Eötvös Loránd Tudományegyetem
Informatikai Kar
Komputeralgebra Tanszék

Általánosított számrendszerek vizsgálatát támogató könyvtár fejlesztése

Radix rendszerek periodikus pontjainak keresése

TDK dolgozat

Dr. Kovács Attila
Egyetemi docens

Németh Bence
Programtervező Informatikus MSc
Modellalkotó Informatikus szakirány

Budapest, 2015

Tartalomjegyzék

Bevezetés **3**

1. Matematikai alapok **4**
1.1. Általánosított számrendszerek 4
1.2. A kifejtés, mint diszkrét dinamikai rendszer 7
1.2.1. Kifejtések osztályozása 9
1.2.2. Operátornorma konstrukció 10
1.3. A törtek halmaza 11
1.3.1. A törtek halmazának egy lefedése 12
1.4. Az eldöntési és az osztályozási feladatok 14

2. Adattípusok kezelése **17**
2.1. Típusok 17
2.2. Típusok összekapcsolása 18
2.2.1. Konvertálás típusok között 19
2.3. Konstansok és műveletek 20
2.3.1. Komplex számok 23
2.3.2. Egész számok 24

3. Lineáris algebra **25**
3.1. Adatszerkezetek 26
3.1.1. Vektor 26
3.1.2. Ritka vektor 27
3.1.3. Mátrix 28
3.1.4. Ritka mátrix 29
3.2. Műveletek megvalósítása 31
3.2.1. Vektor műveletek 31
3.2.2. Mátrix műveletek 35
3.2.3. Szorzatok számítása 37
3.3. Normák 41
3.3.1. p-normák 42

3.3.2. Frobenius-norma 42
3.3.3. Operátornorma 43
3.4. Megvalósított algoritmusok 44
3.4.1. Determináns számítás 44
3.4.2. Inverz számítás 45
3.4.3. Adjungált számítás 45
3.5. Schur-felbontás és Jordan-féle normálforma 46
3.5.1. QR felbontás 46
3.5.2. Felső Hessenberg-alakra hozás 47
3.5.3. Schur-felbontás 48
3.5.4. Jordan-féle normálforma 49

4. Általánosított számrendszerek **51**
4.1. Jegyrendszerek konstrukciója 52
4.1.1. A j-kanonikus és j-szimmetrikus jegyrendszer 52
4.1.2. Adjungált jegyrendszer 53
4.1.3. Sűrű jegyrendszer 54
4.2. A $-H$ halmaz lefedésének javítása 56
4.3. A Φ függvény számítása 58
4.3.1. Adjungált módszer 58
4.3.2. Smith-normálforma 59
4.3.3. A Φ függvény megvalósítása 60
4.4. Az osztályozási és az eldöntési algoritmus megvalósítása 63
4.5. Szimultán rendszerek 66
4.5.1. Sűrű jegyrendszer keresése 66

Összefoglalás **68**

Irodalomjegyzék **69**

Bevezetés

A dolgozat célja az általánosított számrendszerek vizsgálatát támogató, a téma további kutatását megkönnyítő C++ könyvtár fejlesztése.

A könyvtár az általánosított számrendszerek vizsgálatát lehetővé tévő adatszerkezetek és algoritmusok megvalósítását, valamint az ezekhez szükséges további, lineáris algebrai adatszerkezeteket és algoritmusokat tartalmazza.

A cél egy olyan eszköz kifejlesztése, amely megkönnyíti a téma kutatását, lehetőséget ad olyan rendszerek esetében is számítások elvégzésére, amelyek elvégzése megfelelően hatékony megvalósítás hiányában korábban szinte lehetetlen volt. Reményeink szerint a jövőben a könyvtár által biztosított hatékony algoritmusokat használva az általánosított számrendszerekkel kapcsolatos új eredményekhez juthatunk.

Az algoritmusok hatékony implementációjához a már meglévő elméleti eredményeket használjuk fel.

A megvalósított algoritmusokon belül is külön figyelmet fordítunk az eldöntési és osztályzási kérdéskörhöz kapcsolódó adatszerkezetek és algoritmusok lehető leghatékonyabb implementációjára. Megmutatjuk azokat az eredményeket, amelyek segítségével az eldöntési, illetve osztályozási feladatokat az adott radix rendszer periodikus pontjainak keresésére vezetjuk vissza. Megvalósítunk egy módszert, amely segítségével a keresési tér méretét nagyságrendekkel csökkenthetjük, továbbá bemutatjuk azokat a rendelkezése álló elméleti eredményeket, amelyekkel számításaink hatékonyságát növeltük.

1. fejezet

Matematikai alapok

1.1. Általánosított számrendszerek

1.1.1. Definíció. (Rács) Amennyiben $\{v_1, v_2, \ldots, v_n\}$ bázis $\mathbb{R}^n$ felett, akkor a

$$\Lambda = \left\{ \sum_{i=1}^{n} a_i v_i : a_i \in \mathbb{Z} \right\}$$

halmaz *rács* $\mathbb{R}^n$ felett.

Legyen Λ rács $\mathbb{R}^n$ felett, $M : \Lambda \to \Lambda$ invertálható, valamint legyen $D \subset \Lambda$ véges halmaz, melyre $0 \in D$.

1.1.2. Definíció. (Számrendszer) A (Λ, M, D) hármast *számrendszernek* nevezzük, ha minden $z \in \Lambda$ egyértelműen felírható

$$z = a_0 + Ma_1 + M^2 a_2 + \cdots + M^l a_l$$

véges alakban, ahol $a_i \in D$.

M a számrendszer alapja, vagy más néven *radix*, valamint azt mondjuk, hogy z M alapú kifejtése $(a_l a_{l-1} \ldots a_1 a_0)_M$. A *kifejtés hossza* ebben az esetben $l + 1$.

Az általánosított számrendszerek vizsgálatának egyik fő feladata annak eldöntése, hogy egy adott (Λ, M, D) hármas számrendszert alkot-e. Ennek eldöntésére az alábbi elméleti eredmények – szükséges, illetve elégséges feltételek – ismertek.

A tételek kimondása előtt definiáljuk az alábbi fogalmakat.

1.1.3. Definíció. Ha $z_1, z_2 \in \Lambda$ és z_1, z_2 $\Lambda/M\Lambda$ azonos mellékosztályába tartoznak, akkor azt mondjuk, hogy kongruensek *modulo* M és a

$$z_1 \equiv z_2 \pmod{M}$$

jelölést használjuk.

1.1.4. Állítás. A $\Lambda/M\Lambda$ faktorcsoport rendje $|\det M|$.

1.1.5. Definíció. (Expanzivitás) Az $M : \Lambda \to \Lambda$ leképezés *expanzív*, amennyiben minden sajátértékének abszolútértéke nagyobb egynél.

1.1.6. Tétel. (Szükséges feltétel) Ha (Λ, M, D) számrendszer, akkor a következők teljesülnek.

1. D teljes maradékrendszer *modulo* M,
2. M expanzív,
3. $\det(I - M) \neq \pm 1$.

Bizonyítás: 1. Ha $z \in \Lambda$ kifejtése $(a_l a_{l-1} \dots a_1 a_0)_M$, akkor $z \equiv a_0 \pmod{M}$. Mivel tetszőleges Λ-beli elemnek van kifejtése, így D biztosan tartalmaz teljes maradékrendszert.

Lássuk be, hogy D nem tartalmazhat egyazon maradékosztályból több elemet. Tegyük fel, hogy $c, d \in D$ és $c \equiv d \pmod{M}$. Ekkor $\exists e \in \Lambda$, melyre $c - d = Me$. Ha e kifejtése $(a_l a_{l-1} \dots a_1 a_0)_M$, akkor

$$(c)_M = c = Me + d = (a_l a_{l-1} \dots a_1 a_0 d)_M$$

Így azonban c két különböző kifejtéssel rendelkezne, amely ellentmond a számrendszer tulajdonságnak.

2. A bizonyítást [4] tartalmazza.

3. Ha (Λ, M, D) számrendszer, akkor $\nexists b \in \Lambda$, melyre

$$b = a_0 + Ma_1 + \dots + M^{l-1}a_{l-1} + M^l b,$$

ahol $a_i \in D$ $(i = 0 \dots l-1)$, $l \in \mathbb{N}^+$.

M expanzivitásából következik, hogy $\forall n \in \mathbb{N}^+ : \det(I - M^n) \neq 0$, azaz $I - M^n$ invertálható. Ezt felhasználva kapjuk, hogy

$$(I - M^l)^{-1}(a_0 + Ma_1 + \dots + M^{l-1}a_{l-1}) \notin \Lambda$$

Azonban ha $\det(I - M) = \pm 1$, akkor $(I - M)\Lambda = (I - M)^{-1}\Lambda = \Lambda$, ami ellentmond az előző állításunknak.

□

1.1.7. Következmény. Mivel D teljes maradékrendszer, ezért ha egy tetszőleges $z \in \Lambda$ elem rendelkezik véges kifejtéssel, akkor a kifejtés egyértelmű.

1.1.8. Definíció. (Radix rendszer) Ha (Λ, M, D) teljesíti az 1.1.6. tétel 1. és 2. pontját, akkor a (Λ, M, D) hármast *radix rendszernek* nevezzük.

1.1.9. Tétel. (Elégséges feltétel) Ha egy (Λ, M, D) radix rendszerre teljesül, hogy

1. a Λ rácsnak létezik olyan bázisa, amelyben minden bázisvektornak létezik véges kifejtése,
2. a $D \pm D$ halmaz minden eleme rendelkezik $a_0 + M^j a_j$, $(a_0, a_j \in D, j \in \mathbb{N})$ alakú kifejtéssel,

akkor (Λ, M, D) számrendszer.

Bizonyítás: Az 1.1.7. következmény alapján elegendő megmutatnunk, hogy bármely $z \in \Lambda$ elemnek létezik véges kifejtése.

Legyenek Λ bázisvektorai $b_1, b_2, \ldots, b_d$. Az 1. feltétel alapján ezek a vektorok rendelkeznek véges kifejtéssel. Mivel $z \in \Lambda$, ezért felírható a bázisvektorok segítségével, azaz

$$z = \sum_{i=1}^{n} \alpha_i b_i,$$

valamely α_i egészekre. Így elegendő belátnunk, hogy két véges kifejtéssel rendelkező elem összege, illetve különbsége is rendelkezik véges kifejtéssel. Ez viszont következik a 2. feltételből.

□

1.1.10. Tétel. (Számrendszerek ekvivalenciája) Ha M_1 és M_2 hasonló mátrixok, $M_2 = QM_1Q^{-1}$, akkor a következő állítás teljesül

$$(\Lambda, M_1, D) \text{ számrendszer} \iff (Q\Lambda, M_2, QD) \text{ számrendszer}$$

Bizonyítás: D akkor és csak akkor teljes maradékrendszer Λ felett *modulo* M_1, ha QD teljes maradékrendszer $Q\Lambda$ felett *modulo* M_2.

Legyen $z \in \Lambda$, valamint legyen z kifejtése (Λ, M_1, D) felett $\sum_{i=0}^{l} M_1^i a_i$. Ekkor

$$Qz = \sum_{i=0}^{l} QM_1^i a_i = \sum_{i=0}^{l} QM_1^i Q^{-1} Q a_i = \sum_{i=0}^{l} M_2^i (Q a_i)$$

Vagyis z kifejtése (Λ, M_1, D) felett pontosan akkor véges, ha Qz kifejtése véges $(Q\Lambda, M_2, QD)$ felett. □

1.1.11. Következmény. A tétel számunkra igazán fontos következménye, hogy tetszőleges (Λ, M, D) hármas helyett — bázistranszformációt alkalmazva — vizsgálhatjuk a $(Q\Lambda, QMQ^{-1}, QD)$ hármast, ahol

$$Q\Lambda = \mathbb{Z}^n, \quad QMQ^{-1} : \mathbb{Z}^n \to \mathbb{Z}^n, \quad \text{illetve} \quad QD \subset \mathbb{Z}^n$$

1.2. A kifejtés, mint diszkrét dinamikai rendszer

A következőkben azt vizsgáljuk, mi történik egy adott radix rendszerben egy tetszőleges pont kifejtésének kiszámítása közben. Ehhez vezessük be a következő függvényt.

1.2.1. Definíció. (Φ függvény)[1] Legyen (Λ, M, D) radix rendszer. Definiáljuk a $\Phi : \Lambda \to \Lambda$ függvényt a következő módon

$$\Phi(z) = M^{-1}(z - a_i),$$

ahol $a_i \in D$ és $a_i \equiv z \pmod{M}$.

Szemléletesen a Φ függvény meghatározza, majd "levágja" a kifejtés utolsó jegyét.

Azt szeretnénk vizsgálni, hogy mi történik egy adott ponttal a Φ függvény többszöri alkalmazása során, más szavakkal a kifejtések dinamikáját szeretnénk vizsgálni. Ehhez definiálnunk kell az alábbi fogalmakat.

1.2.2. Definíció. (Pálya) Legyen $\Phi^n(z) = \Phi(\Phi^{n-1}(z))$, valamint $\Phi^0(z) = z$. Ekkor a

$$z, \Phi(z), \Phi^2(z), \Phi^3(z), \ldots$$

sorozatot a z pont pályának nevezzük.

1.2.3. Definíció. (Periodikus pont) Egy $z \in \Lambda$ pontot periodikusnak nevezünk, ha létezik olyan $n \in \mathbb{N}^+$ szám, melyre $\Phi^n(z) = z$. Ilyenkor azt mondjuk, hogy a z pont n hosszú periódussal periodikus. A legkisebb ilyen n számot alapperiódusnak nevezzük.

1.2.4. Definíció. A periodikus pontok halmazát jelölje $\mathcal{P}$.

1.2.5. *Megjegyzés.* Egy $z \in \Lambda$ pont pályája végül periodikus, amennyiben $\exists n \in \mathbb{N}$ és $p \in \mathcal{P}$ periodikus pont, melyekre $\Phi^n(z) = p$.

1.2.6. *Megjegyzés.* Egy $z \in \Lambda$ pontot fixpontnak nevezünk, ha $\Phi(z) = z$, azaz 1 hosszú periódussal periodikus.

A Φ függvényt felhasználva az alábbi szükséges és elégséges feltételt tudjuk adni annak eldöntésére, hogy (Λ, M, D) számrendszer-e.

1.2.7. Tétel. (Szükséges és elégséges feltétel) A (Λ, M, D) radix rendszer akkor és csak akkor számrendszer, ha

$$\forall z \in \Lambda,\ \exists n \in \mathbb{N} : \Phi^n(z) = 0$$

Bizonyítás: Ha $\Phi(z) = 0$, akkor $\exists a_0 \in D$, melyre $z \equiv a_0 \pmod{M}$. Könnyen látható, hogy $\Phi^n(z) = 0$ akkor és csak akkor állhat fenn, ha z felírható

$$z = a_0 + Ma_1 + \cdots + M^{n-1}a_{n-1}$$

alakban, ahol $a_0, a_1, \ldots, a_{n-1} \in D$, vagyis z rendelkezik véges kifejtéssel. □

[1] A Φ függvényt D. W. Matula [3] vizsgálta először.

Mivel (Λ, M, D) radix rendszer, ezért M expanzív, így M^{-1} kontraktív. Ez azt jelenti, hogy $\varrho(M^{-1}) < 1$. Ekkor létezik egy megfelelő $\|.\| : \mathbb{R}^n \to \mathbb{R}$ norma úgy, hogy a hozzá tartozó indukált operátornormára teljesül az

$$\|M^{-1}\| = \sup_{\|x\|=1} \|M^{-1}x\| < 1$$

egyenlőtlenség.

