## Az áramkör és a SUNRED összekapcsolása

Az áramkör önálló, nem SUNRED specifikus, így más megoldó is használható.

Egy csomóponthoz akárhonnan akárhány alkatrész kapcsolódhat, a csomópont adatai egy példányban léteznek.

A SUNRED új változata viszi magával a cellaközi csomópontok indexét (I). Az egyszerűség kedvéért minden SUNRED cella viszi. A cellák egyesítésekor egyszerűen szét kell másolni ugyanabban a sorrendben, ahogy az Y-t és a J-t. A JB kiszámításakor a SUNRED lenyúl a csomópontokba, és kiolvassa a megfelelő hibaáramokat (d), és hozzáadja JB-hez, így pontosan egyszer lesznek hozzáadva. A BW alatt pedig a BW ugyanebben a fázisban kiírja a csomópontba a hibafeszültséget (v). Így biztosan csak egyszer lesz írva a v, nem kell, hogy atomic legyen. Centroid csomópontnál nincs d olvasás és v írás.

Y, J, I

Y, J, I

## Csomópont alapértelmezett tere

A cella (subcircuit) definíciójakor a belső csomópontokra lehet definiálni, hogy az adott csomópont melyik alapértelmezett értéket kapja (elektromos, termikus, stb.). Mivel a vezérlési célból kivezetett csomópont a külső csomópontok között van definiálva, a külső csomópontokra is lehet definiálni ugyanezt. Emiatt elvi akadálya nincs annak, hogy bármelyik külső csomópontra is legyen definiálható az alapértelmezett index. Ha egy csomópontra több helyen is definiálva van alapértelmezett index, és ezek ellentmondanak egymásnak, akkor ezek közül bármelyik lehet a csomópont alapértelmezett indexe. Ha sehol sincs definiálva, akkor az alapértelmezett index 0 lesz, így, ha ez jó, akkor nem kell definiálni. (ModelSubCircuit::forcedNodes)

## AC analízis tranziens közben

Elvileg lehetséges AC analízist futtatni a tranziens analízis egyes lépéseinél, mivel megmarad a tranziens lépés eredménye, hiszen kell a nemlineáris komponensek inicializálásához. Ekkor a time emlékszik, hogy honnan kell tovább folytatni a tranzienst, a dt viszont nem kell az AC-hez, ezért a dt és az AC frekvencia lehet ugyanaz a változó (GND::dt\_or\_freq).

## Iterációk

Új lépés

* ha volt strukturális változás, akkor buildOrReplace
* A szimulációs módtól függő mátrixok és vektorok lefoglalása
* ha AC, akkor emlékezni kell, hogy volt-e strukturális változás a DC/transient során az előző AC után, vagy ha nem volt AC. Ekkor létre kell hozni az AC node-okat.
* init:
  + az egész lépésre állandó paraméterek beállítása
  + v

DC lépés

* Kontrollerek számítása, ha vannak
* Komponensek értékének számítása (vezérlésre használható)
* Áramok számítása
* Amíg a hiba nem elég kicsi:
  + Hibafeszültségek kiszámítása, a subcircuit-re beállított szimulációs módtól függően:
    - tartalmazott komponensek jakobi mátrixainak és hibaáram vektorainak kiszámítása
    - saját jakobi mátrix és hibaáram vektor számítása a külső csomópontokra
      * full mátrix: jakobi mátrix és hibaáram vektor létrehozása a cellára és mátrixműveletekkel a külső csomópontokra redukálás: forwsubs
      * sunred: a sunred vezérlő adatok alapján forwsubs a külső csomópontokig
      * multigrid: a vezérlésnek megfelelően hibaáramok restrikciója a durvább gridre rekurzívan a legdurvább gridig, multigridnél nem keletkezik jakobi mátrix és hibaáram vektor
    - hibafeszültség számítása
      * full mátrix és sunred: backsubs
      * multigrid: interpoláció a vezérlésnek megfelelően
  + Kontrollerek számítása, ha vannak
  + Áramok számítása (nem kell külön lépés a komponensek frissítésére, mert ekkor történik)

## Kontrollerek és párhuzamosítás

Pl. sugárkövetésnél elég masszív lehet a kontrollerek használata, milliós nagyságrend lehet belőlük.

~~A kontrollerek kimenetére érdemes árnyék puffert tenni:~~ => a bemenetről van másolás a függvény bemenetére

* 1. lépés: a kontroller kiszámítja az értékét egy árnyék kimenetre
* 2. lépés: a kiszámított érték átmásolása a tényleges kimenetre

Ha az 1. lépés lezajlott az összes kontrollernél, akkor mehet a 2. lépés.

Összetett kontroller lehet a **kontrollerlánc**. A kontrollerláncban egy vektorban vannak kontrollerek, ezeket egy szál értékeli ki egymás után, így egy iterációs lépésben végig tud haladni az adatfeldolgozás egy sor kontrolleren. Meg kell oldani, hogy egy láncon belül a kiszámított értéket a láncban egy következő kontroller azonnal fel tudja használni, de a láncon kívüli kontrollerek csak a korábbi értéket lássák.

Egy adott iteráción belül csak kontrollerlánc esetén van lehetőség egymásra építeni a kontrollereket.

Kontroller kimenete lehet normál (varnode) vagy rvt. Utóbbira csak másik kontroller megfelelő bemenete csatolható (esetleg expression?). rvt lehet árnyékolt vagy árnyékolás nélküli. Ezek lehetnek egy tömbben. Esetleg egy kontrollerlánc összes adata egy tömbben?

A kontrollermag olyan, mint egy kifejezés? A munkaterületet paraméterként kapja?

Kontrollercella

*IC1*

*O0*

*P0*

*IC0*

Kontrollermag

Kontrollermag

Munkamemória

## Summer kontroller vs. sokparaméteres áramforrás

* Lehetséges egy 3. lépés: ha lehetnek summer kontrollerek, amelyek ugyanolyanok, mint a normál kontrollerek, csak a 3. lépésben értékelődnek ki. A fő funkciójuk, hogy a bemenetükre kapcsolt kontrollerek értékeit összegezzék (de bármilyen számítást végezhetnek, akár különbsége, stb. is számolhatnak). Ha előírjuk, hogy summer kimenete nem kapcsolódhat summer bemenetére, akkor nem kell shadow kimenet, és nem kell 4. lépés.
* Lehet sokparaméteres áramforrás komponens, amelyik összegzi a paraméterként kapcsolódó változók/node-ok értékét, és annyi áramot ad ki magából. Így nem kell 3. lépést futtatni, és gyorsabb is lehet, hiszen csak összegezni tud, tehát nem kell utasításokat feldolgozni, mint egy kontrollernek.

Akár mindkét fajta lehet. A 3. lépés csak akkor fut, ha van az áramkörben summer kontroller.

## Paraméterek

Egy komponens paraméterként kaphat értéket, a tároló komponens paraméterét, golobális és lokális változót, a tároló komponens valamely külső vagy belső node-ját. Ha node-ot kap paraméterként, akkor azt is konstansnak tekinti, tehát nem veszi figyelembe a deriváltját a jakobi mátrix kiszámításánál. Ha figyelembe akarjuk venni, akkor vezérelt komponenst kell használni, és a vezérlőfüggvényben node-ként kell megadni. (A vezérlő függvényben is lehet paraméterként, de akkor ott is konstansnak tekinti.)

