ELEKTROTEHNIŠKI VESTNIK 79(4): 149-152, 2012
ERK 2012 CONFERENCE ISSUE

Optimizacija vezij s knjižnico PyOPUS

Árpád Bűrmen[1]

Fakulteta za elektrotehniko
Univerza v Ljubljani
Tržaška cesta 25, SI-1000 Ljubljana, Slovenija
E-pošta: arpadb@fides.fe.uni-lj.si

PyOPUS Library for Automated Circuit Sizing

Circuit sizing is a process where one tries to pick the values of circuit components so that the circuit either satisfies or exceeds the design requirements. Developement of automated circuit-sizing software is a challenging task. It requires detailed knowledge of numerical algorithms, circuit simulation, parallel computing, data visualisation, and several other areas of expertise. Representation of design requirements involes non-trivial data structures which are tedious to implement and manage in low- and medium-level programming languages. Most of the computational time is spent by the simulator and only a small fraction of time for the surrounding circuit-sizing algorithms. This makes a scripting language with a wide selection of extensions an optimal choice for their implementation. The PyOPUS library is a collection of these algorithms implemented in Python. The internal structure of the library is presented along with an example of circuit optimization.

1 Uvod

Postopek določanja vrednosti parametrov elementov vezja (v nadaljevanju načrtovalskih parametrov vezja) temelji na optimizacijskih postopkih. Naloga optimizacijskega postopka je, da generira vrednosti načrtovalskih parametrov. S pomočjo simulatorja nato ovrednotimo vezje, ki ga ti parametri določajo in iz rezultatov simulacije določimo vrednost kriterijske funkcije (KF). Ta številsko ovrednoti kakovost vezja. Optimizacijski postopek na osnovi izračunane vrednosti KF določi nove vrednosti načrtovalskih parametrov. Opisani postopek se ponavlja, dokler ne najdemo vezja ki zadošča načrtovalskim zahtevam, oziroma optimizacijski potopek ne „obupa“ nad problemom. Slednje je običajno posledica previsokih načrtovalskih zahtev, včasih pa tudi izbire neprimernega optimizacijskega postopka.

Zaradi precejšnjega števila simulatorjev vezij, ki so lahko namensko tudi zelo ozko specializirani (npr. za analize harmonskega ravnovesja), se ne moremo omejiti na zgolj en simulator. Posledično moramo povezavo med optimizacijskim postopkom in simulatorjem zastaviti čim bolj modularno. Rezultati simulacije so ponavadi poteki veličin v vezju v časovnem ali frekvenčnem prostoru. Iz teh potekov moramo izluščiti lastnosti vezja, kot so naprimer pasovna širina, ojačenje, fazna varnost, dvižni čas, ipd. Načrtovalske zahteve so v splošnem neenačbe, ki postavijo omejitve za vrednosti lastnosti vezja (npr. ojačenje mora biti večje od $80\mathrm{dB}$, dvižni čas mora biti manjši od $20\mathrm{ns}$). Vrednost KF odraža vrednosti lastnosti vezja v primerjavi z načrtovalskimi zahtevami. Večje vrednosti KF ustrezajo vezjem, ki manj uspešno izpolnjujejo načrtovalske zahteve [1]. Vidimo, da lahko celoten postopek od spodaj navzgor razdelimo na štiri sloje: 1. simulacija, 2. določanje lastnosti vezja, 3. določanje kriterijske funkcije, 4. optimizacijski postopek.

En tek optimizacijskega postopka lahko vključuje več deset tisoč simulacij. Ker je dolžina ene simulacije reda velikosti sekund ali več, lahko en tek optimizacijskega postopka traja ure ali celo dneve. Čas računanja skrajšamo z uporabo vzporednih optimizacijskih postopkov, kjer delo razdelimo med več vzporedno računajočih procesorskih enot. Pri tem simulacija ostane nedeljivo opravilo in se še vedno izvaja v celoti zgolj na eni procesorski enoti. Za izvedbo vzporednih postopkov sta na voljo dve knjižnici, ki olajšata programiranje vzporednih postopkov: PVM [2] in MPI [3].