1.2.8. Definíció. (Operátornorma) A dolgozat további részeiben $\|.\|$ jelölje ezt a megfelelő normát.

Vezessük be a következő jelöléseket, legyen

$$K := \max_{a_i \in D} \|a_i\|, \qquad \text{valamint} \qquad r := \|M^{-1}\|$$

Ezen kívül legyen

$$L := \frac{rK}{1-r}$$

1.2.9. Lemma. A most bevezetett jelöléseket használva, valamit $\|z\|$ értékét ismerve a következő felső becslést adhatjuk $\|\Phi(z)\|$ értékére.

$$\|\Phi(z)\| \leq r\|z\| + rK$$

Bizonyítás: A Φ függvény definíciójából kiindulva és a normákra vonatkozó ismert egyenlőtlenségeket felhasználva

$$\|\Phi(z)\| = \|M^{-1}(z - a_i)\| = \|M^{-1}z - M^{-1}a_i\| \leq \|M^{-1}z\| + \|M^{-1}a_i\| \leq r\|z\| + rK$$

□

1.2.10. Tétel. Minden $z \in \Lambda$ pont pályája végül periodikus.

Bizonyítás: Használjuk az előző lemmában kapott felső becslést és vizsgáljuk a következő két esetet. Ha $\|z\| \leq L$, akkor

$$\|\Phi(z)\| \leq rL + rK = r(L + K) = r\left(\frac{Kr}{1-r} + \frac{K(1-r)}{1-r}\right) = r\frac{K}{1-r} = L$$

Ellenkező esetben, vagyis ha $\|z\| > L$, akkor

$$\|\Phi(z)\| \leq r\|z\| + rK = r\|z\| + (1-r)L < r\|z\| + (1-r)\|z\| = \|z\|$$

Így azt kaptuk, hogy $\|z\| \leq L$ esetén $\|\Phi(z)\| \leq L$, $\|z\| > L$ esetén pedig $\|\Phi(z)\| < \|z\|$.

Tudjuk, hogy tetszőleges nemnegatív $c \in \mathbb{R}$ esetén csak véges sok olyan $z \in \Lambda$ pont létezik, melyre $\|z\| \leq c$. Ebből következik, hogy tetszőleges $\|z\| > L$ esetén $\exists n \in \mathbb{N}^+$, amelyre $\|\Phi^n(z)\| \leq L$. Ez azt jelenti, hogy minden pont pályája véges sok lépésben az origó középpontú, operátornorma szerinti L sugarú gömbbe vezet és ezután onnan nem lép ki.

Az olyan pontokból, amelyekre $\|z\| \leq L$ szintén csak véges sok van, azonban minden pont pályája végtelen sok pontból áll. Így minden pont pályája végül periodikus kell legyen. □

1.2.11. Következmény. A tétel bizonyításából azt is megkaptuk, hogy az összes periodikus pont a normája legfeljebb L, azaz a periodikus pontok kereséséhez elegendő az origó középpontú, operátornorma szerinti L sugarú gömb elemeit vizsgálnunk.

1.2.12. Következmény. Az 1.2.7. tétel értelmében a (Λ, M, D) radix rendszer akkor és csak akkor számrendszer, ha

$$\mathcal{P} = \{0\}$$

1.2.1. Kifejtések osztályozása

Az 1.2.10. tételben beláttuk, hogy minden pont pályája végül periodikus, ezért ha egy adott pont véges kifejtéssel nem is rendelkezik, egy általánosabb, de egyértelmű kifejtést így is meg tudunk adni hozzá.

1.2.13. Definíció. Legyen $z \in \Lambda$ tetszőleges pont, valamint legyen $l : \Lambda \to \mathbb{N}$, ahol $l(z)$ értéke az a legkisebb szám, melyre $\Phi^{l(z)}(z) = p \in \mathcal{P}$.

Ekkor z felírható

$$z = a_0 + Ma_1 + \cdots + M^{l(z)-1}a_{l(z)-1} + M^{l(z)}p$$

alakban, ahol $a_0, a_1, \ldots, a_{l(z)-1} \in D$. Jelölje ezt a fajta kifejtést

$$\left(a_{l(z)-1} \ldots a_1 a_0 \mid p\right),$$

illetve amennyiben $z \in \mathcal{P}$ a jelölés legyen $(* \mid z)$.

1.2.14. Tétel. (A kifejtés hossza) Létezik egy megfelelő c konstans, melyre

$$l(z) \leq -\frac{\log \|z\|}{\log \|M^{-1}\|} + c$$

1.2.15. Definíció. (Vonzási tartomány) Legyen $p \in \mathcal{P}$ periodikus pont. A p pont vonzási tartományának a

$$\mathcal{B}(p) = \{\, z \in \Lambda \,:\, \exists n \in \mathbb{N} : \Phi^n(z) = p \,\}$$

halmazt nevezzük. Hasonlóan ha $X \subseteq \mathcal{P}$, akkor az X halmaz vonzási tartománya a

$$\mathcal{B}(X) = \{\, z \in \Lambda \,:\, \exists n \in \mathbb{N} : \Phi^n(z) \in X \,\}$$

halmaz lesz.

Osztályozhatjuk Λ pontjait aszerint, hogy melyik periodikus pont vonzási tartományához tartoznak.

1.2.16. Definíció. Azt mondjuk, hogy $x, y \in \Lambda$ azonos osztályhoz tartoznak, amennyiben

$$\exists p \in \mathcal{P} : x, y \in \mathcal{B}(p)$$

1.2.17. Definíció. Legyen $p \in \mathcal{P}$ periodikus pont n hosszú alapperiódussal. Jelölje a p pont által generált kört, vagyis azon pontoknak a halmazát, amelyeken egy perióduson belül áthalad p pályája

$$\mathcal{C}(p) = \left\{\Phi(p), \Phi^2(p), \ldots, \Phi^n(p)\right\}$$

1.2.2. Operátornorma konstrukció

Legyen $M : \mathbb{R}^n \to \mathbb{R}^n$ egy invertálható expanzív lineáris operátor. Egy olyan $\|.\|_{op}$ vektornormára adunk konstrukciót, hogy a hozzá tartozó indukált operátornormára teljesüljön az

$$\|M^{-1}\|_{op} < 1$$

egyenlőtlenség.

Legyen $J = TM^{-1}T^{-1} = \text{diag}(J_{\lambda_i,j})$ az M^{-1} mátrix Jordan-féle normálformája. Amennyiben M^{-1} sajátértékei egyszeres multiplicitásúak, akkor $J_{\lambda_i,j} = [\lambda_i]$, azaz a Jordan-blokkok 1×1 méretűek és csak a sajátértéket tartalmazzák. Ebben az esetben

$$\|J\|_\infty = \max_i \|J_{\lambda_i,j}\|_\infty = \varrho(M^{-1}) < 1$$

teljesül.

Amennyiben M^{-1} rendelkezik többszörös multiplicitású sajátértékkel, akkor az ezen sajátértékekhez tartozó Jordan-blokkok

$$J_{\lambda_i,j} = \text{tridiag}(0, \lambda_i, 1) \in \mathbb{C}^{m_{i,j} \times m_{i,j}}$$

alakúak. Ekkor azonban

$$\|J_{\lambda_i,j}\|_\infty = |\lambda_i| + 1 > 1$$

Válasszunk egy megfelelő $\mu_i > 0$ számot úgy, hogy $|\lambda_i| + \mu_i < 1$ teljesüljön, valamint legyen $D_i := \text{diag}_{j=1}^m(\mu_i^{m-j})$. Hasonlósági transzformációt végezve

$$D_i^{-1} J_{\lambda_i} D_i = \text{tridiag}(0, \lambda_i, \mu_i), \quad \text{valamint} \quad \|D_i^{-1} J_{\lambda_i} D_i\|_\infty = |\lambda_i| + \mu_i < 1$$

Azokban az esetekben, amikor a sajátérték egyszeres multiplicitású, legyen $D_i = 1$. Így a

$D := \text{diag}(D_i)$ mátrixra

$$\|D^{-1}JD\|_\infty < 1$$

Legyen $S := DT$, ekkor a fenti eredményeket felhasználva

$$\|SM^{-1}S^{-1}\|_\infty < 1$$

Ez azt jelenti, hogy az

$$\|M^{-1}\|_{op} = \|SM^{-1}S^{-1}\|_\infty$$

operátornorma számunkra megfelelő tulajdonságokkal rendelkezik.

1.2.18. Lemma. Legyen $A : \mathbb{R}^n \to \mathbb{R}^n$ lineáris operátor, $x \in \mathbb{R}^n$. Az $\|A\|_{op} = \|SAS^{-1}\|_\infty$ operátornorma az $\|x\|_{op} = \|Sx\|_\infty$ vektornormához tartozó indukált operátornorma.

Bizonyítás: Használjuk fel $\|.\|_{op}$ definícióját, és a $\|.\|_\infty$ norma tulajdonságait.

$$\|Ax\|_{op} = \|SAS^{-1}Sx\|_\infty \leq \|SAS^{-1}\|_\infty \|Sx\|_\infty = \|A\|_{op}\|x\|_{op}$$

Átrendezve az egyenlőtlenséget azt kapjuk, hogy

$$\|A\|_{op} \geq \frac{\|Ax\|_{op}}{\|x\|_{op}}$$

Az egyenlőtlenség éles, egyenlőség teljesül például $x = S^{-1}\,(1, 0, \ldots, 0)^T$ esetén, így

$$\|A\|_{op} = \sup \frac{\|Ax\|_{op}}{\|x\|_{op}}$$

□

1.3. A törtek halmaza

1.3.1. Definíció. (Törtek halmaza) A (Λ, M, D) feletti törtek halmazának a

$$H = \left\{ \sum_{n=1}^{\infty} M^{-n} a_n : a_n \in D \right\} \subset \mathbb{R}^n$$

halmazt nevezzük.

1.3.2. Állítás. Belátható, hogy H kompakt halmaz $\mathbb{R}^n$ felett.

1.3.3. Definíció. Legyen E tetszőleges kompakt halmaz $\mathbb{R}^n$ felett. Az E halmazban lévő rácspontok halmazát jelölje

$$I(E) = E \cap \Lambda$$

1.3.4. Lemma. Tetszőleges $z \in \Lambda$ esetén létezik $n_0 \in \mathbb{N}$, melyre minden $n \geq n_0$ esetén $\Phi^n(z) \in I(-H)$.

Bizonyítás: Jelölje $B(-H, r)$ a $-H$ halmaz r sugarú nyílt környezetét. Mivel H $\mathbb{R}^n$ kompakt részhalmaza, így léteznie kell egy olyan $\varepsilon > 0$ számnak, melyre

$$(B(-H, \varepsilon) \setminus -H) \cap \Lambda = \emptyset$$

Legyen $z \in \Lambda$ tetszőleges pont. Ekkor z-t n tagra kifejtve kapjuk, hogy

$$z = a_0 + Ma_1 + \dots M^{n-1}a_{n-1} + M^n\Phi^n(z)$$

ahol $a_1, \dots, a_{n-1} \in D$.

Ezt M^{-n}-nel szorozva és átrendezve a következő összefüggést kapjuk

$$\Phi^n(z) = M^{-n}z - \left(M^{-n}a_0 + M^{1-n}a_1 + \dots + M^{-1}a_{n-1}\right)$$

Látható, hogy $-\left(M^{-n}a_0 + M^{1-n}a_1 + \dots + M^{-1}a_{n-1}\right) \in -H$, $M^{-n}z$ normája pedig M expanzivitása miatt megfelelően nagy $n \geq n_0$ esetén kisebb, mint ε, így $\Phi^n(z) \in \Lambda \cap (-H)$. $\square$

1.3.5. Következmény. Minden $z \in \Lambda$ pont pályája belép az $I(-H)$ halmazba, és miután belépett a halmazba nem hagyja el azt.

Ezzel hasonló következtetésre jutunk, mint az 1.2.10. tétel bizonyítása során. Minden pont pályája végül periodikus, és az összes periodikus pont az $I(-H)$ halmazban található.

Az 1.2.12. következmény értelmében így annak eldöntésére, hogy egy (Λ, M, D) radix rendszer számrendszer-e elegendő ellenőriznünk, hogy $I(-H)$ pontjaira Φ a 0 ponthoz tart-e.

1.3.1. A törtek halmazának egy lefedése

A következőkben egy olyan könnyen számítható halmazra adunk konstrukciót, amely tartalmazza a $-H$ halmazt. A célunk olyan $l_i, u_i \in \mathbb{R}$ $(i = 1, \dots, n)$ határok meghatározása, melyre

$$T := \{(x_1, x_2, \dots, x_n) : l_i \leq x_i \leq u_i\} \supseteq -H$$

Ekkor az összes periodikus pont a T halmazban található, így elegendő azon rácspontokat ellenőriznünk, amelyek T-ben találhatóak.

Vezessük be a következő jelöléseket. Jelölje $c_{j,k}(d)$ az $M^{-j}d$ vektor k. komponensét, valamint legyen

$$a_{j,k} = \min_{d \in D} c_{j,k}(d)$$

$$b_{j,k} = \max_{d \in D} c_{j,k}(d)$$

Válasszunk egy megfelelően kicsi, nullánál nagyobb ε valós számot és legyen $m \in \mathbb{N}$ olyan, hogy

$$\|M^{-m}\|_\infty = \delta \leq \varepsilon,$$

ezen kívül definiáljuk a

$$\mathcal{W} := \left\{ (x_1, x_2, \ldots, x_n) : -\sum_{j=1}^{m} b_{i,j} \leq x_i \leq -\sum_{j=1}^{m} a_{i,j} \right\}$$

halmazt. Ekkor tetszőleges $d_1, \ldots, d_m$ jegysorozat esetén $-\sum_{j=1}^{m} M^{-j} d_j \in \mathcal{W}$. Ebből következik, hogy

$$-H \subseteq \left(I + M^{-m} + M^{-2m} + \ldots \right) \mathcal{W}$$

Felhasználva, hogy $\|M^{-m}\|_\infty = \delta$ a következő felső becslés tudjuk adni

$$\begin{aligned} \left\| I + M^{-m} + M^{-2m} + \ldots \right\|_\infty &\leq \|I\|_\infty + \left\|M^{-m}\right\|_\infty + \left\|M^{-2m}\right\|_\infty + \ldots \\ &= 1 + \delta + \delta^2 + \ldots \\ &= \frac{1}{1-\delta} \end{aligned}$$

Ezt a becslést alkalmazva a

$$T := \left\{ (x_1, x_2, \ldots, x_n) : -\sum_{j=1}^{m} b_{i,j} \frac{1}{1-\delta} \leq x_i \leq -\sum_{j=1}^{m} a_{i,j} \frac{1}{1-\delta} \right\}$$

halmaz magában kell foglalja a $\left(1 + M^{-m} + M^{-2m} + \ldots\right) \mathcal{W}$ halmazt és így a $-H$ halmazt is.

Így azt kaptuk, hogy

$$l_i = -\sum_{j=1}^{m} b_{i,j} \frac{1}{1-\delta}, \qquad u_i = -\sum_{j=1}^{m} a_{i,j} \frac{1}{1-\delta}$$

megfelelő alsó, illetve felső korlátok lesznek.

1.3.6. Definíció. (Lefedés mérete) Legyen $T := \{(x_1, x_2, \ldots, x_n) : l_i \leq x_i \leq u_i\}$ a $-H$ halmaz lefedése. Legyen a T mérete a T halmazban található rácspontok száma.

1.3.7. *Megjegyzés.* Mivel a gyakorlatban $(\mathbb{Z}^n, M, D)$ rendszereket vizsgálunk, ezért a T halmazban lévő rácspontok az egész koordinátájú pontok lesznek. Ekkor a lefedés mérete

$$\prod_{i=1}^{n} (\lfloor u_i \rfloor - \lceil l_i \rceil + 1).$$

1.4. Az eldöntési és az osztályozási feladatok

Az általánosított számrendszerek vizsgálatának egyik fő feladata annak eldöntése, hogy egy adott (Λ, M, D) hármas számrendszert alkot-e, illetve amennyiben a (Λ, M, D) hármas nem alkot számrendszert, akkor a periodikus pontok meghatározása.

Ennek a két feladatnak az elvégzésére az alábbi két algoritmust definiáljuk.

