Eötvös Loránd Tudományegyetem

Informatikai Kar

Numerikus Analízis Tanszék

Nemlineáris egyenletrendszerek megoldása a folytatás módszerével

Dr Gergó Lajos
egyetemi docens

Huszti Péter
Programtervező Informatikus
nappali tagozat

Budapest, 2014

Tartalomjegyzék

1. fejezet

Bevezetés

1.1. Motiváció

A gyakorlati életben sokszor fordulhat elő olyan probléma, hogy adott egy függvény, amely valamilyen viselkedést ír le, és meg kell határozni, hogy milyen feltételek mellett lesz az adott függvény értéke nulla. Azaz keresnünk kell egy x-t úgy, hogy f(x)=0, ahol f a függvény, ami ismert. Az ilyen x-eket az f függvény gyökeinek nevezzük. A fenti probléma megoldására különböző közelítő eljárásokat tudunk alkalmazni, amik véges lépésben, egy meghatározott hibával előállítják az egyenlet gyökeit.

Nagyon sokszor futhatunk olyan problémába, hogy több ismeretlen változónk van, és ezek között több összefüggés is fennáll. Ha ezekben az egyenletekben a változók egymással lineáris kombinációban állnak, akkor beszélünk *lineáris* egyenletrendszerről.

Előfordulhat azonban az is, hogy a változók fokszáma nincs 1-ben maximalizálva, ekkor mondjuk azt, hogy az egyenletrendszer *nemlineáris*, melynek általános alakja:

$$F_i(x) = 0 \quad (i = 1, \ldots, m)$$

ahol $F_1, \ldots, F_m$ $\mathbb{R}^n$-ből $\mathbb{R}$-be képező függvények úgy, hogy legalább az egyik közülük nemlineáris. Az ilyen egyenletrendszerek megoldására általában a többváltozós Newton-módszereket használjuk. Ezek többsége ún. *lokális módszer*, azaz szükség van az iterációhoz induló érték vagy értékekre, és előfordulhat, hogy nem fog konvergálni, vagy konvergálni fog, de nem a várt megoldáshoz.

1.2. A program célja

A program célja, hogy a felhasználó könnyen és gyorsan tudjon megoldani nemlineáris egyenletrendszereket. A program egy globális módszert, a folytatás módszerét alkalmazza a megoldás kiszámítására, ami azt jelenti, hogy a kezdővektor (ahonnan a közelítőeljárás indul) tetszőleges lehet. Ellentétben a hagyományos Newton-módszerrel, a program biztos, hogy megtalálja az egyenletrendszer egy gyökét.

A program feladata egy interfész biztosítása, aminek segítségével a felhasználó meg tud adni kézzel, vagy be tud olvasni fájlból egy egyenletrendszert képletekkel, és hozzá egy kezdővektort, hogy honnan indítsa a program az iterációt. Lehetőség van még megadni a pontos megoldását az egyenletrendszernek, ha az ismert, továbbá a megoldó programegység paraméterezésére is van lehetőség.

A program képes különböző módokon ábrázolni a megoldás kiszámításának folyamatát. Ha megadta a felhasználó a pontos megoldást, akkor az alapján is képes szemléltetni a módszer működését. Ezeket a grafikonokat lehetőség van elmenteni, csak úgy mint az egész projektet. A program futása során egy naplófájl is készül a megoldás menetéről.

A programnak nem feladata, hogy az egyenletrendszer összes megoldását megtalálja, ha több is van. Az, hogy melyik gyökhöz konvergál az eljárás, azt előre nem lehet megmondani, de egy megoldást biztosan találni fog. Ha másik kezdővektorból indítjuk az iterációs eljárást, vagy más paraméterezést használunk, akkor megtalálhatjuk az egyenletrendszer más megoldásait is.

Továbbá nem képes a program megoldani a túl-, vagy alulhatározott egyenletrendszereket, tehát csak $\mathbb{R}^n \to \mathbb{R}^n$, $(n > 1)$ alakú függvények egy gyökét képes megkeresni.

A program jól használható a gyakorlati életben például fizikai feladatokban, főleg akkor, ha valamilyen természetes paraméter van a feladatban.

A program C++ programozási nyelvvel van implementálva. A grafikus kezelőfelület a *wxWidgets* könyvtár alkalmazásával készült.

2. fejezet

Felhasználói dokumentáció

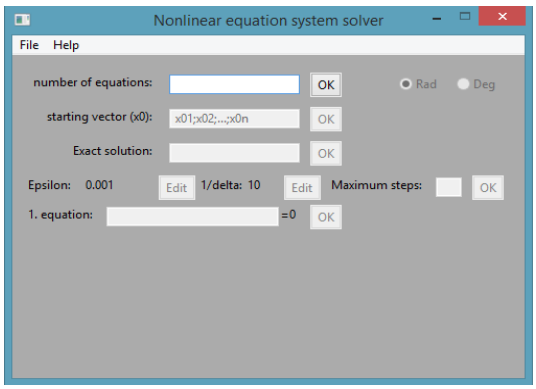
2.1. A program funkciói, felépítése

A felhasználónak kétféle lehetősége van megadni a megoldani kívánt egyenletrendszert: billentyűzetről manuálisan, vagy fájlból beolvasással.

Adatbevitel manuálisan

Ha kézzel szeretné bevinni az adatokat a felhasználó, akkor a kívánt adatokat a megfelelő szövegdobozba kell beírni, és utána az 'OK' gombra kattintani, vagy az 'ENTER' billentyűt leütni.

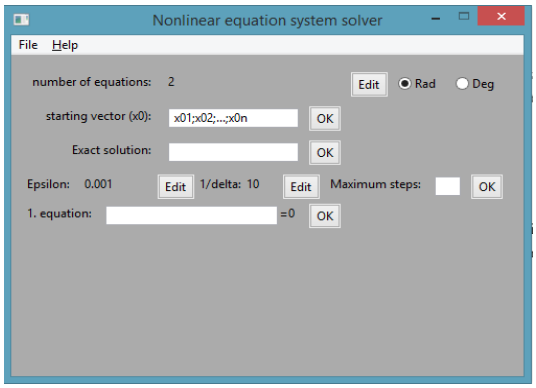
Először az egyenletrendszerben szereplő egyenletek számát kell megadnia a felhasználónak (*number of equations*). Amíg ezt nem tette meg, addig a több funkció nem él. Ha nem megfelelő adatot (kettőnél kisebb számot vagy szöveget) adtunk meg, akkor a program figyelmeztet a hibára, továbbá ha tört számot írtunk be, akkor a program a szám egész részét veszi.



2.1. ábra. A program indulás utáni képernyőképe

Ha megadtuk az egyenletek számát, akkor elérhetővé válik az összes többi funk-

ciója a programnak. Megadhatjuk a kezdővektort, az egyenleteket és a pontos megoldást, de ez utóbbi nem kötelező.



2.2. ábra.

Továbbá meg lehet adni a megoldó programegység paraméterezését is. Ezek a paraméterek a következők:

- ε (Epsilon): a számolás során végzett Broyden-módszerek megállási feltételét lehet vele szabályozni. A közelítő eljárás akkor áll meg, ha

$$||x_m - x_{m-1}|| <= \varepsilon(0,5 + ||x_m||). \qquad [1] \tag{2.1}$$

Az alapértelmezett érték $\varepsilon = 0,001$.

- Maximum steps: ez a paraméter a Broyden-módszerek során elvégzett maximális lépések számát adja meg. Ennek az paraméternek nincs alapértelmezett értéke, mert a program a függvény, a kezdővektor és az ε paraméter alapján egy optimális értékkel számol, ha nem írjuk felül azt:

$$kmax := \lceil -1 \cdot ln(\frac{\varepsilon}{||f(x_0)||} \rceil. \qquad [1] \tag{2.2}$$

- 1/delta: a folytatás módszerének lépéseinek számát lehet megadni vele. Ha x lépést kell megtennie a programnak, akkor a lépésköz 1/x. Az alapértelmezett érték $\Delta t = 0,1$.

A kezdővektor megadási formátuma, mint ahogyan az a 2.2-es ábrán is látható: $x_0^{(1)}; x_0^{(2)}; \ldots; x_0^{(n)}$. Tehát ez azt jelenti, hogy a vektor tagjait **pontosvesszővel** kell elválasztani egymástól. Fontos megjegyezni még, hogy tizedes törtek írásakor a tizedesvesszőt **vesszővel** kell jelölni mind a vektorok megadása, mind az egyenletek megadása során.

Ha a felhasználó nem ad meg kezdővektort, akkor a program figyelmezteti a

hibáról. Ha túl rövid vektort adunk meg, akkor a program kiegészíti 0-kkal, ha pedig túl hosszút, akkor eltárolja az első n darabot a megadott vektor elemei közül, ahol n az egyenletek száma. Továbbá ha a felhasználó szöveget ír be a vektor valamelyik értéke helyére, akkor a program azt úgy kezeli, mint ha 0-t adtak volna meg.

A pontos megoldás formátumára hasonló megkötések vannak, mint a kezdővektorra, csak nem kötelező megadni azt.

Fontos megjegyezni, hogy a vektorok megadásánál nincs lehetősége a felhasználónak képletek használatára, sem racionális törtszámok megadására az osztás művelet segítségével. A program nem szimbolikusan végzi a számításokat, hanem mindig kerekít. Ha tudjuk, akkor adjuk meg minél több tizedesjegy pontossággal a kívánt számot, de a program a standard C++ által kínált *double* típust használja a lebegőpontos számok ábrázolására, aminek pontossága 15 tizedesjegy.

Az egyenletek során a változókra xi formátumban hivatkozhatunk ($i = 1, \ldots, n$). A program által támogatott műveletek:

- sin, cos, tan, ctg, asin, acos, atan, actg a szögfüggvények, és azok inverzei
- sqrt, cbrt a négyzetgyök és a köbgyökvonás
- ln a természetes alapú logaritmus
- ^ a hatványozás (használat: x^y $\rightarrow x^y$)
- +, -, *, / a hagyományos műveletek, de a - operátort lehet használni negálásra is, például: $-x_1$

Továbbá lehetőség van kerek zárójelek: (,) használatára is.

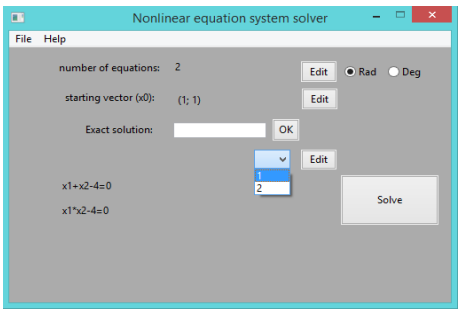
A program támogatja az Euler-féle szám és a π konstansokat. Előbbire hagyományos módon az e betűvel hivatkozhatunk, utóbbira pedig a Pi szó beírásával (a program nem érzékeny a kis- és nagybetűkre).

Az egyenleteket 0-ra rendezve kell megadni, de csak a bal oldalt, vagyis az '=0'-t nem. Például, ha az $x_1 \cdot x_2 = 4$ egyenletet szeretnénk megadni, akkor a következőt kell írni: $x1 * x2 - 4$. Ha a program által nem támogatott karaktert adunk meg, akkor a program jelzi a hibát, és nem fogadja el az egyenletet, valamint azt is figyelembe veszi, hogy nem végződhet az egyenlet operátorral, illetve helyesen van-e zárójelezve a kifejezés.