## Vezérlő csomópontok

Azok a belső csomópontok, amelyekre csak vezérlő bemenetek és vezérlő kimenet kapcsolódik, normál kimenet nem, azok a vezérlő csomópontok.

Ezeket a belső csomópontok vektorában a normál csomópontok után kell tenni, mert ezekre sem admittancia, sem hibaáram nem keletkezik.

## Komponensek csomópontjai

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Component node order

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// External nodes:

// --------------

// IONodes

// NormalINodes: Y values can be nonzero

// ControlInodes: no Y values

// NormalONodes: >= 2 component nodes connected, must be reduced

// ForwardedONodes

// Internal nodes:

// --------------

// NormalInternalNodes: >= 2 component nodes connected, must be reduced

// ControlInternalNodes

// InternalVars

// UnconnectedONodes

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// When a component is created, all external nodes must be connected

// because it gets the external nodes from out.

// If nothing uses ONodes, then an internal node must be created in the

// storage component for each ONode and must be attached to it.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Subcircuit-ekről beszélünk.

A normál kapcsokat követik a bemenő vezérlő kapcsok, mert ezekhez tartozhat admittancia X irányban (Y irányban 0 az admittancia, mert áram nem folyik rajtuk). A normál bemenő vezérlő kapcsok közvetlenül vezérlik az áramkör működését, azaz admittancia tartozhat hozzájuk. A vezérlő bemenő vezérlő kapcsok viszont kontrollert szabályoznak, azaz nem hatnak közvetlenül a Jakobi mátrix admittanciáira. Ha egy komponensnek nincs NormalINode-ja, csak ControlInode-ja, akkor lehet szimmetrikus a Jakobi mátrixa.

Utána jönnek a kimenő vezérlő csomópontok, kettéosztva: a normál kimenő csomópontok kívülről ugyanolyanok, mint a forward kimenő csomópontok. A normál kimenő csomópont valójában egy normál internal node kivezetve, vagyis (full mátrix és sunred esetén) redukálni kell, azaz megjelenik az XA, XB, YB mátrixokban. Ilyen szempontból jobban passzolt volna, ha a normál internal node-ok között hozzuk létre, majd a kimeneti csomópont ide mutat, csakhogy a program logikája nem ez. A program logikája az, hogy minden külső csomópont kívülről kapja a tényleges node-ot, emiatt lett ez a bonyolultabb megoldás.

A forwarded kimenő node a komponensen belül egy vezérlő kimenete valamelyik tartalmazott komponensnek. A redukció során ignoráljuk.

A normál internal csomópontok tényleges elektromos csomópontok, redukálandók. A vezérlő internal csomópontokra vezérlő kimenet és vezérlő bemenetek csatlakozhatnak. Az ide csatlakozó vezérlő kimenethez tartozó admittanciákat eldobjuk (már ha vannak egyáltalán). A vezérlő csomópontok nem vesznek részt a redukcióban, lényegében ugyanolyanok, mint a változók, csak a módosításuk nem akkor történik.

Az internal változókra vezérlő bemenetek csatlakoznak.

## Nem használt vezérlő kimenetek

~~A gyakorlatban előfordulhat, hogy~~ **~~egy komponensnek vannak vezérlő kimenetei, de ezeket nem akarjuk felhasználni~~**~~. Logikus lenne ezeket csak úgy lógva hagyni. Ezt azonban nem szabad! Mivel minden külső csomópont a tényleges csomópontot kívülről kapja, ezekhez létre kell hozni egy-egy internal node-ot a tartalmazó komponensben, és ezekre rákötni, hogy legyen tényleges csomópontjuk. Természetesen azt sem lehet, hogy spórolás céljából egy közös külső csomópontra kötnénk ezeket, hiszen ezzel rövidre zárnánk a csomópontokat, amelyek különböző belső komponensekben vannak.~~

~~A logikus és a szükséges működés úgy oldható fel, ha a SPICE netlistában lógva lehet hagyni a szükségtelen kimenő vezérlő kapcsokat, azonban a netlistát feldolgozó program pótolja a hiányzó internal node-oket és összeköttetéseket.~~

Ha a komponensnek van olyan vezérlő kimenete (ONode), amelyiket lógva hagyunk (nem kapcsolunk rá semmit), akkor a komponens létrehoz magának oda egy node-ot, (amit akár le is lehet kérdezni).

## Subcircuit value

~~A komponens ősosztályból öröklődik a value, ami tartalmazhat konkrét értéket, vagy mutathat csomópontra vagy változóra. Megvalósítható a subcircuitben, hogy ezt beállítsuk. (Jelenleg fixen 0-t ad.)~~

Minden komponensnek van értéke, a kontrollereknek is.

A kontrollereknek nincs árama, a többi komponensnek van.

A valódi komponenseknek van érték és áram változója, a tároló komponenseknek nincs. A subcircuit az első tárolt komponens értékét adja vissza. (Ha nincs, akkor a földet.)

## Full mátrix redukció

Áramok: a tárolt komponensek getJ(i) hívása a komponens belső csomópontjainak redukcióval kapott áramát adja.

Az aktuális komponens J[i] = szum(komponensek erre a csomópontra adott getJ(i)-jei) + az i. csomópont d-je.

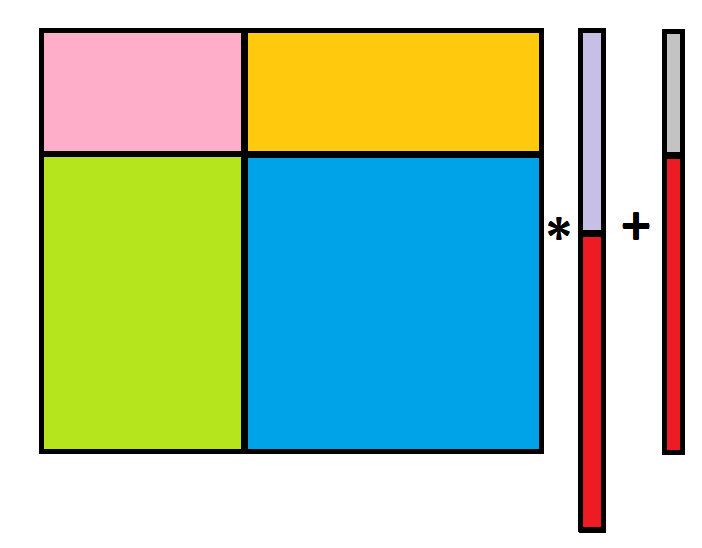
A **vezérlő kimenet** a B részben van a redukciónál, a hozzá tartozó csomópontot redukáljuk, a getJ és getY hívások 0-t adnak majd ezekre. (Úgy hatékony, ha a vezérlő kimenetekre nem is kérjük le a J-t és az Y-t.)

A **vezérlő bemenet**hez tartozó sor Y értékek és J-k 0-k, az oszlop Y értékek lehetnek nem 0-k.

A subcircuit vezérlő bemeneteire viszont leképezzük a csatlakozó vezérlő bemenetek admittancia mátrixát.