1.4.1. Definíció. (Eldöntési algoritmus) Az eldöntési algoritmus feladata, hogy egy bemenetként kapott (Λ, M, D) hármasról meghatározza, hogy a hármas számrendszert alkot-e vagy sem. Az algoritmus kimenete IGAZ, amennyiben (Λ, M, D) számrendszer, különben a kimenet HAMIS.

Az 1.2.7. következmény alapján az eldöntési algoritmus kimenete akkor HAMIS, ha létezik $p \neq 0$ periodikus pont.

1.4.2. *Megjegyzés.* Az eldöntési algoritmusnak elegendő egyetlen tanút találnia. Amennyiben találtunk egy $p \neq 0$ periodikus pontot az algoritmus megállhat.

1.4.3. Definíció. (Osztályozási algoritmus) Az osztályozási algoritmus feladata, hogy egy adott (Λ, M, D) hármas esetén meghatározza a rendszer összes periodikus pontját.

1.4.4. *Megjegyzés.* Amennyiben az eldöntési algoritmus elfogadja a bemenetet, akkor tudjuk, hogy az egyetlen pediodikus pont a 0.

Összefoglalva, mind az eldöntési, mind az osztályozási algoritmus a bemenetként kapott (Λ, M, D) hármas periodikus pontjait keresi. A periodikus pontok helyének behatárolására két eredményt tudunk felhasználni.

Egyrészt az 1.2.10. tétel 1.2.11. következménye szerint, mivel az összes pont pályája végül az origó középpontú, operátornorma szerinti

$$L := \frac{\|M^{-1}\|}{1 - \|M^{-1}\|} \max_{a_i \in D} \|a_i\|$$

sugarú gömbbe vezet és azután onnan nem lép ki, így az összes periodikus pont ebben a gömbben található.

Másrészt pedig az 1.3.5. következmény szerint, mivel az összes pont pályája végül az $I(-H)$ halmazba vezet és azután onnan nem lép ki, így összes periodikus pont az $I(-H)$ halmaznak eleme.

A gyakorlatban ezen halmazok helyett általában egy bővebb, könnyebben bejárható halmaz elemeire ellenőrizzük, hogy melyek periodikus pontok. Ennek eldöntéséhez az egyes pontok pályáját kell vizsgálnunk.

Elegendő minden pontot egyszer vizsgálnunk. Amennyiben egy adott pont pályája elér egy olyan pontot, amelyet már vizsgáltunk, akkor nem kell a pályát tovább vizsgálnunk. Ez két

esetben fordulhat elő: a pontot, amelyet már vizsgáltunk az adott pálya számítása közben vizsgáltuk, ekkor kört találtunk. A második eset, amikor az elért, már vizsgált pontot egy korábbi pálya számításakor vizsgáltuk. Az utóbbi esetben, mivel — a végül periodikus tulajdonság miatt — minden ponthoz pontosan egy kör tartozik, így a pálya kiindulópontjához tartozó kört már egy korábbi pálya vizsgálata során megtaláltuk.

A bővebb halmazokban előfordulhatnak olyan elemek, amelyek pályája kilép a halmazból. Mivel az ilyen pontok pályájára is teljesül, hogy végül visszatér a halmazba, így amennyiben elhagyjuk a halmazt a pálya számítását nem kell folytatnunk, mivel a visszatérési pontot már vizsgáltuk, vagy vizsgálni fogjuk.

Ezek alapján az osztályozási algoritmust az alábbi módon tudjuk megvalósítani.

Algoritmus 1 Osztályozási algoritmus

```
1:  function CLASSIFY(Λ, M, D)
2:      finished ← { };
3:      P ← {};
4:      for all z ∈ K(Λ, M, D) do
5:          if z ∉ finished then
6:              orbit ← {};
7:              repeat
8:                  orbit ← orbit ∪ {z};
9:                  finished ← finished ∪ {z};
10:                 z ← Φ(z);
11:             until z ∉ finished and z ∈ K(Λ, M, D);
12:             if z ∈ orbit then
13:                 P ← P ∩ { GetCycle(z) };
14:             end if
15:         end if
16:     end for
17:     return P;
18: end function
```

Az algoritmusban a $K(\Lambda, M, D)$ halmaz olyan, hogy biztosan magában foglalja a rendszer periodikus pontjait. Ilyen például a törtek halmazának egy lefedése. Az adott periodikus ponthoz tartozó kör meghatározására használhatjuk a következő algoritmust.

Algoritmus 2 Kör meghatározása

```
1:  function GETCYCLE(z)
2:      C ← {};
3:      z' ← z;
4:      repeat
5:          C ← C ∪ {z'};
```

```
        z' ← Φ(z');
    until z' ≠ z;
    return C;
end function
```

Az eldöntési algoritmus nem különbözik jelentősen az osztályozási algoritmustól. Az eldöntési algoritmus esetén az első nem 0 periodikus pont megtalálása után az algoritmus megállhat, ekkor a bemenetet elutasítjuk, azaz visszatérési érték HAMIS lesz. Amennyiben nem találunk ilyen periodikus pontot, akkor az algoritmus elfogadja a bemenetet, a kimenet IGAZ lesz.

Algoritmus 3 Eldöntési algoritmus

```
function DECIDE(Λ, M, D)
    finished ← { };
    for all z ∈ K(Λ, M, D) do
        if z ∉ finished then
            orbit ← {};
            repeat
                orbit ← orbit ∪ {z};
                finished ← finished ∪ {z};
                z ← Φ(z);
            until z ∉ finished and z ∈ K(Λ, M, D);
            if z ≠ 0 and z ∈ orbit then
                return FALSE;
            end if
        end if
    end for
    return TRUE;
end function
```

Az eldöntési algoritmus esetén az algoritmus futásidejét az határozza meg, hogy milyen hamar találunk nem 0 periodikus pontot, így ha egy adott rendszer esetén sejtjük merre lehetnek ilyen periodikus pontok, akkor érdemes a $K(\Lambda, M, D)$ halmaz átvizsgálását azokon a helyeken kezdeni.

2. fejezet

Adattípusok kezelése

Szeretnénk úgy megvalósítani az adatszerkezeteinket és az algoritmusainkat, hogy lehetőség szerint bármely megfelelő adattípussal használhatóak legyek, akár egész számokkal, akár lebegőpontos számokkal, akár racionális számokkal, akár ezek tetszőleges pontosságú vagy komplex változataival szeretnénk műveleteket végezni.

Szeretnénk kezelni az egyes típusok közti függéseket – például két egész szám hányadosa racionális szám –, illetve a különböző típusok közti konverziókat is.

Továbbá az egyes adattípusok gyakran használt függvényeihez szeretnénk egy egységes interfészt biztosítani, amelyet az algoritmusaink implementálásához felhasználhatunk.

Ezen feladatok megvalósítására **template** specializációval megvalósított **trait** osztályokat használunk. Bár bizonyos esetekben tudunk adni általános implementációt, de ennek a módszernek a hátránya, hogy általában csak azokra a típusokra tudjuk használni ezeket a **trait** osztályokat, amelyek rendelkeznek specializált implementációval.

2.1. Típusok

Az fix pontosságú egész számok közül a könyvtár az **int**, illetve a **long long** típusokat támogatja. A fix pontosságú lebegőpontos számok közül pedig a **double** típushoz nyújtunk támogatást. A cél architektúrán az **int** típus 32 bit, a **long long**, illetve **double** típusok 64 bit szélességűek.

A könyvtár azon komplex számok kezelését is támogatja, amelyek valós és képzetes része támogatott típusú.

A könyvtárat úgy valósítjuk meg, hogy fix pontosságú és tetszőleges pontosságú számokkal is használható legyen. A tetszőleges számok kezeléséhez az ingyenesen hozzáférhető, nyílt forráskódú **GMP** könyvtárat használjuk. A **GMP** könyvtár képes tetszőleges pontosságú egész számok, racionális számok és lebegőpontos számok kezelésére. Ezen kívül nagy számú aritmetikai, számelméleti és egyéb függvényeket is biztosít ezen típusok használatához.

2.2. Típusok összekapcsolása

Bizonyos esetekben egy függvény kimenetének típusa nem egyezik meg a bemenet típusával, viszont a bemeneti és a kimeneti típus között valamiféle függés van.

Tekinthetjük az alábbi példákat: általánosan csak annyit tudunk mondani, hogy egy egész szám abszolútértéke egész szám, egy valós szám abszolútértéke valós szám lesz, ezekben az esetekben az eredmény típusa megegyezik a bemenet típusával, azonban egy olyan komplex szám esetén, amelynek valós és képzetes része is egész ez már nem áll fenn. Ebben az esetben abszolútérték valós, az abszolútérték négyzete pedig egész lesz. Két egész szám hányadosa racionális, míg két valós szám hányadosa valós. Valamint azt szeretnénk, hogy tetszőleges pontosságú bemenet esetén a kimenet is tetszőleges pontosságú legyen.

Ha az elemi műveletek visszatérési értékeinek típusát tudjuk az egyes bemeneti típusokra, akkor tetszőleges függvénynek meg tudjuk állapítani a kimeneti típusát a bemenet típusa alapján.

A leggyakrabban használt műveletek közül az osztás, a gyökvonás, és az abszolútérték számítás esetén változik meg a kimenet típusa. Ezen kívül egyes számítások elvégzéséhez szükségünk van komplex számokra való áttérésre is, így meg kell tudnunk határozni az adott típushoz megfelelő komplex típust.

A típusok összekötésére bevezetjük a **GeNuSys::TypeTraitsBase template** osztályt.

```
1 template<typename Type>
2 struct TypeTraitsBase {
3 };
```

2.1. ábra. Az **TypeTraitsBase** osztály

Az osztály nem specializált változata üres, a specializált változatok pedig a következő típusokat definiálják. A továbbiakban az adott $\mathbb{K}$ típushoz kapcsolt típusokat is az alábbiak szerint jelölhetjük.

Név	Leírás	Jelölés
RealType	valós számok típusa	$\mathbb{K}_R$
RationalType	racionális számok típusa	$\mathbb{K}_Q$
ComplexType	a típussal kompatibilis komplex típus	$\mathbb{K}_C$
AbsType	abszolútérték számítás eredménye	$\mathbb{K}_{Abs}$
AbsSqrType	abszolútérték számítás eredményének négyzete	$\mathbb{K}_{AbsSqr}$

Példaként megvizsgálhatjuk az osztály **int** típushoz tartozó specializált változatát:

```
template<>
struct TypeTraitsBase<int> {
  typedef double RealType;
  typedef double RationalType;
  typedef std::complex<int> ComplexType;
  typedef int AbsType;
  typedef int AbsSqrType;
};
```

2.2. ábra. Az **TypeTraitsBase<int>** osztály

Miután rendelkezésünkre áll a **TypeTraitsBase trait** osztály, az alábbi módon tuduk például az osztás elvégzésére egy függvényt definiálni.

```
template<typename Type>
typename TypeTraitsBase<Type>::RationalType div(const Type& a, const Type&
    b);
```

2.2.1. Konvertálás típusok között

Szükségünk van a különböző típusok közti konverziók elvégzésére is. A továbbiakban kétféle konverziót különböztetünk meg: a biztonságos (safe) konverziót, illetve a veszélyes (unsafe) konverziót.

Akkor beszélhetünk biztonságos konverzióról, amikor egy adott típusú elemet minden esetben veszteség nélkül tudunk konvertálni az új típusra. Ilyen konverziók például egész ábrázolásról lebegőpontosra való áttérés, a típushoz tartozó komplex típusra való konvertálás, vagy az adott típushoz tartozó tetszőleges pontosságú típusra való átváltás.

Azokat az eseteket, amikor az átváltás elvégzésekor veszteség léphet fel veszélyes konverziónak nevezzük. Ez többek között akkor fordulhat elő, ha lebegőpontos ábrázolásról egész ábrázolásra váltunk, vagy tetszőleges pontosságról fix pontosságra váltunk át.

A különböző konverziók elvégzésére a **GeNuSys::TypeTraits template** osztály biztosítja a következő metódusokat.

```
template<typename Type>
struct TypeTraits : TypeTraitsBase<Type> {
  template<typename ConvertedType>
  static ConvertedType asType(const Type& value);
  template<typename ConvertedType>
  static ConvertedType asTypeUnsafe(const Type& value);
};
```

2.3. ábra. Az **TypeTraits<int>** osztály

A **GeNuSys::TypeTraits** osztály örökli a **GeNuSys::TypeTraitsBase** osztályban definiált típusokat, így azok elérhetőek a **GeNuSys::TypeTraits** osztályon keresztül is.

Az egyes konverziók megvalósítását az osztály metódusainak specializált verziói tartalmazzák. Minden elvégezni kívánt konverziónak rendelkeznie kell egy megfelelő implementált specializációval.

Ezeket a konvertáló metódusokat fogjuk használni a későbbiekben implicit és explicit konverziók elvégzésére is.

2.3. Konstansok és műveletek

A **GeNuSys::NumberTraits** osztály biztosít egységes interfészt az egyes típusokhoz tartozó konstansok, valamint a típusokkal tipikusan végzendő műveletek eléréséhez.

Az interfészen elérhetővé tesszük az additív és a multiplikatív egységelemeket — **zero**, illetve **one** —, valamint egy ε (**epsilon**) állandót úgy, hogy amennyiben egy adott elem abszolútértéke kisebb ennél az adott állandónál, akkor az elem értékét 0-nak tekinthetjük.

A **GeNuSys::NumberTraits** osztály által megvalósított műveletek azok, amelyeket az algoritmusaink implementálása során használni szeretnék, azonban az egyes típusok nem rendelkeznek egységes interfésszel ezek használatához. A megvalósított **trait** osztály ezen műveletek közül az előjel, illetve abszolútérték számításra, a hatványozásra és a gyökvonásra, valamint — mivel a legtöbb esetben a típushoz tartozó osztás operátor egész egészosztást valósít meg — az osztás elvégzésére is lehetőséget ad.

Amennyiben lehetséges az egyes műveletekhez adunk általános, tetszőleges adattípus esetén is működő megvalósítást, de ilyen esetekben is ahol az egyes típusokhoz elérhetőek hatékonyabb megvalósítások, akkor **template** specializációt alkalmazunk.

Abszolútérték és előjel számítás

Általános esetben az abszolútérték függvényt a következő módon definiáljuk:

$$|x| = \sqrt{\mathrm{Re}(x)^2 + \mathrm{Im}(x)^2}$$

Amennyiben valós szám abszolútértékét szeretnénk meghatározni, akkor használhatjuk az

$$|x| = \begin{cases} x & \text{ha } x \geq 0, \\ -x & \text{különben.} \end{cases}$$

függvényt.

Láthatjuk, hogy a gyökvonás miatt a fenti függvény visszatérési értékének típusa változhat. Az eredmény egész valós és képzetes rész esetén is valós lesz, a valós esetben viszont a

függvény megőrzi a típust, így a típusok összekapcsolásánál látottakat felhasználva a függvény visszatérési értékének típusa **TypeTraits<Type>::AbsType** lesz.

Mivel komplex számok esetén gyakran elegendő az abszolútérték négyzetét meghatároznunk, így erre külön függvényt biztosítunk.

Az előjel függvényt az abszolútérték függvény alapján az alábbi módon definiáljuk:

$$\operatorname{sgn}(x) = \begin{cases} \dfrac{x}{|x|} & \text{ha } x \neq 0, \\ 0 & \text{különben.} \end{cases}$$

Valós esetben a függvény értéke -1, 0, vagy 1, komplex esetben viszont az érték $e^{i \arg x}$.

```
1 template<typename Type>
2 typename TypeTraits<Type>::AbsType abs(const Type& value);
3 template<typename Type>
4 typename TypeTraits<Type>::AbsSqrType absSqr(const Type& value);
5 template<typename Type>
6 Type sgn(const Type& value);
```

2.4. ábra. Az **abs**, az **absSqr**, valamint a **sgn** függvények

Hatványozás és gyökvonás

Lehetőséget biztosítunk tetszőleges típusú szám egész hatványainak és gyökeinek meghatározására. A hatványozáshoz a bináris gyors hatványozást használjuk, mely segítségével ha $n \in \mathbb{N}$, valamint a tetszőleges szám, akkor a^n értékét n szorzás elvégzése helyett $2\log_2(n)$ szorzással meghatározhatjuk.