A bevitt adatokat lehet módosítani is, ehhez mindössze a megfelelő 'EDIT' gombra kell kattintanunk. Az egyenletek módosításához először ki kell választanunk egy legördülő listából a módosítani kívánt egyenlet sorszámát. Az egyenletek számát és a kezdővektort bármikor, az egyenleteket pedig csak az után módosíthatjuk, hogy megadtuk már az összes egyenletet.

Ha az egyenletek számát csökkentjük, akkor a kezdővektor méretét és az egyen-

letek számát is csökkenti a program úgy, hogy kitörli a többletet. Az egyenletek számának növelésekor a program újra bekéri a kezdővektort, és annyi egyenletet még, amennyi hiányzik az új egyenletszám eléréséhez.



2.3. ábra. Egyenletek módosítása

Ha mindennel kész vagyunk, és úgy gondoljuk megfelelően vittük be az adatokat, akkor a megjelenő 'SOLVE' gombbal meg lehet oldani az egyenletrendszert.

Beolvasás fájlból

Ha az adatokat fájlból szeretnénk beolvasni, akkor azt a File/Open menüpont, vagy a CRTL+O billentyűparancs segítségével tehetjük meg. Az input fájlnak a következő feltételeknek kell megfelelnie:

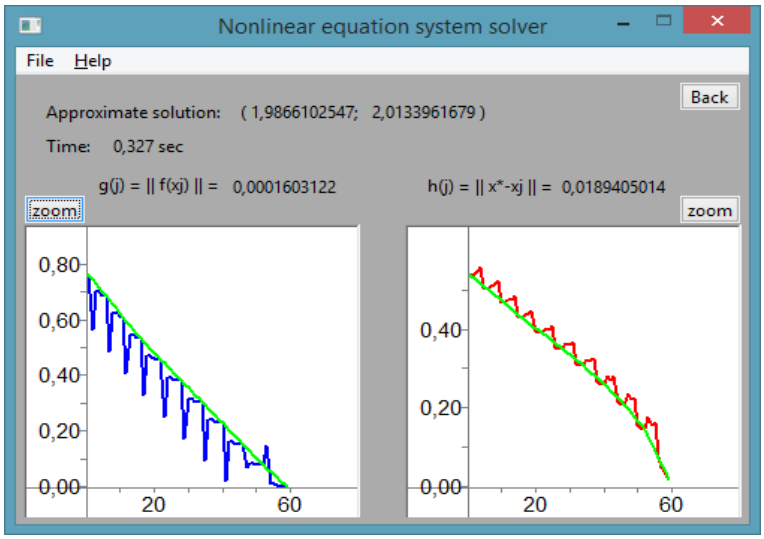
- az első sorban az egyenletek száma szerepel
- a második sorban kell feltüntetni a kezdővektort, ugyanolyan formátumban, mint ahogy a manuális adatbevitel során elvárt
- a harmadik sorban szerepel a pontos megoldás, illetve ha ez nem ismert, vagy nincs megadva, akkor az 'unknown' kifejezés
- a negyedik sortól kezdve az egyenletek számának megfelelő számú egyenlet következik, soronként tagolva.

Ezek után hasonlóan a 'SOLVE' gombra kattintással oldhatjuk meg az egyenletrendszert.

2.2. Az eredmény megjelenítése

A program által kiszámolt megoldást az 'Approximate solution:' felirat után láthatjuk tíz tizedesjegy pontossággal. A második sorban a *Time* : felirat után szerepel az, hogy a megoldás mennyi időt vett igénybe.

A program kétféle diagram ábrázolására képes:



2.4. ábra. A program egy futásának eredménye

A bal oldali ábrát (kékkel) mindig elkészíti a program, ez az aktuális közelítővektor helyén vett függvényérték normáját mutatja:

$$g(j) = ||f(x_j)||, \tag{2.3}$$

ahol f a megadott egyenletrendszernek megfelelő függvény.

A jobb oldali (pirossal) ábrát csak akkor készíti el a program, ha megadtuk neki a pontos megoldását az egyenletrendszernek. Ez az aktuális közelítővektor megoldástól vett eltérését (normáját) adja meg:

$$h(j) = ||x^* - x_j||,$$

ahol x^* az egyenletrendszer gyöke.

Mindkét esetben az abszcissza-tengely mutatja az iterációs lépések számát (j). Szintén mindkettő esetre igaz, hogy akkor működik megfelelően a program, ha a függvények értéke a 0-hoz konvergál. Előbbinél az átviteli elv miatt

$$\lim_{j\to\infty} x_j = x^* \Leftrightarrow \lim_{j\to\infty} f(x_j) = f(x^*),$$

de $||f(x^*)|| = 0$, ezért

$$\lim_{j\to\infty} ||f(x_j)|| = 0$$

Utóbbinál triviálisan minél jobban közelíti x_j vektor x^*-t, annál kisebb a távolságuk,

vagyis általánosan a különbségük normája, tehát:

$$\lim_{j\to\infty} x_j = x^* \Leftrightarrow \lim_{j\to\infty} ||x^* - x_j|| = 0.$$

Mindkettő esetben a grafikon címében az egyenlőség jobb oldalát úgy kapjuk, hogy a g, illetve h függvényekben x_j helyére a kiszámolt megoldást helyettesítjük.

Míg a fenti két grafikon a megoldás során az összes kiszámolt közelítővektor alapján ábrázol, addig a zöld grafikonok csak azokkal a vektorokkal számolnak, amelyek a közbülső megoldott egyenletrendszerek gyökei. Ez így sokkal kevesebb ábrázolható érték lenne, ezért a meglévő értékek közötti részeket lineáris interpolációval számolja ki a program.

Ha rákattint a felhasználó valamelyik grafikon felett lévő 'zoom' gombra, akkor lehetőség van annak nagyobb méretben való megtekintésére egy külön ablakban.

A grafikonokon a jobb egérgombot nyomva tartva lehet eltolni a koordinátatengelyeket.

A CTRL+egér görgő felfelé használatával lehet közelíteni, a CTRL+egér görgő lefelével pedig távolítani a grafikonon.

2.3. Egyéb funkciók

Lehetősége van a felhasználónak új projektet kezdeni a File/New menüpont, vagy a CTRL+N billentyűparancs lenyomásával. Ilyenkor minden eddigi adat, ami az aktuális projektben van elveszik, de előtte el lehet menteni a projektet. A mentésre akkor van lehetőség, ha már minden lényeges adatot kitöltött, tehát meg van adva az egyenletek száma, a kezdővektor és az egyenletek. A projekt elmentését a File/Save menüpont, vagy a CTRL+S billentyűkombináció használatával lehet megtenni.

A projektet szöveges fájlként, .txt kiterjesztéssel lehet elmenteni. A program egy ugyan olyan formátumú fájl generál, mint amilyenből adatokat képes beolvasni, tehát az elmentett projektet később könnyedén folytatni lehet.

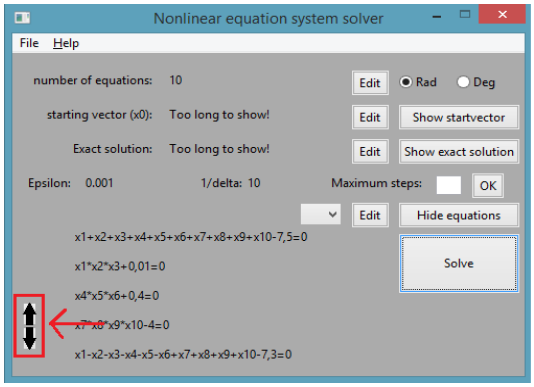
Szögfüggvények használatakor meg lehet adni, hogy fokokkal vagy radiánban számoljon a program. Ennek módosítására a jobb felső sarokban elhelyezkedő rádiógombokkal van lehetőség. Az alapértelmezett beállítás a radiánnal való számolás. Oda kell azonban ügyelni a kezdővektor és az egyenletek megadásánál, hogy melyik számolási módszert szeretnénk alkalmazni. Ha π-t szeretnénk írni, akkor a program automatikusan radiánban értelmezi azt, tehát $\pi \approx 3,14$-et helyettesít a szimbólum helyére, természetesen több tizedesjegy pontossággal.

Ügyelni kell még arra is, hogy az értelmezési tartományoknak megfelelő értéke-

ket adjunk meg, ugyanis a sin^{-1} és $cos{-}1$ függvények a [-1;1] intervallumon vannak értelmezve, a logaritmus és gyökvonás függvényeké pedig $[0; +\infty]$.

Ha nagy egyenletrendszert szeretnénk megoldani, akkor az alapértelmezés szerint nem jeleníti meg a program sem a kezdő és pontos megoldás vektorokat, sem az egyenleteket. Ha szeretnénk mégis látni őket, akkor a mellettük megjelenő megfelelő 'Show *' gombra kattintva válnak láthatóvá, illetve a 'Hide *' gombra kattintva újra el lehet rejteni őket. Ha változtatunk az adatokon, akkor az elrejtés gomb megnyomásával együtt el is menti a program a módosításokat.

Az egyenletek esetében egyszerre öt egyenletet jelenít meg a program, de a bal alsó sarokban lévő nyilak segítségével navigálhatunk közöttük.



2.5. ábra. A nyilakkal lehet navigálni az egyenletek között

Lehetőség van továbbá a grafikonok elmentésére Bitmap képként, .bmp kiterjesztéssel. Ha mindkettő grafikon meg van jelenítve, akkor program rákérdez egy dialógus segítségével, hogy melyik kép kerüljön elmentésre. A grafikonok elmentését a File/Save graph menüpont használatával tehetjük meg.

Ha megoldottunk egy egyenletrendszert, és újra meg szeretnénk oldani más kezdővektorból indítva vagy más paraméterekkel, akkor a jobb felső sarokban található 'back' gombra kattintva visszaléphetünk a kezdőképernyőre.

Ha a felhasználó szeretné nyomon követni a megoldás folyamatát, akkor a program által létrehozott *log.txt* naplófájlból ezt megteheti.