Po končani optimizaciji (pogosto pa tudi med optimizacijskim postopkom) je zelo koristno imeti vpogled v lastnosti vezja. Najkrajša pot do njega vodi preko grafične predstavitve lastnosti vezja. Obstaja cela vrsta programov in knjižnic za vizualizacijo rezultatov simulacij. Sama izvedba prikazovanja rezultatov tako ni posebej problematična, saj njen težji del predstavlja izvedba povezave med optimizacijskim postopkom in knjižnico za vizualizacijo rezultatov.

Nenazadnje ne smemo pozabiti, da vsi opisani postopki predstavljajo le temelj programske opreme za optimizacijo vezij. Da sam postopek približamo inženirjem in ga naredimo uporabnega tudi v primeru, ko ta ni vešč programiranja, potrebujemo učinkovit in enostaven uporabniški vmesnik. Za gradnjo uporabniških vmesnikov je na voljo veliko število knjižnic (npr. wxWidgets [4], Qt [5]).

2 Izbira platforme

Sama izvedba celotne programske opreme za optimizacijo vezij torej zahteva združevanje velikega števila knjižnic, ki so specilaizirane za posamezne naloge. Ponavadi se tovrstnih nalog lotimo v programskem jeziku C ali C++. Na žalost sta oba jezika statična in zahtevata, da ob vsaki spremembi prevedemo in ponovno povežemo celoten program. Njuna glavna prednost je velika hitrot izvajanja programa. Vendar ta prednost izgine, če upoštevamo dejstvo, da $90\% - 95\%$ procesorskega časa porabimo za simulacijo vezja. Preostalih $5\% - 10\%$ odpade na prej omenjene knjižnice in kodo, ki jih povezuje. Tudi če ta del izvedemo na zelo učinkovit način (C/C++), celoten postopek ne bo bistveno hitrejši.

Za vso povezovalno kodo med knjižnicami je mnogo bolj primeren intepretiran jezik, ki lahko nudi precej večjo fleksibilnost, kot C/C++. Tovrstnih jezikov je veliko. Če upoštevamo še numerično plat vseh postopkov, se izbor precej zoži. Postavimo še nekaj dodatnih zahtev: 1. jezik mora biti objektno orientiran, da lahko lažje zgradimo modularne programe, 2. imeti mora širok nabor knjižnic in 3. omogočati mora enostavno uporabo zunanjih prevedenih knjižnic, ki se uporabljajo pri programiranju v jezikih C/C++.

Vse opisane zahteve izpolnjuje programski jezik Python [6]. Python je na voljo tako za sisteme Windows, kot tudi Linux. Za numerično matematiko sta na voljo knjižnici NumPy in SciPy [7], ki ponujata primerljive možnosti, kot programski paket MATLAB [8]. Za prikazovanje rezultatov lahko uporabimo številne grafične knjižnice, od katerih se odločimo za kombinacijo MatPlotLib-a [9], in na WxWindows-ih temelječe knjižnice WxPython [10], ki služi gradnji uporabniških vmesnikov.

Programi v Pythonu so kompaktni, a kljub temu enostravno berljivi zaradi zahtev, ki jih postavlja slovnica jezika. Poleg objektno orientiranega programiranja Python ponuja tudi možnost organizacije objektov v module. Slednje je skoraj nujno, saj številne knjižnice s sabo prinesejo veliko število razredov in funkcij, katerih imena se včasih tudi prekrivajo.