A vezérlő node-ok nincsenek benne a mátrixokban és vektorokban, ezek értékét vagy kontroller számítja ki, vagy valamelyik belső/külső normál csomópont.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | IONodes | normalINodes | normalInternalNodes | normalONodes |
| IONodes | YA | | XB | |
| normalINodes |  |  |  |  |
| normalInternalNodes | XA | | YB | |
| normalONodes |



## Időállandó spektrum

A VSUN3-ból + SzékelySolverből megpróbálom visszafejteni.

Időállandó analízis a VSUN3-ban: analtype==3

tca\_angle = pi + 1.5 \* log(10) / step\_pro\_decade

A log az ln-t jelenti. ln(10) = 2.3 => 1.5\*ln(10)=3.45

Székely komment a tca\_angle-hez: 3 samples for the half-value width

fmin-től fmax-ig ciklus (f = fmin):

* omega = 2\*pi\*f
* s =
  + AC/BODE: s = cplx { 0, omega }
  + TIMECONST: s = cplx { omega\*cos(tca\_angle), omega\*sin(tca\_angle) }
* forwsubs + backsubs
* f \*= pow(10, 1/step\_pro\_decade)

Ilyenkor a kiszámított érték nem hőmérséklet, hanem a komplex impedancia. (Driving point vagy transfer impedancia is értelmes.)

Mentéskor frekvencia helyett idő(álandó): 1.0/(2.0\*M\_PI\*fact; érték (hőellenállás dimenzióval, °C/W/oktáv) a probe feszültségét veszi, de csak az im részét menti (a re-t nem), így: value.im\*log(10.0)/M\_PI.

## A Newton módszer

A Trottenberg multigrid könyv frappánsan összefoglalja a módszert (149. o. környéke).

A megoldandó egyenlet:

A Jakobi-mátrix az elsőrendű parciális deriváltak mátrixa:

A hibaáram az m-edik lépésben:

A megoldandó egyenlet:

A feszültség következő értéke:

Egyszerűsített Newton: ha a K mátrix az u0-hoz tartozó értéket tartalmazza, azaz konstans. => lehetne úgy, hogy pár normál iteráció után tesszük konstanssá, pl. ha a hiba már 1% alatt van. Az alkatrészek értékének és az áramok kiszámítása természetesen marad nemlineáris.

Az egyszerűsített Newton konvergenciája lineáris, míg a normál Newtoné négyzetes. Esetleg kombinált módszer: pár egyszerűsített iteráció után beiktatni egy normál Newtont.

Szokás először egy robosztusabb iterációval kezdeni, majd a gyök közelében Newtonra váltani a gyorsabb konvergencia érdekében.

## Komponensek

### Feszültségvezérelt áramforrás

2 normál node és 2 control node.

Ahogy a normál áramgenerátornál, itt is lehet t<=0 és t>0 időpontban más S, valamint AC esetben is más S, és AC esetben fázistolás is használható. Ez 4 db paraméter.

### Girátor

*S1∙(UB1-UB2)*

*S2∙(UA1-UA2)*

*UA1*

*UA2*

*UB1*

*UB2*

*IA*

*IB*

IA iránya ellentétes a könyvbelivel!

Tipikusan S1=-1, S2=1.

S1 és S2 mindig ugyanannyi, függetlenül az időponttól és a DC vagy AC esettől, így csak 2 paramétere van. (Mivel a paraméter változóra is köthető, ez a korlátozás megkerülhető.)

A kódban a getY valamiért -1-szeres értéket ad vissza. Nem tudom, miért. Ezt meg kéne nézni.

### Feszültségforrás girátorból

*S1∙(UB1-UB2)*

*S2∙(UA1-UA2)*

*UA1*

*UA2*

*UB*

*IA*

*IB*

*V*

*G*

Redukálva:

Ha S1=-1, S2=1:

Ez lényegében az a modell, amit eddig is használtunk, csak a G helyett 1/G jelenik meg az admittanciamátrixban. Kicsi G-t kell használnunk, hogy a V áramgenerátor áramának nagy része az IB-ben jelenjen meg.

Vigyázat! JRED-ben a V helyett az UB csomópont hibaárama kell, hogy megjelenjen!

Az érdekesség most jön, a backsubs: UB = NZBXA∙UA + NZBJB = -UA1/G + UA2/G - V/G = (UA2 - UA1 - V)/G.

Ha nem lenne a G, akkor a bemeneti feszültség V lenne, de így van egy kis különbség, és ebből kiszámítható az áram. Óvatosnak kell lenni, G megválasztásakor, mert minél kisebb, annál kisebb a feszültségkülönbség, kevés értékes jegy marad az áramra.

Ha a feszültségforrást nagyon nagy ellenállásra kell kapcsolni, akkor e helyett a modell helyett a cellába egy girátort és egy rá kapcsolt áramgenerátort teszünk, így elég, ha a cella redukálható (= cellán belül kell nem 0 vezetés a fesz.gen kapcsai közé), nem kell önmagában a feszültségforrásnak annak lennie. Mivel a girátor admittanciamátrixa nem szimmetrikus, a Hex nem tekinti szimmetrikusnak a keletkező struktúrát.

### Függvények

Ugyanaz a függvény sokszor hívódhat különböző példányokban.

### Fényút tárolása

Controller-line: kontrollerek vektora, a kontrollerek index szerinti sorrendben értékelendők ki a függőségek miatt. Így egy CL- egy szál számol végig.

A kontroller bemenete a cellahőmérséklet. A cella hőmérséklet node-ját ONode-ként kivezetjük, így a kontroller bemenete egy node lesz.

A memóriaspórolás miatt érdemes lehet a kontrollerek sima változóit floatnak definiálni, vagy csinálni float kontrollereket és double kontrollereket.

A memóriaspórolás miatt lehet pl. egyszerre 4 cellát feldolgozó fényút kontrollert csinálni.

### SPICE kifejezések

.EXPRESSION EXPR P4 V0 // = G0\*exp(-gamma\*(T-T0))   
RET = P1\*.exp(-P2\*(P3-P4))   
.ENDF

4 paramétere van, 0 lokális változója

.component xxx [IO2] [IN1] [IC2] [ON1] [OF0] ~~[NI2] [CI1] [V1]~~ [P2] = ős\_komponens IO0 IO1 értéke

Ahol IO: IO node, IN: normal input, IC: control input, ON: normal output, OF: forwarded output, ~~NI: normal internal node, CI: control internal node, V: variable~~, P: parameter

Egyelőre belső node és változó nem lesz, mert ritkán kellene, és a vektora sok helyet foglalna. Ha kell belső csomópont, ONode-ként tökéletes, még azt is eldönthetjük, hogy kell-e redukálni (normal/forwarded).

Pl.

.component RthT IO2 IN1 P3 = \_G1 IO1 IO2 EXPR(P0, P1, IN0.stepstart, P2)

Az \_G1 a vezetésével megadott ellenállást jelenti.

Példányosítás

.instance R1 RthT IO0 \_gnd0 T (10m, 0.0025, 25)

Az .instance utasítással létrehozott komponensnek bármilyen neve lehet, itt R1.

