Универзитет у Београду — Електротехнички факултет

Дипломски рад

из предмета

Теорија електричних кола

АУТОМАТИЗОВАНО ИЗВОЂЕЊЕ БРАНИНОВИХ ЈЕДНАЧИНА ПОМОЋУ СЛОБОДНОГ СОФТВЕРА SYMPY

Катарина Станковић
2018/0183
Одсек: Рачунарска техника и информатика

септембар 2022.

Садржај

1 Увод **2**
1.1 SymPy алат 2
1.2 Препоруке за праћење рада 2

2 Извођење Бранинових једначина **3**
2.1 Програмирање на основу правила 3
2.2 Извођење Бранинових једначина уз помоћ *rewrite rules* 3

3 Примена Бранинових једначина **11**
3.1 Заступљеност у софтверским алатима 11
3.2 Решени примери 11
3.2.1 Решавање кола помоћу SymPyCAP 11
3.2.2 Поступно решавање кола помоћу SymPy 13

4 Закључак **19**

A SymPy наредбе **20**

B Python наредбе **22**

C IPython наредбе **23**

Литература **24**

Поглавље 1

Увод

1.1 SymPy алат

SymPy [1] је бесплатна (*free open-source software*) Python [2][3] библиотека за симболичку математику. Овај алат тежи да постане потпун CAS (*computer algebra system*) пакет, написан искључиво у Python језику, који је један од најпопуларнијих програмских језика данашњице. Неколико предности SymPy алата који креатори наводе су зависност од само језика Python и једне његове библиотеке, самим тим и мало меморијско заузеће (нетипично за CAS пакете), као и могућност коришћења SymPy библиотеке у било ком окружењу које подржава Python.
Коришћењем SymPy библиотеке можемо извести Бранинове једначине [6].

1.2 Препоруке за праћење рада

За праћење овог рада препоручена је Anaconda [4], бесплатна дистрибуција (*free open-source distribution*) програмског језика Python. У оквиру ње се налази препоручено окружење Jupyter Notebook [5].
Верзије алата коришћене у овом раду су:

- Anaconda Navigator - 2.1.4
- conda - 4.12.0
- SymPy - 1.10.1
- Jupyter Notebook - 6.4.8

Поглавље 2

Извођење Браниновиx једначина

2.1 Програмирање на основу правила

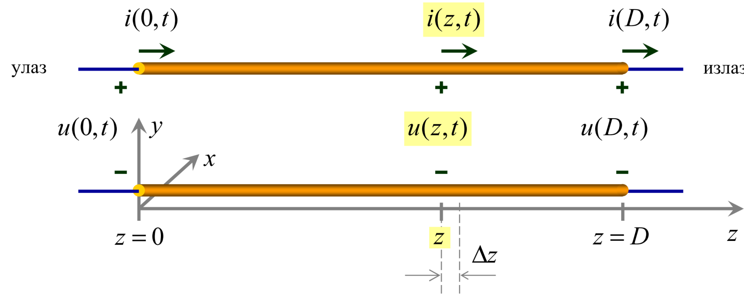
Програмирање на основу правила (енг. *Rule-based programming*) је програмирање базирано на механичком примењивању правила (енг. *rewrite rules*). На основу правила, замењујемо леву страну правила, десном страном. Природан пример правила би биле тригонометријске формуле, нпр. $\sin\theta = \dfrac{2\tan\theta/2}{1+\tan^2\theta/2}$. Да бисмо заменили леву страну десном у неком изразу, неопходно је пронаћи тај израз (енг. *pattern matching*). Овај концепт је имплементиран у оквиру SymPy.

2.2 Извођење Браниновиx једначина уз помоћ *rewrite rules*

Идеалан вод је униформан вод без губитака са хомогеним диелектриком и савршеним проводником на коме се простире трансверзалан електромагнетски талас.

Једначине телеграфичара идеалног вода су:

$$\left\{ \begin{array}{c} -\dfrac{\partial u(z,t)}{\partial z} = L'\dfrac{\partial i(z,t)}{\partial t} \\ -\dfrac{\partial i(z,t)}{\partial z} = C'\dfrac{\partial u(z,t)}{\partial t} \end{array} \right\}$$



Слика 2.2.1: Идеалан вод

а његове таласне једначине напона и струје су:

$$\left\{ \begin{array}{c} \dfrac{\partial^2 u(z,t)}{\partial z^2} = L'C' \dfrac{\partial^2 u(z,t)}{\partial t^2} \\ \dfrac{\partial^2 i(z,t)}{\partial z^2} = C'L' \dfrac{\partial^2 i(z,t)}{\partial t^2} \end{array} \right\}$$

Производ $C'L'$ можемо изразити преко подужног кашњења вода $T' = \sqrt{C'L'}$.

