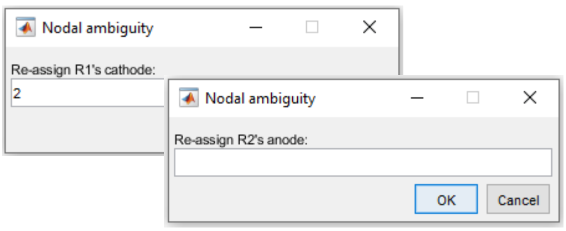
Adding or removing elements are used by several higher-level functions, but are also available to be used directly by the user. Say, we want to add an additional resistor to a simple voltage divider. If the element is defined to be connected to existing nodes, for example when adding an element in parallel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

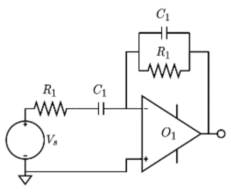
Of course, one may want to add an element "onto a wire", i.e. onto a single node, the process is slightly more complicated. When -1 is used to denote a nodal connection, it means that the node has yet to be created, so in the next case, the resistor R_x will go onto the wire between R_1 and R_2. However, 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 $s = -1/\,(R_1C_1)$ and $s = -1/\,(R_2C_2)$. 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 =
```

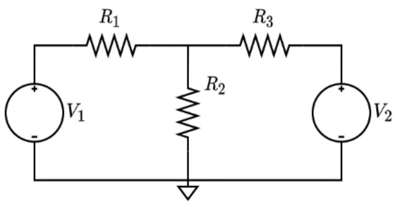
$$\frac{v_4}{v_1} = -\frac{C_1\,R_2\,s}{C_1\,R_1\,s+1}$$

The transfer functions now look like this. Note, that the nodes have been appropriately relabelled 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 $s = -1/\,(R_1C_1)$ as expected, and that opening C_2 will remove the pole at $s = -1/\,(R_2C_2)$.

6.1.4 Thevenin/Norton Equivalents

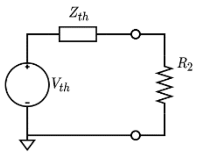
Finding Thevenin or Norton Equivalent circuits is often useful for circuit analysis, so of course there exists a high-level function for just this. To illustrate the use of these functions, we load a circuit and decide to find the Thevenin-equivalent. In this case, let us see what the equivalent circuit looks like, if we view R_2 as the load impendance. Again, the choise 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'
```

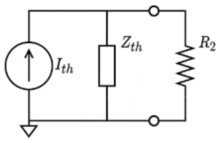


The Thevenin and Norton functions leverages lower-level functions like clone to make a copy of the load element, open to open-circuit the load, simplify to reduce to a single impedance, and then finally the add function to place the clone of the load back into the circuit.

```
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'
```



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.

```
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'
```

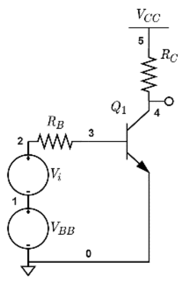
And again, because the simplify function can handle simplifying elements - as if they were generic impedances – down to a single impedance, the thevenin and norton functions can handle any type of RLC-circuit as well.

6.1.5 Dependent Sources

6.1.6 AC/DC-equivalents

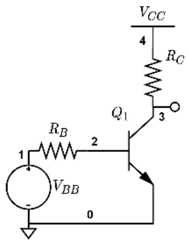
It is often useful when performing circuit analysis to view the circuit as operating in AC and DC separately. This is the case when performing analysis on non-linear circuits, such as a transistor amplifier. We start by loading in a circuit. In this case, it's a simple common-source amplifier with biasing. 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'
```

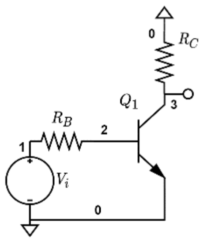


Typically, when analyzing such circuits, we split up the analysis into a DC-part and an AC-part. To find the DC-equivalent of the circuit above, one can simply call the dc_eq function (short for direct-current-equivalent). This function uses lower-level functions such as short, open and clean to modify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



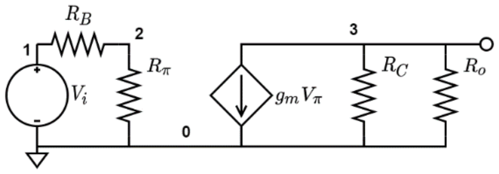
```
ELAB.ac_eq(circuit);
circuit.list

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'
```



6.1.7 Non-linear Modelling

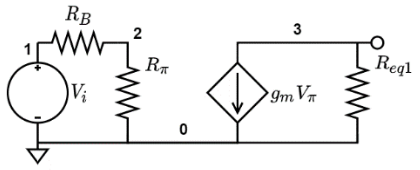
Extending from the previous example, we may use the the AC-equivalent circuit to gain further insight. From the AC-equivalent, you may model the transistor as linear elements, using the hybrid_pi function, which can either assume the circuit will operate in low-frequency "lf" or high-frequency "hf". 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 =

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

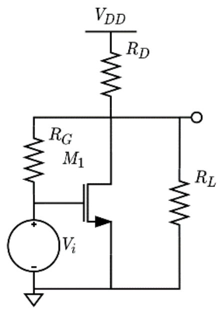
```
ELAB.ec2sd(circuit,1,3)
```

ans =

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

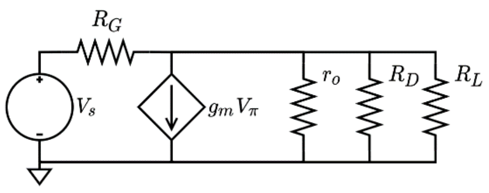
Of course, all of the functionality shown on BJT's also works for circuits with MOSFETs or mix of the two. Let us load in a similar circuit, but this time with a MOSFET.