A függvény minden sora értékadás. => nem feltétlenül

R1 = IF(P1>P2, N1, N2) // ha igaz, N1, ha hamis, N2 kerül R1-be   
R1 = IF(P1>P2, N1) // ha igaz, N1 kerül R1-be, ha hamis, R1 nem változik   
R1 = IFR(P1>P2, N1) // ha igaz, N1 kerül R1-be, és return-öl is a függvényből. Ha hamis, R1 nem változik, de megy tovább a következő függvénysorra.

Az IF/IFR értéke nem lehet kifejezés, csak változó (itt a node és a paraméter is változó). Ha kifejezés kell, akkor egy korábbi sorban számoljuk ki, tegyük egy változóba, és az kerüljön az if-be.

Paraméterek

Munkaterület

Megvalósítás:

* IF(P1>P2, N1, N2): HmgF\_IfValue2
* IF(P1>P2, N1): HmgF\_IfValue1
* IFR(P1>P2, N1): HmgF\_IfR

Továbbá:

* HmgF\_If1: IF(P1>P2): ha hamis, kihagyja a következő sort
* HmgF\_Else1: ELSE(P1>P2): ha igaz, kihagyja a következő sort

### GND és FixV

Bármelyik komponensben / subcircuit-ben hivatkozhatunk GND-re és FixV-re egyaránt.

Pl.: R1 IO1 \_GND0 10k   
 R2 IO2 \_FIXV0 20k

A kettő közötti különbség az, hogy a \_fixv után megadott index abszolút, a \_gnd után megadott index pedig felüldefiniálható.

Ez azt jelenti, hogy ha a szimuláció előtt beállítom, hogy .set \_FIXV1 25, akkor bárhol is szerepel az áramkörleírásban a \_fixv1, az a 25 V-ra beállított node-ra kapcsolódik.

\_gndN esetén N = 0…7 lehet, más nem. \_fixvN-nél N viszont nincs limitálva.

A \_gndN jelentése:

.hsubckt cella IO2   
R1 IO0 \_gnd0 10k   
.ends

.hsubckt big NI1   
.instance c1 cella NI0 {\_gnd1,\_fixv1}   
.ends

.fullckt big {\_gnd1,\_gnd2,\_gnd0}

A fullckt-k a definiálásuk sorrendjében jönnek létre, hivatkozni rájuk \_fullckt0, stb.-ként lehet.

Ebben az esetben a big típusú fullckt \_gnd0 = \_fixv1, \_gnd1 = fix2, \_gnd2 = \_fix0. Ha a big-ben lenne 2-nél nagyobb \_gnd hivatkozás, akkor az a megfelelő \_fix-re hivatkozna, pl. \_gnd3 = \_fix3. (Ha úgy vesszük, a \_fullckt0 a [1,2,0,3,4,5,6,7] indextömböt veszi át.

A c1 cella \_gnd0-ként a \_fulckt 1-es indexű \_gnd-jét veszi át, tehát NEM a \_fiv1-et (!), hanem a \_fixv2-t, hiszen a \_fullckt1-nek a \_fixv2 a \_gnd1-e. De ha biztosan valamelyik \_fixv-re akarjuk kötni a tartalmazott komponens valamelyik \_gnd-jét, akkor \_fixv-vel is definiálhatjuk. A c1 tehát a [2,1,0,3,4,5,6,7] indextömböt veszi át. (Az indextömbben előfordulhat többször is ugyanaz az index, ennek semmi akadálya.)

Egy adott komponens (subcircuit) node-jainak alapértelemzett kezdeti értéke a \_gnd0. Ezt felüldefiniálhatjuk:

.hsubckt cella IO2   
.defaultvalue IO0=\_gnd1, IO1=\_fixv1   
R1 IO0 \_gnd0 10k   
.ends

Ha ugyanazt a node-ot több helyen (pl. a big-ben is és a cell-ban is) .defaultvalue-zzuk, akkor nem definiált, hogy melyik értéket fogja felvenni. (Valószínűleg azt, amelyik a legmélyebben van, de ez nem garantált.) Az egyiket biztosan felveszi, tehát a szimulációt le lehet futtatni.

### Custom component és Controlled component

A két tároló komponens sokan hasonlít egymásra, és a custom component gyakran vezérelt is egyben (de nem mindig). Érdemes lehet egyben megvalósítani, így csökken az indirekciók száma => gyorsabb és remélhetőleg kevesebb memóriát eszik.

megj. ha a vezérlő függvény konstans értékű, akkor az eredmény egy nem vezérelt komponens

Két komponenssel modellezzük a szituációt (az egyszerűség kedvéért egy konkrét, általánosítható esetet bemutatva):

Vezérelhető komponens

*1*

*I = g(U1, U2, U3)*

*U1*

*U2*

*U3*

*C*

**A vezérelt áramgenerátor + G komponensnek két normál bemenete van, az alja nincs automatikusan földelve!**

A kifejezés komponens vezérli a vezérelhetőt. (Ha csak sima custom componentet csinálunk, akkor nem kell vezérelhető komponens, ilyenkor a redukcióban csak 1 komponens lesz.) A kifejezés komponens egy darab IO porttal rendelkezik, ill. vezérlő bemenetekkel és paraméterekkel.

Ha a vezérelhető komponens több vezérlő bemenettel rendelkezik, akkor ennyi kifejezés komponens lesz a cellában.

A vezérlő komponens egyenletei:

A kifejezés komponens egyenlete:

Az admittanciamátrixot a összefüggés adja. A vezérlő bemenetekre a deriváltak nullák, hiszen itt az áram mindig 0. A mátrixok vezérlő bemenetekhez tartozó sorait ezért fel sem tüntetem, ezek csupa 0 sorok, viszont a vezérlő bemenetekhez tartozó oszlopok nem 0-k.

A vezérelt komponens admittancia egyenlete:

A kifejezés komponens admittancia egyenlete:

Az összekapcsolt egyenlet:

NZB = , JB = , NZBJB =

JRED = JA

**A kifejezés komponens legyen egyszerűen egy vezérelt áramforrás beállítható belső ellenállással.** Ha vezérlésre akarjuk használni, akkor a második node-ját gnd0-ra kell kötni.

AC eset: a vezérelt komponens a vezérlő node DC értékét veszi figyelembe. => inkább a kifejezés komponens adja AC-ben is a DC értékét.

### Adaptív deriválás

Először kiszámítjuk a kifejezés értékét (y), majd ∆x = (abs(x) + abs(y))\*1.0e-9.

### Függvények megvalósítása

A függvény rvt paramétereket kap és egy rvt értéket ad vissza.

Egy függvénynek bármennyi paramétere lehet, a paraméterátadást meg kell oldani.

Takarékoskodni kell a memóriával.

A függvény minden sorában egy függvényhívás történik, a kifejezéseket így bontjuk szét.

Pl.

y = (x1 \* x2 + sqrt(x3)) \* x4

kifejezés sorokra bontva:

függvény(ret, x1, x2, x3, x4)

* v1 = x1 \* x2
* v2 = sqrt(x3)
* v3 = v1 + v2
* ret = v3 \* x4

Az elemi műveleteket is függvényhívás valósítja meg.

A függvény munkaterülete tartalmazza a paramétereket, visszatérési értéket, munkaváltozókat.

Mivel ugyanazt a függvényt több szálon is hívhatjuk, a munkaterületet a hívó adja át.