Применом Лапласове трансформације, сматрајући да нема почетне енергије, добијају се комплексне једначине телеграфичара:

$$\left\{ \begin{array}{c} -\dfrac{d\underline{U}(z)}{dz} = \underline{s}L'\underline{I}(z) \\ -\dfrac{d\underline{I}(z)}{dz} = \underline{s}C'\underline{U}(z) \end{array} \right\}$$

и комплексне таласне једначине:

$$\left\{ \begin{array}{c} \dfrac{d^2\underline{U}(z)}{dz^2} = (\underline{s}T')^2\underline{U}(z) \\ \dfrac{d^2\underline{I}(z)}{dz^2} = (\underline{s}T')^2\underline{I}(z) \end{array} \right\}$$

Почећемо извођење Бранинових једначина увођењем целе библиотеке SymPy, ради једноставности, као и иницијализовати такозвано *pretty printing*, који прилагођава изглед исписа окружењу. Ово смо постигли првим двема командама. Дефинисаћемо неке од симбола, користећи функцију `symbols`, које ћемо даље користити у извођењу. Приликом именовања симбола, захваљујући *pretty printing*, можемо да искористимо изразе који ће бити препознати као LaTeX ентитети [10]. Такав пример је `prime`. Ако не наведемо особине симбола, подразумевано је да представљају комплексне бројеве.

```
from sympy import *
init_printing()
A1, A2, Uz, Iz, Tprim, Lprim = symbols('A1, A2, Uz, Iz, Tprime, Lprime')
s,z = symbols('s,z')
```

Решавањем обичне хомогене линеарне диференцијалне једначине другог реда са константним коефицијентима добија се опште решење за комплексан напон и комплексну струју вода у пресеку на координати z.
Саставићемо леву и десну страну комплексне таласне једначине за напон користећи `diff` коме прослеђујемо, редом, функцију, параметар по коме тражимо извод, као и ред извода. Стварамо функцију тако што симбол U декларишемо као функцију параметра z, уз помоћ `Function`.

```
U = Function('U')(z)
levo = diff(U, z, 2)
levo
```

$$\frac{d^2}{dz^2}\mathrm{U}(z)$$

```
desno = (s*Tprim)**2*U
desno
```

$$T'^2 s^2 \mathrm{U}(z)$$

Спајамо леву и десну страну у једначину помоћу `Eq`. Функција ставља једнакост између ова два израза.

```
diferencijalna_jednacina = Eq(levo, desno)
diferencijalna_jednacina
```

$$\frac{d^2}{dz^2}U(z) = T'^2 s^2 U(z)$$

Решавање добијене диференцијалне једначине се ради функцијом `dsolve`.

```
resenje = dsolve(diferencijalna_jednacina)
resenje
```

$$U(z) = C_1 e^{-T'sz} + C_2 e^{T'sz}$$

Можемо дохватити десну страну израза уз помоћ атрибута `rhs` *(right-hand side)* објекта `resenje`, коме приступамо уз помоћ тачке. Овако смо добили објекат који представља израз и над којим можемо позивати методе, такође уз помоћ тачке. Заменићемо константе C_1 и C_2 константама A_1 и A_2, да не бисмо помешали са капацитивношћу. Проследићемо методи `subs` *(substitute)*

структуру под називом *dictionary*, која се састоји из парова кључ:вредност. Наша функција ће мењати у изразу сваки кључ својом наведеном вредношћу.

```
Uz = resenje.rhs
C1, C2 = symbols('C1, C2')
Uz = Uz.subs({C1:A1, C2:A2}) #preimenovanje radi izbegavanje zabune
Iz = -diff(Uz,z)/(s*Lprim) #kompleksna jednačina telegrafičara
Uz, Iz
```

$$\left(A_1e^{-T'sz} + A_2e^{T'sz},\ \frac{A_1T'se^{-T'sz} - A_2T'se^{T'sz}}{L's}\right)$$

Уводимо карактеристичну импедансу вода $Z_c = \frac{T'}{L'} = \sqrt{\frac{L'}{C'}}$. Можемо навести особину карактеристичне импедансе, а то је да је позитивна. Свака особина симбола, коју наведемо, се узима у обзир приликом симболичке анализе.

```
Zc = symbols('Z_c', positive = True)
```

Изразићемо L' преко карактеристичне импедансе вода и уврстити у Iz, користећи методу `subs`. Функција `expand`, као што и име сугерише, проширује израз. У нашем случају ће представити израз као суму.

```
Iz = expand(Iz.subs({Lprim:Zc*Tprim}))
```

Опште решење постаје:

```
Uz, Iz
```

$$\left(A_1e^{-T'sz} + A_2e^{T'sz},\ \frac{A_1e^{-T'sz}}{Z_c} - \frac{A_2e^{T'sz}}{Z_c}\right)$$

Узимајући координату $z = 0$ можемо добити комплексан напон и комплексну струју на почетку вода (U_0, I_0), а увођењем кашњење вода $\tau = T'd$ можемо изразити комплексан напон и комплексну струју на крају вода (U_d, I_d), али за неусаглашене смерове (као што су и приказани на слици 2.2.1).

```
U0, I0, Ud, Id = symbols('U0, I0, U_d, I_d')
d = symbols('d')
tau = symbols('tau')
```

Правимо једначине мењајући координату z и T' и постављамо атрибут `simultaneous` такав да забрани оптимизације израза пре него што се уврсте и z и T'.
Кренућемо са празном листом једначина, и додати на њен крај сваку једначину