A naplófájl tartalmazza a program futásának minden lényeges lépését. A fájl elején láthatjuk a kiindulási paramétereket, majd utána a folytatás módszere által, t-vel jelölt változó léptetését 0-tól 1-ig Δt-nek megfelelően. Utána látható a közbülső egyenletrendszer megoldásának alakulása, majd megvizsgálja a program,

hogy konvergál-e az eljárás. Ha nem, akkor visszalép (*Corrected t*), és csökkenti a lépésközt. Nyomon követhetjük folyamatosan a részeredményeket is, amiket a program grafikonokon ábrázol. Az kiszámolt közelítővektorok a zárójelekben szereplő szám n-esek.

```
   ( 1.844074948; 2.389034048; -1.053164554; )        0.1615215329
   ( 1.859516811; 2.394344663; -1.053016498; )        0.11228355
   ( 1.873782658; 2.401487989; -1.077038178; )        0.04680530205

   || F(x_i+1) || = 0.04680530205 < || F(x_i) || = 0.8591337938
   The method is convergent.
   || x* - x_i || = 3.652046279
   || f(x_i) || = 5.205857349

t = 0.5
   Starting Broyden-method from ( 1.873782658; 2.401487989; -1.077038178; )
   ( 2.175552658; 3.248017989; -0.9722881779; )        3.072613003
   ( 2.739372022; 1.546738933; -1.295164972; )        2.103648104
   ( -205.3082559; 173.9194602; 284.4101812; )        10155396
   ( 2.517039403; 2.263271358; -1.889468744; )        6.792531254
   ( 2.907215629; 2.322419558; -1.641426468; )        7.097012098
   ( 2.778099266; 2.432133037; -1.801868537; )        8.17533258
   ( 2.805255777; 2.480839659; -1.811966572; )        8.610247865
   ( 2.816209304; 2.499998478; -1.816210543; )        8.7871
   ( 2.816483564; 2.499998594; -1.81648466; )        8.7902

   || F(x_i+1) || = 8.7902 >= || F(x_i) || = 0.9047932008
   The method is not convergent, stepping back.

Corrected t = 0.45
   ( 2.025552658; 2.848017989; -1.022288178; )        1.60635359
   ( 2.320379487; 1.967325176; -1.195650682; )        1.298645868
   ( -0.8504938743; 4.823414866; 2.679482028; )        7.513099154
   ( 4.47254082; 1.683036421; -3.572131198; )        22.51524094
   ( -3.182769262; 3.738588032; 3.414862946; )        36.29189163
```

2.6. ábra. A naplófájl egy részlete

A menüben található *Help* menüpont alatt érhetjük el a programhoz tartozó dokumentációt a *Documentation* almenü segítségével *.pdf* formátumban. A program ennek megnyitásához az adott számítógépen (amin futtatva van a program) alapértelmezett pdf olvasót használja.

2.4. CD tartalma

A *documentation* mappában található a programhoz tartozó dokumentáció .pdf formátumban.

A *program* mappában található a futtatható fájl, ennek neve *nonlinsolver.exe*. Ezt fájlt futtatva indíthatjuk el a programot.

A tesztfájlok az *input* mappában helyezkednek el, azon belül az *error* mappában vannak a hibás bemenetet tesztelő fájlok. Ebben a mappában található még az egyenletek görgetéséhez szükséges két kép az *images* mappán belül.

Az *output* mappába készíti a program a naplófájlt, aminek neve *log.txt*. Természetesen a CD-re nem tud naplófájlt készíteni a program, ezért ennek a funkciónak a használatához a CD tartalmát át kell másolni egy olyan lemezterületre, ahol van jogunk írásra.

A *sources* mappán található a program forráskódja.

A *tester* mappában található a fehér doboz teszteket elvégző *main.cpp*, a később leírtak szerint módosított forrásfájlok, valamint a futtatható fájl, a *tester.exe*.

3. fejezet

Fejlesztői dokumentáció

3.1. Matematikai háttér

Egyváltozós Newton-módszer

A Newton-módszer függvények zérushelyeinek meghatározására használt iteratív módszer. Fontos feltétel, hogy a függvénynek differenciálhatónak kell lennie. Ha a függvény kétszer deriválható, akkor a módszer alkalmazható a függvény szélsőértékeinek meghatározására is.

Az alapötlet a következő: kiindulunk egy pontból, ami a gyökhöz elég közel van, ebből a pontból indítjuk az iterációt: x_0. Ezek után, mivel ismerjük a függvényt, ki tudjuk számítani a függvény x_0-ban felvett értékét: $f(x_0)$. Ezután könnyen kiszámolható a függvényhez az $(x_0, f(x_0))$ pontban húzott érintő metszéspontja az x tengellyel: x_1. Ez általában jobb közelítése a gyöknek, mint az eredeti pont. A módszert folytatjuk, most x_1-ből kiindulva addig, amíg el nem érjük a zérushely adott pontosságú közelítését.

Egy adott (x_0, y_0) ponton áthaladó, m meredekségű egyenes egyenlete a következő:

$$(y - y_0) = m(x - x_0)$$

Jelen esetben $m = f'(x_i)$, mivel a deriváltfüggvény megadja egy adott pontban húzott érintő egyenes meredekségét. Ha a fenti egyenlet alapján felírjuk az $(x_i, f(x_i))$ ponton áthaladó, az f függvényhez húzott árintő egyenletét, akkor meg tudjuk határozni, hogy hol metszi az abszcissza-tengelyt:

$$\left.\begin{array}{r} f(x) - f(x_i) = f'(x_i)(x - x_i) \\ f(x) = 0 \end{array}\right\} \quad \Longrightarrow \qquad x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Szelőmódszer

A szelőmódszer a Newton-módszer egy változata. A probléma az a Newton-módszerrel, hogy meg kell adni a függvény deriváltját, ami azért gond, mert előfordulhat, hogy nem, vagy csak nagy költségek árán tudjuk azt meghatározni.
Egy f függvény x_i helyen vett deriváltja:

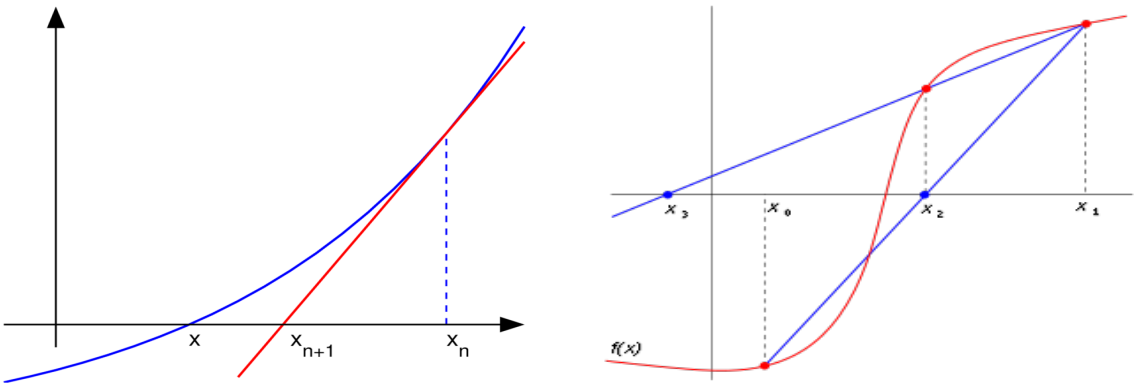
$$f'(x_i) = \lim_{h \to 0} \frac{f(x_i + h) - f(x_i)}{h}$$

Ha h kellően kicsi ($h := x_{i-1} - x_i$), akkor ez a határérték helyettesíthető a differenciahányadossal:

$$f'(x_i) \approx \frac{f(x_i + x_{i-1} - x_i) - f(x_i)}{x_{i-1} - x_i} = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

Ha ezt behelyettesítjük a Newton-módszer iterációs képletébe, akkor megkapjuk a szelőmódszer egy lépését:

$$x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$



3.1. ábra. Példa a Newton-módszer (balra) és a szelőmódszer működésére.

Mint ahogyan a 3.1 ábrán is látható, a szelőmódszer geometriai jelentése a következő: kiválasztunk két kezdeti pontot, x_0-t és x_1-et. Ezek után szerkesztünk egy egyenest az $(x_0, f(x_0))$, és az $(x_1, f(x_1))$ pontokra. Ennek a szelőnek az abszcisszatengellyel való metszéspontja legyen $(x_2, 0)$. Ezután folytatjuk ezt az eljárást x_1-re és x_2-re. Ennél a módszernél is fontos megjegyezni, hogy akkor fog x_n konvergálni a függvény gyökéhez, ha a kezdeti pontokat a gyökhöz elég közel választottuk meg.

Többváltozós Newton-módszer

A fent ismertetett módszereket akkor tudtuk alkalmazni, ha egy egyenletünk és egy változónk volt. Lehetőség van azonban arra, hogy kiterjesszük ezeket az eljárásokat többváltozós függvényekre is. Az f függvény ennek megfelelően most képezzen az n dimenziós térből az m dimenziós térbe: $f : \mathbb{R}^n \to \mathbb{R}^m$. (tehát n darab ismeretlen van, és m darab összefüggés)

Ebben az esetben nincs értelme a derivált-függvényről beszélnünk, így az f Jacobi-mátrixával kell helyettesíteni, jelöljük ezt J-vel:

$$J_{i,j}(x) := \frac{\partial f_i(x)}{\partial x_j}, \qquad i, j = 1, \ldots, n$$

Ezek után felírható az iterációs eljárás képlete nagyon hasonlóan, mint az egyváltozós esetben:

$$x_{i+1} = x_i - J_f(x_i)^{-1} f(x_i)$$

A való életben egy függvény Jacobi-mátrixának kiszámítása nehéz lehet. Általában a deriváltakat differenciálképletekkel közelítjük. Ha minden m iterációs lépésre az $f'(x_m)$ Jacobi-mátrix elemeit az

$$f_{ij} = \frac{\partial f_i(x)}{\partial x_j} \approx \frac{f_i(x_m + h_{ij} e_j) - f_i(x_m)}{h_{ij}}, \quad i, j = 1, \ldots, n$$

közelítéssel számoljuk ki, ahol e_j a j-edik koordináta egységvektor, h_{ij}-k alkalmas, nullától különböző számok, akkor az eljárást diszkretizált Newton-módszernek nevezzük.

Előfordulhat az is, hogy a Jacobi-mátrix a rendelkezésünkre áll. Ilyenkor ügyelni kell a tárolási struktúrára, ugyanis nagy n-re a nemlineáris egyenletrendszerek Jacobi-mátrixai tipikusan ritkák.

Broyden-módszer

A Broyden-módszer a szelőmódszer általánosítása, amelynél a cél az, hogy ne kelljen a Jacobi-mátrixot alkalmazni a fent említett számítási nehézségek miatt.

Broyden javaslatára a Jacobi-mátrix közelítését (jelöljük ezt B-vel) iteratívan számoljuk a következő módon:

$$B_{i+1} = B_i + \frac{(f(x_{i+1}) - f(x_i) - B_i \Delta x_i)\ \Delta x_i^T}{\Delta x_i^T \Delta x_i},$$

ahol $\Delta x_i := x_{i+1} - x_i$. Kezdő B_0 mátrixnak az egységmátrixot szokták használni.

Az iteráció így a következő alakban írható fel:

$$x_{i+1} = x_i - B_i(x_i)^{-1} f(x_i)$$

Tovább lehet javítani az algoritmust, ha nem végezzük el B invertálását, ami költséges. Helyette, B QR-felbontásának segítségével számítsuk x_{i+1}-t.

$$B_i^{-1} = (Q_i R_i)^{-1} = R_i^{-1} Q_i^{-1} = R_i^{-1} Q_i^T$$

Q_i ortogonális, ezért $Q_i^{-1} = Q_i^T$, valamint R_i felső háromszögmátrix, ezért inverze viszonylag gyorsan $\Theta(n^2)$ művelettel számolható. Már csak azt kell megmutatni, hogy Q_{i-1}-ből és R_{i-1}-ből $O(n^2)$ művelettel közvetlen kiszámítható Q_i és R_i. [1]

$$u := \frac{(B_i - B_{i-1})\Delta x_i}{||\Delta x_i||^2}$$

$$v := Q_{i-1}^T u$$

Ekkor $B_i = Q_{i-1}(R_{i-1} + v\Delta x_i^T)$, és kiszámítandó $R_{i-1} + v\Delta x_i^T = QR$. Itt lényeges, hogy R_{i-1} felső háromszögmátrix, és $v\Delta x_i^T$ rangja 1. A QR-felbontásra használjuk a Givens-eljárást. $R_{i-1} + v\Delta x_i^T$ speciális alakja miatt elegendő $2n-2$ forgatást végezni. Ezek után $Q_i = Q_{i-1}Q$ és $R_i = R$.