Algoritmus 4 Gyors hatványozás

```
1:  function FastPow(a, n)
2:      result ← 1;
3:      acc ← a;
4:      while n > 0 do
5:          if n ≡ 1 (mod 2) then
6:              result ← result × acc;
7:          end if
8:          n ← n/2;
9:          acc ← acc × acc;
10:     end while
11:     return result;
12: end function
```

A gyökvonást a Newton-módszer segítségével tudjuk elvégezni. Legyen $n \in \mathbb{N}^+$, $a \in \mathbb{R}$ pozitív, az a szám n. gyökét keressük. Legyen $f(x) = x^n - a$. Ezeket a jelöléseket használva

$$f(x) = 0 \Longleftrightarrow x = \sqrt[n]{a}$$

A Newton-módszer alapján, kihasználva f tulajdonságait, amennyiben

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},$$

valamint $x_0 = 1$, akkor $\lim_{k\to\infty} x_k = \sqrt[n]{a}$. Amennyiben $f(x_k) < \varepsilon$ az algoritmust megállíthatjuk.

Algoritmus 5 Gyökvonás Newton-módszerrel

1: **function** ROOT(a, n)
2: $x \leftarrow 1$;
3: **repeat**
4: $x \leftarrow x - \dfrac{x^n - a}{nx^{n-1}}$;
5: **until** $|x^n - a| \geq \varepsilon$
6: **return** x;
7: **end function**

Ezen felül a négyzetgyök számításához külön függvényt adunk az interfészen, mivel az adattípusok többségéhez rendelkezésre áll a négyzetgyököt az általános implementációnál gyorsabban kiszámító függvény.

```
1 template<typename Type>
2 Type pow(const Type& value, unsigned int n);
3 template<typename Type>
4 typename TypeTraits<Type>::RealType sqrt(const Type& value);
5 template<typename Type>
6 typename TypeTraits<Type>::RealType root(const Type& value,
7                                           unsigned int n);
```

2.5. ábra. A **pow**, az **sqrt** és a **root** függvények

Osztás

Mivel a legtöbb esetben az adott típushoz tartozó osztás operátor egész egészosztást valósít meg, így szükségünk van egy egységes osztás művelet bevezetésére is.

```
1 template<typename Type>
2 typename TypeTraits<Type>::RationalType div(const Type& a, const Type& b);
```

2.6. ábra. A **div** függvény

2.3.1. Komplex számok

Mint ahogy korábban láttuk a megvalósítandó műveletek számítási módja jelentősen különbözhet komplex számok esetén a valós eset számítási módjától. Az abszolútérték függvény, illetve az előjel függvény esetén például

$$\text{abs}(x) = \sqrt{\text{Re}(x)^2 + \text{Im}(x)^2}, \quad \text{valamint} \quad \text{sgn}(x) = e^{i \arg x}.$$

A komplex számokhoz ezen felül szükségünk van egy új művelet, a konjugálás bevezetésére is, valós számok esetén a függvény az értéken nem módosít, komplex esetben visszaadja a bemenet konjugáltját.

```
1 template<typename Type>
2 Type conj(const Type& value);
```

2.7. ábra. A **conj** függvény

Komplex számok esetén az általános n. gyökvonás műveletét nem implementálja a könyvtár, a négyzetgyök függvény esetén pedig, amennyiben a képzetes rész nem 0, a következő képlet szerint biztosítunk egyetlen kijelölt gyököt:

$$\text{sqrt}(a + bi) = \sqrt{\frac{a + \sqrt{a^2 + b^2}}{2}} + \text{sgn}(b)\sqrt{\frac{-a + \sqrt{a^2 + b^2}}{2}}i$$

Az osztás műveletét pedig az alábbi módon valósítjuk meg $z_1, z_2 \in \mathbb{C}$ komplex számok esetén.

$$\frac{z_1}{z_2} = \frac{z_1\overline{z_2}}{z_2\overline{z_2}} = \frac{z_1\overline{z_2}}{|z_2|^2},$$

vagyis a komplex számmal való osztást felcseréljük egy komplex számmal való szorzás és egy abszolútértékkel valós osztásra egymásutánjára.

A fenti műveletek megvalósítását szintén a **GeNuSys::NumberTraits** osztály tartalmazza. Az implementáció **template** specializációval történik.

2.3.2. Egész számok

Egész számokkal tipikusan végzett műveletek eléréséhez a megfelelő típusok esetén a **GeNuSys::IntegerTraits** osztály biztosít egységes interfészt.

Az osztály metódusait használva lehetőség van oszthatóság ellenőrzésére, egészosztásra, maradékszámításra, valamint szimmetrikus maradékszámításra is. Maradékszámítás esetén a maradék értéke

$$\mathrm{mod}(a,b) = a \pmod b \in \{0,1,\ldots,b-1\},$$

szimmetrikus maradékszámítás esetén pedig

$$\mathrm{mods}(a,b) = a \pmod b \in \left\{\left\lfloor -\frac{b}{2}\right\rfloor + 1,\ldots,-1,0,1,\ldots,\left\lfloor\frac{b}{2}\right\rfloor\right\}.$$

A **GeNuSys::IntegerTraits** osztály ezen kívül implementálja a bővített Euklideszi algoritmust is az alábbi módon.

Algoritmus 6 Bővített Euklideszi algoritmus

1: **function** EXTENDEDGCD(a, b)
2: $(r_0, u_0, v_0) \leftarrow (a, 1, 0)$;
3: $(r_1, u_1, v_1) \leftarrow (b, 0, 1)$;
4: **while** $r_1 \neq 0$ **do**
5: $q \leftarrow r_0$ quo r_1;
6: $(r, u, v) \leftarrow (r_0 - qr_1, u_0 - qu_1, v_0 - qv_1)$;
7: $(r_0, u_0, v_0) \leftarrow (r_1, u_1, v_1)$;
8: $(r_1, u_1, v_1) \leftarrow (r, u, v)$;
9: **end while**
10: **return** (r_0, u_0, v_0);
11: **end function**

A bővített Euklideszi algoritmus azon kívül, hogy megadja $a, b \in \mathbb{Z}$ legnagyobb közös osztóját, előállítja azon $u, v \in \mathbb{Z}$ együtthatókat is, melyekre

$$ua + tb = \gcd(a,b).$$

```
1  bool divisible(const Type& a, const Type& b);
2  Type idiv(const Type& a, const Type& b);
3  Type mod(const Type& a, const Type& b);
4  Type mods(const Type& a, const Type& b);
5  ExtendedGCD<Type> egcd(const Type& a, const Type& b);
```

2.8. ábra. A **GeNuSys::IntegerTraits** osztály

3. fejezet

Lineáris algebra

Az általánosított számrendszerekhez kapcsolódó számítások elvégzése közben nagyrészt vektorokkal és mátrixokkal kell műveleteket végrehajtanunk. Az általánosított számrendszerek vizsgálatához ezért lineáris algebrai adatszerkezetek és algoritmusok megvalósítására van szükségünk.

Mivel a lineáris algebrai műveletek elvégzése általában költséges — bizonyos esetekben az elemek számával négyzetesen, vagy akár köbösen arányos —, így a lehető leghatékonyabb implementációt szeretnénk biztosítani.

Általában összetett, nagyobb méretű adatszerkezeteket kell kezelnünk, így a memória hatékony kezelése, a másolások, allokációk elkerülése szintén javíthatja az elvégzett műveletek futásidejét.

A megvalósított algoritmusok és adatszerkezetek a **GeNuSys::LinAlg** névtében találhatóak.

3.1. Adatszerkezetek

A lineáris algebrai műveletek elvégzéséhez szükségünk van megfelelő adatszerkezetek, elsősorban vektorok és mátrixok implementációjára. Ezen adatszerkezetek hatékony megvalósítása nagyban befolyásolja az algoritmusaink futásidejét. Ezen felül olyan megvalósítást szeretnénk létrehozni, amely a vektorokkal, mátrixokkal végzendő algoritmusok hatékony alkalmazását is lehetővé teszi.

Az is célunk továbbá, hogy a megvalósított adatszerkezeteink képesek legyenek tetszőleges típusú elemek kezelésére, azaz ugyanazon implementáció használható legyen például egészekkel, lebegőpontos számokkal, vagy tetszőleges pontosságú számokkal is. Ehhez az adatszerkezeteinket **template** osztályokként fogjuk megvalósítani és felhasználjuk az előző fejezetben bemutatott interfészeket is.

A könyvtár a következő négy adatszerkezet implementációját biztosítja: vektor, ritka vektor, mátrix, ritka mátrix. Az adatszerkezeteinkben az indexelés minden esetben 0 alapú.

3.1.1. Vektor

Egy tetszőleges $v \in \mathbb{K}^n$ vektort ábrázoló adatszerkezetet a **GeNuSys::LinAlg::Vector** osztály valósítja meg. Az implementáció tárolja a vektor méretét, valamint folytonosan tárolja a vektor elemeit egy **std::vector** konténer segítségével.

```
1 template <class Type>
2 class Vector {
3   unsigned int length;
4   std::vector<Type> elem;
5 };
```

3.1. ábra. A **Vector** osztály mezői

Az adatszerkezet memóriaigénye megegyezik az ábrázolt vektor méretével, azaz $\Theta(n)$.

Az egyes komponensek értékének lekérésére, illetve módosításra kétféle lehetőséget adunk. Az osztály publikus interfésze biztosít erre a célra egy-egy metódust, illetve meghatározott osztályoknak lehetőségük van közvetlenül hozzáférniük az osztály mezőihez.

```
1   const Type& operator [](unsigned int idx) const;
2   void set(unsigned int idx, const Type& value);
```

3.2. ábra. Hozzáférés az vektor komponenseihez

Mivel a vektor minden komponense ábrázolva van, így ha az i. komponenshez szeretnénk hozzáférni, akkor az **elem** tömb i. eleméhez kell csak hozzáférnünk. Ez $\mathcal{O}(1)$ művelettel végrehajtható, így a vektor komponenseinek olvasása és írása is $\mathcal{O}(1)$ műveletigényű.

A vektor adatszerkezet implementációját a **vector.h**, illetve **vector.hpp** fájlok tartalmazzák.

3.1.2. Ritka vektor

3.1.1. Definíció. (Ritka vektor) Legyen $v \in \mathbb{K}^n$. Amennyiben a v vektorban az elemek túlnyomó része 0, akkor azt mondjuk, hogy a vektor *ritka*.

Legyen a vektor nemnulla komponenseinek száma NNZ. A fenti definíció alapján amennyiben a vektor ritka, akkor NNZ $\ll d$. Ritka vektorok esetén ezt a tulajdonságot kihasználva próbáljuk a sűrű vektorokhoz képest javítani a vektor memóriaigényét, illetve az elvégzett számítások műveletigényét.

A ritka vektor adatszerkezetet a **GeNuSys::LinAlg::SparseVector** osztály valósítja meg. A megvalósított adatszerkezet tárolja a vektor méretét, illetve a nemnulla elemeket a hozzájuk tartozó indexekkel, a **Vector** osztályhoz hasonlóan folytonos sorrendben.

```
1 template <class Type>
2 class SparseVector {
3   struct Entry {
4     unsigned int idx;
5     Type value;
6   };
7   unsigned int length;
8   std::vector<Entry> elem;
9 };
```

3.3. ábra. A **SparseVector** osztály mezői

A tömörített ábrázolás hátránya, hogy egy tetszőleges indexű komponenshez való hozzáférés bonyolultabbá válik, mint a **Vector** osztály esetén. Mivel a nemnulla komponensek index szerint növekvő sorrendben vannak tárolva, így a megfelelő indexű bejegyzést *bináris keresés*t alkalmazva találhatjuk meg az **elem** tömbben. Ezzel a módszerrel az i. komponenshez való hozzáférés műveletigénye $\mathcal{O}(\log(\text{NNZ}))$.

Amennyiben a vektort módosítani szeretnénk, nemnulla elemet törölni, illetve hozzáadni, akkor az ábrázolás invariánsának fenntartásához – nevezetesen, hogy az elemeket hézagok nélkül, index szerint növekvő sorrendben tároljuk – az elemek másolására, esetlegesen új memória allokálására van szükségünk. Így ezek a műveletek jellemzően $\mathcal{O}(\text{NNZ})$ művelettel járnak.

Ezért amennyiben egy művelet eredménye ritka vektor, akkor a művelet lépéseit úgy szeretnénk szervezni, hogy ne legyen szükség törlésre, beszúrásra, hanem a vektor komponenseit index szerint növekvő sorrendben kapjuk meg.

A ritka vektor összes komponensének felsorolásának műveletigénye nem nő a vektornál látottakhoz képest, sőt megtehető a nemnulla elemek számával arányos számú, azaz $\Theta(\text{NNZ})$ művelettel. A ritka vektorokkal végzett műveleteknél ezért előnyös, ha a vektor elemeit sorban kell elérnünk.

A ritka vektor adatszerkezet implementációját a **sparse_vector.h**, illetve **sparse_vector.hpp** fájlok tartalmazzák.

3.1.3. Mátrix

Egy tetszőleges $A \in \mathbb{K}^{n \times m}$ mátrixot ábrázoló adatszerkezetet a **GeNuSys::LinAlg::Matrix** osztály valósítja meg. Az adatszerkezet tárolja a mátrix sorainak, illetve oszlopainak számát, valamint *sorfolytonosan* tárolja a mátrix elemeit egy **std::vector** konténer felhasználásával.

```
1 template <class Type>
2 class Matrix {
3   unsigned int rows;
4   unsigned int cols;
5   std::vector<Type> elem;
6 };
```

3.4. ábra. A **Matrix** osztály mezői

Az adatszerkezet a mátrix minden komponensét ábrázolja, így memóriaigénye arányos az ábrázolt mátrix méretével, azaz $\Theta(nm)$.

A vektor osztályokhoz hasonlóan a mátrix osztály is kétféle hozzáférési lehetőséget biztosít a komponensekhez: az osztály interfésze tartalmaz egy metódust az olvasáshoz és egy metódust az íráshoz, ezen felül egyes osztályok itt is rendelkeznek közvetlen hozzáféréssel az osztály mezőihez.

```
1   const Type& operator ()(unsigned int i, unsigned int j) const;
2   void set(unsigned int i, unsigned int j, const Type& value);
```

3.5. ábra. Hozzáférés a mátrix komponenseihez

Az i. sor j. elemének tömbbeli indexe közvetlenül számolható az

$$i \times m + j$$

képlettel. Így a mátrix egyes elemeihez $\mathcal{O}(1)$ művelettel férhetünk hozzá. Az elemek olvasása, illetve írása így szintén $\mathcal{O}(1)$ műveletet igényel.

Az algoritmusok általában sor- vagy oszlopfolytonosan férnek hozzá a mátrix elemeihez. Amennyiben egy adott elem tömbbeli indexe idx, akkor a sorban következő elem indexe $idx+1$, az oszlopban következő elem indexe pedig $idx + m$. Így nem kell minden lépésben újra kiértékelnünk az indexre vonatkozó képletet, ezt felhasználva hatékonyan tudunk mind sorfolytonosan, mind oszlopfolytonosan hozzáférni az elemekhez. Természetesen figyelnünk kell arra, hogy ekkor ne haladjunk túl a sor, illetve oszlop határain.

A mátrix adatszerkezet implementációját a **matrix.h**, illetve **matrix.hpp** fájlok tartalmazzák.

3.1.4. Ritka mátrix

A ritka vektorokhoz hasonlóan definiáljuk a ritka mátrixokat.

3.1.2. Definíció. (Ritka mátrix) Legyen $A \in \mathbb{K}^{n \times m}$. Amennyiben az A mátrixban az elemek túlnyomó része 0, akkor azt mondjuk, hogy a mátrix *ritka*.