користећи методу `append`. До сада смо испис радили тако што наведемо само име на крају блока кода, што је за окружење довољно. Овде ћемо исписати све једначине, па је потребно да искористимо `display`. Њој прослеђујемо по једну једначину из листе, којој приступамо преко петље, тачније, преко променљиве петље, `jednacina`. Није довољно искористити функцију `print`, јер тако не користимо моћ иницијализованог *pretty printing*.

```
jednacine = []
jednacine.append(Eq(U0, Uz.subs({z:0, Tprim:tau/d}, simultaneous=True)))
jednacine.append(Eq(I0, Iz.subs({z:0, Tprim:tau/d}, simultaneous=True)))
jednacine.append(Eq(Ud, Uz.subs({z:d, Tprim:tau/d}, simultaneous=True)))
jednacine.append(Eq(Id, Iz.subs({z:d, Tprim:tau/d}, simultaneous=True)))
for jednacina in jednacine:
    display(jednacina)
```

$$U_0 = A_1 + A_2$$

$$I_0 = \frac{A_1}{Z_c} - \frac{A_2}{Z_c}$$

$$U_d = A_1 e^{-\tau s} + A_2 e^{\tau s}$$

$$I_d = \frac{A_1 e^{-\tau s}}{Z_c} - \frac{A_2 e^{\tau s}}{Z_c}$$

Циљ нам је да добијемо једначине U_0 и U_d. Постижемо тај циљ тако што коефицијент A_1 извучемо из једне једначине за струју, нпр. за I_0, а коефицијент A_2 извучемо из система једначина за U_d и I_d. Када уврстимо добијене коефицијенте, обезбедили смо десну страну једначине U_0 без непознатих. На сличан начин радимо за једначину U_d.
Налазимо A_1 из друге наведене једначине тако што решимо ту једначину по променљивој A_1 користећи функцију `solveset`, док налазимо A_1 и A_2 из последње две једначине, које представљају систем линеарних једначина које решавамо функцијом `linsolve`. Обе функције враћају структуру под називом `Set`, односно скуп. Када решавамо једначину по једној променљивој, добићемо скуп са само једним чланом, а кад решавамо систем по више непознатих, опет добијемо скуп са једним елементом, где тај елемент представља уређену n-торку, где је n број непознатих. Да бисмо лакше приступили решењима једначина, претворићемо скуп у листу користећи истоимену функцију и приступити том једном јединoм елементу индексирањем са индексом 0. Фунцкија `simplify` користи хеуристику да би поједноставила израз.

```
A1_iz_J2 = simplify(list(solveset(jednacine[1], A1))[0])
A1A2_iz_J3J4 = list(linsolve(jednacine[2:], A1, A2))[0]
A1_iz_J2, A1A2_iz_J3J4
```

$$\left(A_2 + I_0 Z_c,\ \left(\frac{I_d Z_c e^{\tau s}}{2} + \frac{U_d e^{\tau s}}{2},\ \frac{\left(-I_d Z_c + U_d\right) e^{-\tau s}}{2}\right)\right)$$

Уврстићемо A_1 из друге једначине и A_2 извученог из система једначина, у прву једначину (U_0). Када функцији `subs` проследимо структуру по којој можемо итерирати, тј. која има редослед, као што је листа, обезбеђујемо редослед по којем се извршава замена. Овако како смо проследили, прво се замени свако појављивање коефицијента A_1, па тек онда A_2. Ово нам одговара јер A_1 је изражен преко A_2. Приступамо коефицијенту A_2 добијеном из система тако што индексирамо другом елементу претходно добијене листе решења система једначина. Функцијом `expand` ћемо се ослободити заграда.

```
U0jednacina = expand(jednacine[0].subs([(A1, A1_iz_J2), (A2, A1A2_iz_J3J4[1])]))
U0jednacina
```

$$U_0 = I_0 Z_c - I_d Z_c e^{-\tau s} + U_d e^{-\tau s}$$

Сада налазимо A_2 из четврте једначине тако што решимо ту једначину по променљивој A_2 користећи функцију `solveset`, док налазимо A_1 и A_2 из прве и друге једначине, које представљају систем линеарних једначина које решавамо функцијом `linsolve`. Редослед непознатих у решењу је одређен редоследом којим смо проследили непознате. Функција `solve` је застарела.

```
A2_iz_J4 = simplify(list(solveset(jednacine[3], A2))[0])
A1A2_iz_J1J2 = list(linsolve(jednacine[:2], A1, A2))[0]
A2_iz_J4, A1A2_iz_J1J2
```

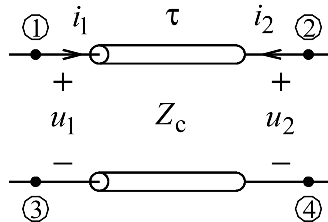
$$\left(\left(A_1 - I_d Z_c e^{\tau s}\right) e^{-2\tau s},\ \left(\frac{I_0 Z_c}{2} + \frac{U_0}{2},\ -\frac{I_0 Z_c}{2} + \frac{U_0}{2}\right)\right)$$

Уврстићемо A_2 из четврте једначине и A_1 извученог из система једначина, у трећу једначину (U_d). Опет прослеђујемо структуру по којој можемо итерирати, тј. листу n-торки, где је $n = 2$, и обезбеђујемо редослед по којем се извршава замена. Прво ће да се замени свако појављивање коефицијента A_2, па тек онда свако појављивање коефицијента A_1. Приступамо коефицијенту A_1 добијеном из система тако што индексирамо први елемент претходно добијене листе решења система једначина.

```
Udjednacina = expand(jednacine[2].subs([(A2, A2_iz_J4), (A1, A1A2_iz_J1J2[0])]))
Udjednacina
```

