Symbolic Analysis of Linear Electric Circuits with MIT Julia Symbolics.JI Package

GLORIJA Z. DOŠLO, University of Belgrade, School of Electrical Engineering, Belgrade
KRISTINA D. RAJKOVIĆ, University of Belgrade, School of Electrical Engineering, Belgrade

This paper presents novel free open-source software JuliaCAP for symbolic analysis of linear time-invariant electric circuits in the complex domain of the Unilateral Laplace Transform or Phasor Transform. Modified Nodal Analysis (MNA) has been chosen for formulation of circuit equations in JuliaCAP. JuliaCAP has been developed by Kristina Rajković, Glorija Došlo, Nikola Radojević, Tamara Petković and Ivana Stanoević, at the University of Belgrade – School of Electrical Engineering, Belgrade, Serbia. JuliaCAP is written in Julia, a free open-source (MIT licensed) programming language. Whereas Julia is a relatively new language, there will be some words about it. As a free open-source software package, the underlying algorithm is revealed in full detail, so anybody can get a better understanding of the used methods. We offer JuliaCAP as a useful tool for solving linear time-invariant electric circuits. The future directives should be the improvement of simplifying expressions and the absence of the possibility to solve electric circuits in the complex domain of the Unilateral Laplace Transform. Also, it is shown through examples how this software operates. Electric circuits that are used in examples are Riordan gyrator network, 3rd order Butterworth Low-pass Filter, Subtractor and Adder.

Key Words: Symbolic analysis, Linear time-invariant electric circuit, Julia programming language, Symbolic simulation, Modified Nodal Analysis (MNA), Unilateral Laplace Transform

1. INTRODUCTION

Symbolic simulation is a formal technique for determining the behavior or a characteristic of a system such as a digital system, electronic circuit, or continuous-time system with an independent variable such as time, frequency, or sample index, the dependent variables such as voltages, currents, signals, and sample values, and symbols representing (some or all of) the element values [1].

A computer program known as a symbolic simulator can automatically perform symbolic analysis [2]-[3] on the system description given as input, and thus produce the symbolic expression for the desired system characteristic [1]. Symbolic analysis might be an important tool for contemporary education, see for example [4].

The use of symbolic computation has been prevalent in the modeling, simulation, and synthesis of VLSI systems and analog circuits [1]JuliaCAP is the first or at least among the first programs written in Julia for the symbolic analysis of linear time-invariant circuits. We wrote this project based on two other programs for symbolic analysis of electrical circuits. They are SymPyCAP [5] and SALECx [1]. SymPyCAP was written in Python while SALECx was written in Maxima CAS [6]. Both SymPyCAP and SALECx have good functions for solving system of equations and simplifying the results, which currently Julia lacks. SALECx has method linsolve, SymPyCAP is using linsolve and solve methods and JuliaCAP is using solve_for method.

There is another project for symbolic analysis of electrical circuits which is called SCAM (Symbolic Circuit Analysis in MATLAB) [7] which is written in MATLAB.

JuliaCAP supports same elements as SALECx and SymPyCAP and much more electrical elements than SCAM. SCAM supports Resistor, Inductor, Capacitor, Ideal Voltage Source, Ideal Current Source, Ideal Ope-
rational Amplifier while JuliaCAP supports all elements as SCAM and much more. Those elements are Voltage Controlled Current Source, Voltage Controlled Voltage Source, Current Controlled Current Source, Current Controlled Voltage Source, Ideal Transformer, Linear Inductive Transformer, Two-port specified by ABCD-parameters, Impedance, Admittance and Transmission line. Unlike SCAM, JuliaCAP has support for adding initial conditions for capacitor and inductor to solve circuit in initial state. Mathematica offers a convenient and comprehensive environment called SchematicSolver for drawing, analyzing, solving, designing, and implementing systems, if you want to know more about SchematicSolver see for example [8].

There is also a CircuitSolver which is similar but not the same program that is designed to be used as an instructional tool to guide the student step-by-step in the process of formulating and solving the circuit equations, see [9] for more on that subject and if you are interested in how symbolic analysis is applied in telecommunications, refer to [10].