```
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.

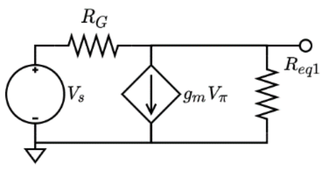
```
ELAB.hybrid_pi(circuit,'lf');
circuit.list

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 =
```

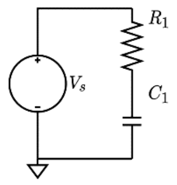
$$\frac{v_2}{v_1} = \frac{R_{\mathrm{eq1}}\ (G_{M,1}\, R_G + 1)}{R_G + R_{\mathrm{eq1}}}$$



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'
```



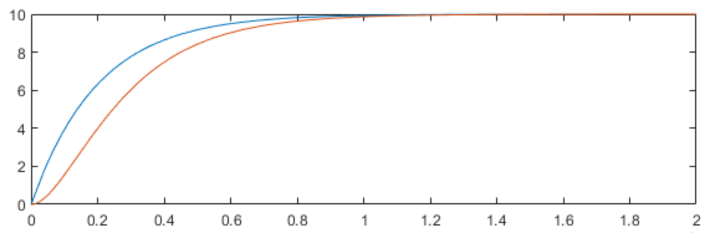
We analyze the circuit as usual to obtain the circuit equations. If evaluate detects that the circuit has not been analyzed, it will call the analyze function automatically. In this case, we would like to know the voltage across the capacitor. Evaluate has saved the results in the circuit object. We 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 = $10\,e^{-10000\,t} - 20\,e^{-5000\,t} + 10$

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.

6.2 Summation of Results

7 Conclusion

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 simple 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
│   └── ELAB.m
├── examples/
│   ├── circuits/
│   ├── pdfs/
│   └── scripts/
├── gui/
│   └── ...
├── resource/
│   └── ...
├── readme.md
└── plan.md
```

7.2 Unreleased functionality

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 Appendix I

This appendix is the code 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 correctly and contained within separate files.

9.1 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 Engine

9.2.1 Analyzer.m

9.2.2 Modeller.m

9.2.3 Transmuter.m

9.2.4 Visualizer.m

9.3 Elements

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

```matlab
% 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
        gain; gain_val;
        num_terminals = 3;
    end

    methods(Abstract)
        % internal(obj)
    end
end
```

BJT.m

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

classdef BJT < Transistor
% Bipolar Junction Transistor.

    properties
        base_node; collector_node; emitter_node;
        R_bb; R_cc; R_ee; % Optional internal resistances.
    end

    methods
        function obj = BJT(id, B, C, E, gain_val, R_bb, R_cc, R_ee)
            obj.id = id;
            obj.base_node = B;
            obj.collector_node = C;
            obj.emitter_node = E;
            obj.gain = sym(sprintf('beta_%s', id));

            if isempty(gain_val), obj.gain_val = obj.gain;
            else, obj.gain_val = sym(gain_val); end

            if isempty(R_bb), obj.R_bb = sym(sprintf('R_bb_%s', id));
            else, obj.R_bb = sym(R_bb); end

            if isempty(R_cc), obj.R_cc = sym(sprintf('R_cc_%s', id));
            else, obj.R_cc = sym(R_cc); end

            if isempty(R_ee), obj.R_ee = sym(sprintf('R_ee_%s', id));
            else, obj.R_ee = sym(R_ee); end

            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.gain_val);
        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.gain);
        end
    end
end
```

MOSFET.m

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

classdef MOSFET < Transistor
% Metal Oxide Semiconductor Field Effect Transistor.

    properties
        gate_node; drain_node; source_node;
        R_gg; R_dd; R_ss; % Optional internal resistances.
    end

    methods
        function obj = MOSFET(id, G, D, S, gain_val, R_gg, R_dd, R_ss)
            obj.id = id;
            obj.gate_node = G;
            obj.drain_node = D;
            obj.source_node = S;
            obj.gain = sym(sprintf('beta_%s', id));

            if isempty(gain_val), obj.gain_val = obj.gain;
            else, obj.gain_val = sym(gain_val); end

            if isempty(R_gg), obj.R_gg = sym(sprintf('R_gg_%s', id));
            else, obj.R_gg = sym(R_gg); end

            if isempty(R_dd), obj.R_dd = sym(sprintf('R_dd_%s', id));
            else, obj.R_dd = sym(R_dd); end

            if isempty(R_ss), obj.R_ss = sym(sprintf('R_ss_%s', id));
            else, obj.R_ss = sym(R_ss); end

            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.gain_val);
        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.gain);
        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;
            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

```
% 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

```
% 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

```
% 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

```
% 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
```

10 Appendix II

This appendix contains exports of livescripts in which various use-cases for the program are showcased.