$U_d = I_0 Z_c e^{-\tau s} - I_d Z_c + U_0 e^{-\tau s}$

Усвајамо ознаке уобичајене за елементе са два приступа, за стандардне смерове:



Слика 2.2.2: Усаглашени смерови вода

```
U1, I1, U2, I2 = symbols('U1, I1, U2, I2')
```

Уврстићемо уобичајене ознаке у једначине U_0 и U_d, где је $I_1 = I_0$, $U_1 = U_0$, $I_2 = -I_d$ и $U_2 = U_d$, користећи речник, с обзиром да нам није важан редослед којим ћемо уврстити ове ознаке. Кренућемо са празном листом једначина, и додати на њен крај досад добијене једначине напона користећи методу `append`.

```
idealanVodLT = []
idealanVodLT.append(expand(U0jednacina.subs({I0:I1, Id:-I2, Ud:U2, U0:U1})))
idealanVodLT.append(expand(Udjednacina.subs({I0:I1, Id:-I2, Ud:U2, U0:U1})))
```

Сада можемо видети комплексне Бранинове једначине користећи петљу и произвољно названу променљиву петље `jednacina`:

```
for jednacina in idealanVodLT:
    display(jednacina)
```

$U_1 = Z_c I_1 + Z_c I_2 e^{-\tau s} + U_2 e^{-\tau s}$
$U_2 = Z_c I_1 e^{-\tau s} + Z_c I_2 + U_1 e^{-\tau s}$

Примењујући $\underline{U} e^{-\underline{s}\tau} \leftrightarrow u(t-\tau)$ (инверзна Лапласова трансформација), добијају се једначине идеалног вода.

```
e = Symbol('e')
```

Прво ћемо заменити експоненцијалну функцију $e^{-s\tau}$ симболом e функцијом `replace`, која проналази идентично написане изразе, не узимајући у обзир математичке еквиваленције, за разлику од `subs`. Пример би био:

```
z, x = symbols('z, x')
exp(2*x).subs(exp(x), z)
```

z^2

```
exp(2*x).replace(exp(x), z)
```

e^{2x}

Овако ћемо лакше уврстити трансформације. Да би се препознало грчко слово у изразу, користићемо LaTex назив тог слова \tau, а да би Python препознао да је коса црта део назива слова, користићемо *raw string* тако што ставимо слово r испред ниске.

```
for i in range(len(idealanVodLT)):
    idealanVodLT[i] = idealanVodLT[i].replace(exp(-tau*s), e).subs( \
                                    { \
                                    U1*e:Symbol(r'u_{1}(t-\tau)'), \
                                    U2*e:Symbol(r'u_{2}(t-\tau)'), \
                                    I1*e:Symbol(r'i_{1}(t-\tau)'), \
                                    I2*e:Symbol(r'i_{2}(t-\tau)'), \
                                    U1:Symbol('u_1(t)'), \
                                    U2:Symbol('u_2(t)'), \
                                    I1:Symbol('i_1(t)'), \
                                    I2:Symbol('i_2(t)') \
                                    })
```

```
display(idealanVodLT[0])
display(idealanVodLT[1])
```

$$u_1(t) = Z_c i_1(t) + Z_c i_2(t-\tau) + u_2(t-\tau)$$
$$u_2(t) = Z_c i_1(t-\tau) + Z_c i_2(t) + u_1(t-\tau)$$

Ове једначине су познате као Бранинове једначине [6].

Поглавље 3

Примена Бранинових једначина

3.1 Заступљеност у софтверским алатима

Браниновим једначинама можемо да моделујемо идеалне водове еквивалентним шемама [6].

У софтверском алату SPICE (*Simulation Program with Integrated Circuit Emphasis*), идеални вод се имплементира као временски закашњени контролисани генератори на основу Бранинових једначина [7].

Графичко програмско окружење заснован на MATLAB, Simulink, има више врста водова на располагању. Идеални вод је моделован Браниновим једначинама и пружа најефикаснију симулацију од понуђених врста водова [8].

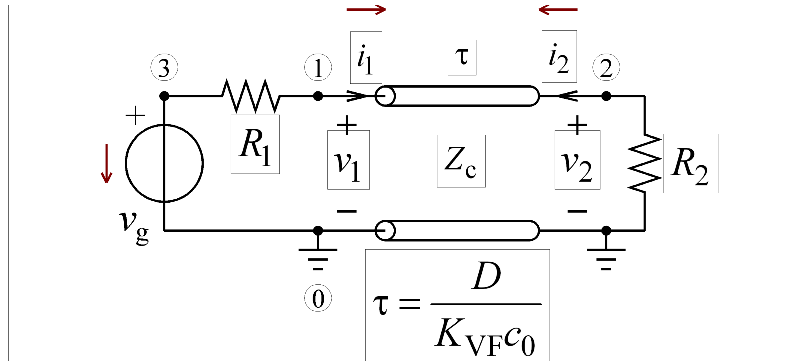
У *open-source* алату SymPyCAP, могуће је имати идеалне водове као елементе кола. Ако се приликом описа вода наведу карактеристична импеданса вода Z_c и кашњење вода τ, идеалан вод ће бити моделован Браниновим једначинама. [12].