A folytatás módszere

A Newton-féle módszer nem mindig konvergens, ezért fontos a módszer globalizálása, ugyanis a dimenzió növekedésével együtt csökkennek a megoldásokat körülvevő vonzási gömbök átmérői, és egyre valószínűbb, hogy olyan kezdőpontot fogunk választani, ahonnan kiindulva a módszer nem konvergál a megoldáshoz. Ennek általános lehetősége a folytatás módszere, ami egy esetleg kedvezőtlenül megválasztott x_0-ból kiindulva olyan x_{k-1} kezdeti vektort gyárt a Newton-módszer beindítására, amely a lokális konvergencia vonzási gömbjében fekszik. Beágyazzuk az $f(x) = 0, f(x) \in \mathbb{R}^n$ egyenletrendszerünket egy n+1 dimenziós feladatba, bevezetve egy t paramétert úgy, hogy t=0-ra ismert a megoldása, t=1-nél viszont az eredeti feladathoz kell jutnunk.

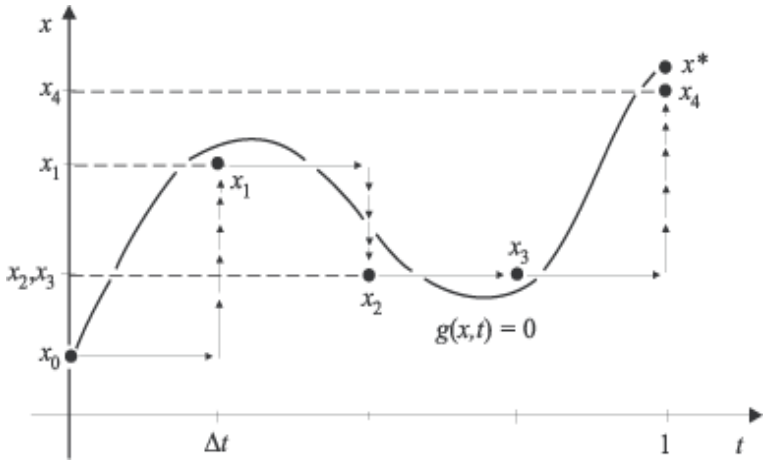
$$g(x,t) := f(x) + (t-1)f(x_0), \quad x_0 \textit{ adott.}$$

Itt t-t futtatva 0-tól 1-ig egyenletrendszereket kapunk, amelyeket meg kell oldanunk:

$$g(x_i, t_i) := f(x_i) + (t_i - 1)f(x_0) = 0, \quad i = 1 \ldots k \tag{3.1}$$

ahol $t_i = t_{i-1} + \Delta t = \frac{i}{k}$, és a k lépésszám elég nagy. A közbülső egyenletrendszereket megoldhatjuk például fent részletezett Broyden-módszerrel.

A gyakorlatban nem mindig jó megoldás fix lépésközzel haladni. Ha a Broyden-módszer nem konvergens, akkor nagyobb k-t kell választani, ami csökkenti a lépésközt. Az konvergenciára vonatkozó elégséges feltétel ([1], 6.4 tétel) kimondja, hogy $\forall\varepsilon$-hoz lehet olyan lépésközt választani, hogy a feladat megoldható legyen. Ezt viszont előre nem lehet meghatározni, és a program minden sikertelen kísérlet után felezi a lépéstávot. Ha viszont sokadszorra sem konvergens, akkor már annyira kicsi lesz a lépéshossz, hogy megéri inkább más paraméterezést választani, mert a számítás túl pontatlan lenne.



3.2. ábra. A folytatás módszere. [1]

Az 2.4-es ábrán jól látszódik, hogy a módszer lineárisan konvergál a megoldáshoz, ami nem meglepő. Igaz ugyan, hogy a Newton-módszereknek általában nagyobb a konvergenciarendjük, de itt több ilyen eljárás van elvégezve egymás után. A folytatás módszere folyamatosan transzformálja az adott függvényt $t = 0$-tól $t = 1$-ig, ezért a közbülső egyenletrendszerek megoldásai is lineárisan konvergálnak az eredeti függvény gyökéhez. Az ilyen folytonos deformációt, amely egy objektumot átvisz egy másikba homotópiának nevezzük.

Előfordulhat, hogy lassul a konvergencia, de ekkor is lineáris marad. Ez akkor lehetséges, ha az egyik közbülső Broyden-módszer nem konvergál, és vissza kell lépnie a programnak, hogy kisebb lépésközzel próbálkozzon.

A folytatás módszer alkalmazása nem mindig célszerű. Lehetséges, hogy túl sok idő megy el a közbülső egyenletrendszerek megoldására, pedig azok nem érdekesek a feladat szempontjából. Előfordulhatnak azonban olyan feladatok, ahol másképp nem jutunk megoldáshoz. A módszer feltétlenül akkor ajánlott, ha természetes paraméter van a feladatban, amelynek bizonyos értékére ismerünk megoldást.

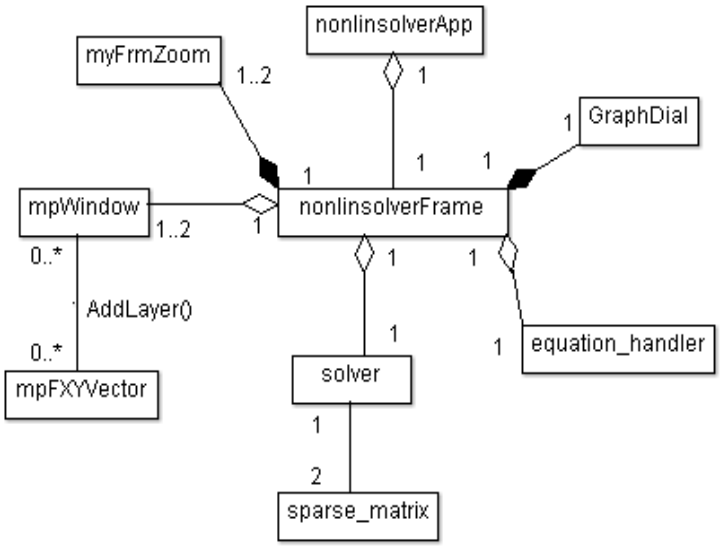
3.2. A program szerkezete

A programnak három fő szerkezeti egysége van: a grafikus kezelőfelület, röviden GUI, a modell réteg, ami tárolja a programhoz szükséges adatokat és a számításokért felelős réteg, ami megoldja az egyenletrendszert. Ezek jól elkülönülnek egymástól, ez csökkenti a szerkezeti bonyolultságot és növeli a rugalmasságot.

A GUI a nonlinsolverFrm, a GraphDial és a myFrmZoom osztályokból áll. Felhasználja továbbá a *mathplot* csomag mpWindow és mpFXYVecotor osztályait is.

A *mathplot* egy ingyenes csomag a *wxWidgets* könyvtárhoz, ami egy keretrendszert biztosít függvények két dimenziós ábrázolására. 2007-től a projektért Davide Rondini felelős (`http://wxmathplot.sourceforge.net/`).

A modell réteget az equation_handler és a sparse_matrix osztályok, valamint a linalg csomag alkotják. Az egyenletrendszer megoldásáért felelős osztály pedig a solver osztály.



3.3. ábra. A program osztálydiagramja

A program indításakor a*nonlnsolverApp* osztály *OnInit()* metódusa fut le először. Ez nem csinál többet, mint létrehozza a fő ablakát a programnak, vagyis példányosítja a *nonlinsolverFrame* osztályt, és beállítja kezdőablaknak a létrehozott példányt. A GUI szerkezetét, megjelenítését és a felhasználói interakciókat ez fogja kezelni.

A továbbiakban vannak részletezve a különböző osztályok. Az osztálydiagramokban nincs minden adattag és eljárás feltüntetve, csak a fontosabbak, például a getter-setter metódusok ki vannak hagyva.

3.3. A grafikus kezelőfelület felépítése

A felhasználói interfész kinézetéért és a felhasználói interakciók kezeléséért a *nonlinsolverFrame* osztály a felelős, ami a *wxFrame* osztályból van származtatva. Példányosításakor a konstruktor beállítja a segédváltozókat, példányosítja az *equation_handler* és *solver* osztályokat, valamint létrehozza az interfészt alkotó elemeket. Ez utóbbit a *CreateGUIControls* metódus végzi el.

Vegyük észre, hogy az ablakon lévő elemek mutatókkal vannak megvalósítva. Erre azért van szükség, mert így csak az ablak bezárásakor szűnnek meg az elemek. Ez azért fontos, mert így elkerüljük az olyan hibákat, hogy egy olyan objektumra hivatkozzon egy esemény (például az egérrel rákattint a felhasználó), ami már megszűnt. Másrészről azért is fontos mutatókkal létrehozni az elemeket, mert egy ablak, amikor megszűnik törli az összes gyerek objektumát is a *Destroy()* metódus meghívásával. Így elkerüljük az olyan hibákat is, amik abból adódnának, ha egy objektumot kétszer akar a program törölni.

Ennek megfelelően a konstruktorban a *new* paranccsal kell létrehozni az objektumokat. Az elemek konstruktorának paraméterezése ugyan olyan, mert mindegyik a *wxControl* osztályból van származtatva. A paraméterek, amiket meg kell adni sorban: szülő, azonosító, felirat, pozíció, méret, stílus.

A programhoz tartozó menü létrehozását a *wxMenuBar* osztály példányosításával lehet megtenni. Ez az osztály létrehozza a menüsort. Hogy menüpontokat tudjunk felvenni, ahhoz a *Menu* osztályt kell példányosítani, ehhez hozzáadni a kívánt almenüpontokat az *Append* parancssal, majd az egészet fel kell fűzni az eredeti menüsorra.

Elnevezési konvenciók

Mind az egyenletek száma, a kezdővektor, a pontos megoldás, a megoldó paraméterei és az egyenleteknek vannak közös pontjaik. Ami az egyenletek számához tartozik, annak elnevezése numbeq_* (number of equations), ami a kezdővektorhoz, annak startvect_* (starting vector), ami a pontos megoldáshoz, annak ex_* (exact result), ami az egyenletekhez, annak eq_* (equations), a három paraméternek, az ε-nak, a lépésköznek és a maximum lépésszámnak pedig rendre epsilon_*, k_* és kmax_* az elnevezése. A csillag helyére kerül a feliratok esetében (ami az ablakon az első oszlopban szerepel) a *stat*, a szövegdobozok esetében a *box*, az 'OK' gombnál a *but* és az 'EDIT' gombok esetében pedig az *edit* szó. Az egyenleteket leszámítva közös elnevezésű még a felirat is, ami az 'OK' gombra kattintás után kerül megjelenítésre a szövegdoboz helyén, ennek neve *stat2*. Tehát például ha a kezdővektor szövegdobozára akarunk hivatkozni, akkor az így tehetjük meg: *startvect_box*.