2. METHODS FOR SOLVING ELECTRIC CIRCUITS

2.1. Mesh Current Method

The Mesh current method is used in electrical engineering to solve complex circuit problems. This method is based on the assumption that currents in the circuit can be represented as sums of currents flowing through closed loops (contours) within the circuit.

The steps to solve a circuit using the method of contour currents include:

- Identify all contours in the circuit.
- Determine the direction of current in each contour.
- Apply Kirchhoff’s Voltage Law for each contour that does not have current source.
- Solve the system of equations for the currents flowing through each contour.
- Using the calculated currents, calculate all other circuit parameters, such as currents, voltages and powers.

Let’s show this method with an example:

Let us choose contours and directions of currents in this circuit:

![Figure 1 – Example circuit](image)

Let us choose contours and directions of currents in this circuit:

![Figure 2 – Circuit with marked contours and currents](image)

Apply Kirchhoff’s Voltage Law to both contours:

Contour 1: \((R_1 + R_2)I_{k1} - R_3I_{k2} = E_1\)  \(\text{(1)}\)

Contour 2: \(-R_3I_{k1} + (R_2 + R_3)I_{k2} = -E_2\)  \(\text{(2)}\)

After solving this system of equations, we can calculate currents of branches:

\[I_1 = I_{k1}\]
\[I_2 = I_{k2}\]
\[I_3 = I_{k1} - I_{k2}\]

Now we can use the currents to calculate the rest of circuit parameters.

The mesh current method is particularly useful for analyzing circuits with multiple current sources because every current source removes one of the equations from the circuit.

2.2 Nodal Analysis Method

The nodal analysis method is a fundamental technique for analyzing electric circuits. It is based on the principle of Kirchhoff’s Current Law (KCL), which states that the sum of currents flowing into a node (or junction) in a circuit must be equal to the sum of currents flowing out of the node.

The basic nodal analysis method involves three steps:

- Select one node in the circuit to be the reference node (usually the one with the most connections).
- For each non-reference node in the circuit, write an equation based on KCL that relates the currents flowing into and out of the node.
- Solve the resulting system of equations for the node voltages using standard algebraic techniques.
- Using the calculated potentials, calculate all other circuit parameters, such as currents, voltages and powers.
- We will use the same circuit like in the previous method (Fig. 1). Let’s choose node 1 to be refe-
rence node and choose the directions of the currents:

![Circuit diagram](image)

**Figure 3 – Circuit with marked directions of the currents**

Now let’s write equations for other nodes excluding the reference node:

**Node 2:**

\[
\frac{V_2 - E_1}{R_1} + \frac{V_2 - E_2}{R_2} + \frac{V_2}{R_3} = 0 \tag{3}
\]

From this equation we can get potential of node 2 and then we calculate the rest of circuit parameters.

The nodal analysis method can be used to solve any number of unknown node voltages in a circuit, making it a powerful tool for analyzing complex circuits. However, it can be more difficult to apply in circuits with a large number of nodes since number of equations is increasing and it may not be the most efficient method for certain types of circuits (such as those with many current sources).

Overall, the nodal analysis method is a valuable technique for solving electric circuits, and it forms the basis for many other circuit analysis methods.

### 2.3. Modified Nodal Analysis (MNA)

Modified Nodal Analysis (MNA) is an extension of nodal analysis which not only determines the circuit’s node voltages (as in classical nodal analysis), but also some branch currents [11]-[12].

Let’s show this method with an example:

![Non-Inverting Amplifier circuit](image)

**Figure 4 – Non-Inverting Amplifier circuit**

First, we have to mark the nodes (here we started numeration from one because we aimed to be consistent with the numeration in our project). Then we choose directions of currents that cannot be expressed through the equations of the elements and node voltage. In this example, it’s the current of \( u_{g1} \) and the output of the operational amplifier.