3.2 Решени примери

3.2.1 Решавање кола помоћу SymPyCAP

Двоструко прилагођен идеалан вод је приказан на слици 3.2.1. Симболичка анализа, изведена у s-домену и приказана у наставку, потврђује да се ово коло понаша као линија кашњења. [12]

```
from symPyCAP import Circuit
from sympy import *
Zc = Symbol('Zc')
```



Слика 3.2.1: Двоструко прилагођен идеалан вод

```
tau = symbols('tau')
TLine_schematic = [
    ["V", "Vg", 3, 0],
    ["R", "R1", 3, 1],
    ["T", "T1", [1,0], [2,0], [Zc, tau]],
    ["R", "R2", 2, 0]
    ]
TLine_circuit = Circuit(TLine_schematic)
TLine_circuit.symPyCAP(replacement = {"R1" : Zc, "R2" : Zc})
TLine_circuit.print_specific_solutions()
```

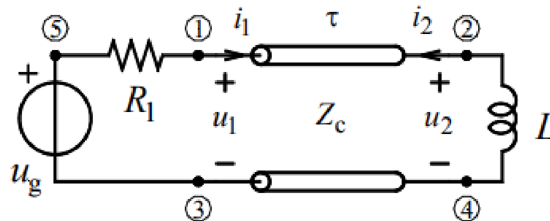
$V1 : Vg/2$
$V2 : Vg * exp(-s * tau)/2$
$V3 : Vg$
$IVg : -Vg/(2 * Zc)$
$IT1_1 : Vg/(2 * Zc)$
$IT1_2 : -Vg * exp(-s * tau)/(2 * Zc)$

```
TLine_circuit.get_specific_solutions()["V2"]
```

$$\frac{V_g e^{-s\tau}}{2}$$

3.2.2 Поступно решавање кола помоћу SymPy

Идеалан вод дужине D има примарне параметре C' и L'. Вод и калем су без почетне енергије, $R_1 = Z_c$, $u_g = U\theta(t)$, $U > 0$, L је познато. Одредити карактеристичну импедансу вода Z_c, кашњење вода τ, напон калема $u_2(t)$, његов домен, улазну струју $i_1(t)$ и њен домен. [13]



Слика 3.2.2: Двоструко прилагођен идеалан вод

Функција времена $u(t)$ је тренутна вредност и каузална функција времена, а $U(s)$ комплексна функција и Лапласов трансформат (комплексан представник). Заједно, $(u(t), U(s))$ чине Лапласов трансформациони пар.

Сажети систем једначина електричног кола је:

$$u_g(t) = R_1 i_1(t) + u_1(t)$$

$$0 = L\frac{di_2(t)}{dt} + u_2(t)$$

$$u_1(t) = Z_c i_1(t) + Z_c i_2(t-\tau) + u_2(t-\tau)$$

$$u_2(t) = Z_c i_2(t) + Z_c i_1(t-\tau) + u_1(t-\tau)$$

Последње две једначине су Бранинове једначине.

Карактеристична импеданса вода је $Z_c = \sqrt{\frac{L'}{C'}}$, а кашњење вода је $\tau = D\sqrt{L'C'}$

```
from sympy import *
init_printing()
```

Искористићемо изразе који ће бити препознати као LaTeX ентитети [10] као `prime`, а и доњу црту да бисмо имали следећи карактер написан у индексу. Такође ћемо нагласити алату особине индуктивности и карактеристичне импедансе, а то је да су позитивне.

```
Cprim, Lprim, R1, D, U = symbols('Cprime, Lprime, R_1, D, U')
tau, t, s = symbols('tau, t, s')
L = Symbol('L', positive = True)
Zc = Symbol('Z_c', positive = True)
u1 = Function('u_1')(t)
u2 = Function('u_2')(t)
i2 = Function('i_2')(t)
i1 = Function('i_1')(t)
ug = Function('u_g')(t)
```

Стварамо једначине уз помоћ функције `Eq`, а да бисмо исписали све једначине, користимо `display` за сваку од њих.

```
jednacina1 = Eq(ug, R1*i1 + u1)
jednacina2 = Eq(0, u2+L*diff(i2, t))
Braninovajednacina1 = Eq(u1, Zc*i1 + Zc* Function('i_2')(t-tau) + \
                                            Function('u_2')(t-tau))
Braninovajednacina2 = Eq(u2, Zc*i2 + Zc* Function('i_1')(t-tau) + \
                                            Function('u_1')(t-tau))
display(jednacina1)
display(jednacina2)
display(Braninovajednacina1)
display(Braninovajednacina2)
```

$$\mathrm{u_g}(t) = R_1 \,\mathrm{i_1}(t) + \mathrm{u_1}(t)$$

$$0 = L \frac{d}{dt} \mathrm{i_2}(t) + \mathrm{u_2}(t)$$

$$\mathrm{u_1}(t) = Z_c \,\mathrm{i_1}(t) + Z_c \,\mathrm{i_2}(t-\tau) + \mathrm{u_2}(t-\tau)$$

$$\mathrm{u_2}(t) = Z_c \,\mathrm{i_1}(t-\tau) + Z_c \,\mathrm{i_2}(t) + \mathrm{u_1}(t-\tau)$$