A ritka mátrix adatszerkezet a **GeNuSys::LinAlg::SparseMatrix** osztály valósítja meg. A mátrix megvalósításához hasonlóan a ritka mátrix is tárolja a mátrix sorainak, illetve oszlopainak számát. Az mátrix elemeinek tárolását *sortömörített formában* végezzük.

```
template <class Type>
class SparseVector {
  struct Entry {
    unsigned int col_idx;
    Type value;
  };
  unsigned int rows;
  unsigned int cols;
  std::vector<unsigned int> row_ptr;
  std::vector<Entry> elem;
};
```

3.6. ábra. A **SparseMatrix** osztály mezői

Legyen a mátrix összes nemnulla elemének száma NNZ, az i. sor nemnulla elemeinek száma pedig NNZ_i. Ha a mátrix ritka, akkor $\text{NNZ} \ll nm$.

A sortömörített formátumban az **elem** tömb sorfolytonosan tartalmazza az ábrázolt mátrix nemnulla elemeit a hozzájuk tartozó oszlop indexszel együtt. Az egyes sorok gyors elérésének érdekében a **row_ptr** tömb tartalmazza az egyes sorok első tárolt elemének indexét az **elem** tömbben. Az **elem** tömb mérete NNZ, valamint

$$\text{row\_ptr}[i] = \sum_{k=0}^{i-1} \text{NNZ}_i$$

A ritka mátrixokkal elvégzendő műveletek implementációjának megkönnyítése érdekében a **row_ptr** tömb végére rakunk egy segédelemet, így a tömb mérete $n + 1$ lesz, valamint az $\text{row\_ptr}[n]$ segédelem értéke az **elem** tömb mérete. Ezt felhasználva a mátrix i. sorában lévő elemek **elem** tömbbeli indexeinek halmaza

$$\{idx : \text{row\_ptr}[i] \leq idx < \text{row\_ptr}[i+1]\}$$

A ritka vektorok esetében ismertetett módszereket felhasználhatjuk ritka mátrixok esetén is.

Az i. sor j. elemének eléréséhez *bináris keresés*t alkalmazunk az **elem** tömb row_ptr$[i]$. és row_ptr$[i+1]$. indexe közt. Így az i. sor egyes elemek elérése $\mathcal{O}(\text{NNZ}_i)$ műveletet igényel.

Nemnulla elemek törlésére, illetve beszúrására ritka mátrixok esetén is van lehetőség, de ezeknek a műveleteknek az elvégzése szintén másolással, illetve esetleges memóriafoglalással járhat, így ezek a műveletek ritka mátrixok esetén is jellemzően $\mathcal{O}(\text{NNZ})$ műveletet igényelnek. Ezért itt is fennáll, hogy azoknál a műveleteknél, amelyek eredménye ritka mátrix, sorfolytonos sorrendben szeretnénk megkapni a mátrix elemeit.

Az elemek sorfolytonos felsorolása ritka mátrixok esetén is megtehető a nemnulla elemek számával arányos számú, azaz $\Theta(\text{NNZ})$ művelettel. Az oszlopfolytonos felsorolásra, vagy akár egy tetszőlegesen kiválasztott oszlop elemeinek felsorolására azonban nem tudunk a mátrixnál látott hatékony módszert adni. Így a ritka mátrixokkal végzett műveleteknél is előnyös, hogyha a mátrix elemeit sorfolytonos sorrendben érjük el.

Ha az összes elemet akarjuk oszlopfolytonosan felsorolni, akkor azt megtehetjük egy segédtömb használatával, amiben azt vezetjük, hogy egy adott sor hányadik elemét dolgoztuk már fel. Ekkor a felsorolás megtehető $\mathcal{O}(\text{NNZ})$ lépésben, azonban szükséges $\Theta(n)$ memória, illetve egy lépés végrehajtása is bonyolultabbá vállik. Ha csak egy adott oszlopban keressük a következő elemet, akkor a következő sorra bináris keresést kell futtatnunk. Ezzel a módszerrel $\mathcal{O}(n \log(\text{NNZ}))$ műveletet igényel egy teljes oszlop felsorolása.

A ritka mátrix adatszerkezet implementációját a **sparse_matrix.h**, illetve **sparse_matrix.hpp** fájlok tartalmazzák.

3.2. Műveletek megvalósítása

A megvalósított vektor, illetve mátrix adatszerkezetekkel szeretnénk különböző elemi lineáris algebrai műveleteket végezni. Ilyen műveletek például egy adott vektor, illetve mátrix összes elemének számmal való szorzása, osztása — egészosztása, maradékos osztása —; összeadások, kivonások, különböző szorzások elvégzése; illetve összehasonlítások elvégezése.

A különféle műveleteket megvalósítjuk sűrű, illetve ritka ábrázolású adatszerkezetek kombinációira is, továbbá a műveletek eredményének esetében is lehetővé tesszük a megfelelő ábrázolás kiválasztását.

A műveletek implementációját a **GeNuSys::LinAlg::Operations** osztály tartalmazza, de az egyszerűbb használat érdekében, az egyes adatszerkezetek is rendelkeznek a megfelelő operátorokkal, amelyek a **GeNuSys::LinAlg::Operations** osztály megfelelő metódusait hívják.

A **GeNuSys::LinAlg::Operations** osztály metódusai úgy vannak kialakítva, hogy nem végeznek memórafoglalást. Az adott művelet vagy helyben elvégezhető, vagy az eredmény egy előre lefoglalt, megfelelő típusú bementként kapott objektumba kerül.

```
1 template<typename Type>
2 void vct_sub(const Vector<Type>& op1, const SparseVector<Type>& op2,
     Vector<Type>& result);
3 template<typename Type>
4 void vct_sub(Vector<Type>& op1, const SparseVector<Type>& op2);
```

3.7. ábra. Példa a **GeNuSys::LinAlg::Operations** osztályban definiált műveletekre

Az adatszerkezetek interfészén lévő operátorok elvégzik a memóriafoglalást, létrehoznak egy megfelelő típusú objektumot az eredménynek, meghívják a művelethez tartozó metódust és az általuk létrehozott példányt adják vissza eredményként.

Mivel az összes lineáris algebrai adatszerkezeteket használó algoritmusunk ezen műveletek elvégzésére támaszkodik, ezért a célunk, hogy a lehető leghatékonyabb implementációt adjuk az egyes műveletekhez. Így például a ritka, illetve sűrű bemenetek és kimenetek különböző variációit külön-külön kell implementálnunk, hogy a különböző ábrázolások előnyeit ki tudjuk használni. Ez egyes esetekben, akár 8 különféle megvalósítást is jelenthet egyetlen művelethez.

A következőkben három csoportban vizsgáljuk a megvalósított műveleteket: vektor műveletek, mátrix műveletek, illetve különböző szorzatok előállítása.

Az egyes vektor, illetve mátrix műveletek műveletigényét a különböző reprezentációk esetén alapvetően az befolyásolja, hogy hány lépésben tudjuk az operandusok elemeit felsorolni.

3.2.1. Vektor műveletek

Legyen $v \in \mathbb{K}^n$ vektor. A következő komponensenkénti vektor műveleteket szeretnénk megvalósítani: szorzás, osztás, egészosztás, illetve maradékszámítás.

Ezen műveletek végrehajtásához elegendő a vektor minden komponensével elvégezni az adott műveletet, ezért ezeknek a műveleteknek a műveletigénye arányos az ábrázolt elemek számával: sűrű vektorok esetén $\Theta(n)$, ritka vektorok esetén $\Theta(\text{NNZ})$.

A műveleteket kétféle módon végezhetjük el: helyben, ekkor a meglévő vektor elemeit módosítjuk, illetve kiszámíthatjuk az eredményt egy külön vektorba is. Az utóbbi esetben a művelet elvégzése előtt memóriafoglalást kell végeznünk, vagy már lefoglalt memóriát kell újrahasznosítanunk.

Egyedül az osztást nem tudjuk helyben elvégezni, mivel megváltoztathatja az elemek típusát.

Az **GeNuSys::LinAlg::Operations** osztály a következő függvényeket biztosítja.

Művelet	**Metódus neve**	**Műveletigény**		**Helyben**	
		Sűrű	**Ritka**	**Sűrű**	**Ritka**
szorzás	vct_mul	$\Theta(n)$	$\Theta(\text{NNZ})$	$\Theta(n)$	$\Theta(\text{NNZ})$
egészosztás	vct_idiv	$\Theta(n)$	$\Theta(\text{NNZ})$	$\Theta(n)$	$\Theta(\text{NNZ})$
osztás	vct_div	$\Theta(n)$	$\Theta(\text{NNZ})$	—	—
maradék	vct_mod	$\Theta(n)$	$\Theta(\text{NNZ})$	$\Theta(n)$	$\Theta(\text{NNZ})$
szimmetrikus maradék	vct_mods	$\Theta(n)$	$\Theta(\text{NNZ})$	$\Theta(n)$	$\Theta(\text{NNZ})$

3.1. táblázat. Vektor műveletek költsége – sűrű, illetve ritka operandusok esetén

A fenti komponensenkénti vektor műveleteken kívül megvalósítjuk a összeadás, kivonás, illetve összehasonlítás komponensenkénti vektor – vektor műveleteket. Ezek a műveletek szintén elemenként dolgozzák fel a bemenetként kapott vektorokat úgy, hogy az azonos indexű elemekkel végzik el az adott elemi műveleteket.

A komponensenkénti vektor – vektor műveletek megvalósításához a két operandus elemeit úgy kell felsoroljuk, hogy az azonos indexű elemeket egyszerre tudjuk feldolgozni.

Sűrű vektorok esetén ez egyszerűen kivitelezhető. Mivel mindkét vektor összes komponense ábrázolva van, így a következő módszert használhatjuk:

Algoritmus 7 Sűrű vektorok összefésülése

```
1: function COMBINE(v_1, v_2)
2:     for i from 0 to n − 1 do
3:         Process(i, v_1[i], v_2[i]);
4:     end for
5: end function
```

Ezt az algoritmust használva a komponensenkénti vektor – vektor műveleteket sűrű vektorok esetén $\Theta(n)$ lépésben el tudjuk végezni.

Sűrű – ritka esetben a következő módon fésülhetjük össze a sűrű és a ritka vektor elemeit szintén $\Theta(n)$ lépésben:

Algoritmus 8 Ritka és sűrű vektorok összefésülése

```
1: function COMBINE(v_1, v_2)
2:     i ← 0; j ← 0;
3:     while i < n and j < size(v_2) do
4:         if i < v_2[j].idx then
5:             Process(i, v_1[i], 0);
6:             i ← i + 1;
7:         else
8:             Process(i, v_1[i], v_2[j].value);
9:             i ← i + 1; j ← j + 1;
10:        end if
11:    end while
12:    while i < n do
13:        Process(i, v_1[i], 0);
14:        i ← i + 1;
15:    end while
16: end function
```

Ez a módszer alkalmazható ritka – sűrű vektorok esetén is az operandusok felcserélésével.

Sűrű – ritka esetben lehetőségünk van az összeadás, illetve kivonás műveleteket helyben elvégeznünk úgy, hogy a műveletigény $\Theta(\text{NNZ})$ lépésre csökken, mivel ilyenkor elegendő felsorolnunk a ritka vektor elemeit. A sűrű vektor elemeit a ritka vektor elemeinek indexe alapján érjük el és helyben módosítjuk.

Algoritmus 9 Ritka és sűrű vektorok összefésülése

```
1: function COMBINE(v_1, v_2)
2:     for i from 0 to size(v_1) − 1 do
3:         Process(v_1[i].idx, v_1[i].value, v_2[v_1[i].idx]);
4:     end for
5: end function
```

Ritka – sűrű esetben nem tudjuk alkalmazni ezt az algoritmust, mivel lehetséges, hogy a ritka vektorban nem ábrázolunk olyan komponenseket, amelyeket a művelet módosít, így pedig költséges beszúrás műveleteket kellene végeznünk.

Az algoritmus ritka – ritka esetben válik a legösszetettebbé. Ekkor a két vektor elemeit a következő algoritmus segítségével fésüljük össze:

Algoritmus 10 Ritka vektorok összefésülése

```
1: function COMBINE(v_1, v_2)
2:     i ← 0; j ← 0;
3:     while i < size(v_1) and j < size(v_2) do
```

```
        if v1[i].idx < v2[j].idx then
            Process(v1[i].idx, v1[i].value, 0);
            i ← i + 1;
        else if v1[i].idx > v2[j].idx then
            Process(v2[j].idx, 0, v2[j].value);
            j ← j + 1;
        else
            Process(v1[i].idx, v1[i].value, v2[j].value);
            i ← i + 1; j ← j + 1;
        end if
    end while
    while i < size(v1) do
        Process(v1[i].idx, v1[i].value, 0);
        i ← i + 1;
    end while
    while j < size(v2) do
        Process(v2[j].idx, 0, v2[j].value);
        j ← j + 1;
    end while
end function
```

Az összefésülés során azon indexekhez tartozó értékeket dolgozzuk fel, ahol vagy az egyik, vagy a másik vektor értéke nem 0, így a ritka – ritka összefésülés műveletigénye $\Theta(\text{NNZ}^{(v_1)} + \text{NNZ}^{(v_2)})$. Azonban az összefésülés elágazásai miatt, amennyiben az ábrázolt vektorok nem kellően ritkák, akkor könnyen előfordulhat, hogy az implementált ritka – ritka összefésülés futásideje rosszabb lesz a sűrű – sűrű összefésülés futásidejénél.

Művelet	**Metódus neve**	**csak sűrű**	**ritka, sűrű**	**csak ritka**
Összeadás	vct_add	$\Theta(n)$	$\Theta(n)$	$\Theta(\text{NNZ}^{(op_1)} + \text{NNZ}^{(op_2)})$
Kivonás	vct_sub			
Összehasonlítás	vct_eq, vct_neq			

3.2. táblázat. Koordinátánkénti vektor – vektor műveletek költsége

Művelet	**Metódus neve**	**sűrű – sűrű**	**sűrű – ritka**
Összeadás	vct_add	$\Theta(n)$	$\Theta(NNZ)$
Kivonás	vct_sub		

3.3. táblázat. Helyben elvégezhető koordinátánkénti vektor – vektor műveletek költsége

3.2.2. Mátrix műveletek

Legyen $A \in \mathbb{K}^{n \times m}$ mátrix. A komponensenként vektor műveletekhez hasonlóan mátrixok esetén is a következő komponensenkénti műveleteket szeretnénk megvalósítani: szorzás, osztás, egészosztás, illetve maradékszámítás.

Ezen műveletek végrehajtásához szintén elegendő a mátrix minden komponensével elvégezni az adott műveletet, ezért ezeknek a műveleteknek a műveletigénye arányos az ábrázolt elemek számával: sűrű mátrixok esetén $\Theta(nm)$, ritka mátrixok esetén $\Theta(\text{NNZ})$.

A ritka mátrixok esetén az alábbi algoritmus segítségével járhatjuk be a mátrix elemeit sorfolytonos sorrendben:

Algoritmus 11 Ritka mátrix elemeinek felsorolása

```
1: function ITERATE(A)
2:     for i from 0 to n − 1 do
3:         for j from A.row_ptr[i] to A.row_ptr[i + 1] − 1 do
4:             ...
5:         end for
6:     end for
7: end function
```

A műveleteket itt is kétféle módon végezhetjük el: helyben, ekkor a meglévő mátrix elemeit módosítjuk, illetve kiszámíthatjuk az eredményt egy külön mátrixba is. Az utóbbi esetben a művelet elvégzése előtt memóriafoglalást kell végeznünk, vagy már lefoglalt memóriát kell újrahasznosítanunk.

Az osztást a mátrixok esetében sem tudjuk helyben elvégezni, mivel megváltoztathatja az elemek típusát.

A komponensenkénti mátrix műveletekhez az **GeNuSys::LinAlg::Operations** osztály a következő függvényeket biztosítja.