Kontrollernél a munkaterület megmarad a hívások között (ill. nem csak ott, de egyéb használatán még nem gondolkodtam). Semmi akadálya annak sem, hogy paraméter helyett valamilyen kiszámított értéket adjunk vissza, de akkor ezt a paramétert bemenetként ne használjuk, mert pl. a deriváltszámításhoz újra meg kell hívni a függvényt.

A custom függvény definiálásához szükség van a függvény strukturális leírására, ezt a HgmCustomFunctionModel tartalmazza:

* paraméterek száma
* lokális változók száma
* sorok leírása
  + hívandó függvény
  + visszatérési érték és paraméterek leírása
    - típusa: paraméter vagy lokális változó
    - indexe

A tényleges custom függvény a konstruktorában kapja a modellt, ami alapján felépíti magát. A custom függvény típusa HmgF\_CustomFunction. Három adata van:

* a modell
* az indexterület mérete
* a munkaterület mérete

Az indexterület azonosítja, hogy paraméterként vagy visszatérési értékként a munkaterület mely adatával kell dolgoznia a függvénynek és az általa hívott függvényeknek (a munkaterület közös, ezáltal memóriát és másolásokat spórolunk).

Az indexterület felépítése:

1. A visszaadott érték indexe a munkaterületen (a meghívott függvények is a teljes munkaterületet kapják, ezért nekik meg kell mondani, hogy hová tegyék az eredményt).
2. A hívott függvény számára használható munkaterület kezdetének indexe.
3. A paraméterek indexei.
4. Ezután minden sorhoz tartozó függvényhívásokhoz ugyanezek az adatok.

Egy adott függvény számára az 1-3 adatokat az őt hívó állítja be, tehát ezeket már kapja. A függvény dolga, hogy az általa hívott függvények számára kitöltse az ő 1-3 adatait.

A beállítást a fillIndexField tagfüggvény végzi. Ez, miután feltöltötte a következő meghívandó függvény 1-3 adatait, meghívja a függvény fillIndexField tagfüggvényét, hogy az a saját meghívott függvényeinek feltöltse ugyanezeket az adatait a közös indexterületre.

Minden függvény rendelkezik getN\_IndexField függvénnyel, amely megmondja, hogy ő mekkora helyre tart igényt az indexterületen (az IndexField által visszaadott érték tartalmazza az általa meghívott függvények indexterület-igényét is), így le lehet foglalni a szükséges memóriaterületet, ill. megfelelően léptetni az indexterület pointert a feltöltés során. **A getN\_IndexField a tényleges teljes indexterületigényt megadja.**

A függvényobjektumok getN\_WorkingField tagfüggvénye a munkaterületet méretét adja a, az indexterület függvényéhez hasonlóan. Lényeges különbség, hogy ez a lokális változók területigényét adja csak, a paraméterek és visszatérési érték helyigényét nem, mert ezek általában az őt hívó függvény lokális változóiból/paramétereiből jönnek ill. oda kerülnek, ezért a memóriafoglaláskor **az alapfüggvény getN\_WorkingField-e által adott mérethez az alapfüggvény paramétereinek számát + 1-et a visszaadott értéknek, hozzá kell adni**.

Az index tömb [0] eleme a visszatérési érték, [1] eleme a munkaterület kezdete, [2..n] a paraméterei. A munkaterület tömb alakja az alapfüggvényben tipikusan [0] a visszatérési érték, [1..n-1] a paraméterek.

### A vezérelt áramforrás (feszültségforrás)

*G*

*I = g(IO[], IC[], P[])*

*IO0*

*IO1*

*IC2*

*IC0*

*IC1*

A modell vezérelt feszültségforrásnak is tekinthető U = I / G. Áramforrásként az a jó, ha G kicsi, feszültségforrásként pedig az a jó, ha G nagy, ha „ideális” forrást akarunk. Ha belső ellenállással rendelkező forrást modellezünk, akkor persze a belső ellenállást használjuk. (Ha áramgenerátorként használjuk, akkor G lehet 0 is.)

AC-nél a ∂g-s tagok 0-k, tehát csak a G-s tagokra ad nem 0-t a getY, a többire 0-t ad.

### Diódamodell

A vezérelt forrás segítségével jól konvergáló diódamodellt készíthetünk.

A

diódaegyenletet feszültség alakba írva

Ahol .

A dióda modelljét egy vezérelt áramforrás és egy árammérésre használt 0V-os feszültségforrás soros kapcsolásával kapjuk. A dióda soros ellenállása G0 lesz. Ha külön akarjuk modellezni, akkor lehet pl. 1000 S, az nem okoz érdemi hibát, és levonhatjuk a tényleges soros ellenállásból, vagy lehet a soros ellenállás konstans tagja, vagy annak egy része.

Áram meghajtás esetén az első iterációban nagy kapocsfeszültség adódik, ami a disszipáción keresztül a hőmérsékletben okozhat gondot, de első végiggodolásra konvergensnek tűnik mindenféle beavatkozás nélkül is.

*G0*

*I = G0∙m∙UT∙ln(NI1/I0 + 1)*

*IO0*

*IO1*

*NI0*

*NI1*

*0V*

*IN0 = T*

### Függvénylánc / kontrollerlánc

Kontrollerlánc helyett óriáskontrollereket lehet használni. Ennek az elemei egy-egy cellát kiszámító függvények. Egy függvény több értéket is kiszámíthat, ezeket paraméterként adhatja vissza. A belső függvényhívások ugyanazt a munkamemóriát használják, viszont paraméterként megkaphatják a kontroller fő függvényének lokális változóit: ezek nem csak egy-egy belső függvényhívás, hanem a kontroller újra-kiértékelései között is megőrzik az értéküket.

### Kontroller

A kontroller a loadNodesAndParamsToFunction() függvénnyel másolja a külső csomópontokból és a paraméterekből az értékeket a függvény paramétersorába, közvetlenül a munkaterületre. A model.functionSources.sources vektor mérete mondja meg, hogy hány paraméter másolódik, közte lehetnek sReturn típusúak, ezeket a paramétereket nem írja felül (kimeneti paraméterek). A sources vektor lehet kisebb, mint a paraméterek száma, ha a maradék paraméter kimeneti. Érdemes a kimeneti paramétereket a bemeneti paraméterek után tenni, így a sources-ban nem foglalnak helyet és hatékonyabb.

A storeNodes() függvény ennek az ellentéte: a megadott függvényparamétereket menti a külső csomópontba. Itt a 0 paraméterindex (srcParamIndex) a függvény visszatérési értékét jelenti, tehát az is menthető.

Belső változók!

A HexOPEN doksiban írtam, hogy a komponenseknek csak svar-ja van, mvar-ja nincs, ezért ott nincs teendő.

Kontrollernél az mVar-ok használata a következő utasításokkal történik:

* HmgF\_Load\_Controller\_mVar\_Value: a tároló kontroller megadott indexű mVars elemét adja vissza ret értékként.
* HmgF\_Set\_Controller\_mVar\_Value: a visszatérési értékének megadott változót másolja az mVars megfelelő indexű helyére.
* HmgF\_Load\_Controller\_mVar\_StepStart: az mVar stepStart értékét másolja be a függvény visszatérési értékének.
* HmgF\_Set\_Controller\_mVar\_ValueFromStepStart: az mVar értékét írja felül a saját stepStartjával, a függvény munkaterületét nem változtatja (ret-nek bármit beállíthatunk, nem használja).