Now we check if the graph is connected. If the graph is not connected then one should (1) identify the disconnected components, (2) choose one node in each component, and (3) connect the chosen nodes to make the graph connected [1]. In our case the graph is already connected:

![Non-Inverting Amplifier graph](image)

**Figure 5 – Graph of Non-Inverting Amplifier**

Now we have to write Kirchhoff’s current law (KCL) equations for every node except for the ground node (which is node one). Potentials of the nodes mark as \( V_1, V_2, V_3, \ldots \).

Equations are:

\[
2: \frac{V_2}{R_1} + \frac{V_2 - V_3}{R_2} = 0 \tag{4}
\]

\[
3: \frac{V_3 - V_2}{R_2} + \frac{V_3}{R_3} + i_{\text{op\text{Amp}}} = 0 \tag{5}
\]

\[
4: i_{u_g} + \frac{V_2 - V_4}{R_4} = 0 \tag{6}
\]

\[
5: \frac{V_5 - V_4}{R_4} = 0 \tag{7}
\]

Since \( i_{u_g} \) and \( i_{\text{op\text{Amp}}} \) cannot be expressed over the voltage of the nodes then we add equation of each element:

\[
V_4 = u_g \tag{8}
\]

\[
V_5 - V_2 = 0 \tag{9}
\]

After solving this system of equations, we get:

\[
V_5 = V_2 = V_4 = u_g \tag{10}
\]

\[
V_3 = \frac{R_3 u_g + R_1 u_g}{R_1} \tag{11}
\]

\[
i_{\text{op\text{Amp}}} = - \frac{R_3 u_g + R_2 u_g + R_1 u_g}{R_1 R_3} \tag{12}
\]

\[
i_{u_g} = 0 \tag{13}
\]

The reason why we use MNA method in our code and not other mentioned methods, even though other methods generate smaller system of equations, is because MNA is easy to automate and implement in software. It is a well-defined algorithmic procedure for...
formulating the system of equations that need to be solved. Also MNA is suitable for analyzing circuits with both linear and nonlinear elements. This means that it can handle circuits that cannot be solved using other methods and can be optimized for efficient matrix operations, making it suitable for use in computer programming.

3. JULIACAP

JuliaCAP is free open-source software for solving linear time-invariant electric circuits, written by Kristina Rajković, Glorija Došlo, Nikola Radojević, Tamara Petković and Ivana Stanojević, in the complex domain of the Unilateral Laplace Transform or Phasor Transform. Reference node is one node labeled by 1. The node voltage of this node is set to 0. Other nodes are labeled by consecutive integers, starting from 1.

Here is a pseudo code of two important structs and function `solveCircuit`:

```plaintext
struct Edge
{
    type :: TypeOfEdge
    name :: String
    node1 :: Vector[Int]
    node2 :: Vector[Int]
    param :: Vector()
    currentOrVoltage :: Vector{} //initial conditions
}

struct Graph
{
    nodeEquations :: Vector[Equation]
    edgeEquations :: Vector[Equation]
    edges :: Vector[Edge]
    maxNode :: Int
}

function solveCircuit(graph :: Graph, args :: Dict)
{
    omega = ""
    for (a in keys(args)) {
        if a == "ω" || a == "omega"
            omega = args[a]
        if ...
    }
    ...
    if (omega == "")
    { s = "s"
    } else
    { s = j * omega
    }
    nodeCurrents = Vector{}
    symbols = Vector{}
    for (g in graph.edges)
    {
        if (g.type == R)
        {
            U1 = Symbol("U" + g.node1[1])
            U2 = Symbol("U" + g.node2[1])
            symbols.push(U1, U2) // save all symbols
            I = (U1 - U2) / g.param[1]
            nodeCurrents[g.node1[1]] += I
            nodeCurrents[g.node2[1]] -= I
        } elseif (g.type == Vg)
        {
            ...
        } elseif (g.type == L)
        {
            U1 = Symbol("U" + g.node1[1])
            U2 = Symbol("U" + g.node2[1])
            symbols.push(U1, U2) // save all symbols

            if (isempty(g.currentOrVoltage))
                g.currentOrVoltage = 0
            I1 = (U1 - U2) / g.param[1] * s
            g.currentOrVoltage[1] / s
            I2 = (U2 - U1) / g.param[1] * s
            g.currentOrVoltage[1] / s
            nodeCurrents[g.node1[1]] += I1
            nodeCurrents[g.node2[1]] += I2
        } elseif (g.type == C)
        {
            U1 = Symbol("U" + g.node1[1])
            U2 = Symbol("U" + g.node2[1])
            symbols.push(U1, U2) // save all symbols

            if (isempty(g.currentOrVoltage))
                g.currentOrVoltage = 0
            nodeCurrents[g.node1[1]] += (U1 - U2) * s * g.param[1] - g.currentOrVoltage[1] * g.param[1]
```
In this pseudo code you can see the most important function in JuliaCAP that creates equations and uses `solve_for` function to get result of system of equations and `simplify` function that simplifies result.