Művelet	**Metódus neve**	**Műveletigény**		**Helyben**	
		Sűrű	**Ritka**	**Sűrű**	**Ritka**
szorzás	mat_mul	$\Theta(nm)$	$\Theta(\text{NNZ})$	$\Theta(nm)$	$\Theta(\text{NNZ})$
egészosztás	mat_idiv	$\Theta(nm)$	$\Theta(\text{NNZ})$	$\Theta(nm)$	$\Theta(\text{NNZ})$
osztás	mat_div	$\Theta(nm)$	$\Theta(\text{NNZ})$	—	—
maradék	mat_mod	$\Theta(nm)$	$\Theta(\text{NNZ})$	$\Theta(nm)$	$\Theta(\text{NNZ})$
szimmetrikus maradék	mat_mods	$\Theta(nm)$	$\Theta(\text{NNZ})$	$\Theta(nm)$	$\Theta(\text{NNZ})$

3.4. táblázat. Mátrix műveletek költsége – sűrű, illetve ritka operandusok esetén

A komponensenkénti mátrix – mátrix műveletek közül szintén az összeadást, kivonást, valamint az összehasonlításokat valósítjuk meg.

Kihasználva a mátrixok sorfolytonos ábrázolását ezen műveletek implementációja során a komponensenkénti vektor – vektor műveleteknél látott módszereket alkalmazzuk a mátrixok egyes soraira. Így a fent látott algoritmusokat egy az egyben alkalmazni tudjuk mátrixok esetén mind a helyben, mind a nem helyben végzett műveletek esetén is. Ebből adódóan a műveletigények lényegében a sorok számával arányosan nőnek.

Művelet	**Metódus neve**	**csak sűrű**	**ritka, sűrű**	**csak ritka**
Összeadás	vct_add	$\Theta(nm)$	$\Theta(nm)$	$\Theta(\text{NNZ}^{(op_1)} + \text{NNZ}^{(op_2)})$
Kivonás	vct_sub			
Összehasonlítás	vct_eq, vct_neq			

3.5. táblázat. Koordinátánkénti mátrix – mátrix műveletek költsége

Művelet	**Metódus neve**	**sűrű – sűrű**	**sűrű – ritka**
Összeadás	vct_add	$\Theta(nm)$	$\Theta(NNZ)$
Kivonás	vct_sub		

3.6. táblázat. Helyben elvégezhető koordinátánkénti mátrix – mátrix műveletek költsége

3.2.3. Szorzatok számítása

Az általunk megvalósított elemi lineáris algebrai műveletek közül a legköltségesebbek a vektor – vektor, mátrix – vektor, illetve mátrix – mátrix szorzatok számítása. Fontos, hogy ezeket a műveleteket a lehető leghatékonyabban tudjuk elvégezni.

A ritka ábrázolást használó adatszerkezetek esetén a klasszikus szorzási algoritmusok módosított változatait használhatjuk. Ritka ábrázolás esetén szeretnénk kihasználni ábrázolás előnyeit — például, hogy a ritka adatszerkezeteink nem ábrázolják a szorzások elvégzése szempontjából kihagyható komponenseket —, ugyanakkor figyelembe kell vennünk az adatszerkezet korlátait is.

A leghatékonyabban akkor használjuk a vektor, illetve mátrix adatszerkezeteinket, ha az elemeiket sorfolytonos sorrendben tudjuk a szorzat számításához felhasználni.

A szorzások elvégzése során az operandusok eleminek sorfolytonos sorrendben történő felhasználása vektor – vektor, illetve mátrix – vektor szorzások esetén könnyen kivitelezhető, de mátrix – mátrix szorzások esetén is adunk egy módszert, amely sorfolytonos sorrendben használja fel az elemeket.

3.2.3.1. Skaláris szorzás

Legyen $v_1, v_2 \in \mathbb{K}^n$. Szeretnénk a

$$\langle v_1, v_2 \rangle = \sum_{j=0}^{n-1} v_1[j] v_2[j]$$

skaláris szorzat értékének számítására hatékony implementációt adni. A skaláris szorzásokat a komponensenkénti vektor – vektor műveleteknél látott módszerek segítségével fogjuk elvégezni.

Sűrű vektorok skaláris szorzata esetén az azonos indexű komponensek szorozását $\Theta(n)$ lépésben megtehetjük a következő algoritmus felhasználásával.

Algoritmus 12 Sűrű vektorok skaláris szorzata

```
1: function MULTIPLY(v_1, v_2)
2:     result ← 0;
3:     for i from 0 to d − 1 do
4:         result ← result + v_1[i] × v_2[i];
5:     end for
6:     return result;
7: end function
```

Sűrű és ritka vektorok skaláris szorzása esetén kihasználhatjuk, hogy egy sűrű ábrázolású vektor tetszőleges elemének elérése $\mathcal{O}(1)$ műveletigényű, így a helyben végzett vektor – vektor összeadáshoz, illetve kivonáshoz hasonlóan a vektorok szorzását is $\Theta(\mathrm{NNZ})$ szorzással elvégezhetjük.

Algoritmus 13 Sűrű és ritka vektorok skaláris szorzata

```
1: function MULTIPLY(v_1, v_2)
2:     result ← 0;
3:     for i from 0 to size(v_1) − 1 do
4:         result ← result + v_1[v_2[i].idx] × v_2[i].value;
5:     end for
6:     return result;
7: end function
```

Ritka vektorok skaláris szorzása esetén ismét a komponensenkénti vektor – vektor műveleteknél látott összefésüléses módszert alkalmazzuk. Ebben az esetben elegendő csak azokat a komponenseket vizsgálnunk, amelyek indexei megegyeznek, más esetekben az adott komponensek szorzata nulla. Ezek alapján az algoritmust a következők szerint egyszerűsíthetjük, azonban a szorzat kiszámítása még így is $\mathcal{O}(\text{NNZ}^{(v_1)} + \text{NNZ}^{(v_2)})$ műveletet igényel.

Algoritmus 14 Ritka vektorok skaláris szorzata

```
1: function MULTIPLY(v_1, v_2)
2:     result ← 0;
3:     i ← 0; j ← 0;
4:     while i < size(v_1) and j < size(v_2) do
5:         if v_1[i].idx < v_2[j].idx then
6:             i ← i + 1;
7:         else if v_1[i].idx > v_2[j].idx then
8:             j ← j + 1;
9:         else
10:            result ← result + v_1[i].value × v_2[j].value;
11:            i ← i + 1; j ← j + 1;
12:        end if
13:    end while
14:    return result;
15: end function
```

A különböző esetek műveletigényét megvizsgálva láthatjuk, hogy a sűrű – ritka, illetve ritka – sűrű szorzás végezhető el a legkevesebb művelettel. Ezekben az esetekben tudjuk a legjobban kihasználni a két ábrázolás előnyeit: a sűrű ábrázolásban tetszőleges elemet $\mathcal{O}(1)$ művelettel elérünk, valamint a ritka ábrázolásban a szorzat szempontjából lényegtelen komponenseket kihagyjuk.

A skaláris szorzatok számításánál is fennáll, hogy ritka – ritka esetben az összefésülés elágazásai miatt, amennyiben az ábrázolt vektorok nem kellően ritkák, akkor könnyen előfordulhat, hogy az implementált ritka – ritka szorzás futásideje rosszabb lesz a sűrű – sűrű szorzás futásidejénél.

3.2.3.2. Mátrix – vektor szorzatok

Legyen $A \in \mathbb{K}^{n\times m}$, $x \in \mathbb{K}^m$. Mátrix – vektor szorzás esetén a $b := Ax \in \mathbb{K}^n$ szorzat értékét szeretnénk meghatározni.

Felhasználva, hogy

$$b[i] = \sum_{j=0}^{m-1} A[i,j]x[j], \quad (i = 0, \ldots, m-1)$$

az eredményt megkaphatjuk, ha a skaláris szorzást a mátrix minden sorával elvégezzük az vektor – vektor szorzatok számításánál ismertetett módokon.

Így a mátrixot elegendő egyszer sorfolytonosan, a vektort pedig a mátrix minden sorával együtt szintén folytonosan bejárnunk. Ez sűrű és ritka ábrázolás esetén is könnyen megtehető, így a mátrix – vektor szorzatok előállításának költsége a mátrix sorainak számával arányosan nő a skaláris szorzásnál látottakhoz képest.

A vektor – vektor szorzásokhoz hasonlóan a mátrix – vektor szorzás esetén is a sűrű – ritka, illetve ritka – sűrű szorzások számíthatóak a leghatékonyabbak a gyakorlatban.

3.2.3.3. Mátrix – mátrix szorzatok

Legyen $A \in \mathbb{K}^{n\times m}$, $B \in \mathbb{K}^{m\times k}$, mátrix – mátrix szorzások számításakor a $C := AB \in \mathbb{K}^{n\times k}$ szorzat értékét szeretnénk meghatározni.

A mátrix – vektor szorzatokhoz hasonlóan itt is kihasználhatjuk, hogy

$$B[i,j] = \sum_{l=0}^{m-1} A[i,l]B[l,j],$$

így a skaláris szorzatoknál látott módszereket mátrix – mátrix szorzások számítása esetén is felhasználhatjuk.

A szorzat kiszámításához a B mátrixot oszlopfolytonosan kell bejárnunk. Mivel a sűrű mátrixokat oszlopfolytonosan is hatékonyan be tudjuk járni, így a sűrű – sűrű, illetve ritka – sűrű esetekben közvetlenül használhatjuk a skaláris szorzatoknál látott algoritmusokat:

Az eredmény elemeit sorfolytonos sorrendben számítjuk ki. Az A mátrixban soronként haladunk, minden sorra elvégzünk egy-egy skaláris szorzást B összes oszlopvektorával. Ilyen módon, amennyiben A és B is sűrű, akkor a szorzás eredménye $\Theta(nmk)$ lépésben meghatározható. Amennyiben A ritka mátrix, B pedig sűrű, akkor a műveletigény $\Theta(NNZ^{(A)} \times k)$.

Amennyiben B ritka mátrix, vagyis oszlopfolytonosan nem tudjuk hatékonyan bejárni, akkor az eredmény elemenkénti kiszámítása helyett az eredmény sorait fogjuk kiszámítani. Ezt a módszer sűrű – ritka mátrixok szorzata esetén az alábbi $\Theta(n \times NNZ^{(B)})$ szorzás végrehajtását igénylő módon tudjuk megvalósítani.

Algoritmus 15 Sűrű és ritka mátrixok szorzata

```
1: function MULTIPLY(A, B)
2:     C ← ((0, ..., 0), ..., (0, ..., 0));
3:     for i from 0 to n − 1 do
4:         for j from 0 to m − 1 do
5:             for l from B.row_ptr[j] to B.row_ptr[j + 1] − 1 do
6:                 result[i, B[l].col_idx] ← result[i, B[l].col_idx] + A[i, j] × B[l].value;
7:             end for
8:         end for
9:     end for
10:    return result;
11: end function
```

Ezt az eljárást ritka – ritka mátrixok szorzása esetén is megvalósítható. Ekkor a műveletigény nagyban függ a mátrix nemnulla elemeinek elhelyezkedésétől.

3.2.3.4. Szorzások műveletigénye

A fenti eredményeket összefoglalva, a különböző szorzások elvégzéséhez ritka, illetve sűrű ábrázolás esetén kapott műveletigényeket a következő táblázat tartalmazza.

Szorzás	Metódus neve	csak sűrű	ritka, sűrű	csak sűrű
vektor – vektor	vct_mul	$\Theta(n)$	$\Theta(NNZ)$	$\mathcal{O}(\text{NNZ}^{(v_1)} + \text{NNZ}^{(v_2)})$
mátrix – vektor	mat_mul	$\Theta(nm)$	$\Theta(n \times NNZ^{(v)})$ $\Theta(NNZ^{(A)})$	$\mathcal{O}(\text{NNZ}^{(A)} + n \times \text{NNZ}^{(v)})$
mátrix – mátrix	mat_mul	$\Theta(nmk)$	$\Theta(n \times NNZ^{(B)})$ $\Theta(NNZ^{(A)} \times k)$	változó

3.7. táblázat. Szorzások műveletigénye

3.3. Normák

A könyvtár biztosítja a gyakran használt normák implementációját.

Egyes normákat példányosítás nélkül is szeretnénk használni, ilyen norma például a $\|.\|_2$ norma, vagy a $\|.\|_\infty$ norma. Más esetekben szükség van a norma példányosítására, például az konstruált operátornorma esetén, ahol a norma értékét befolyásolja a konstruált hasonlósági mátrix.

A különböző normák visszatérési értékének típusa szintén bemeneti típusonként változhat, ezért minden implementált norma tartalmaz egy **NormType template** osztályt, amely az egyes típusokhoz tartalmazza a visszatérési érték típusának definícióját.

```
1 struct Norm {
2   template<typename Type>
3   struct NormType {
4     // Visszatérési érték típusának definiálása
5   };
6 };
```

3.8. ábra. A **NormType** osztály

A normák interfészét úgy alakítjuk ki, hogy az egyes normák könnyen felcserélhetőek legyenek. A példányosítás nélkül használható normák rendelkeznek minden megfelelő vektor, illetve mátrix típushoz egy statikus **norm** függvénnyel.

```
1 template<typename Type>
2 static typename NormType<Type>::Value norm(...);
```

3.9. ábra. A **norm** függvény

A példányosítás utáni funktorként való használathoz elegendő túlterhelni a megfelelő operátorokat.

```
1 template<typename Type>
2 typename NormType<Type>::Value operator ()(...) const;
```

3.10. ábra. A normához tartozó operátor

Mivel a norma osztályok nem egy közös ősosztály leszármazottai, így azoknál az osztályoknál, illetve függvényeknél, ahol különböző normákat szeretnék használni, szükséges lehet a norma típusát **template** paraméterként kezelni.

3.3.1. p-normák

3.3.1. Definíció. (p-norma) Legyen $x \in \mathbb{K}^n$. Az x vektor p-normája

$$\|x\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}$$

Az $\|.\|_p$ vektornormához tartozó indukált mátrixnorma értéke egy $A \in \mathbb{K}^{n \times n}$ mátrixra

$$\|A\|_p := \sup_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p}.$$

A **GeNuSys::LinAlg::PNorm** osztály tetszőleges p értékre implementálja a p-vektornormát, illetve **template** specializációval hatékonyabb implementációt ad az $\|.\|_1$, $\|.\|_2$, illetve $\|.\|_\infty$ normák esetén, valamint ezen normák esetén biztosítja az indukált mátrixnormákat is. Az A mátrix normája ezekben az esetekben

$$\|A\|_1 = \max_j \sum_{i=1}^{n} |a_{ij}|, \qquad \|A\|_2 = \sqrt{\lambda_{\max}(A^H A)}, \qquad \|A\|_\infty = \max_i \sum_{j=1}^{n} |a_{ij}|,$$

ahol A^H az A mátrix konjugált transzponáltja, $\lambda_{\max}(A^H A)$ pedig az $A^H A$ mátrix legnagyobb sajátértéke.

A vektornormák esetén a norma értéke meghatározható a vektor elemeinek egyszeri bejárásával. Az $\|.\|_1$, valamint $\|.\|_\infty$ mátrixnorma esetén is hasonló a helyzet. A 2 mátrixnorma esetén az $A^H A$ mátrix sajátértékeinek meghatározására a mátrix Schur-felbontását használjuk.

A p-normák implementációját a **p_norm.h**, illetve **p_norm.hpp** fájlok tartalmazzák.

3.3.2. Frobenius-norma

3.3.2. Definíció. (Frobenius-norma) Legyen $A \in \mathbb{K}^{n \times n}$. Az A mátrix Frobenius-normája

$$\|A\|_F := \sum_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|^2.$$

A **GeNuSys::LinAlg::FrobeniusNorm** osztály implementálja a Frobenius-norma számítását tetszőleges mátrix esetén. Ahogy a norma definíciójából látható, a Frobenius-norma értéke könnyen meghatározható a mátrix elemeinek egyszeri bejárásával.