Применом унилатералне Лапласове трансформације добијамо комплексне сажете једначине електричног кола. Такође смо увели једнакост која је дата у тексту задатка, $R_1 = Z_c$, користећи `subs`.

$$\mathrm{U_g}(s) = Z_c \,\mathrm{I_1}(s) + \mathrm{U_1}(s)$$

$$0 = Ls \,\mathrm{I_2}(s) + \mathrm{U_2}(s)$$

$$\mathrm{U_1}(s) = Z_c \,\mathrm{I_1}(s) + Z_c \,\mathrm{I_2}(s) e^{-s\tau} + \mathrm{U_2}(s) e^{-s\tau}$$

$$\mathrm{U_2}(s) = Z_c \,\mathrm{I_1}(s) e^{-s\tau} + Z_c \,\mathrm{I_2}(s) + \mathrm{U_1}(s) e^{-s\tau}$$

```
U1 = Function('U_1')(s)
U2 = Function('U_2')(s)
I2 = Function('I_2')(s)
I1 = Function('I_1')(s)
Ug = Function('U_g')(s)
```

```
jednacina1LT = Eq(Ug, R1*I1 + U1).subs(R1, Zc)
jednacina2LT = Eq(0, U2 + L*s*I2)
Braninovajednacina1LT = Eq(U1, Zc*I1 + Zc*I2*exp(-s*tau) + U2*exp(-s*tau))
Braninovajednacina2LT = Eq(U2, Zc*I2 + Zc*I1*exp(-s*tau) + U1*exp(-s*tau))
display(jednacina1)
display(jednacina2)
display(Braninovajednacina1)
display(Braninovajednacina2)
```

$\mathrm{U_g}(s) = Z_c \mathrm{I_1}(s) + \mathrm{U_1}(s)$

$0 = Ls \mathrm{I_2}(s) + \mathrm{U_2}(s)$

$\mathrm{U_1}(s) = Z_c \mathrm{I_1}(s) + Z_c \mathrm{I_2}(s) e^{-s\tau} + \mathrm{U_2}(s) e^{-s\tau}$

$\mathrm{U_2}(s) = Z_c \mathrm{I_1}(s) e^{-s\tau} + Z_c \mathrm{I_2}(s) + \mathrm{U_1}(s) e^{-s\tau}$

Извлачимо $U_1(s)$ из прве једначине и $I_2(s)$ из друге једначине. Користимо `solveset` да бисмо решавањем једначина дошли до тражених променљивих. Одмах претварамо резултат из `Set` са једним елементом у листу дужине један, и приступамо том елементу индексирањем. Индексирање креће од нула.

```
U1_iz_j1 = list(solveset(jednacina1LT, U1))[0]
I2_iz_j2 = list(solveset(jednacina2LT, I2))[0]
```

Убацићемо их у другу Бранинову једначину да бисмо одредили комплексан напон калема $U_2(s)$. Прослеђујемо методи `subs` структуру по којој можемо итерирати, тј. листу парова, и обезбеђујемо редослед по којем се извршава замена. Овде нам редослед и није важан, па смо могли и структуром `dictionary`.

```
Braninovajednacina2LT = Braninovajednacina2LT.subs([(U1, U1_iz_j1), (I2, I2_iz_j2)])
Braninovajednacina2LT
```

$$\mathrm{U_2}(s) = Z_c \mathrm{I_1}(s) e^{-s\tau} + \left(-Z_c \mathrm{I_1}(s) + \mathrm{U_g}(s)\right) e^{-s\tau} - \frac{Z_c \mathrm{U_2}(s)}{Ls}$$

Можемо извући $U_2(s)$ решавањем по тој променљивој и да упростимо израз користећи `simplify`.

```
U2_vrednost = simplify(list(solveset(Braninovajednacina2LT, U2))[0])
U2_vrednost
```

$$\frac{Ls\,\mathrm{U_g}\,(s)e^{-s\tau}}{Ls+Z_c}$$

Кажемо да је $U_2(s) = V(s)\ e^{-s\tau}$, где је $V(s) = \dfrac{LU}{Ls+Z_c}$. Одзив (тренутна вредност), као функцију у временском домену можемо наћи из Лапласовог трансформационог пара. Користићемо теорему да су $v(t-T)$ и $V(s)e^{-sT}, T = const$ Лапласов трансформациони пар.

$u_2(t) = LT^{-1}(U_2(s)) = LT^{-1}(V(s)e^{-sT}) = v(t-T)$,

$v(t) = LT^{-1}(V(s)) = LT^{-1}(\dfrac{LU}{Ls+Z_c}) = Ue^{-\frac{Z_c t}{L}}\theta\,(t)$

Коначно:

$$u_2(t) = Ue^{-\frac{Z_c}{L}(t-\tau)}\theta\,(t-\tau)\,,\ -\infty < t < +\infty$$