Reserved symbols:
- Vg – MNA voltage variables,
- Ig – MNA current variables,
- w – frequency for the Phasor Transform analysis,
- omega – another name for w.

To input a circuit user needs to create the corresponding graph using the following method:

```python
newGraph()
```

Then add edges using the method `addEdge` with different parameters:

```python
addEdge(graph, Edge(type, name, n1, n2, par))
addEdge(graph, Edge(type, name, n1, n2, par, ic))
```

For one-port elements:
- n1 – positive terminal,
- n2 – negative terminal.

For two-port elements:
- n11 – positive terminal of the first port,
- n12 – negative terminal of the first port,
- n21 – positive terminal of the second port,
- n22 – negative terminal of the second port.

`par` – element parameter in the form of string or number. Also element can have more parameters if par is a list.

`ic` - initial conditions at 0-minus also in the form of string or number.

For capacitors, „Io“ for inductors,
For linear inductive transformers.

3.1. Other supported methods
- `solveCircuit(graph, Dict(„omega“ => „w“))` – solves electric circuit
printResults(results) – prints result of circuit, node voltages and currents.

printEquations() – prints equations on the basis of which the solution is formed.

printLatexEquations() – prints equations in latex.

printLatexResults(results) – prints result in latex.

printSpecificResult(results, “U2”) – prints specific result specified by second argument of function.

printLatexSpecificResult(results, “IopAmp1”) – prints specific result in latex specified by second argument of function.

printCircuitSpecifications(graph) – prints number of nodes, elements, basic equations and variables.

The further simplification of results could not be done because the function simplify from Symbolics.jl has some limitations. We were not able to predict when the mathematical expression that we were trying to process using this method would end up simplified and when we would get some unexpected behavior. In order to avoid errors that would occur during the use of our module, the whole idea of simplifying was forsaken. There is still hope that in the future Julia will get more developed in this segment, so the utilization of this method will be possible without obstacles.

4. JULIACAP SYMBOLIC SIMULATION EXAMPLES

All of the presented examples have been created using Jupyter Notebook. Also, it is assumed that JuliaCap.jl has been uploaded.

All of the electrical circuits are examples taken from course „Electric Circuit Theory“ [13].

4.1. Example 1 – Riordan Gyrator Network

The first example is Riordan gyrator network shown in Figure 7.

The code for this example is shown in Figure 8.

First, JuliaCap is loaded in the user program. Then, a new circuit is created with the method newGraph and the circuit object is assigned to the variable “graph”. The second case has two variants, the first one being creating a module and putting all the code in that module, then inside the module can be used using JuliaCap. The second variant is importing JuliaCap code with import .JuliaCap and this way requires putting JuliaCap before calling the function.

3.3. Problems that we have encountered

Because our software is fully developed in Julia, using only libraries that it offers, JuliaCap lacks some of the functionalities that SALECx and SymPyCap have.