A Frobenius-norma implementációját a **frobenius_norm.h**, illetve **frobenius_norm.hpp** fájlok tartalmazzák.

3.3.3. Operátornorma

Az operátornorma implementációja során az 1.2.2. pontban leírtakat követjük.

Legyen $M \in \mathbb{K}^{n\times n}$ mátrix, elkészítjük M^{-1} Jordan-féle normálformáját, így kapjuk, hogy $M^{-1} = T^{-1}JT$. Amennyiben M^{-1} rendelkezik többszörös multiplicitású sajátértékkel, azaz J nem diagonális, akkor elkészítjük a D hasonlósági mátrixot, amelyre, $\|D^{-1}JD\|_\infty < 1$ teljesül. Ezeket a lépéseket elvégezve megkapjuk az $S := DT$ hasonlósági mátrixot.

Legyen $x \in \mathbb{K}^n$, valamint $A \in \mathbb{K}^{n\times n}$, ekkor

$$\|x\| = \|Sx\|_\infty, \quad \text{valamint} \quad \|A\| = \|SAS^{-1}\|_\infty$$

A **GeNuSys::LinAlg::OperatorNorm** osztály az operátornormát a szorzások elvégzése után a már implementált $\|.\|_\infty$ norma felhasználásával számolja.

Az operátornorma implementációját az **operator_norm.h**, illetve **operator_norm.hpp** fájlok tartalmazzák.

3.4. Megvalósított algoritmusok

Az általánosított számrendszerek kezeléséhez szükségünk van mátrixok inverzének, adjungáltjának és determinánsának számítására, a mátrix sajátértékeinek meghatározására, Jordan-féle, illetve Smith-normálformára hozására.

Mivel ezen érékeket egy adott (Λ, M, D) rendszer esetén csak egyszer kell kiszámítanunk, így az itt megvalósított algoritmusok hatékonysága lényegében nem befolyásolja a további algoritmusok futásidejét. Ennek ellenére igyekszünk lehetőleg hatékony megvalósításokat adni az egyes eljárásokhoz.

A **GeNuSys::LinAlg::Algorithms** osztály tartalmazza ezen lineáris algebrai algoritmusok megvalósítását. Az osztály implementációja a **linalg_algorithms.h**, illetve **linalg_algorithms.hpp** fájlokban található.

3.4.1. Determináns számítás

Legyen $A \in \mathbb{K}^{n \times n}$, valamint legyen $PA = LU$, az A mátrix PLU felbontása. Kihasználva, hogy

$$\det(P)\det(A) = \det(L)\det(U),$$

valamint, hogy P permutációs mátrix, így determinánsa ± 1, továbbá L alsóháromszög, míg U felsőháromszög mátrix, azaz a determinánsuk a főátlóbeli elemeik szorzata, egy könnyen elvégezhető módszert kapunk A determinánsának számítására.

Mivel a PLU felbontás során kapott L mátrix minden főátlóbeli eleme 1, azaz determinánsa is 1, így valójában L kiszámítására nincs is szükségünk.

A PLU felbontást részleges főelem-kiválasztással végezzük el, és mivel $\det(P) = \pm 1$ — attól függően, hogy hány sorcsere történt a PLU felbontás számítása közben —, így a determináns értékének számításához elegendő minden sorcsere után előjelet váltani.

Amennyiben a mátrix nem invertálható, azaz a determináns 0, akkor az algoritmus végrehajtása közben eljutunk egy olyan részleges főelem-kiválasztásig, ahol csak 0 választható főelemként. Ha eljutunk egy ilyen állapotba, akkor a számítást meg is állíthatjuk, a determináns értéke 0.

A determináns kiszámítása ezzel a módszerrel elvégezhető $\mathcal{O}(n^3)$ lépésben.

```
1 template<typename Type>
2 typename TypeTraits<Type>::RationalType det(const Matrix<Type>& mat);
```

3.11. ábra. A **det** függvény

3.4.2. Inverz számítás

Legyen $A \in \mathbb{K}^{n\times n}$, amennyiben létezik, az A mátrix inverzének előállítása megtehető a Gauss elimináció egyik változata, a Gauss-Jordan elimináció segítségével.

Vegyük az

$$\left[\begin{array}{c|c} A & I \end{array}\right]$$

mátrixot. Amennyiben A invertálható, részleges főlemkiválasztást alkalmazva, az eliminációt elvégezve, előállítjuk az

$$\left[\begin{array}{c|c} I & B \end{array}\right]$$

mátrixot. Ekkor a kapott $B \in \mathbb{K}_Q^{n\times n}$ mátrix az A mátrix inverze.

Így az inverz kiszámítás szintén elvégezhető $\mathcal{O}(n^3)$ lépésben.

```
1 template<typename Type>
2 Matrix<typename TypeTraits<Type>::RationalType> invert(const Matrix<Type>&
    mat);
```

3.12. ábra. Az **invert** függvény

3.4.3. Adjungált számítás

3.4.1. Definíció. (Adjungált) Az $A \in \mathbb{K}^{n\times n}$ mátrix adjungáltjának nevezzük az előjeles aldeterminánsaiból alkotott mátrix transzponáltját. A adjungáltjának jelölése A^* vagy adj(A).

3.4.2. *Megjegyzés.* Ha $A \in \mathbb{K}^{n\times n}$, akkor $A^* \in \mathbb{K}^{n\times n}$.

Amennyiben a definíció alapján számolnánk az adjungált értékét, akkor n^2 darab $\mathbb{K}^{(n-1)\times(n-1)}$ mátrix determinánsának kiszámítását kellene elvégeznünk, amely $\mathcal{O}(n^2 \times n^3)$ lépésben végezhető el.

3.4.3. Állítás. Legyen $A \in \mathbb{K}^{n\times n}$ invertálható mátrix. A mátrix inverze és adjungáltja közt az alábbi kapcsolat áll fenn:

$$A^{-1} = \frac{A^*}{\det A}$$

Felhasználva ezt az állítást kapjuk, hogy $A^* = \det(A)A^{-1}$. Így az adjungáltat az egyes aldeterminánsok kiszámítása nélkül, a determináns, illetve az inverz felhasználásával $\mathcal{O}(n^3)$ lépésben elő tudjuk állítani.

```
1 template<typename Type>
2 Matrix<Type> getAdjoint(const Matrix<Type>& mat);
```

3.13. ábra. A **getAdjoint** függvény

3.5. Schur-felbontás és Jordan-féle normálforma

Az operátornorma konstrukciójához szükségünk van invertálható mátrixok Jordan-féle normálformájának előállítására.

Mivel az operátornorma konstrukcióját egy adott (Λ, M, D) rendszer esetén csak egyszer kell elvégeznünk, így a Jordan-féle normálforma előállításának hatékonysága lényegében nem befolyásolja az algoritmusaink futásidejét. Ennek ellenére igyekszünk egy megfelelően hatékony módszert megvalósítani.

A Jordan-féle normálforma előállításához szükségünk van a mátrix sajátértékeinek meghatározására. A sajátértékek meghatározásához a mátrix Schur-felbontását fogjuk használni. A Schur-felbontás előállítását pedig a QR-algoritmus segítségével végezzük, amelyhez szükségünk van mátrixok QR felbontásának számolására.

3.5.1. QR felbontás

3.5.1. Állítás. (QR felbontás) Legyen $A \in \mathbb{K}^{n\times n}$. Az A mátrix QR felbontása során egy $Q \in \mathbb{K}_R^{n\times n}$ ortogonális és egy $R \in \mathbb{K}_R^{n\times n}$ felsőháromszög mátrixot állítunk elő úgy, hogy $A = QR$ teljesüljön.

Az A mátrix QR felbontása megvalósítható Gram-Schmidt ortogonalizációval, illetve Householder transzformációkkal is. A könyvtárban implementált algoritmus Householder transzformációkat alkalmaz.

A Householder transzformációval végzett QR felbontás alapötlete, hogy olyan vetítéseket keresünk, amelyek az adott mátrix első oszlopát úgy vetítik, hogy az oszlopnak csak az első komponense ne legyen 0, azaz egy olyan ortogonális Q_1 mátrixot konstruálunk, amelyre

$$Q_1 A = \begin{bmatrix} \alpha & * & \dots & * \\ 0 & & & \\ \vdots & & A_2 & \\ 0 & & & \end{bmatrix}$$

Legyen x az A mátrix első oszlopvektora,

$$\alpha = -\text{sgn}(x_1)\|x\|, \quad u = x - \alpha e_1, \quad \text{valamint} \quad v = \frac{u}{\|u\|}.$$

Ekkor a $Q_1 = I - 2vv^T$ mátrix megfelelő transzformáció, mivel Q_1 ortogonális mátrix, valamint

$$Q_1 x = (\alpha, 0, \dots, 0)\,.$$

Miután a $Q_1 A$ szorzást elvégeztünk, az módszert a következő lépésben az A_2 mátrixra folytatva $n-1$ lépésben felsőháromszög alakra hozható a mátrix.

Az algoritmus k. lépésében kapott $Q_k \in \mathbb{K}_R^{(n-k+1)\times(n-k+1)}$ mátrixot az alábbi módon kiegészítve kapott

$$Q'_k = \begin{bmatrix} I_{k-1} & 0 \\ 0 & Q_k \end{bmatrix}$$

mátrixok szorzatából $Q = Q'_{n-1}Q'_{n-2}\dots Q'_1$, melyre $QA = R$, azaz $A = Q^T R$.

```
template<typename Type>
QR<Type> decomposeQR(const Matrix<Type>& mat);
```

3.14. ábra. A **decomposeQR** függvény

3.5.2. Felső Hessenberg-alakra hozás

3.5.2. Definíció. (Felső Hessenberg-alak) Azt mondjuk, hogy az $A \in \mathbb{K}^{n\times n}$ mátrix felső Hessenberg-alakú, amennyiben $i > j+1$ esetén $a_{ij} = 0$, azaz a mátrix

$$A = \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ & * & * & * \\ & & * & * \end{pmatrix}$$

alakú.

3.5.3. Állítás. Legyen $A \in \mathbb{K}^{n\times n}$, az A mátrix felső Hessenberg-alakra hozása során egy $H \in \mathbb{K}_R^{n\times n}$ felső Hessenberg-alakú, valamint egy $Q \in \mathbb{K}_R^{n\times n}$ ortogonális hasonlósági mátrixot állítunk elő úgy, hogy $A = QHQ^{-1}$ teljesüljön.

A felső Hessenberg-alakra hozáshoz a QR felbontás elvégzésénél látotthoz hasonló módon szintén Householder transzformációkat alkalmazunk.

Legyen Q_1 az $(a_{21}, a_{31}, \dots, a_{d1})$ oszlopvektorból képzett Householder transzformáció, valamint legyen

$$Q'_1 = \begin{bmatrix} I_1 & 0 \\ 0 & Q_1 \end{bmatrix}.$$

A szorzást elvégezve kapott $Q'_1 A$ mátrix első oszlopának utolsó $n-2$ eleme így már 0 lesz, azaz

$$Q'_1 A = \begin{bmatrix} * & * & \dots & * & * \\ * & & & & \\ 0 & & A_2 & & \\ \vdots & & & & \\ 0 & & & & \end{bmatrix}$$

Továbbá mivel a Q'_1 transzponáltjával jobbról való szorzás nem változtatja meg az A mátrix első oszlopát, így a $Q'_1 A Q'^T_1$ szorzás elvégzése után A első oszlopa továbbra is számunkra megfelelő lesz. Ezt az eljárást megismételve a módszert az A_2 mátrixszal folytathatjuk tovább.

Az eljárás $n-2$ lépésben véget ér, és a kapott Q'_k mátrixok szorzatából

$$Q = Q'_{n-2} Q'_{n-3} \dots Q'_1, \quad \text{melyre} \quad QAQ^T \text{ felső Hessenberg-alakú.}$$

```
template<typename Type>
HessenbergForm<Type> getHessenbergForm(const Matrix<Type>& mat);
```

3.15. ábra. A **getHessenbergForm** függvény

3.5.3. Schur-felbontás

3.5.4. Definíció. (Schur-felbontás) Legyen $A \in \mathbb{K}^{n\times n}$, akkor az A mátrix felírható $A = QUQ^{-1}$ alakban, ahol $U \in \mathbb{K}_C^{n\times n}$ felsőháromszög mátrix, $Q \in \mathbb{K}_C^{n\times n}$ pedig ortogonális. Az U mátrixot A Schur formájának nevezzük.

3.5.5. *Megjegyzés.* Mivel A és U hasonló mátrixok, valamint U felsőháromszög mátrix, így U főátlójában A sajátértékei szerepelnek.

Az A mátrix Schur-felbontását előállíthatjuk a QR-algoritmus segítségével. A könyvtárban a QR-algoritmus explicit eltolásos változatát valósítottuk meg.

Az algoritmus első lépéseként a mátrixot felső Hessenberg-alakra hozzunk, így is gyorsítva az algoritmust.

Algoritmus 16 QR-algoritmus

1: **function** QRALGORITHM(A)
2: $A_1 \leftarrow$ HESSENBERGFORM(A);
3: **for** $i = 1, 2, \dots$ **do**
4: Válasszunk egy megfelelő δ_i eltolást.
5: $(Q_i, R_i) \leftarrow$ QRDECOMPOSE($A_i - \delta_i I$);
6: $A_{i+1} \leftarrow R_i Q_i + \delta_i I$;
7: **end for**
8: **end function**

```
template<typename Type>
SchurForm<Type> getSchurForm(const Matrix<Type>& mat);
```

3.16. ábra. A **getSchurForm** függvény

3.5.4. Jordan-féle normálforma

Egy tetszőleges $A \in \mathbb{C}^{n\times n}$ mátrix nem feltétlenül diagonalizálható, azonban Jordan-féle normálformával minden négyzetes mátrix rendelkezik.

3.5.6. Definíció. (Jordan-féle normálforma) Legyen $A \in \mathbb{C}^{n\times n}$. Az A mátrix hasonló egy J blokkdiagonális mátrixhoz úgy, hogy

$$J = \begin{bmatrix} J_1 & & \\ & \ddots & \\ & & J_k \end{bmatrix}, \quad \text{ahol} \quad J_i = \begin{pmatrix} \lambda_i & 1 & & \\ & \lambda_i & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_i \end{pmatrix}.$$

A J mátrix főátlójában lévő J_i mátrixokat Jordan-blokkoknak nevezzük.

Legyen λ az A mátrix sajátértéke m algebrai multiplicitással. Amennyiben λ geometriai multiplicitása kisebb mint m, akkor az $(A-\lambda I)v = 0$ egyenlet megoldásaként nem kapunk m lineárisan független sajátvektort.

3.5.7. Definíció. (Általánosított sajátvektor) Legyen $A \in \mathbb{C}^{n\times n}$. A $0 \neq v \in \mathbb{C}^n$ vektor az A mátrix λ sajátértékéhez tartozó k-ad rangú általánosított sajátvektora, amennyiben

$$(A-\lambda I)^k v = 0, \text{ de } (A-\lambda I)^{k-1} v \neq 0.$$

3.5.8. *Megjegyzés.* Belátható, hogy a λ sajátvektorhoz pontosan m darab lineárisan független általánosított sajátvektor létezik.

Az $A \in \mathbb{C}^{n\times n}$ mátrix Jordan-féle normálformáját a következőkben ismertetett módon állíthatjuk elő.

Legyenek λ az A mátrix sajátértéke, továbbá legyen λ algebrai multiplicitása m.

Állítsuk elő λ általánosított sajátvektorait. Legyen m_i a λ sajátértékhez tartozó i-ed rendű, lineárisan független általánosított sajátvektorok száma. Ahogy a lineárisan független általánosított sajátvektorok száma nő $m_1 \leq m_2 \leq \ldots m_N = m$, vagyis előbb–utóbb előállítjuk az összes lehetséges sajátvektort. Ekkor a λ sajátértékhez tartozó legalább $j \times j$ méretű Jordan-blokkok száma s_j lesz, ahol $s_1 = m_1$, valamint $s_i = m_i - m_{i-1}$, amennyiben $i > 1$. Így a pontosan $j \times j$ méretű Jordan-blokkok száma d_j lesz, ahol $d_N = s_N$, valamint $d_i = s_i - s_{i+1}$, ha $i < N$.