3 Zgradba knjižnice

Knjižnica PyOPUS je razdeljena v 7 modulov. Modul `pyopus.simulator` skrbi za povezavo med simulatorjem in programom v Pythonu. Simulacijo opišemo s podatkovno strukturo, ki je sestavljena in ene ali več nalog (angl. jobs). Vsaka naloga podaja seznam datotek, ki skupaj opišejo vezje. Vse lastnosti elementov, ki jih želimo spreminjati preko programa v Pythonu (parametri), morajo biti parametrizirane (namesto konkretnih vrednosti imamo v opisu vezja le imena parametrov). Kot del naloge podamo tudi vrednosti parametrov vezja, nastavitve simulatorja (npr. število iteracij in natančnost), seznam rezultatov, ki jih želimo od simulatorja, in ukaz, ki izvede simulacijo. Nabor razpoložljivih ukazov je odvisen od simulatorja in analiz, ki jih ta ponuja. Seznam nalog podamo simulatorskemu objektu, ki te naloge optimalno razvrsti, pripravi vhodne datoteke za simulacijo, požene simulator in zbere rezultate. Slednje lahko po končani simulaciji preberemo iz simulatorskega objekta.

Trenutno sta podprta SPICE OPUS [11] in HSPICE [12]. Dodajanje novih simulatorjev v PyOPUS ni pretirano zahtevno, saj moramo definirati le nov simulatorski razred, ki opiše posebnosti simulatorja. Opis simulatorja HSPICE tako obsega dobrih 700 vrstice kode napisane v jeziku Python. Nekoliko več dela imamo, če tip izhodne datoteke, ki jo ustvari simulator ni podprt v PyOPUS-u. V tem primeru moramo napisati funkcijo za uvoz podatkov v Python, ki pa mora biti zaradi učinkovitosti napisana v jeziku C. Modul za uvoz izhodnih datotek HSPICE-a obsega 800 vrstic programa v jeziku C. Ker se izhodni formati med simulatorji ponavljajo, je za pričakovati, da bo prej ali slej vsak simulator pokrit z vsaj enim izhodnim formatom, ki ga podpira PyOPUS.

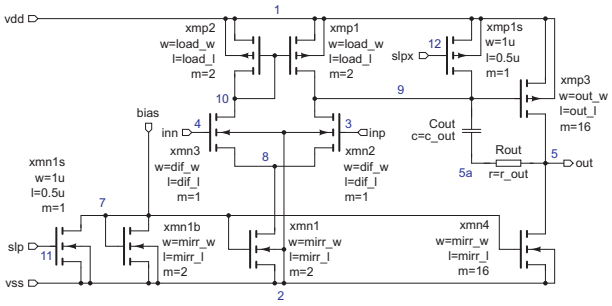
Modul `pyopus.evaluator.performance` skrbi za določanje lastnosti vezja. Lastnosti, ki nas zanimajo, opišemo v podatkovni strukturi. Ta najprej našteje vse simulatorje, ki so potrebni za ovrednotenje vezja, skupaj s pripadajočimi seznami datotek s parametriziranimi opisi vezja. Sledi seznam analiz in vogalnih točk (ang. corners), v katerih bodo izvedene analize. Pri tem je za vsako analizo potrebno navesti simulator, ki jo bo izvajal. Sledi seznam lastnosti vezja (ang. permormance measures), ki jih želimo določiti. Za vsako lastnost moramo navesti analizo, katere rezulati predstavljajo izhodišče za določanje njene vrednosti, in vogalne točke, v katerih nas ta vrednost zanima. Nazadnje navedemo še izraz s pomočjo katerega se lastnost izračuna iz rezultatov analize. Pri tem nam je v pomoč modul pomožnih funkcij `pyopus.evaluator.measure`. Z njihovo pomočjo opišemo določanje lastnosti vezja podobno kot na sodobnih digitalnih osciloskopih.

Osrednji razred v modulu se imenuje `PerformanceEvaluator`. Ta na osnovi zgoraj opisanih podatkovnih struktur sam poskrbi za pripravo simulatorskih objektov (iz modula `pyopus.simulator`) in pripadajočih seznamov nalog. Ob klicu mu podamo vrednosti parametrov vezja. Objekt nato izvede vse simulacije in iz njihovih rezultatov izračuna lastnosti vezja, ki jih vrne v tabeli. Prvi indeks v tej tabeli predstavlja ime lastnosti vezja, drugi indeks pa ime vogalne točke.