Az események bekövetkezésekor végrehajtandó függvények nevei is hasonlóan

vannak meghatározva, mint az elemeké: *név_ *click*. A * helyére kerülhetnek a *but, edit és show* szavak akkor, ha van ilyen esemény. Ha az előzőek közül egyik sem létezik, például a 'Solve' gomb esetében, akkor nem kerül a * helyére semmi.

Felhasználói interakciók

A wxWidgets a felhasználói interakciókat egy ún. *event table* segítségével kezeli, és az interakciókat, vagy eseményeket *event*-eknek, vagy eseményeknek nevezi. Ez a tábla a *nonlinsolverFrame.cpp* fájlban van deklarálva. A programban ötféle event kezelése van megoldva. A felhasználó használhatja a menüt - EVT_MENU, kattinthat gombokra - EVT_BUTTON, adatbevitelnél az enter gomb lenyomásával is elfogadhatja a beírt szöveget, nem csak a megfelelő gombra kattintva - EVT_TEXT_ENTER, a radián és fok közötti váltás rádiógombok segítségével van megoldva, ezekre külön event van - EVT_RADIOBUTTON és végül, ha bezárul a program, akkor az EVT_CLOSE esemény következik be. Általában igaz az az event tábla elemeire (ez alól a bezáró esemény a kivétel), hogy a következőképpen kell paraméterezni őket:

1. paraméter annak az elemnek (gomb, szövegdoboz stb.) az azonosítója, amelyik kiváltja az eventet. Az azonosítók a könnyebb kezelhetőség érdekében a header fájlban egy enumerátorral, vagyis felsorolóval vannak megadva. Ez annyit jelent, hogy minden elemnek, ami deklarálva van, létre van hozva egy azonosító, ami nevesítve van, hogy ne számként kelljen rá hivatkozni, hanem beszédesebb neve legyen. Az elnevezések egy mintára épültek, vagyis például a fentebb említett kezdővektor szövegdobozának azonosítója: ID_STARTVECT_BOX, tehát az 'ID_' szó után következik magának az elemnek a neve csupa nagybetűvel.

2. paraméter az a függvény, ami meghívásra kerül az event bekövetkezésekor. Például, ha azt akarjuk lekezelni, hogy a felhasználó kattintott egy gombra, akkor a következőt kell írni:

```
BEGIN_EVENT_TABLE(nonlinsolverFrame,wxFrame)
...
    EVT_BUTTON(button_id,nonlinsolverFrame::butClick)
...
END_EVENT_TABLE
```

A bezáró esemény deklarálásakor elég megadni paraméterben a bezáráskor végrehajtandó metódus nevét.

Kivételkezelés

A program kétféle kivételkezelést alkalmaz. Ha az adatbevitel során azonnal el lehet dönteni, hogy hibás az adat (például az egyenletek számának negatív számot

adtunk meg), akkor a program egy üzenetet küld a felhasználónak, hogy értesítse a hibáról. Ha nem lehet egyből eldönteni az adat helyességét (például hibás egyenlet), vagy az egyenletrendszer megoldása során ütközik hibába a program, akkor a hagyományos *try-catch* szerkezetet használja.

Azt, hogy egy egyenlet hibás-e, azt az egyenletek lengyelformára hozása közben lehet eldönteni. Az equation_handler osztályban az *errors* felsoroló tartalmazza a lehetséges hibákat. Ezek a következők:

- UNKNOWN: az egyenletben ismeretlen karaktert talált a program.
- OPERATOR_END: az egyenlet nem végződhet operátorral.
- XI_ERROR: nem a megfelelő változóindexelést alkalmaztuk. A használható változó indexek: $x1, x2 \dots xn$, ahol n az egyenletek száma.
- INVALID_BRACKET: helytelenül van az egyenlet zárójelezve.

Ha az adott paraméterekkel nem jut el a program a feladat pontos megoldásához, akkor a NOT_CONVERGENT hibát generálja.

Az egyenlet megoldása

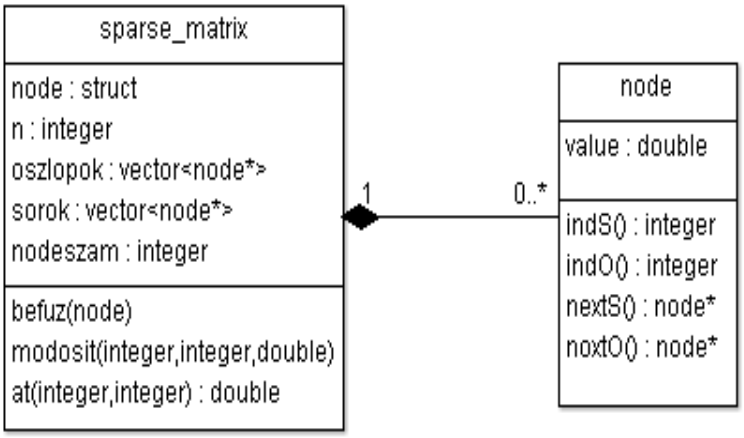
Az egyenlet megoldását akkor kezdi el a program, amikor a felhasználó a 'SOLVE' gombra kattint. A kattintáskor a *solveClick* metódus hajtódik végre. Először elrejt minden mezőt a program, majd beállítja a megoldó programegység paramétereit, a pontos megoldást, ha van, illetve azt, hogy radiánban vagy fokban történjen a számolás. Ha kivételt dob a megoldó, akkor visszaállít mindent az eredeti helyzetbe. Ha viszont ki tudta számítani a pontos megoldást a program, akkor lekérdezi a megoldás során létrehozott közelítővektorokat, és ábrázolja őket. A 2.4 ábrán zölddel jelölt függvény esetében még lineáris interpolációt végez.

3.4. A modell réteg

A modell réteg három részből tevődik össze:

Ritka mátrix

A sparse_matrix osztály a számítások hatékonyabbá tételéhez kell, a ritka mátrixok hatékony eltárolását valósítja meg.



3.4. ábra.

1. Definíció. *Egy $M \in \mathbb{R}^{nxn}$ mátrixot akkor nevezünk ritka mátrixnak, ha $m \ll n^2$, ahol m a mátrix nemnulla elemeinek száma.*

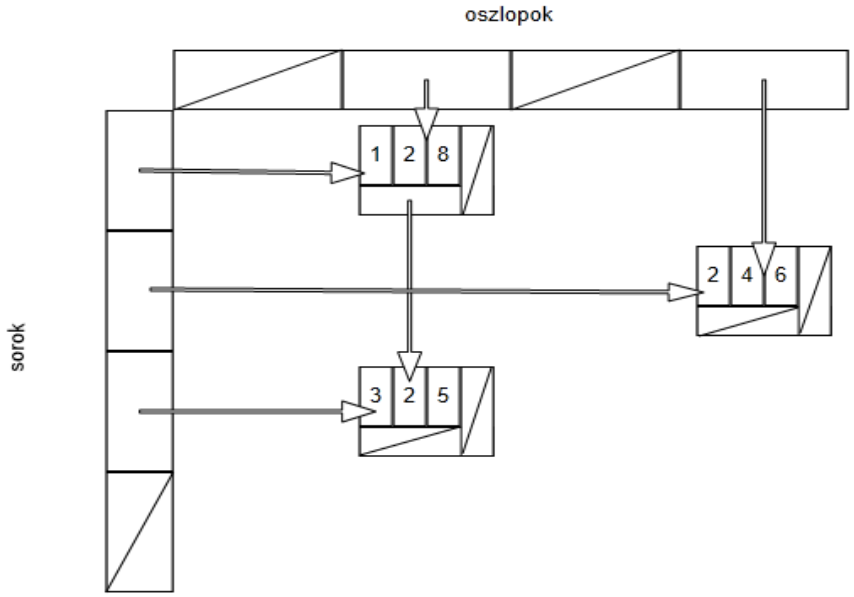
Ha egy mátrix ritka, akkor nem célszerű az összes elemét tárolni, hanem csak elég a nemnullákat. Ez úgy van megvalósítva, hogy a program létrehoz és karbantart két tömböt, aminek az tagjai mátrixelemekre mutató mutatók, vagyis *pointerek*. Az ilyen elemek a *node* struktúra példányai. A fenti ábra szerint egy mátrixhoz akármennyi elem tartozhat. Ez persze nem teljesen igaz, mert csak n^2 darab, de ez a konkrét példánytól függ, nem az osztálytól.

Egy elem tartalmazza azt, hogy melyik sorban, illetve oszlopban van, az adott sorban és oszlopban a következő nemnulla elemre mutató pointer (ha nincs ilyen akkor a mutató NULL), és egy értéket. A két tömb tartalmazza a sorokat és az oszlopokat. Például a sorok tömb ötödik eleme a mátrix ötödik sorának első nemnulla elemére mutat.

Erre az osztályra a QR-felbontás hatékony elkészítése miatt, és utána a felsőháromszög mátrix R-rel való számítások gyorsítása miatt van szükség, ugyanis a mátrixszorzások ritka mátrixokkal gyorsabban elvégezhetőek, mint sűrű mátrixokkal. Két sűrű mátrix összeszorzását $\Theta(n^3)$ idő alatt lehet elvégezni, ha a szorzás

a releváns művelet. Ha az egyik mátrix ritka (nemnulla elemeinek száma legyen $m \ll n^2$), akkor a szorzás elvégezhető $\Theta(m \cdot n) = \Theta(n)$, azaz lineáris időben. Ez azért lehetséges, mert a sok nullával való szorzást ki lehet így hagyni, ugyanis a nullákat nem tároltuk el, és azokat az elemeket átugorja az algoritmus. Ha pedig két ritka mátrixot akarunk összeszorozni, akkor még gyorsabban elvégezhető a szorzás, de erre pontos műveletigényt nem lehet adni, mert függ a mátrixok ritkasági struktúrájától is, azaz hogy hogyan helyezkednek el bennük a nemnulla elemek.

A ritka mátrixokba való adatmódosítás körülbelül ugyan olyan gyors, mint sűrű mátrixok esetében. Az adatbefűzés valamivel lassabb, mivel két vektorba kell az elemet elhelyezni, de még így is lineáris időben elvégezhető.



3.5. ábra. Példa egy 4x4-es ritka mátrix ábrázolására, a számok rendre a sorindex, oszlopindex, érték.

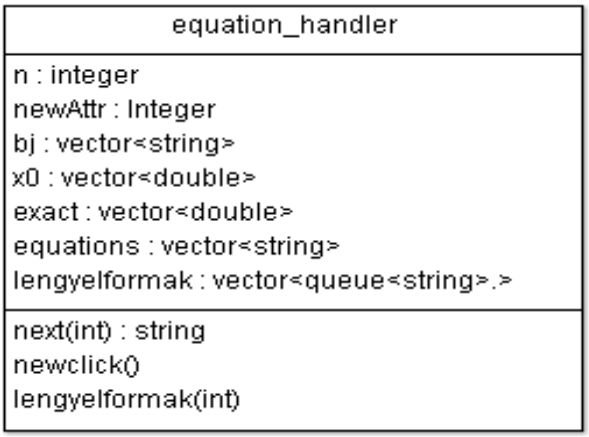
A ritka mátrixok kezelésére három metódus van implementálva az osztályban:

- befuz(node): ezzel az eljárással lehet feltölteni a mátrixot elemekkel. Szükséges még, hogy *node*-okat tudjunk létrehozni. Ennek paraméterezése a példányosításkor rendre: sorindex, oszlopindex, érték.
- modosit: egy adott sor és oszlopindexű elem módosítását lehet vele elvégezni.
- at: egy adott sor és oszlopindexű elem értékének lekérdezésére szolgál.