One of the main problems is the absence of the possibility to solve electrical circuits in the complex domain of the Unilateral Laplace Transform. Through time, Julia developed and in the latest version 1.9.3, the function solve_for has been upgraded. It now has internal simplification and can be used for more complex equations.
the method `solveCircuit(graph, Dict("omega" => "w"))` which contains all the node’s voltages and element’s currents. Using `printEquations` equations that are formed during solving are printed. In the end, using `printLatexResults(result)` the final results are printed in Latex form. The circuit equations and the corresponding response generated by JuliaCAP, that is the program output, are as follows:

```
U2 = Vg
U3 = U2 + 8
U2 = U3 + 0
IVg = (U2 - U4) / R5 = 0
(U3 + U4) / R4 + (U3 - U5) / R3 = 0
I0 = U4 / R2 + U5 / R3 + (U4 - U5) / R4 = 0
U5 / R1 + C2*w*(U5 - U6) = 0
I0 = (U5 - U3) / R3 + C2*w*(U6 - U5) = 0
```

**Figure 9 – The equations of the example 1**

In order to have "pretty" expressions for the final presentation, the user can set the cell type to Markdown and copy and paste the results.

```
U1 = Vg
U2 = U3 + 0
IVg = (U2 - U4) / R5 = 0
(U3 + U4) / R4 + (U3 - U5) / R3 = 0
I0 = U4 / R2 + U5 / R3 + (U4 - U5) / R4 = 0
U5 / R1 + C2*w*(U5 - U6) = 0
I0 = (U5 - U3) / R3 + C2*w*(U6 - U5) = 0
```

**Figure 10 – The solution of the example 1**

The next example is the 3rd order Butterworth Low-pass Filter.

```
U1 = U
U3 = \frac{L_1 w}{R_1 + L_1 w} + \frac{C_1 w}{R_1 + C_1 w}
U2 = \frac{U}{R} - \frac{L_1 w}{R_1 + L_1 w} + \frac{C_1 w}{R_1 + C_1 w}
```

**Figure 14 – The result of the example 2**

4.3. Example 3 – Subtractor

The next example is Subtractor, realized with an operational amplifier.
The result is shown in Figure 20.

4.4. Example 4 – Adder

The next example is Adder, realized with an operational amplifier.

\[
\begin{align*}
U_4 &= -R_3 \left( \frac{U_{g1} (\frac{1}{R_1} + \frac{1}{R_2}) + U_{g2} (\frac{1}{R_1} + \frac{1}{R_3})}{R_1 \left( \frac{1}{R_2} + \frac{1}{R_3} \right)} - R_1 (\frac{1}{R_2} + \frac{1}{R_3}) \right) \\
U_{g2} &= \frac{-U_{g3}}{R_3} \\
U_6 &= -R_1 \left( \frac{U_{g1} (\frac{1}{R_1} + \frac{1}{R_2}) + U_{g2} (\frac{1}{R_1} + \frac{1}{R_3})}{R_1 \left( \frac{1}{R_2} + \frac{1}{R_3} \right)} - R_1 (\frac{1}{R_2} + \frac{1}{R_3}) \right) \\
U_3 &= U_{g2} \\
U_5 &= R_0 \left( \frac{U_{g1} (\frac{1}{R_1} + \frac{1}{R_2}) + U_{g2} (\frac{1}{R_1} + \frac{1}{R_3})}{R_1 \left( \frac{1}{R_2} + \frac{1}{R_3} \right)} - R_1 (\frac{1}{R_2} + \frac{1}{R_3}) \right) \\
I_{OpAmp} &= \frac{U_{g1} (\frac{1}{R_1} + \frac{1}{R_2}) + U_{g2} (\frac{1}{R_1} + \frac{1}{R_3})}{R_1 \left( \frac{1}{R_2} + \frac{1}{R_3} \right)} - R_1 (\frac{1}{R_2} + \frac{1}{R_3}) \\
I_{Ug1} &= -\frac{U_{g1}}{R_1} \\
\end{align*}
\]

The code for this example is shown in Figure 19.
and object-oriented programming. Julia is appropriate for scientific and numerical computing, but also supports general programming [14].