A HmgF\_Load\_Controller\_Node\_StepStart utasítás valamelyik node stepStart értékét tölti be, és teszi a ret helyre. (A node-ok értékének kezelésére nincs külön utasítás, mert azt normál eljárásban olvassuk és írjuk.)

A vezérelt áramforrásnak is van ilyen függvénye: HmgF\_Load\_ControlledI\_Node\_StepStart.

### Ugróutasítások függvényekben

Minden függvényobjektum evaluate függvényének van egy egész szám visszatérési értéke (ez nem felhasználó által kezelt tulajdonság, hanem bedrótozott). Ez általában 0, ami azt jelenti, hogy nincs ugrás.

Ha a visszaadott érték returnInstructionID (ez jelenleg 1'000'000'000), akkor a függvényobjektum hátralévő utasításait nem hajtjuk végre, hanem visszatérünk a hívóhoz.

Egyéb esetben annyi utasítást lépünk előre vagy vissza (pozitív vagy negatív visszaadott érték), ahány az evaluate által visszaadott érték. Ezzel megvalósítható akár if, akár ciklus jellegű utasítás is.

### SUNRED

Levél cella:

* forwsubs: magába másolja a forrás cella Yred és Jred értékeit, közben figyeli, hogy volt-e változás.
* backsubs: nincs teendő

Belső cella:

* forwsubs: két cellát egyesít. Ha kell redukció, a bemenő admittanciamátrixokat megfelelően bemásolja az YA, XA, XB, YB mátrixokba. (Lehetnek olyan összeadandók, amelyek az YA-ba kerülnek: a centroid csomópontok.) A JA, JB feltöltése a forrás J-kből. A JB-be kerülő értékekhez hozzá kell adni a közös csomópont defect értékét (mivel a centroid JA-ba kerül, így automatikus, hogy nem adódik hozzá, viszont amikor a centroid megszűnik, akkor igen).
* backsubs: a két forrás UA alapján kiszámítja UB-t. Az UB-t vissza kell írnia a megfelelő csomópontokba.

Nem sablon osztály, hanem a csomópont tartalmazza az AC és a DC mátrixokat is.

**Ha változik az engedélyezett komponensek köre a sunred szintjén, akkor újra kell generálni a sunred adatszerkezetet!**

### Multigrid

#### DC nemlineáris teljes iteráció

f0=0

u0 = **solve0**(f0) // normál módon kiszámítjuk a 0. szintű hálózat megoldását

ciklus célszint = 1 to n

ucél = **P**(ucél-1)

V ciklus 1-szer vagy 2-szer

lefelé ciklus szint = célszint to 1

uh = **relax**(uh, fh) // 1-szer, de akár többször

dh = **Lh**(uh)

uH = **R**(uh)

dH = **LH**(uH)

fH = **R**(fh) + dH – **R**(dh) // Ha szint == célszint => trerr += (dHi – R(dh)i)^2

lefelé ciklus vége

~~trerr = sqrt(trerr) / nodenum(célszint)~~ // minek?

u0 = **solve**(f0)

felfelé ciklus szint = 1 to célszint

uh = uh + **P**(uH – **R**(uh)) // uH – R(uh) => a dH lehet a temp

uh = **relax**(uh, fh) // 1-szer, de akár többször

felfelé ciklus vége

dcél = **LXcél**(ucél) // az fh-t ne adjuk hozzá, így levonni sem kell ! // vagy mégis kell???

//~~d~~~~cél~~ ~~-= f~~~~cél~~ // mert a lop fv szintén fh nélkül számol

~~res = sqrt(sum(dcéli^2)) / nodenum(célszint)~~ // minek?

res = sum(dcéli^2)

ha res < 0.33 \* trerr => break

V ciklus vége

célszint ciklus vége

#### DC Lh

A megoldandó téregyenlet diszkretizált alakja az Lh(uh) = 0, miközben lineáris esetben Lh(uh) = fh. Emiatt lineáris esetben dh = Lh(uh) – fh van az algoritmusban, míg nemlineáris esetben dh = Lh(uh). A nemlineáris Lh azonban tartalmazza a gerjesztést is, tehát mindkét algoritmus a hibaáramot számolja.

A numRec kódjában ugyanakkor a lop() nem használja. Itt az fh tartalmazza a gerjesztést (ez a kezdeti értéke). Sehol nem indokolja, hogy miért nem használja a lop(), ugyanakkor a levezetése szerint használnia kéne.

A relax() hibaáramot számol az rhs-t, azaz az fh-t is hozzáadva. Nálam maga a hibaáram számítása tartalmazza a gerjesztést, az rhs pedig nem, mert ott a dH – R(dh) során elvileg kiesik. Tehát elvileg jó kéne legyen.

Szóval Lh a hibaáram számítása (ComponentBase::calculateCurrent()).

Ezzel még lehet gond. Meglátjuk.

#### DC relax

20230417 A jacobiIteration függvényekkel szemben elvárás, hogy a lefutás után a komponens belső node-jain a megemelt feszültség legyen.

20230417 Mivel itt külön lépés a currentCalculation, a Gauss-Seidel iterációt nem igazán tudjuk használni, csak a Jakobit. Ha jól értékelem, akkor két Jakobi iteráció az kb. egy Gauss-Seidelnek felel meg.

A könyvek Gauss-Seidel-Newton relaxációról beszélnek, ami azt jelentené, hogy a Newton a Gauss-Seidelen belül van, azaz

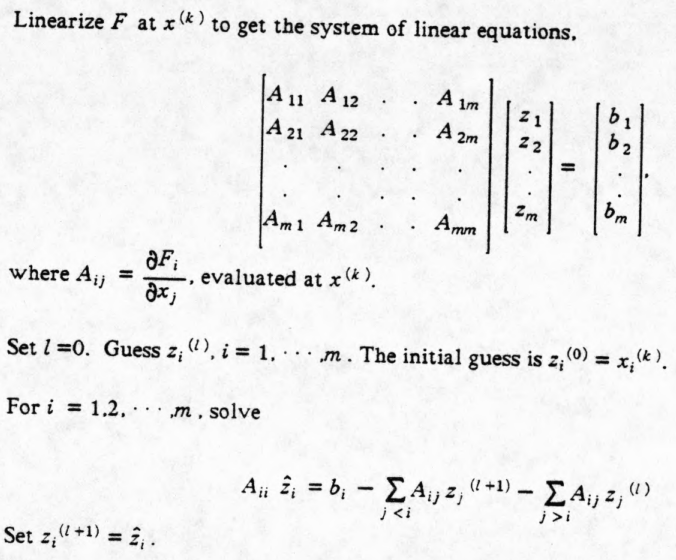
* kiszámolom a Jakobi-mátrix első sorát
* kifejezem du1-et: y11\*du1 = d1 – szum(y1i\*dui) // i != j-re
* u1 += du1
* Az új u1-gyel kiszámolom a Jakobi-mátrix második sorát
* kifejezem du2-t…
* …