Fontos megjegyezni, hogy ezen osztály metódusainak paraméterezésénél az indexelés a programozási nyelvektől eltérően 1-től, és nem 0-tól kezdődik.

Az egyenletrendszer kezelése

Az equation_handler osztály: tárolja az egyenletrendszer adatait: az egyenleteket, azok lengyelformáját és a kezdővektort.



3.6. ábra.

Az n változó tartalmazza az egyenletek számát, az *x0* és az *exact* tömb a kezdővektort és a pontos megoldást. Ha nincs megadva a pontos megoldás, akkor a tömb mérete 0. Az *eqations* tömb tartalmazza az egyenleteket, a *lengyelformak* pedig ezeknek a lengyelformára hozott alakját, mindet egy sor struktúrában, hogy megkönnyítse a kiértékelést. A *prec* map tartalmazza az operátorok precedenciáit. Minél nagyobb számot kapott egy művelet, annál nagyobb a precedenciája. A precedencia sorrend a következő:

1. az összes függvény: szinusz, koszinusz, logaritmus, gyökvonás ...
2. hatványozás
3. szorzás, osztás
4. összeadás, kivonás
5. negálás

A *bj* tömb tartalmazza azokat az operátorokat, amelyeket balról-jobbra kell kiértékelni (a négy alapművelet).

2. Definíció. *A lengyel forma (RPN - Reverse Polish Notation) a matematikai kifejezések olyan megadása, ahol az operátorok nem az operandusok között infix módon, hanem azok után postfix módon helyezkednek el.*

Továbbá a lengyel formában felírt kifejezések nem tartalmaznak zárójeleket és műveleti sorrendet, hanem balról-jobbra kell kiértékelni a kifejezést. Az *equation_handler* osztálynak nem feladata a kifejezések kiértékelése. Amikor a program futása során sorban adjuk meg az egyenleteket, akkor ez az osztály lengyel formára hozza azokat, és mindkettő megadási módon eltárolja az egyenleteket.

Egy kifejezés lengyel formára hozásához szükség van egy veremre, és egy eljárásra, ami képes szimbólumonként olvasni a kifejezésből. Ez jelen esetben a *next* függvény. A *next* függvény hasonlóan működik egy fordítóprogram lexikális elemzőjéhez. Először olvas egy karaktert, és ha van olyan szimbólum, ami ezzel az olvasott karakterrel kezdődik, akkor tovább olvas (az értelmező nem érzékeny a kis- és nagybetűkre). Ha x-et, vagy számjegyet olvasott, akkor utána addig olvas, amíg továbbra is számjegyeket olvas. Képes kiszűrni az olyan hibákat, hogy operátor nem állhat az egyenlet végén, és az x_i változók esetében az $i > n$ és az $i = 0$ változókat nem értelmezzük, ahol n az egyenletek száma. Továbbá karban tartja az *open* segédváltozót, amivel képes felismerni egy helytelenül zárójelezett egyenletet. Ha '('-t olvas, akkor $open := open + 1$, ha ')'-t, akkor $open := open - 1$. Ha ')' olvasásakor $open = 0$ vagy az egész kifejezés feldolgozása után $open \neq 0$, akkor a kifejezés helytelenül van zárójelezve.

Azt is képes a *next* eljárás felismerni, hogy a ' - ' operátort negálásra vagy kivonásra akarjuk-e használni. Ennek eldöntésére van bevezetve az *elozo* segédváltozó. Akkor lehetséges az, hogy negálásra akarjuk használni a ' - ' jelet, ha az operátor előtt közvetlenül nincs operandus, esetleg '(' lehet. Az osztály példányosításakor beállítja a konstruktor ezt a változót: $elozo := 1$. Ennek megfelelően ha '('-t olvas az értelmező, akkor $elozo := 1$, különben $elozo := -1$. Tehát ha a *next* függvény '-'-t olvas, és $elozo \neq 1$, akkor kivonás, ha $elozo = 1$ akkor pedig negálás. Ha negálást ismert fel az eljárás, akkor '- -' a visszatérési érték, hogy a program többi részegsége el tudja egyértelműen dönteni, hogy melyik operátorról van szó.

Az egyenletek lengyelformái sor struktúrában vannak tárolva, mert folyamatosan tölti fel az algoritmus úgy, hogy mindig a lengyelforma végére ír. Kiértékeléskor pedig szinéán jól fog jönni ez a tárolási módszer, mert szekvenciálisan, balról-jobbra haladva értékeli majd ki a program a lengyelformákat. Egy kifejezés lengyel formára hozásának algoritmusa:
Ha a *next* függvény által olvasott következő szimbólum:
– operandus: beírjuk a lengyelformájának végére
– operátor: a vermet üríti a lengyelforma végére az első nálánál kisebb vagy egyenlő (balról-jobbra típusú operátor esetén szigorúan kisebb) operátorig (ezt már nem beleértve), vagy a verem aljáig, vagy '('-ig. Utána a verembe teszi az operátort.

– '(': verembe teszi
– ')': üríti a vermet a lengyelforma végére '('-ig. Utána kiveszi a veremből a '('-t is, de nem írja a lengyelformához hozzá.
Ha nincs több szimbólum a kifejezésben, akkor kiürítjük a vermet a lengyelforma végére. [10]

Lineáris algebra csomag

A *linalg* csomag nem tartalmaz osztálydefiníciót, csak matematikai függvényeket, főleg mátrix és vektorműveleteket. A csomag a következő eljárásokat tartalmazza:

- myAtof: egy karakterláncot konvertál lebegőpontos számmá. A különbség ez és C++-ban megszokott *atof* eljárás között, hogy a *myAtof* függvényben a tizedesvesszőt a vessző jelöli, míg a hagyományos társában a pont.
- T: mátrix transzponáltat számol
- operator*: a következő műveletek elvégzésére lehet használni: mátrix · mátrix, mátrix · konstans, vektor · konstans, vektor · vektor=mátrix, vektor · mátrix
- vectorMultScalar: két vektor skaláris szorzása
- operator+: lehet használni mátrixok és vektorok összeadására
- inverseUpperTriangular: egy felsőháromszög mátrixnak számolja ki az inverz mátrixát [5]
- isUpperTriangular: egy mátrixról eldönti, hogy felsőháromszög struktúrájú-e
- toSpare, toDense: sűrű és ritka mátrixok közötti konverziót teszi lehetővé
- javit: kijavítja a számítási hibákat egy mátrixban a nulla körül. A mátrix i.-ik sorának j.-ik oszlopának eleme legyen $m_{i,j}$. Ekkor ha $|m_{i,j}| < thres$, akkor $m_{i,j} := 0$, ahol thres egy megfelelően kicsi küszöbindex.
- norm: egy vektor 2-es normáját számítja ki, vagyis: $norm(v) = \sqrt{\sum_{i=1}^{n} v_i^2}$, ahol n a vektor hossza
- setRCS, choose: ezeket a givens eljárás alkalmazza (ld. később)
- givens: egy mátrix QR-felbontását készíti el

A csomag tartalmaz még kettő segédstruktúrát:

- qr: kettő mátrixot tartalmaz, a *givens* eljárás visszatérési értéke
- dd: egy indexpárt tartalmaz, a *choose* függvény visszatérési értéke

Megtalálható még a csomagban egy küszöbindex:

- thres: ez a szám határozza meg, hogy mekkora kerekítési hibát enged meg a program a futása során, például a *javit* függvény működése során. Ez az érték 10^{-12}.

A QR-felbontást a givens() függvény készíti el, ahogy a nevéből is látszik a Givens-eljárással. [6] A Givens-forgatások lényege, hogy egy alkalmas mátrixszal balról megszorozva az eredeti mátrixot ki lehet nullázni egy elemet a főátló alatt.
A forgatási mátrix:

$$G(i,j,\Theta) = \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & -s & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \tag{3.2}$$

ahol $c = cos(\Theta)$ és $s = sin(\Theta)$ az i.-ik és j.-ik oszlopban és sorban vannak. Ha G-vel megszorzunk egy mátrixot balról, akkor csak az i.-ik és a j.-ik sorok változnak, tehát elég ezt a problémát megoldani:

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \tag{3.3}$$

Θ kiszámítása lényegtelen, csak keressük c-t, s-t és r-t.
Triviális megoldás: $r := \sqrt{a^2 + b^2}$, $c := a/r$ és $s := -b/r$. Azonban ezek számításánál gyakran jegyveszteség léphet fel, ezért a program az Edward Anderson által javasolt algoritmust használja, ezt valósítja meg a *setRCS* eljárás. [11]

Ha az összes elemet kinullázzuk a mátrix alsóhárömszög részében, akkor kapjuk R-t. Q-t pedig a következőképpen:

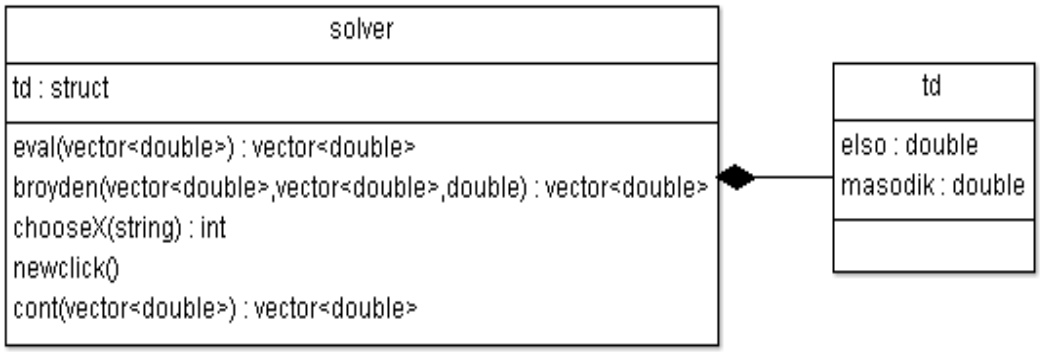
$$Q = G_1^T G_2^T \dots G_k^T,$$

ahol k az elvégzett forgatások száma, és G_i az i.-ik forgatómátrix. Azt, hogy mikor melyik elemet nullázzuk ki, a *choose* eljárás határozza meg.

A maximális forgatások száma, amit el kell végezni, az $\frac{n^2-n}{2}$ darab. Ez akkor lehetséges, ha nincs nulla a mátrix alsóháromszög részében. A G_i forgatómárixok elemszáma $n+2$. Mivel G_i ritka mátrixként van tárolva a programban, ezért összesen maximum $\Theta(\frac{n^2-n}{2} \cdot (n \cdot (n+2)) \cdot 2) = \Theta(n^4)$ szorzást kell elvégezni. Ellenben ha hagyományos módon ábrázolnánk G_i-t, akkor $\Theta(\frac{n^2-n}{2} \cdot n^3 \cdot 2) = \Theta(n^5)$ szorzással oldható meg a QR-felbontás.

3.5. Egyenletrendszer megoldása

Az egyenletrendszer megoldását a *solver* osztály végzi.