Če razvijamo večciljne optimizacijske postopke, se na tej točki prične razvoj postopka, nadzor pa prevzamejo objekti, ki jih doda razivijalec. Če pa je naš namen enociljna optimizacija, pa moramo združiti vse dobljene lastnosti vezja v eno samo vrednost - kriterijsko funkcijo. Modul `pyopus.evaluator.cost` ponuja razrede s pomočjo katerih to izvedemo na enostaven način z uporabo kazenskih funkcij [1]. Osrednji razred modula je `CostEvaluator`. Ob klicu njegovega konstruktorja podamo seznam m lastnosti vezja s pripadajočimi kazenskimi funkcijami in seznam n parametrov vezja. Dobljen objekt se obnaša kot funkcija $\mathbb{R}^n \rightarrow \mathbb{R}$, ki jo lahko uporabimo v vlogi KF poljubnega optimizacijskega postopka.

Slednji so zbrani v modulu `pyopus.optimizer`. Poudarek je na direktnih postopkih, ki ne potrebujejo gradientov KF. Poleg optimizacijskih postopkov modul vsebuje tudi nabore matematičnih testnih funkcij za lokalne in globalne optimizacijsek postopke, ki pridejo prav pri razvoju novih postopkov. Nekateri postopki lahko izkoriščajo večprocesorske sisteme za pospešitev optimizacije. Podporo za vzporedno računanje nudi modul `pyopus.parallel`, ki je zasnovan na sistemu PVM [2], v bližnji prihodnosti pa bo dodana tudi podpora za knjižnico MPI [3]. Ostali moduli PyOPUS-a večinoma uporabljajo le modul `pyopus.parallel.evtdrvms`. Ta omogoča enostavno programiranje vzporednih postopkov, kjer je en proces nadrejen vsem ostalim (ang. master-slave). Postopke opišemo z naborom sporočil (ang. messages) in odzivov na prejeta sporočila (ang. event hadlers). Modul `pyopus.parallel.evtdrvms` sam poskrbi za zagon vzporednih procesov in vzpostavitev njihovega začetnega stanja.

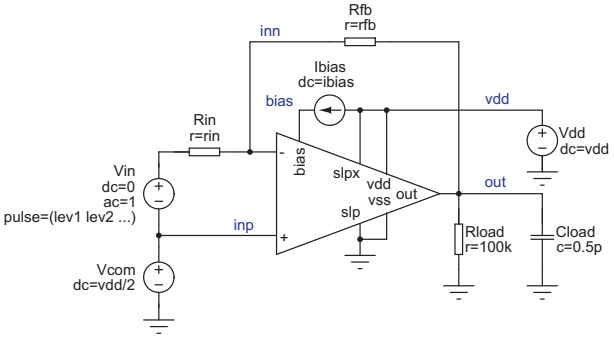
Modul `pyopus.wxmplplot` ponuja podporo za risanje 2D grafov, ki temelji na uporabi knjižnic MatPlotLib [9] in wxPython [10]. Programski vmesnik je zasnovan podobno kot v paketu MATLAB. Za enostaven grafičen prikaz lastnosti vezja skrbi modul `pyopus.visual` z vtičnikom za optimizacijske postopke, ki prikazuje izbrane rezultate simulacije vezja kar med postopkom optimizacije. Ta vtičnik pa ni edini. V drugih modulih so na voljo vtičniki za izpisovanje in shranjevanje rezultatov, ter ustavitev optimizacijskih postopkov ob izpolnjevanju določenih pogojev. Uporabniki knjižnice lahko na enostaven način dodajajo tudi svoje vtičnike.



Slika 1: Vezje operacijskega ojačevalnika.
Figure 1: Two stage opamp schematic.