Mivel a Hex struktúrája ehhez nem passzol, nem lehet lokálisan kifejezni a Jakobi-mátrix valamelyik sorát, inkább a Newton-Gauss-Seidel (Newton-Jacobi?) módszert használjuk:

* kiszámítjuk ~~a Jacobi-mátrixokat~~ => az Yii-ket
* a tárolt komponensek internal node-jaira csinálunk egy Jacobi-iterációt => párhuzamosítható, mert független a többi komponens internal node-jaitól ~~=> az onode-ok nem számíthatók a versenyhelyzet miatt.~~ A node value értéke rögtön megnövelhető.
* a multigridezett subcircuit minden node-jára csinálunk egy Jacobi-iterációt (a node-ok v-iben tároljuk a kiszámított értéket) => ~~a számoláshoz a node-ok (kiinduló) value értékét kell használni, a getValue nem használható! Lehet pl. egy getValue0() ehhez.~~ => Nem kell a node értéke, mert azt a currentCalculation-nál, amikor még a value nem változik, aztán a Jakobi-iterációban már csak az áramot használjuk, így azonnal növelhető a node feszültsége.
* ~~minden node-on futtatható a setValueAcceptedDC() (esetleg lehet csinálni egy olyan variánst, ami alpha nélkül csinálja)~~

~~Ez a subcircuit node-jainak feszültségéhez már az internal node-ok relaxált feszültségét használja. Ez lényegében megfelel a red-black mintázatnak.~~ => Mivel nem a Jakobi-iterációban számítjuk a hibaáramot, hanem előtte, a node-ok relaxált feszültségét nem tudjuk használni.

~~(Hibaáramból számolunk, tehát hibafeszültség jön ki, amit az eddigi feszültséghez kell adni.)~~ Egy túrót, nem ezért.



Az Yii-k azok jellemzően több G összegei (ahány ellenállás vezet a csomópontba, pl. 4G), míg az Yij-k jellemzően egy db. -G-k.

De ebből (nekem) nem jön ki a következő, a szakirodalomban sem találom nyomát. Lehet, hogy Li-t úgy tekinti, hogy csak ui-töl függ, a többi u-t erre a lépésre (soronként) konstansnak tekinti, így a Jakobi-mátrixnak csak a főátlójában vannak elemek.:

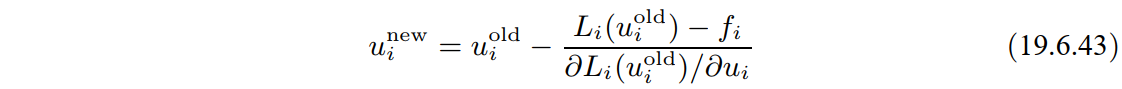
Írjuk fel ezt az azonosságot (nem tudom, miért):

Vagy:

Most legyen a bal oldal az (l + 1). lépés, a jobb oldal az (l). lépés:

Az az iteráció célja, hogy az iteráció után a hibaáram eltűnjön, vagyis legyen. Így a kifejezés így alakul:

És ez nem más, mint a NumRec megfelelő kifejezése:



(Ha csak ui új értékét számolnánk, akkor a hibaáram valóban eltűnne, de mivel rajta kívül az összes u-t újraszámoljuk, a következő iterációra újra lesz hibaáram.)

A kifejezés érdekessége, hogy az yii-n kívül a többi admittanciát nem kell kiszámolni, nem is használjuk. (A sunrednek kell, de a GS-nek nem.)

A relax algoritmus tehát: ~~Itt ne gc.fullCircuitInstances[szint].component legyen! Mélyebben is lehet a multigridizálandó komponens!~~ => nem lehet! itt nincs forwsubs és backsubs

* Az u és f értékek be vannak állítva.
* gc.fullCircuitInstances[szint].component->calculateValueDC();
* gc.fullCircuitInstances[szint].component->deleteY(true);
* gc.fullCircuitInstances[szint].component->node-ok yii értékét beállító függvény, a d beállítóhoz hasonló +=, ez is atomic. (a calculateValue kell előtte, a calculateCurrent nem)
* gc.fullCircuitInstances[szint].component->loadFtoD(true);
* gc.fullCircuitInstances[szint].component->calculateCurrent(true);
* gc.fullCircuitInstances[szint].component->a tárolt komponensek internal node-jaira Jakobi-iteráció (bár lokálisan ez GS-nek is tekinthető). Az eredmény rögtön a value-ba kerül.
* gc.fullCircuitInstances[szint].component->internal ~~+ external~~ node-jaira Jakobi iteráció (párhuzamos) => ~~az hibafeszültség a v-be kerül~~ => Az eredmény rögtön a value-ba kerül.
* ~~gc.fullCircuitInstances[szint].component->internal + external node-jaira setValueAcceptedMGDC() (párhuzamos)~~

Ha több iteráció kell, akkor mehet a loadFtoD-től.

#### Restrikció és prolongáció

Mivel a cellahatárok gyakran anyaghatárok, érdemes a restrikció-interpoláció párost csak az összetartozó összevont-szétbontott cellák értékeiből képezni (cellacsoport). Ha ez a felállás, akkor a kód és a restrikció-prolongáció utasítások is egyszerűbbek. Cserébe az interpolációnál megjelennek negatív súlyok is, de ezt túléjük.

Egy cellacsoportban tetszőleges számú durva és tetszőleges számú finom cella lehessen, így rugalmasabb multigrid modell építhető. Pl. ha durvább felbontású termikus modell van, mint az elektrotermikus, akkor az interfész eltűnésekor több durva cella kerülhet a csoportba. Jó ez az algoritmus az interfész kezeléséhez is?

Az összevonás leírása: a durva subcircuit mely komponenséhez a finom subcircuit mely komponensei tartoznak? A vezérlőkre nincs interpoláció/restrikció.

Restrikció:

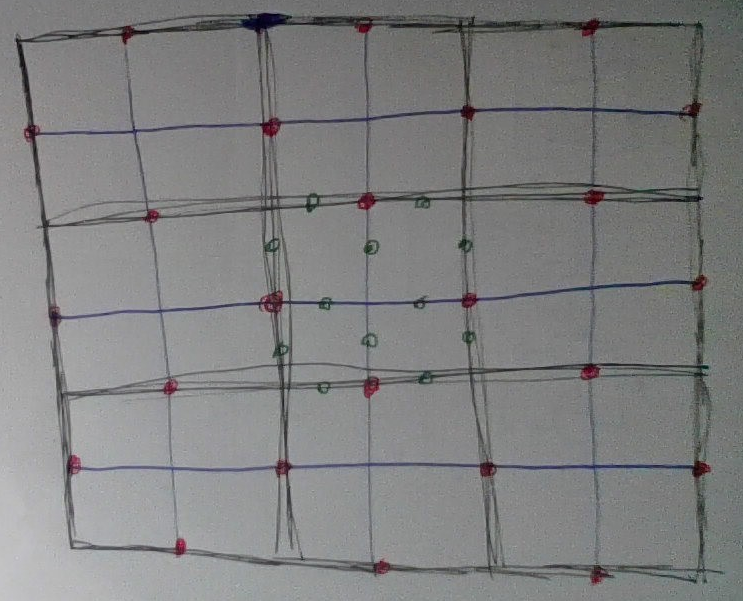
* Normál esetben a nagy cella internal node-jai a kis cellák megfelelő internal node-jainak átlagai.
* Normál eseten nagy cella external node-jai face-enként a kis cellák megfelelő node-jainak átlagai, de a hexMG-ben nincsenek face-ek definiálva, ezért muszáj egyedi utasításokat használni. Viszont lehet egyszerűsített utasításokat használni: egy external méretű vektor elemei vector<uns>-ok: hányas cellákból kell az azonos indexű node-ot venni, és ezeket átlagolni?
* De lehetővé tesszük univerzális restrikció megadását is. Megadható az összes internal és/vagy az összes external node-hoz: az adott indexű internal/external node-hoz mely cella mely node-ját milyen súllyal vegyük figyelembe? (Externalhoz csak external, internalhoz csak internal lehet.)