Julia differs in many ways from typical dynamic languages. One of the most significant departures is the fact that Julia Base and the standard library are written in Julia itself. Nonetheless, it has a rich language of types for constructing and describing objects which can optionally be used to make type declarations. Also, there is the ability to define behavior across many combinations of argument types via multiple dispatch and automatic generation of efficient, specialized code for different argument types [14].

Symbolics.jl package is a fast Computer Algebra System (CAS) built in Julia. It was made with an idea in mind that generic functions in Base Julia work with symbolic expressions. It has many features, but the most important for our software is solving and simplifying symbolic equations [15].

We used generic function `solve_for` for solving circuit equations. It currently only works with linear equations, which is a limitation in terms of solving electric circuits in the complex domain of Unilateral Laplace Transform. Also, we tried using function `simplify` for simplifying the results, but still it is not powerful enough and majority of examples was not simplified properly.

SymPyCAP is written in Python [5]. It uses package SymPy. The main differences between Sym- matics.jl and SymPy are that Symbolics.jl is written in Julia itself, so it is flexible and extendable and in terms of performance, SymPy is slow and developers began writing Symbolics.jl in order to build faster library [16].

From our experience, SymPy is more developed and Symbolics.jl lacks many features in comparison. SymPyCAP uses two functions for solving circuit equations `linsolve` and `solve` [17].

The first one is used for solving linear systems, and the other one for solving complex systems, which is useful for solving electric circuits in the complex domain of Unilateral Laplace Transform. SymPy also have function `simplify` [18]. It tries many kinds of simplifications, so it can be slow, but in the result, you will have the best possible solution.

SALECx is written in Maxima CAS [6]. It uses function `linsolve` for solving and simplifying the system equations. Maxima is a computer algebra system, which is derived from the Macsyma. It is implemented in the Lisp programming language.

SCAM is written in MATLAB. It uses MNA in the matrix form, and function `simplify` for getting system solutions [7]. The main difference between our program and SCAM is that programming language Julia, in which our software is developed, is open-source while MATLAB isn’t.

In Table 1 is shown comparison between used functions for solving equations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Program</th>
<th>Programming language</th>
<th>Programming package</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>solve_for</code></td>
<td>Julia CAP</td>
<td>Julia</td>
<td>Symbolics.jl</td>
<td>equation(s), variables</td>
<td>Any</td>
</tr>
<tr>
<td><code>linsolve</code></td>
<td>SymPyCAP</td>
<td>Python</td>
<td>SymPy</td>
<td>system, symbols</td>
<td><code>FiniteSet</code></td>
</tr>
<tr>
<td><code>solve</code></td>
<td>SymPyCAP</td>
<td>Python</td>
<td>SymPy</td>
<td><code>f (Expr, Poly, Equality, Relational expression, Boolean), symbols, flags</code></td>
<td>Varies according to input</td>
</tr>
<tr>
<td><code>linsolve</code></td>
<td>SALECx</td>
<td>Maxima CAS</td>
<td></td>
<td><code>[expr_1, ..., expr_m], [x_1, ..., x_n]</code></td>
<td>List of labels</td>
</tr>
<tr>
<td><code>simplify</code></td>
<td>SCAM</td>
<td>MATLAB</td>
<td></td>
<td>expression</td>
<td>expression</td>
</tr>
</tbody>
</table>

These two tables represent time (in seconds) and memory (in MB) usage of each program:

<table>
<thead>
<tr>
<th>Circuit</th>
<th>JuliaCAP</th>
<th>SALECx</th>
<th>SymPyCAP</th>
<th>SCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low pass filter</td>
<td>2.654</td>
<td>0.010</td>
<td>0.9</td>
<td>0.516</td>
</tr>
<tr>
<td>Riordan schematic</td>
<td>4.334</td>
<td>0.000</td>
<td>1</td>
<td>0.506</td>
</tr>
<tr>
<td>Subtractor</td>
<td>3.248</td>
<td>0.000</td>
<td>1</td>
<td>1.669</td>
</tr>
<tr>
<td>Adder</td>
<td>4.493</td>
<td>0.000</td>
<td>0.9</td>
<td>0.234</td>
</tr>
</tbody>
</table>
In general comparison, Julia is not as powerful as the other presented programming languages when it comes to symbolic analysis. It is still developing and in the future it may become equal to them. The good thing about Julia is that it is free open-source unlike some languages as MATLAB. Also, Julia has integrated support for some of the presented packages like SymPy, etc. Performance is one of the main characteristics of this language, so it is a lot faster than the others.