4 Primer

Uporabo knjižnice bomo ilustrirali s primerom optimizacije enosmerne karakteristike operacijskega ojačevalnika na sliki 1, ki je opisan v datoteki `opamp.inc`. Ojačevalnik vežemo v testno vezje (slika 2, datoteka `topdc.inc`), modeli tranzistorjev pa se nahajajo v knjižnični datoteki `cmos180n.lib`. Opis problema v jeziku Python začnemo z izbiro simulatorja, seznamom vhodnih datotek in vrednostmi tistih parametrov vezja, ki se ne spreminjajo med optimizacijo.



Slika 2: Testno vezje za operacijski ojačevalnik.
Figure 2: Opamp testbench circuit.

```
heads = {
 'opus': {
  'simulator': 'SpiceOpus',
  'settings': { 'debug': 0 },
  'moddefs': {
   'def': { 'file': 'opamp.inc' },
   'tb':  { 'file': 'topdc.inc' },
   'mos_tm': { 'file': 'cmos180n.lib',
               'section': 'tm' },
  },
  'options': { 'method': 'trap' },
  'params': {
   'lev1': 0.0, 'lev2': 0.5, 'tstart': 1e-9,
   'tr': 1e-9, 'tf': 1e-9, 'pw': 500e-9
}}}
```

Sledijo opis analiz, vogalnih točk in lastnosti vezja, ki nas zanimajo (največje ojačenje v enosmerni karakteristiki in območje izhodnih napetosti, znotraj katerih je ojačenje nad $1/\sqrt{2}$ največje vrednosti).

```
analyses = {
 'dc': {
  'head': 'opus', 'modules': [ 'def', 'tb' ],
  'params': { 'rin': 1e6, 'rfb': 1e6 },
  'saves': [ ],
  'command': "dc(-2.0, 2.0, 'lin', 100, 'vin', 'dc')"
}}

corners = {
  'nominal': {
   'modules': [ 'mos_tm' ],
   'params': { 'temperature': 25, 'vdd': 1.8 }
}}

measures = {
 'gain': {
  'analysis': 'dc', 'corners': [ 'nominal' ],
  'expression':
  "m.gain2dB(m.DCgain(v('out'), v('inp','inn')))"
 },
 'swing': {
  'analysis': 'dc', 'corners': [ 'nominal' ],
  'expression':
  "m.DCswingAtGain(v('out'),v('inp','inn'),0.71,'out')"
}}
```

Nazadnje opišemo še optimizacijske parametre in njihove omejitve ter načrtovalske zahteve, na osnovi katerih se računa kriterijska funkcija. Zahtevamo vsaj $70\mathrm{dB}$ ojačenja, ki ne sme pasti pod $1/\sqrt{2}$ največje vrednosti v območju izhodnih napetosti širine vsaj $1{,}5\mathrm{V}$. Kršitve načrtovaskih zahtev kaznujemo z utežjo 1, presežene zahteve pa nagradimo z utežjo $0{,}001$.

```
# Optimizacijski parametri, začetne vrednosti in meje
w_set = { 'init': 1e-005, 'step': 0.01e-6,
          'lo': 1e-6, 'hi': 95e-6 }
l_set = { 'init': 5e-007, 'step': 0.01e-6,
          'lo': 0.18e-6, 'hi': 4e-6 }
costInput = {
 'mirr_w': w_set, 'mirr_l': l_set,
 'out_w':  w_set, 'out_l':  l_set,
 'load_w': w_set, 'load_l': l_set,
 'dif_w':  w_set, 'dif_l':  l_set
}

# Vrstni red optimizacijskih parametrov
inOrder = [ 'mirr_w', 'mirr_l', 'out_w', 'out_l',
            'load_w', 'load_l', 'dif_w', 'dif_l' ]

# Kriterijska funkcija
costDefinition = [
 { 'measure': 'gain', 'goal': 'MNabove(70)',
   'shape': 'CSlinear2(1.0,0.001)' }
 { 'measure': 'swing', 'goal': 'MNabove(1.5)',
   'shape': 'CSlinear2(1.0,0.001)' }
]
```