Interpoláció:

* Itt nincs egyszerűsített definíció. Minden finom cella minden node-jához definiáljuk, hogy a durva cella mely node-ját milyen súllyal vegyük figyelembe.
* Először az external node-okat számítjuk ki, majd ezek alapján az internal node-okat, így más cella externalja is hatással lehet az adott cella internaljára.
* Az iteráció lépései:
  + Az első ciklus végigmegy minden cellacsoporton, és kéri a finom cellák external node-jainak kiszámítását. Ezt az értéket a csomópont mgExternalNodeVoltage tagváltozójába fetch\_add-oljuk.
  + Ezután végigmegyünk a finom tároló subcircuit összes internal és external node-ján, és frissítjük a node feszültség (value) értékét, ha a mgExternalNodeVoltage tárol értéket, továbbá töröljük a mgExternalNodeVoltage tartalmát (moveToValueIfSet()).
  + Újra végigmegyünk minden cellacsoporton, és kiszámítjuk a finom cellák internal node-jainak értékét, az externalokat is figyelembe véve.

Az interpoláció egyszerűen a legközelebbi pontok távolsággal súlyozott átlagát jelenti, semmi extrát. Pl. ha a két rácson egybeesik a két pont, akkor a durvább pont értékét kapja a finomabb, nincs semmilyen mosás. A következő ábra a külső csomópontokat mutatja sunred celláknál, pirossal a durva rácson, zölddel a finom rácson. A durva kocka belsejébe eső pontokkal egyszerű a helyzet, mert a kék vonalon szomszédos két pont közül a közelebbi ¾, a távolabbi ¼ súllyal átlagolódik.

A fekete vonalon lévőknél viszont nem ilyen egyszerű a helyzet. Ha a két szomszédos piros pötty azonos anyaghoz tartozik (elég, ha az egyik oldalon), akkor itt is számolhatunk így. Ha nem, viszont a metsző fekete vonalon lévő két szomszédos piros igen, akkor azok átlagát számolhatjuk a metszőpontra, és onnan a megfelelő zöld pontot ki tudjuk számolni. Ha nincs azonos anyag, akkor talán a legszerencsésebb az azonos durva cellához tartozó nem szomszédos pirosakat is bevonni az interpolációba. (De érdemes lehet kipróbálni, hogy ha eleve csak azokból interpolálunk, akkor mi van.)



20230418 A cellabelsőre vonatkozó utasítás external node-ot is írhat, amikor elvileg létrejöhet versenyhelyzet. Ugyanakkor normális esetben ez az external node inicializálatlan, így nem használhatjuk más cella inicializálására, tehát ha létrejön versenyhelyzet, akkor az biztosan hibás prolongációs/restrikciós utasítás eredménye.

20230419 A normál restrikció maximum 4 szintű lehet (mert amúgy még több szintet kéne belenyomni manuálisan, vagy rekurzív megoldást kéne csinálni, ami jelentősen megbonyolítaná a kódot).

* 0. szint: a piros és zöld pöttyök szintje, azaz a fullCircuit internal node-jai.
* 1. szint: a cellák internal node-jai
* 2. szint: a cellákban lévő feszültségforrások internal node-jai
* 3.-4. szint: ha bonyolultabb a cellaszerkezet, és vannak mélyebben subcircuit-ek is. (Ez még + két subcircuit szintet jelent, ill. 3-at, ha a legalsóban nincs olyan komponens, amelyiknek van belső csomópontja. Jelenleg nem hiszem, hogy fogok ilyen modellt generálni.)

20230419 A párhuzamosításnál lehet, hogy ez amúgy is több függvényre bomlik, és akkor akár rekurzív verziót is írhatok. Meglátjuk.

20230421 ! Áram prolongáció/restrikció esetén mi van, ha forrásként a destLevel-t állítottuk be?

### Termikus és elektromos terek szétválasztása

A termikus modellt általában sokkal jobban le lehetne egyszerűsíteni a multigrid során, mint az elektromosat + a modell jelentős része lehet csak termikus. Mit kell módosítani, hogy ezt a lehető leghatékonyabban lehessen kezelni?

* mivel az elektrotermikusból képzett termikus cellák gerjesztését az elektromosak adják, nem nagyon van értelme kisebb felbontású termikus multigrid modellt csinálni, mint az elektromos.

Az alap (legdurvább felbontású) modell lehetne ilyen:

* Nem választjuk szét az elektrotermikus cellákat elektromosra és termikusra.
* A minimál felbontású elektrotermikus részt jóval durvább felbontású termikus modellel vesszük körül, amit a kompakt modellek csatolásához hasonlóan oldhatunk meg.
* Ezután lehetnek olyan multigrid lépések, amelyekben csak a termikus rész felbontása nő, az elektromosé nem. Nyilván a termikus rész változása hatással van az elektromos értékekre is, úgyhogy ez a rész is frissítendő.
* Végül normál módon együtt nő az elektrotermikus és a tisztán termikus rész felbontása.

### Vezérlők frissülése

Kéne olyan vezérlőtípus, ami csak az adott lépés elején frissül.

Mi van a multigrid ciklusokkal? Azt is lehetne előírni, hogy csak adott szintű rács első lefutásakor frissüljön. Jó ez?

Ezek még átgondolandók.

## Nodemásolás áramkörök között

Lehetne definiálni a multigridhez hasonlóan súlyozott node érték másolást fullcircuitek között (vagy akár fullcircuit-en belül).

## TODO

* Egy db sunred van beágyazva a subcircuit-be, viszont pl. váltott elektromos-termikus szimulációnál ez nem lesz jó, akkor több is kellene. A setNodesToComponents minden komponenst figyelembe vesz, nem csak az engedélyezetteket. Valószínűleg minden sunred példányhoz saját externalNodesToComponents és internalNodesToComponents kéne. Ezeket rendbe kéne rakni.
* Az Y és J lekérdezést sunredre is meg kéne oldani.
* Konstans Jakobi mátrix használatára utasíthatóvá tenni a rendszert (a sunredet és a full matrixot érinti): egyes lépésekben nem frissül a Jakobi-mátrix, azaz nincs redukció, csak forward.
* Miért kell v az AC-be?
* Áram probe-ot is kellene csinálni. Komponens node-ok áramainak összege (meg lehet adni, hogy mely komponens mely node-ja). Jó lenne, ha ezt vezérlési célra is fel lehetne használni.