Постоји истоимена уграђена функција `Heaviside` (видети Додатак А). Такође постоји и функција која ради Лапласову трансформацију функције коју прослеђујемо.

```
ug = U*Heaviside(t)
Ug_vrednost = laplace_transform(ug, t, s)[0]
Ug_vrednost
```

$$\frac{U}{s}$$

```
U2_vrednost = simplify(U2_vrednost.subs(Ug, Ug_vrednost))
U2_vrednost
```

$$\frac{LUe^{-s\tau}}{Ls+Z_c}$$

```
Vs = U2_vrednost/(exp(-s*tau))
Vs
```

$$\frac{LU}{Ls+Z_c}$$

Постоји и функција која ради инверзну Лапласову трансформацију функције коју прослеђујемо. Да нисмо на самом почетку дефинисали симболе Z_c и L као реалне, знатно би се разликовао резултат функције `inverse_laplace_transform`.

```
v = inverse_laplace_transform(Vs, s, t)
v
```

$$Ue^{-\frac{Z_c t}{L}}\theta\left(t\right)$$

Примењујемо $u_2(t) = LT^{-1}(U_2(s)) = LT^{-1}(V(s)e^{-sT}) = v(t-T)$:

```
Eq(u2, v.subs(t, t-tau))
```

$$\mathrm{u}_2\left(t\right) = Ue^{-\frac{Z_c(t-\tau)}{L}}\theta\left(t-\tau\right)$$

Искористићемо добијено $U_2(s)$ да добијемо $I_1(s)$ из прве Бранинове једначине. Уврстићемо и раније извучене $U_1(s)$ и $I_2(s)$. Поновићемо претходни поступак и уврстити променљиве и решити добијену једначину по $I_1(s)$.

```
Braninovajednacina1LT = Braninovajednacina1LT.subs([(U1, U1_iz_j1), \
                        (I2, I2_iz_j2), (U2, U2_vrednost), (Ug, Ug_vrednost)])
Braninovajednacina1LT
```

$$\frac{U}{s} - Z_c\mathrm{I}_1\left(s\right) = \frac{LUe^{-2s\tau}}{Ls+Z_c} - \frac{UZ_c e^{-2s\tau}}{s\left(Ls+Z_c\right)} + Z_c\mathrm{I}_1\left(s\right)$$

```
I1_vrednost = list(solveset(Braninovajednacina1LT, I1))[0]
I1_vrednost
```

$$\frac{-\frac{LUe^{-2s\tau}}{Ls+Z_c} + \frac{UZ_c e^{-2s\tau}}{s(Ls+Z_c)} + \frac{U}{s}}{2Z_c}$$

Да бисмо успешно извршили инверзну Лапласову трансформацију, поделићемо $I1(s)$ на сабирке, и на сваком појединачном сабирку урадити инверзну Лапласову трансформацију, па вратити их све у један израз. Да бисмо лакше дошли до сабирака, прво ћемо издвојити именилац и бројилац уз помоћ функције `fraction`. Она ће нам вратити пар (бројилац, именилац) коме можемо појединачно приступити индексирањем.

```
brojilac = fraction(I1_vrednost)[0]
brojilac
```

$$-\frac{LUe^{-2s\tau}}{Ls+Z_c} + \frac{UZ_c e^{-2s\tau}}{s\left(Ls+Z_c\right)} + \frac{U}{s}$$

```
imenilac = fraction(I1_vrednost)[1]
imenilac
```

$$2Z_c$$

Сада када смо обезбедили бројилац, до сваког сабирка можемо доћи уз помоћ атрибута израза под називом `args`.

```
fraction(I1_vrednost)[0].args
```

$$\left(\frac{U}{s}, \ -\frac{LUe^{-2s\tau}}{Ls+Z_c}, \ \frac{UZ_ce^{-2s\tau}}{s\left(Ls+Z_c\right)}\right)$$

Python омогућује истовремену вишеструку доделу, па у једној линији кода можемо издвојити и доделити сва три сабирка одређеним промењљивама. Концепт се зове *unpacking* [14].

```
prvi_sabirakLT, drugi_sabirakLT, treci_sabirakLT = fraction(I1_vrednost)[0].args
```

```
prvi_sabirak = inverse_laplace_transform(prvi_sabirakLT, s, t)/imenilac
prvi_sabirak
```

$$\frac{U\theta\left(t\right)}{2Z_c}$$

```
drugi_sabirak = \
    inverse_laplace_transform(drugi_sabirakLT/(exp(-2*s*tau)), s, t)/imenilac
drugi_sabirak
```

$$-\frac{Ue^{-\frac{Z_ct}{L}}\theta\left(t\right)}{2Z_c}$$

```
treci_sabirak = \
    inverse_laplace_transform(treci_sabirakLT/(exp(-2*s*tau)), s, t)/imenilac
treci_sabirak
```

$$\frac{U\theta\left(t\right)-Ue^{-\frac{Z_ct}{L}}\theta\left(t\right)}{2Z_c}$$

Приликом спајања назад сабирака и дефинисања коначног израза, можемо искористити функцију `collect` која тражи за аргументе израз и симболе (шаблон, израз) и која ће извући испред заграде управо тај шаблон/израз.

```
Eq(i1, collect(prvi_sabirak + drugi_sabirak.subs(t, t-2*tau) + \
                treci_sabirak.subs(t, t-2*tau), Heaviside(t-2*tau)))
```

$$\mathrm{i}_1\left(t\right)=\frac{U\theta\left(t\right)}{2Z_c}+\left(-\frac{Ue^{-\frac{Z_c\left(t-2\tau\right)}{L}}}{2Z_c}+\frac{U-Ue^{-\frac{Z_c\left(t-2\tau\right)}{L}}}{2Z_c}\right)\theta\left(t-2\tau\right)$$

Домен је сваки реалан тренутак времена.