6. CONCLUSION

This paper has presented JuliaCAP, original free open-source software for solving linear time-invariant electric circuits, developed by Kristina Rajković, Gloria Đošlo, Nikola Radojević, Tamara Petković and Ivana Stanojević. JuliaCAP is capable of solving circuits in the complex domain of the Unilateral Laplace Transform or Phasor Transform.

JuliaCAP supports the following electrical elements: Resistor, Inductor (with or without initial conditions), Capacitor (with or without initial conditions), Ideal Voltage Source, Ideal Current Source, Ideal Operational Amplifier, Voltage Controlled Current Source, Voltage Controlled Voltage Source, Current Controlled Current Source, Current Controlled Voltage Source, Ideal Transformer, Linear Inductive Transformer, Two-port specified by ABCD-parameters, Impedance, Admittance, Transmission line.

Modified Nodal Analysis (MNA) has been chosen to set up circuit equations in JuliaCAP. The circuit equations are solved by the generic Julia Symbols, jl solve_for function.

JuliaCAP has been exemplified by several practical circuits and its performance, as a circuit solver, has been evaluated.

As it has been shown in this paper, there is plenty of room for improvements. JuliaCAP can be used to solve linear time-invariant electric circuits [6] even though its usage is not convenient for further work without manually simplifying expressions, because of the kernel function simplify.

We are looking forward to Julia’s advancements and will try to improve our module in the future.

Symbolic approximation belongs to our future research directives and currently it is not implemented in our software JuliaCAP [19].

The future version of JuliaCAP should benefit from inverse Laplace Transform when available in Julia Symbolics.jl. In that case JuliaCAP would be able to present the time-domain response of a circuit.

It was a great pleasure working in Julia. We acquired a lot of experience and knowledge. With this project we hope to encourage future generations to use the programming language Julia.

7. ACKNOWLEDGMENTS

We thank Prof. Dr. Dejan Tošić and Prof. Dr. Milka Potrebić for recommending this project to us and valuable discussions and advice regarding JuliaCAP project.

This work was supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia under Grant 2022/200103 and Higher Education Development Project „RF and microwave infrastructure for information and communication systems“ (RFMICS) 2021 - 2022.

REFERENCES


Table 3. Memory usage in megabytes

<table>
<thead>
<tr>
<th>Circuit</th>
<th>JuliaCAP</th>
<th>SALECx</th>
<th>SymPyCAP</th>
<th>SCAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low pass filter</td>
<td>55.523</td>
<td>0.192</td>
<td>4.5734</td>
<td>0.372</td>
</tr>
<tr>
<td>Riordan schematic</td>
<td>178.917</td>
<td>0.320</td>
<td>4.58</td>
<td>0.507</td>
</tr>
<tr>
<td>Subtractor</td>
<td>118.742</td>
<td>0.255</td>
<td>4.56</td>
<td>0.004</td>
</tr>
<tr>
<td>Adder</td>
<td>423.792</td>
<td>0.319</td>
<td>4.52</td>
<td>0.215</td>
</tr>
</tbody>
</table>
REZIME

SIMBOLIČKA ANALIZA LINEARNIH ELEKTRIČNIH KOLA KORISTEĆI MIT JULIA

Symbolic JS Biblioteku


Ključne reči: Simbolička analiza, linearna vremenski nepromenljiva električna kola, programski jezik Julia, simbolička simulacija, modifikovana nodalna analiza, jednostrana Laplasova transformacija