Az *eval* függvény kiértékeli a megadott függvényt a paraméterként átadott vektor helyen. A *broyden* eljárás egy adott kezdővektorból indítva elvégez egy közelítő eljárást a Broyden-módszerrel. A *cont* függvény pedig megoldja az egyenletrendszert a folytatás módszerével. A *td* struktúra egy segédstruktúra, a lengyelformák kiértékelésekor van rá szükség.

A lengyelformák kiértékeléséhez szükség van szintén egy veremre. A program szekvenciálisan dolgozza fel a lengyelformát balról-jobbra. Ezt könnyen megteheti, mert egy sor adatstruktúrában van tárolva a lengyelforma.
Ha a soron következő szimbólum:

- konstans, akkor beleteszi a verembe
- változó, akkor megkeresi a *chooseX* eljárás azt, hogy ez a változó az x vektor hanyadik eleme, és azt beleteszi a verembe
- bináris operátor, akkor kiveszi verem felső két elemét, meghatározza a művelet végeredményét úgy, hogy amelyik változót először vette ki a veremből,az lesz a művelet jobb oldalán, majd visszateszi a verembe az eredményt
- unáris operátor, vagyis valamilyen függvény (például *ln*, *sin* . . .), akkor kiveszi a program a verem legfelső elemét, kiértékeli a műveletet majd visszateszi az eredményt a verembe

Ha nincs több szimbólum a kifejezésben, akkor a verem legfelső eleme a megoldás. Az *eval* eljárás elvégzi a fent részletezett lengyelforma kiértékelést az összes egyenletre, és ennek megfelelően előállítja a függvény értékét a paraméterként megkapott vektor helyen.

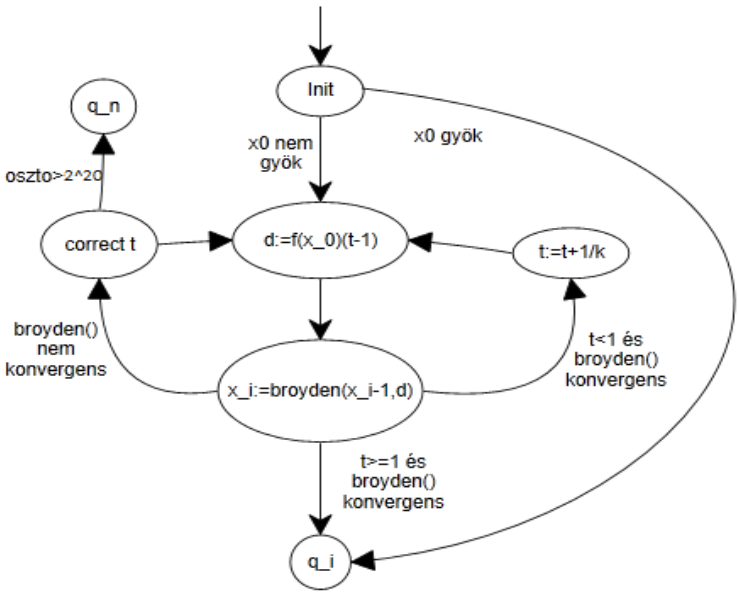
A szögfüggvények kiértékelésénél ügyelni kell arra, hogy radiánban, vagy fokban kéri a felhasználó a megoldást, ezért van bevezetve egy *deg_ rad* segédváltozó, aminek értéke 0, ha radiánban számolunk, és 1 egyébként. Továbbá jelöljön h most egy tetszőleges trigonometrikus függvényt. Ekkor a következő képlettel kell számolni h(x)-et:

$$h((1 - deg\_rad) * x + deg\_rad * \frac{\pi}{180} * x).$$

Az inverz trigonometrikus függvények esetében a következő a számítási módszer, ha h' most egy tetszőleges trigonometrikus függvény inverze:

$$(1 - deg\_rad) * h'(x) * \frac{180}{\pi} + deg\_rad * h'(x).$$

Magát az egyenletrendszert a *cont* metódus meghívásával lehet megoldani, aminek bemenő paramétere egy kezdővektor, kimenő paramétere pedig a függvény gyöke.



3.7. ábra. A program működését leíró Turing-gép

A program első lépése az inicializálás (a 3.7 ábrán *Init*). Itt megvizsgálja azt, hogy a kezdővektor gyöke-e a függvénynek. Ezt a $||f(x_0)|| <= thres$ feltétel eldön-

tésével teszi meg. Még beállítja a program a maximum lépések számát, ha ezt nem adta meg a felhasználó (ld. 2.2 egyenlet).

A *broyden* metódusnak nem az eredeti függvény ($f(x) = 0$) gyökét kell megkeresnie, hanem a 3.1 egyenletnek. Ezért kapja meg az eljárás a $d := (t_i - 1)f(x_0)$ paramétert is. Ha a módszer konvergál, akkor a program növeli t_i-t Δt-vel, így megkapva t_{i+1}-et.

Ha viszont nem volt konvergens a módszer, akkor a fentebb leírtak szerint visszalép program a régi t-re, vagyis $t - \Delta t$-re, majd felezi a lépéstávolságot. Ezzel az új lépésközzel megpróbálja újból a *broyden*() metódust alkalmazni. Ezt addig csinálja, amíg vagy konvergens lesz a módszer, vagy már 20-szor lépett vissza. Ha utóbbi következik be, akkor a program hibát jelez, és kéri, hogy változtassuk meg a program paramétereit. Többször nem érdemes csökkenteni a lépésközt, mert akkor már romlik a számolás pontatlansága jelentősen.

A program karbantart egy *oszto* változót, amivel elosztja Δt-t. Vagyis a program indításakor $oszto := 1$, ha vissza kell lépni, akkor $oszto := 2 \cdot oszto$. Ha sikeres lesz ezek után a Broyden-módszer, akkor $oszto := 1$, tehát az eredeti lépésközzel halad tovább a program.

A *broyden* metódus a fent részletezett Broyden-módszer alapján működik. Nem kell tudni a függvény Jacobi-mátrixát a megoldáshoz, hanem az egységmátrixból (I) kiindulva közelíti a program a QR-felbontás segítségével. A program egy hátultesztelős ciklussal közelíti a megoldást, aminek megállási feltétele vagy 2.1 egyenlet teljesülése vagy a maximum lépésszám elérése. Hátránya ennek a módszernek a hagyományos Newton-módszerhez képest, hogy lassabb a konvergenciája, nem kvadratikus. Viszont sokkal kevesebb számítással lehet elvégezi, ami a legtöbb esetben megéri ezt az áldozatot.

A program a *cont* eljárás során folyamatosan eltárolja a kiszámolt részeredményeket, amik majd ábrázolva lesznek később. Ezzel egy időben naplózást is végez. A naplózás során minden lépésnél kiírja a program, hogy mekkora lépést tesz, és hogy honnan indítja a Broyden-módszert. Ezek alatt nyomon lehet követni az aktuálisan kiszámolt közelítővektorokat. A minden sorban mellettük lévő szám $||F(x_m)||$, ahol x_m az aktuális közelítővektor. Ezután a program vizsgálja, hogy konvergált-e a módszer. Ha igen, akkor naplózza, hogy hol jár a számításban, ha nem, akkor visszalép, és amit eddig számolt az előző Broyden-módszerrel, azt nem fogja ábrázolni.

3.6. Műveletigény, hibabecslés

A program műveletigényét nehéz meghatározni. Vannak olyan elemei a számításnak, amelyeknek nem lehet előre megmondani a számítási költségeit. Ilyen például az *eval* eljárás, ami kiértékel egy függvény egy adott vektor helyen. Nem lehet megmondani előre, hogy milyen és mennyi művelet fog szerepelni az egyenletekben. Nem tudjuk azt se, hogy a végrehajtott Broyden-módszerek sikeresek lesznek-e. Szerencsétlen esetben akár minden lépés után vissza kell lépni többször is, de az is előfordulhat, hogy egyszer sem kell.

A program viszont képes megmérni az adott feladat elvégzéséhez szükséges időt. A sorokban a felhasznált idő szerepel másodpercekben. Az utolsó oszlopban a három szám sorban: ε, Δt és maximum lépésszám. A maximum lépésszám értéke általában 10 körül van, de ez függ természetesen az egyenletrendszertől. A fejlécben szereplő *test** fájlok a programhoz mellékelt tesztfájlok.

test10	test11	test7	test3	test2	test16	
0,420	0,406	0,761	0,735	2,233	1,313	0,001; 1/10; default
0,323	0,363	0,760	0,917	2,427	1,125	0,01; 1/10; default
0,411	0,454	0,794	1,043	2,282	1,531	0,0001; 1/10; default
0,195	0,233	0,621	0,293	1,188	0,625	0,01; 1/5; default
0,944	0,835	2,049	1,365	6,866	2,703	0,01; 1/30; default
0,202	0,226	0,549	0,279	1,408	0,627	0,01; 1/5; 30

Ez a táblázat magában még nem sokat mond, mert nem tudjuk, hogy mennyit veszítünk a pontosságból a gyorsaság érdekében. A következő táblázat a program által ábrázol g függvény (ld. 2.3) értékét ábrázolja az utolsó lépés után, vagyis a megoldásban.

test10	test11	test7	test3	test2	test16	
0,00017	0,00070	0,03574	0,00480	0,00047	0,000002	0,001; 1/10; default
0,05060	0,02646	0,03042	0,03674	0,00047	0,000136	0,01; 1/10; default
0,00000	0,00002	0,03574	0,00002	0,00047	0,000003	0,0001; 1/10; default
0,05068	0,09268	0,01857	0,30350	0,00059	0,000004	0,01; 1/5; default
0,00086	0,00194	0,01596	0,01148	0,00018	0,000027	0,01; 1/30; default
0,05068	0,01146	0,00417	0,00090	0,00001	0,000004	0,01; 1/5; 30

Látható a fenti két táblázat alapján, hogy általában igaz az, hogy ε és Δt csökkentésével, valamint a maximum lépésszám növelésével javul a számítási pontosság, viszont nő a számítási idő. Persze nem lehet jól összehasonlítani a teszteseteket, a megoldhatóság, és hogy mi az optimális paraméterezés nagyban függ az feladat bonyolultságától is. Sokkal jobban függ tőle, mint az egyenletrendszer méretétől. A 16.

tesztesetet leíró függvény például $\mathbb{R}^{10} \to \mathbb{R}^{10}$, míg a 2. teszteset mindössze $\mathbb{R}^5 \to \mathbb{R}^5$ alakú.

Nagyon bonyolult feladatoknál nem biztos, hogy megoldáshoz jutunk minden esetben. Az ilyen egyenletrendszereknél kezdjük a próbálkozást viszonylag nagyobb lépésközzel, és sok lépésszámmal.

3.7. Rendszerkövetelmények

A program futtatásához MS Windows operációs rendszer szükséges.

3.8. Felhasznált modulok

- C++ programozási nyelv

`http://www.cplusplus.com/reference/` (2014.05.12)

- wxWidgets platformfüggetlen API a GUI elkészítéséhez

`https://www.wxwidgets.org/` (2014.05.12)

- wxMathPlot könyvtár a grafikonok ábrázolásához

`http://wxmathplot.sourceforge.net/` (2014.05.12)

- wxDevC++ IDE

`http://wxdsgn.sourceforge.net/` (2014.05.12)

- argoUML az osztálydiagrammok elkészítéséhez

`http://argouml.tigris.org/` (2014.05.12)

3.9. Továbbfejlesztési lehetőségek

- Jelenleg a program csak MS Windows operációs rendszereken működik. A wxWidgets viszont egy platformfüggetlen csomag, tehát működőképes a program Linux-on, Mac OS X-en és más platformokon is minimális kódváltoztatással. Azonban ezeken a platformokon nincs tesztelve.
- A lebegőpontos számoknak véges a számítási pontosságuk. Léteznek azonban nagy pontossággal számolni képes könyvtárak, amiknek a pontosságát mindössze az elérhető memória korlátozza. Át lehet írni a programot, hogy egy ilyen csomag által nyújtott szolgáltatásokat használja, ne pedig a standard C++ által kínált *double* számokkal számoljon.

