JuliaCAP reference manual

Authors

- Kristina Rajković
- Glorija Došlo
- Nikola Radojević
- Tamara Petković
- Ivana Stanojević

University of Belgrade-School of Electrical Engineering

Acknowledgement

We thank Prof. dr Dejan Tošić and Prof. dr Milka Potrebić for recommending this software and help with the project.

About JuliaCap

JuliaCAP is a program for solving linear time-invariant electric circuits.

Why Julia?

- Julia is free and open source.
- Julia has good call support to other languages so if you wish to make use of libraries which have already been written in C and Fortran, Julia makes it easy to do so in a simple and efficient way
- It excels at technical computing. Designed with data science in mind, Julia excels at numerical computing with a syntax that is great for math, with support for many numeric data types, and providing parallelism out of the box.

For monitoring this work we recommend Visual Studio Code and Jupyter Notebook.

Algorithm

Nodes

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.

Modified Nodal Analysis

MNA variables: node voltages and currents which cannot be expressed in terms of node voltages.
Node voltages are labeled by V_2, V_3, V_4,…
V_1=0 by default
Currents are labelled by I_1, I_2, I_3,…

Reserved symbols

Vg-MNA voltage variables
Ig-MNA current variables
w-frequency for the Phasor transform analysis
omega-another name for w

Electric circuit

Struct Graf represents an element of circuit and its parts are:

- tip : TipGrane, an enum that can have one of the following values:
  - R – Resistor
  - Vg – Ideal Voltage Generator
  - Ig – Ideal Current Generator
  - opAmp – Ideal Operational Amplifier
  - VCVS – Voltage Controlled Voltage Source
  - VCCS – Voltage Controlled Current Source
  - CCVS – Current Controlled Voltage Source
  - CCCS – Current Controlled Current Source
  - L – Inductor
  - C – Capacitor
  - IdealT – Ideal Transformer
  - InductiveT - Linear Inductive Transformer
  - ABCD – Two-port specified by ABCD-parameters(transmission parameters, chain parameters)
  - Z – Impedance

    - Y – Admittance
    - T – Transmission line, Phasor Transform,
- ime : String, arbitrary element name,
- cvor1 : vector of Integer, plus term,
- cvor2 : vector of Integer, negative term,
- param : Vector, value of element, type: float or string (string must be different from symbols reserved for element type, tip)

Methods available to the user:

- `noviGraf()`
- `dodajGranu(graph, Grana(tip, ime, cvor1, cvor2, param))`
- `resiKolo(graph; omega = "w")`
- `ispisi_rezultate(rezultat)`
- `ispisi_jednacine()`
- `ispisi_jednacine_latex()`
- `ispisi_rezultate_latex(rezultat)`
- `ispisi_specifican_rezultat_latex(rezultat, "U2")`
- `ispisi_specifican_rezultat(rezultat, "U2")`
- `ispisi_specifikacije_kola(graf)`

• Functions that make a circuit:

- `noviGraf()`
  Makes graph of circuit and returns it.
  Using: graf = noviGraf();

- `dodajGranu(graf, Grana(tip, ime, cvor1, cvor2, param))`
  Adds element (branch) in circuit (graph) .
  graph – graph (circuit) in which we add an element

  Note: if param is a variable it must be different from symbols reserved for the element type and it must be defined before using with:
  ```
  using Symbolics
  Rx = Symbolics.Sym{Num}(Symbol("R"))
  ```
  Symbol is the type of element.
  Additionally the new symbol can be multiplied by a rational number.
  Using:

One-port elements R, Z, Y, Ig, Vg:

```
dodajGranu(graf, Grana(R, "ime", [plusTerm], [minusTerm], [Rx]))
```

or

```
dodajGranu(graf, Grana(R, "ime", [plusTerm], [minusTerm], [5.0]))
```

One-port elements C, L:

```
dodajGranu(graf, Grana(L, "ime", [plusTerm], [minusTerm], ["L1"], [initial value of the current when t=0⁻]))
```

Two-port elements:

opAmp (Operational Amplifier):

```
dodajGranu(graf, Grana(opAmp, "ime", [nonInvertingTerm, invertingTerm], [SecondTerm], [„a"]))
```

ABCD (Two-port specified by ABCD-parameters(transmission parameters, chain parameters)):

```
dodajGranu(graf, JuliaCAP.Grana(ABCD, "ime", [plusFirstTerm, minusFirstTerm], [plusSecondTerm, minusSecondTerm], ["endsOfABCD"]))
```

Controlled sources (VCVS, VCCS, CCVS, CCCS) :

```
dodajGranu(graf, Grana(VCVS, "ime", [plusControlledTerm, minusControlledTerm], [plusControlledTerm, minusControlledTerm], ["g"]))
```

Transmission line T:

```
dodajGranu(graf, Grana(T, "ime", [plusSendingTerm, minusSendingTerm], [plusReceivingTerm, minusReceivingTerm], ["Zc", "tau"])
```

- `resiKolo(graf; omega = "w")`
  Solves circuit graph and replacing omega with number or string.

- Results display functions:
  - `ispisi_rezultate(rezultat)`

Writes result of circuit, node voltages and currents.

- `ispisi_jednacine()`
  Writes equations on the basis of which the solution is formed.

- `ispisi_jednacine_latex()`
  Writes equations in latex so they can be easy to check.

- `ispisi_rezultate_latex(rezultat)`
  Writes result in latex so it can be easy to check because sometimes equations can be disordered.

- `ispisi_specifican_rezultat_latex(rezultat, "U2")`
  Writes just part specified with second argument of result in latex.

- `ispisi_specifican_rezultat(rezultat, "U2")`
  Writes just part specified with second argument of result.

- `ispisi_specifikacije_kola(graf)`
  Writes number of nodes, elements, basic equations and variables.

Calling JuliaCAP

The use of JuliaCAP electric circuit solvers can be realized in several ways. Each of them involves including of a JuliaCap.jl file.
`include("JuliaCap.jl")`
After that first case would be using `using .JuliaCAP` which requires restarting Julia each time (slow way), but does not require using JuliaCAP. when calling function.
Second case has two variants, first one is with creating module and replacing all the code in that module, in module can be used `using .JuliaCAP`.
Second one is importing JuliaCAP code with `import .JuliaCAP` and this way requires specifying the file name before calling the function e.g.

`JuliaCAP.dodajGranu(...)`.

New circuit is made with `graf = noviGraf()`

using .JuliaCAP includes our library with source code.
Function noviGraf() makes a graph or an electric circuit in which we then add elements.
Adding elements is done by calling function dodajGranu(args) as shown above.
Then we can solve the circuit
`rezultat = resiKolo(graf; omega = "w")`
and use one writing functions to display solution.

The problems we had

As the Julia programming language is still in development, we were faced with a number of limitations when designing the project.
Some of them were the absence of Laplace transform and integration that greatly limited our solutions.
Also we noticed that order in which elements are added may have an effect on the solution of the circuit in sense that program will not work.
JuliaSymbolics is also limited in simplifying mathematical equations, so we had to manually simplify equations to solve some problems. But many solutions are still impractical for further work.
Function solve_for from Symbolics package does not return array type any, so we had to make a new variable res2, in which we put the solver solution.