Optimizacijo poženemo s kratkim programčkom.

```
# Naloži komponente iz knjižnice
from pyopus.evaluator.performance import \
  PerformanceEvaluator
from pyopus.evaluator.cost import \
  CostEvaluator, parameterSetup
from pyopus.optimizer import optimizerClass

# Ustvarimo objekt, ki računa lastnosti vezja.
pe=PerformanceEvaluator(heads, analyses, corners,
                        measures, debug=0)
# Ustvarimo kriterijsko funkcijo.
ce=CostEvaluator(pe, inOrder, costDefinition, debug=0)
# Iz opisa opt. parametrov zgradimo vektorje.
(xi, xl, xh, xs) = parameterSetup(inOrder, costInput)

# Metoda Hooke-Jeeves.
cls=optimizerClass("HookeJeeves")
# Optimizator, ki minimizira ce v mejah xl, xh.
opt=cls(ce, xlo=xl, xhi=xh, maxiter=1000)
# Začetna točka.
opt.reset(xi)
# Namestimo vtičnika za izpisovanje rezultatov in
# ustavitev optimizacije ob izponjenih zahtevah.
opt.installPlugin(ce.getReporter())
opt.installPlugin(ce.getStopWhenAllSatisfied())

# Zaženemo optimizacijo.
opt.run()
print("Resitev najdena po %d izracunih KF." %
      opt.bestIter)

# Po končani optimizaciji počistimo začasne datoteke.
pe.finalize()
```

Optimizacijski postopek izračuna kriterijsko funkcijo 334-krat. Na računalniku s procesorjem iz družine Core i7 s sistemsko uro 3.2GHz se zaključi po 21s.

5 Zaključek

Predstavili smo knjižnico PyOPUS, ki omogoča, da opišemo in rešimo parametrične optimizacijske probleme pri načrtovanju vezij na hiter in enostaven način. Navedli smo razloge za izbiro programskega jezika Python in spremljajočih knjižnic ter prikazali uporabo knjižnice PyOPUS na preprostem primeru optimizacije integriranega vezja. Knjižnica je brezplačno dostopna na svetovnem spletu na naslovu `http://fides.fe.uni-lj.si/pyopus`. Trenutno sta podpri platformi Windows in Linux.

V prihodnosti nameravamo razvoj knjižnice usmeriti v podporo večjega števila simulatorjev in knjižnice MPI ter razvoj višjenivojskih postopkov avtomatizacije načrtovanja vezij, kot je naprimer načrtovanje robustnih vezij, večkriterijska optimizacija in modularizirano načrtovanje kompleksih analognih vezij.

6 Zahvala

Raziskavo je sofinancirala Agencija za raziskovalno dejavnost Republike Slovenije v okviru programa P2-0246 - Algoritmi in optimizacijski postopki v telekomunikacijah.

Literatura

[1] Á. Bűrmen, D. Strle, F. Bratkovič, J. Puhan, I. Fajfar, and T. Tuma. Automated Robust Design and Optimization of Integrated Circuits by Means of Penalty Functions. International Journal of Electronics and Communications, 57: 47–56, 2003.

[2] A. Geist et.al. PVM: Parallel Virtual Machine. MIT Press, 1994.

[3] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, Version 2.2. High Performance Computing Center Stuttgart, 2009.

[4] http://www.wxwidgets.org, 2012.

[5] http://qt.nokia.com/products, 2012.

[6] http://www.python.org, 2012.

[7] http://www.scipy.org, 2012.

[8] C.B. Moler. Numerical Computing With MATLAB. Cambridge University Press, 2004.

[9] http://matplotlib.sourceforge.net, 2012.

[10] http://wxpython.org, 2012.

[11] T. Tuma, Á. Bűrmen. Circuit Simulation With SPICE OPUS: Theory and Practice. Birkhäuser, 2009.

[12] HSPICE Simulation and Analysis User Guide. Synopsys, 2006.