Поглавље 4

Закључак

Извођење Бранинових једначина смо урадили уз помоћ библиотеке SymPy јер има уграђен механизам проналаска израза који одговара шаблону (енг. *pattern matching*). Тај механизам нам омогућава програмирање на основу правила (енг. *rule-based programming*) коришћењем правила једнакости (енг. *rewrite rules*) које ми дефинишемо. Таква правила смо дефинисали приликом извођења Бранинових једначина. Користили смо и друге могућности које нам библиотека SymPy пружа, као што су симболичко решавање система једначина и манипулације над обликом самих израза.

Видели смо да се Бранинове једначине користе у софтверским алатима. Показали смо како да, уз помоћ правила једнакости и библиотеке SymPy, сами решимо задатке са водовима користећи Бранинове једначине и како да решимо коло користећи *open-source* алат под називом SymPyCAP [12] који у позадини моделује водове Браниновим једначинама.

Симболичка анализа у решавању електричних кола, какву нам пружа библиотека SymPy, омогућава аутоматизовано решавање, као и проверу електричних кола, која би се, приликом ручног решавања, представила као напорна или склона грешкама за оног који их решава. Још већа апстракција јесте прављење алата користећи библиотеку SymPy која решава кола и може послужити у едукативне сврхе. Управо пример таквог алата јесте SymPyCAP [12].

Додатак А

SymPy наредбе

- `args` - args (14.04.2022.)
- `collect` - collect (14.04.2022.)
- `diff` - diff (29.03.2022.)
- `dsolve` - dsolve (29.03.2022.)
- `Eq` - Eq (29.03.2022.)
- `expand` - expand (29.03.2022.)
- `fraction` - fraction (14.04.2022.)
- `Function` - Function (29.03.2022.)
- `Heaviside` - Heaviside (11.04.2022.)
- `init_printing` - init_printing (29.03.2022.)
- `inverse_laplace_transform` - inverse_laplace_transform (11.04.2022.)
- `laplace_transform` - laplace_transform (11.04.2022.)
- `linsolve` - linsolve (29.03.2022.)
- `replace` - replace (29.03.2022.)
- `rhs` - rhs (29.03.2022.)
- `solveset` - solveset (29.03.2022.)

- `subs` - subs (29.03.2022.)
- `Symbol` - Symbol (29.03.2022.)
- `symbols` - symbols (29.03.2022.)
- `symplify` - symplify (29.03.2022.)

Додатак Б

Python наредбе

- `append` - append (29.03.2022.)
- `list` - list (29.03.2022.)

Додатак Ц

IPython наредбе

- `display` - display (29.03.2022.)

Литература

[1] Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. (2017) SymPy: symbolic computing in Python. PeerJ Computer Science 3:e103 https://doi.org/10.7717/peerj-cs.103

[2] Allen Downey, Think Python: How to Think Like a Computer Scientist, Green Tea Press, 2008.

[3] Paul Gerrard, Lean Python: Learn Just Enough Python to Build Useful Tools, Apress, 2016.

[4] Anaconda Software Distribution. Computer software. Vers. 2-2.4.0. Anaconda, Nov. 2016. Web. <https://anaconda.com>.

[5] Thomas Kluyver and Benjamin Ragan-Kelley and Fernando Pérez and Brian Granger and Matthias Bussonnier and Jonathan Frederic and Kyle Kelley and Jessica Hamrick and Jason Grout and Sylvain Corlay and Paul Ivanov and Damián Avila and Safia Abdalla and Carol Willing and Jupyter development team. Jupyter Notebooks - a publishing format for reproducible computational workflows. In Fernando Loizides and Birgit Scmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90, Netherlands, 2016. IOS Press.

[6] F. H. Branin, "Transient Analysis of Lossless Transmission Lines", Proceedings of the IEEE, Volume 55, Issue 11, Pages 2012-2013, Nov. 1967. DOI: 10.1109/PROC.1967.6033

[7] Paul R. Clayton, Analysis of Multiconductor Transmission Lines, 2/e, Wiley IEEE Press, 2008.

[8] Transmission Line. Retrieved from https://www.mathworks.com/help/physmod/sps/ref/transmissionline.html

[9] Anaconda Installation on Windows. Retrieved from https://docs.anaconda.com/anaconda/install/windows (05.01.2022.)

[10] Expressions that can convert into its LaTeX equivalent. Retrieved from https://github.com/sympy/sympy/blob/master/sympy/printing/latex.py (21.03.2022.)

[11] Roman Maeder, *Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering*, Cambridge University Press, 2000.

[12] Dodović, Matija, Bakić, Jelena, Stanković, Katarina, & Ilić, Nikola. (2021). SymPyCAP workshop (in Serbian: "SymPyCAP radionica"). Zenodo. https://doi.org/10.5281/zenodo.5554568

[13] Electric Circuit Theory. Retrieved from
http://tek.etf.rs/ (8.4.2022)

[14] Unpacking in Python. Retrieved from
https://docs.python.org/3/tutorial/datastructures.html#:~:text=This%20is%20called%2C%20appropriately%20enough%2C%20sequence%20unpacking%20and%20works%20for
(14.4.2022)