4. fejezet

Tesztelés

4.1. Fekete doboz tesztelés

A feketedobozos (black-box) tesztelést specifikáció alapú tesztelésnek is nevezik. Leggyakoribb formája, hogy egy adott bemenetre tudjuk, hogy milyen kimenetet kellene a programnak adnia. Kipróbáljuk, hogy a program az adott bemenettel milyen eredményre jut, és összehasonlítjuk azzal, amit vártunk.

A programhoz számos tesztfájl van mellékelve, amik a *test files* mappában találhatóak meg. A program tesztelése során az összes teszteset ki lett próbálva fájlból olvasással, valamint manuális adatbevitellel is.

Hibás bemenetek

A hibás tesztfájlok az *error* mappában vannak, nevük végén szerepel az *_err* kifejezés, például *test1_err.txt*.

- test1_err ... test4_err: az egyenletek számának tesztelése. Az egyenletek száma $n \geq 2$, valamint nem lehet megadni karaktereket. Ha $n \notin \mathbb{Z}$, akkor az egész részét kell venni az inputnak.

- test5_err ... test10_err: a kezdővektor tesztelése. Ha a vektor nem elég hosszú, vagy valamelyik vektorelem helyén karaktersorozat van, akkor 0-val helyettesíti az értékeket. Ha túl hosszú, akkor levágja a többletet

- test11_err ... test15_err: a pontos megoldás tesztelése. Ugyan az vonatkozik erre, mint a kezdővektorra.

- test16_err ... test23_err: az egyenletek tesztelése. Tesztelve van a helytelen zárójelezés, az ismeretlen karakterek kiszűrése, a változók hibás indexelése és az, hogy operátor nem lehet az egyenletek végén.

- Tesztelhetőek továbbá a megoldó programegység paraméterei is, de ezeket nem lehet fájlból beolvasni. Ha karaktersorozatot, nullát, vagy negatív számot adunk meg, akkor azt nem fogadhatja el a program.

Érvényes bemenetek

Előfordulhat, hogy nem megfelelő paraméterezéssel nem jut megoldásra a program. Ennek oka az, hogy a hagyományos lebegőpontos számokkal való számítások pontatlanok lehetnek. Ilyenkor célszerű más paramétereket vagy kezdővektort megadni.
Előfordulhat az is, hogy a program nem a várt megoldáshoz konvergál. A programnak nem feladata az összes megoldást megtalálnia, ha több is van. Az, hogy melyiket fogja megtalálni, azt előre nem lehet megmondani, függ a kezdővektortól és a paraméterektől.

- test1, test2: lineáris egyenletrendszerek tesztelése. A programnak nem feladata a lineáris egyenletrendszerek hatékony megoldása, de képes megoldani az ilyen feladatokat is.

- test3: példa arra az esetre, amikor a program nem az általunk várt megoldáshoz konvergál, hanem egy másikhoz.

- test4: ennél a példánál a program az alapbeállításokkal nem jut megoldáshoz. Többféle módon is megkereshetjük a megoldást, például ha növeljük ε-t 0,01-re, ekkor viszont elég pontatlan megoldást kapunk. Jobb ennél, ha a lépésközt csökkentjük mondjuk 1/15-re.

- test5: ennek a példának a megoldását lásd a 2.4 ábrán.
  Továbbiakban ha más nincs említve, akkor az ajánlott paraméterezés van feltüntetve, amivel biztos eléri a pontos megoldást a program. A számok rendre ε, 1=delta és maximum lépésszám.

- test6: 0,0001, 5, 30

- test8: 0,001, 4 , 30

- test23: fokkal való számolás esetén: 0,0001, 10, 4

  A test16, 18, 19 nem saját példa: [12]

- test18: neurofiziológiai rendszer: 0,0001, 20, 30

- test19: égési feladat: 0,00001, 5, 30

4.2. Fehér doboz tesztelés

A fehérdobozos (white-box), vagy strukturális teszt során a forráskód alapján tesztelünk. Egyszerre nem az egész programot, hanem annak csak részeit, például metódusokat, osztályokat stb.

A fehér doboz tesztelés egy részét elvégző program megtalálható a mellékelt CD *tester* mappájában. A főprogram osztályai és a tesztelőprogram osztályai (solver, equation_handler, sparse_matrix) megegyeznek. Egyedül a *linalg* csomagba került bele még öt metódus, amik segítségével ki lehet íratni mátrixokat, illetve vektorokat a konzolra (*write_mtx, write_vec*), valamint össze lehet hasonlítani két mátrixot (*equals*). Ennek megfelelően a ritka mátrixot megvalósító osztálya bekerült a kiíratót függvény, mint barát függvény.

Ezért a forrásfájlok nem kerültek rá még egyszer a CD-re, csak azok, amelyek változtak, valamint a futtatható fájl.

A tesztelő képes tesztelni külön a ritka és sűrű mátrixok műveleteit: a transzponálást, a szorzást, a QR-felbontást, és az inverz képzést. A transzponálás ellenőrzéséhez ha kétszer transzponálunk egy mátrixot, akkor a kiindulási mátrixot kell visszakapni. Ha ez jól működik, akkor lehet ellenőrizni a szorzást a következő összefüggés alapján: $(A \cdot A)^T = A^T \cdot$, ahol A a tesztelést végző mátrix. A QR-felbontás ellenőrzéséhez teszteli a program azt, hogy R felsőháromszög mátrix-e, valamint azt, hogy $A = QR$ egyenlőség teljesül-e. Az inverz előállításának helyességét pedig a legegyszerűbben a következő összefüggés alapján lehet ellenőrizni: $A \cdot A^{-1} = I$, ahol I az egységmátrix.

A többi ellenőrzés helyességét nem tudja eldönteni a program, hanem kiírja a konzolra az eredményt, és magunknak ellenőrizhetjük. Ezek a tesztesetek vizsgálják a lengyel formára hozás implementációjának helyességét, valamint a szekvenciális olvasó *next* eljárás működését. A programmal lehet még tesztelni sűrű és ritka mátrixok, valamint vektorok egymással való szorzását is.

A *main* függvényben bármilyen mátrixot megadhatunk, és saját teszteseteket is készíthetünk, de néhány előregyártott tesztmátrix létre van hozva a tesztelés meggyorsítása végett.

Az automatikus tesztelést a *tester.exe* fájl futtatásával lehet elindítani.

A fehér doboz tesztelés másik része a felhasználói interfész tesztelése. Az összes gomb és szövegdoboz kipróbálása minden lehetséges szituációban, ami a program futásakor adódhat. Az adatok módosításánál a program ténylegesen módosítja az adott adatot, és a frissített változatot jeleníti meg.

5. fejezet

Összegzés

A nemlineáris egyenletrendszerek megoldása nem egyszerű feladat. Általában viszonylag sok számítási időt igényel az ilyen feladatok megoldásának kiszámítása. Ahhoz, hogy biztosan meg tudjuk oldani az egyenletrendszereket, globalizálni kellett a feladatot, ami lassítja a megoldást, és pontatlanabbá is teszi azt. A program egy gyors és hatékony módszert biztosít a felhasználónak nagyon sokféle egyenletrendszer megoldására. A kezelőfelület felhasználóbarát, könnyen átlátható és egyszerű.

A program a gyakorlati életben is jól használható. A program megoldó egysége paraméterezhető, így lehetőség van arra, hogy mindig megtalálja a felhasználó a saját maga számára optimális megoldást. Például szükség esetén nagyon pontos megoldást tud találni.

Fontos megemlíteni, hogy nagyobb feldatok megoldásánál mennyire fontos a kód optimalizálása. A ritka mátrixok speciális tárolási módja már kisebb egyenletrendszerek megoldását is gyorsította.

A jövőben a program tovább lesz fejlesztve, hogy még gyorsabb és pontosabb megoldáshoz lehessen jutni, valamint ne csak egy adott operációs rendszert használó felhasználók alkalmazhassák a program által nyújtott szolgáltatást.

Végül szeretném megköszönni a témavezetőm, Dr Gergó Lajos segítségét.

Irodalomjegyzék

[1] **Stoyan Gisbert, Takó Galina**, *Numerikus módszerek 1.*, 6.4. A Newton-módszer és változatai, 2005 `http://www.tankonyvtar.hu/hu/tartalom/tkt/numerikus-modszerek-1/ch07s04.html` (2014.05.12)

[2] **Paláncz Béla**, *Numerikus módszerek*, 3. Nemlineáris egyenletek, 2011, `http://www.fmt.bme.hu/fmt/oktatas/feltoltesek/bmeeoftmkt2/fejezet_03.pdf` (2014.05.12)

[3] **Veress Krisztián**, *A Newton és Gauss-Newton módszerek alkalmazása egyenletrendszerek megoldására és nemlineáris optimalizálásra*, 2007, `http://www.inf.u-szeged.hu/ verkri/bin/newton-gauss.pdf` (2014.05.12)

[4] **Gergó Lajos** *Numerikus módszerek - kidolgozott példák, feladatok*, Eötvös kiadó, 2000, 204 oldal

[5] **Massoud Malek**, *Numerical Analysis*, Inverse of Lower Triangular Matrices, `http://www.mcs.csueastbay.edu/ malek/TeX/Triangle.pdf` (2014.05.12)

[6] **D. Bindel, J. Demmel, W. Kahan, O. Marques** *On computing Givens rotations reliably and efficiently*, 2001 `http://www.netlib.org/lapack/lawnspdf/lawn148.pdf` (2014.05.12)

[7] **Gene H. Golub, Charles F. Van Loan** *Matrix Computations*, 5.2 The QR Factorization, 1996 `http://www.math.pku.edu.cn/teachers/yaoy/reference/golub.pdf` (2014.05.12)

[8] **George Grätzer** *Math into LATEX*, 1996 `http://ctan.ijs.si/tex-archive/info/examples/mil/mil.pdf` (2014.05.12)

[9] **Julian Smart and Kevin Hock with Stefan Csomor** *Cross-Platform GUI Programming with wxWidgets*, 2005 `http://ptgmedia.pearsoncmg.com/images/0131473816/downloads/0131473816_book.pdf` (2014.05.12)

[10] **Tichler Krisztián** Lengyel forma `http://www.cs.elte.hu/ tichlerk/algterv/LF.pdf` (2014.05.12)

[11] **Edward Anderson** *Discontinuous Plane Rotations and the Symmetric Eigenvalue Problem*, 2000 `http://www.netlib.org/lapack/lawnspdf/lawn150.pdf` (2014.05.12)

[12] **Crina Grosan and Ajith Abraham** *A New Approach for Solving Nonlinear Equations Systems*, 2008 `http://www.softcomputing.net/smca-ca-web.pdf` (2014.05.12)