A Jordan-féle normálforma előállításához ilyen módon A összes sajátérékéhez meghatározzuk a Jordan-blokkok méretét és számát. A kapott blokkdiagonális mátrixban méret szerint növekvő sorrendben helyezzük el az adott sajátértékhez tartozó Jordan-blokkokat.

Amennyiben elő szeretnénk állítani a $P \in \mathbb{C}^{n\times n}$ hasonlósági mátrixot, melyre $A = P^{-1}JP$, akkor a következőket kell tennünk.

Legyenek a λ sajátértékhez tartozó Jordan-blokk méretek csökkenő sorrendben $S_1 \geq S_2 \geq \cdots \geq S_k$. Ilyen sorrendben vesszük sorra a blokkokat. Az i. blokkhoz választunk egy $v_{i,1}$ S_i-ed

rangú általánosított sajátvektort, amely lineárisan független az előtte előállított $v_{j,l}$ vektroktól. Ezen $v_{i,1}$ vektor mellé vesszük a $v_{i,2}, \ldots, v_{i,S_i}$ vektorokat, úgy hogy $v_{i,j} = (A - \lambda I)^{j-1} v_{i,1}$.

Ezt a folyamatot elvégezzük az összes blokkra, így megkapjuk a P hasonlósági mátrix λ sajátértékhez tartozó részét a P_λ mátrixot, ahol

$$P_\lambda = \left(v_{k,N_k}, v_{k,N_k-1}, \ldots, v_{1,1}\right).$$

A különböző sajátértékekhez tartozó P_λ mátrixok egymás mellé illesztésével megkapjuk a P hasonlósági mátrixot.

```
1 template<typename Type>
2 JordanForm<Type> getJordanForm(const Matrix<Type>& mat);
```

3.17. ábra. A **getJordanForm** függvény

4. fejezet

Általánosított számrendszerek

A dolgozat fő célja az általánosított számrendszerekhez kapcsolódó adatszerkezetek és algoritmusok hatékony megvalósítása volt. A következőkben ezen általánosított számrendszerek vizsgálatához kapcsolódó konstrukciók implementációjáról lesz szó.

Az egyes algoritmusok, illetve adatszerkezetek megvalósítása során az 1. fejezetben kapott eredményeket használjuk fel. A jelöléseink is megegyeznek az ott használt jelölésekkel.

Az 1.1.10. tétel következményeként kaptuk, hogy tetszőleges radix rendszer helyett vizsgálhatunk egy vele ekvivalens $(\mathbb{Z}^n, M, D)$ rendszert, ahol $M : \mathbb{Z}^n \to \mathbb{Z}^n$, valamint $D \subset \mathbb{Z}^n$.

Ezt kihasználva a következőkben az általánosság megsértése nélkül $(\mathbb{Z}^n, M, D)$ rendszereket vizsgálunk. A továbbiakban legyen $\Lambda = \mathbb{Z}^n$, $M : \mathbb{Z}^n \to \mathbb{Z}^n$, valamint $D \subset \mathbb{Z}^n$.

Az általánosított számrendszerek vizsgálatához kapcsolódó adatszerkezeteket és algoritmusokat megvalósított osztályok a **GeNuSys::NumSys** névtében találhatóak meg.

Mivel a legtöbb megvalósított algoritmus során több, az M mátrixból származtatott értéket használunk, ezért az értékek többszöri kiszámításának elkerülése érdekében egy adott M mátrixhoz, az M mátrixszal végzett további számítások előtt, ezen értékeket előre kiszámítjuk és tároljuk.

A **GeNuSys::NumSys::RadixProperites** osztály egy adott M mátrixhoz tárolja

- a mátrix inverzét $\left(M^{-1}\right)$,
- a mátrix adjungáltját (M^*),
- a mátrix determinánsát $(\det M)$,
- a mátrix determinánsának abszolútértékét $(|\det M|)$,
- a mátrix Smith-normálformáját $(UMV = S)$,
- a mátrixhoz konstruált operátornormát $(\|.\|)$.

4.1. Jegyrendszerek konstrukciója

4.1.1. Definíció. Egy $D \subset \mathbb{Z}^n$ *modulo* M maradékrendszert *jegyrendszer*nek nevezünk, amennyiben teljes maradékrendszer.

A következőkben ilyen speciális $D \subset \mathbb{Z}^n$ jegyrendszerekre adunk konstrukciókat különböző szempontok alapján. Az itt bemutatott jegyrendszerek előállításának implementációját a **GeNuSys::NumSys::DigitSet** osztály tartalmazza.

4.1.1. A j-kanonikus és j-szimmetrikus jegyrendszer

Legyen

$$D_M^{(j)} = \{ie_j : i = 0, \ldots, t-1\},$$

ahol e_j a j. egységvektor.

4.1.2. Definíció. (j-kanonikus jegyrendszer) Amennyiben $D_M^{(j)}$ teljes *modulo* M maradékrendszert alkot, akkor $D_M^{(j)}$ *j-kanonikus jegyrendszer*.

4.1.3. Definíció. (j-kanonikus számrendszer) Ha valamely j esetén $(\mathbb{Z}^n, M, D_M^{(j)})$ számrendszer, akkor $(\mathbb{Z}^n, M, D_M^{(j)})$ *j-kanonikus számrendszer*.

A j-szimmetrikus jegyrendszert a j-kanonikus jegyrendszerhez hasonlóan definiáljuk. Az egyetlen különbség, hogy a jegyrendszer elemeinek komponensei a *modulo* t szimmetrikus maradékrendszerből kerülnek ki, azaz

$$D_M'^{(j)} = \left\{ie_j : i = \left\lfloor -\frac{t}{2} \right\rfloor + 1, \ldots, \left\lfloor \frac{t}{2} \right\rfloor\right\}$$

4.1.4. Definíció. (j-szimmetrikus jegyrendszer) Amennyiben $D_M'^{(j)}$ teljes *modulo* M maradékrendszert alkot, akkor $D_M^{(j)}$ *j-szimmetrikus jegyrendszer*.

4.1.5. Definíció. (j-szimmetrikus számrendszer) Ha valamely j esetén $(\mathbb{Z}^n, M, D_M'^{(j)})$ számrendszer, akkor $(\mathbb{Z}^n, M, D_M'^{(j)})$ *j-szimmetrikus számrendszer*.

Mivel a j-kanonikus, illetve j-szimmetrikus jegyrendszerek esetén nem garantált, hogy tetszőleges j esetén teljes *modulo* M maradékrendszert kapunk, így szükséges ezen tulajdonság ellenőrzése. Erre a legegyszerűbb modszer, ha a 4.3.7. pontban definiált hasítófüggvényt használjuk. Amennyiben a jegyrendszer elemeire páronként különböző értéket kapunk, akkor a jegyrendszer teljes maradékrendszert alkot.

A j-kanonikus, illetve j-szimmetrikus jegyrendszerek elemei ritka vektorok, így az ilyen jegyrendszerek használatának előnye, hogy ezt a tulajdonságot kihasználva a jegyekkel végzett műveleteket jóval kevesebb lépésben el tudjuk végezni, mint tetszőleges jegyrendszerek esetén.

```
1 template<typename Type>
2 std::vector<GeNuSys::LinAlg::SparseVector<Type> > getJCanonical(const
    RadixProperties<Type>& props, unsigned int j);
3 template<typename Type>
4 std::vector<GeNuSys::LinAlg::SparseVector<Type> > getJSymmetric(const
    RadixProperties<Type>& props, unsigned int j);
```

4.1. ábra. A **getJCanonical** és **getJSymmetric** függvény

4.1.2. Adjungált jegyrendszer

Vegyünk egy tetszőleges $D = \{a_1, \dots, a_t\} \subset \mathbb{Z}^n$ teljes *modulo* M maradékrendszert.

A $-H$ halmaz lefedésének előállításakor láthattuk, hogy a lefedés méretét a $M^{-j}d$ vektorok ($d \in D$, $j = 1, 2, \dots$) komponenseinek abszolútértéke határozza meg, azaz amennyiben ezen vektorok $\|.\|_\infty$ normája kicsi, a lefedés mérete is kisebb lesz. Ezért az adjungált jegyrendszer előállításakor az $M^{-1}d$ vektorok $\|.\|_\infty$ normáját minimalizáljuk.

Használjuk ki, hogy $M^* = (\det M)M^{-1}$ és állítsuk elő az

$$M^*D \pmod{(\det M)I} = \{b_1, \dots, b_t\}$$

halmazt úgy, hogy $b_i \equiv M^*a_i \pmod{(\det M)I}$ teljesüljön, valamint b_i komponensei a *modulo* $|\det M|$ szimmetrikus maradékrendszerből kerüljenek ki. Mivel a komponenseket szimmetrikus maradékrendszerből vesszük, így M^*D elemeinek $\|.\|_\infty$ normája minimális lesz, ezért $M^{-1}D$ elemeinek $\|.\|_\infty$ normája szintén minimális lesz. Ekkor a

$$D_{\text{Adj}} := \left\{ \frac{Mb_1}{|\det M|}, \dots, \frac{Mb_t}{|\det M|} \right\}$$

halmaz ismét teljes *modulo* M maradékrendszer.

4.1.6. Definíció. (Adjungált jegyrendszer) A fenti módon konstruált D_{Adj} teljes *modulo* M maradékrendszert *adjungált jegyrendszer*nek nevezzük.

```
1 template<typename Type>
2 std::vector<GeNuSys::LinAlg::Vector<Type> > getAdjoint(const
    RadixProperties<Type>& props);
```

4.2. ábra. A **getAdjoint** függvény

4.1.3. Sűrű jegyrendszer

Az 1.2.11. következmény szerint a periodikus pontok az operátornorma szerinti

$$L := \frac{\|M^{-1}\|}{1 - \|M^{-1}\|} \max_{a_i \in D} \|a_i\|$$

sugarú, origó középpontú gömbben vannak. Ha M adott, vagyis

$$\frac{\|M^{-1}\|}{1 - \|M^{-1}\|}$$

állandó, akkor a gömb sugarát úgy csökkenthetjük, ha minimalizáljuk $\max_{a_i \in D} \|a_i\|$ értékét.

4.1.7. Definíció. (Sűrű jegyrendszerek) Azon $D \subset \mathbb{Z}^n$ jegyrendszereket nevezzük *sűrű jegyrendszerek*nek, amelyekre minden $a_i \in D$ jegy normája minimális, így $\max_{a_i \in D} \|a_i\|$ is minimális.

```
1 template<typename Type>
2 std::vector<GeNuSys::LinAlg::Vector<Type> > getDense(const
    RadixProperties<Type>& props);
```

4.3. ábra. A **getDense** függvény

A sűrű jegyrendszer előállítása a következő módon történik. Amennyiben $z_1, z_2 \in \mathbb{Z}^n$, akkor

$$z_1 \equiv z_2 \pmod{M} \iff M^* z_1 \equiv M^* z_2 \pmod{tI}.$$

Mivel

$$M^* z_1 \equiv M^* z_1 + te_j \pmod{tI},\ (j = 1, \ldots, n),$$

így

$$z_1 \equiv z_1 + \frac{Mte_j}{t} \equiv z_1 + Me_j \pmod{M},\ (j = 1, \ldots, n).$$

Ez azt jelenti, hogy ha a jegyrendszer egy elemét Me_j $(j = 1, \ldots, n)$ értékével növeljük, vagy csökkentjük, akkor az adott elem azonos maradékosztályban marad, valamint ilyen módosítások végrehajtásával a maradékosztály összes elemét megkaphatjuk.

A következő algoritmus segítségével minimalizálhatjuk a jegyrendszer elemeinek operátornormáját. A megfigyelésünk az, hogy az adjungált jegyrendszer elemei közel vannak egy sűrű jegyrendszer elemeihez, így a sűrű jegyrendszert az adjungált jegyrendszerből kiindulva állítjuk elő.

Algoritmus 17 Jegyrendszer operátornormájának minimalizálása

```
1: function ConstructDense(M)
2:     D ← ConstructAdjoint(M);
3:     repeat
4:         improved ← FALSE;
5:         for all a ∈ D do
6:             for k ← 1 to n do
7:                 start ← a;
8:                 diff ← Me_k;
9:                 repeat
10:                    oldNorm ← ‖a‖;
11:                    a ← a + diff;
12:                until oldNorm ≥ ‖a‖
13:                a ← a − diff;
14:                repeat
15:                    oldNorm ← ‖a‖;
16:                    a ← a − diff;
17:                until oldNorm ≥ ‖a‖
18:                a ← a + diff;
19:                if a ≠ start then
20:                    improved ← TRUE;
21:                end if
22:            end for
23:        end for
24:    until improved ≠ TRUE
25:    return D;
26: end function
```

4.2. A $-H$ halmaz lefedésének javítása

Bár a $-H$ lefedésének mérete egy adott $(\mathbb{Z}^n, M, D)$ rendszer esetén állandó, hasonlósági transzformációk alkalmazásával kereshetünk egy olyan $(\mathbb{Z}^n, M', D')$ rendszert, amely ekvivalens a $(\mathbb{Z}^n, M, D)$ rendszerrel, de a hozzá tartozó lefedés mérete jelentősen kisebb. Szemléletesen a $-H$ halmazt szeretnénk úgy forgatni, hogy az egyes tengelyeken minél kisebb vetülettel rendelkezzen.

Bár az optimum meghatározására nincsen jó módszerünk, a következőkben ismertetett genetikus algoritmussal is nagyságrendekkel csökkenthető a lefedés mérete.

Legyen $T \in \mathbb{Z}^{n\times n}$ felsőháromszög mátrix, ahol minden főátlóbeli elem 1.

$$T = \begin{pmatrix} 1 & * & * & \cdots & * \\ & 1 & * & \cdots & * \\ & & 1 & \cdots & * \\ & & & \ddots & \vdots \\ & & & & 1 \end{pmatrix}$$

Az algoritmus ilyen T hasonlósági mátrixok főátló feletti elemeinek véletlenített módosításával próbálja csökkenteni a lefedés méretét. Mivel a főátlót, illetve a mátrix alsóháromszög részét nem módosítjuk, így ezen T mátrixok végig unimodulárisak maradnak, ebből adódóan $T\mathbb{Z}^n = \mathbb{Z}^n$ minden esetben fennáll. Az alábbiakban a $(T\mathbb{Z}^n, TMT^{-1}, TD)$ rendszerhez tartozó lefedés méretét jelölje $\mathrm{Vol}(T)$.

Az algoritmus az egységmátrixból indul ki és a legjobb CandNum darab jelöltet tartja számon. Egy körön belül az összes jelölt méretét megpróbálja javítani egyenként MutateNum véletlenszerűen választott pozíció módosításával. A következő körben a legjobb CandNum javított méretű jelöltel folytatja a próbálkozást.

Az algoritmus akkor fog megállni, ha az utolsó NoImprLimit számú körben a minimális méret nem javult. Az implementációba beépíthetünk további megállási feltételeket. Korlátozhatjuk a körök számát, esetlegesen megadhatunk egy cél méretet, amelyet ha a legjobb jelölt elér, az algoritmus megáll.

Algoritmus 18 Lefedés méretének csökkentése

```
1: function FindTransformation(M, D, CandNum, MutateNum, NoImptLimit)
2:     Candidates ← [I, ..., I];
3:     NoImpr ← 0;
4:     while NoImpr < NoImprLimit do
5:         NewCandidates ← [ ];
6:         for all T ∈ Candidates do
7:             for k ← 1 to MutateNum do
8:                 Legyen (i, j) véletlenszerűen választott főátló fölötti pozíció.
9:                 U ← T;
10:                while Vol(incr(U, i, j)) < Vol(U) do
11:                    U ← incr(U, i, j);
12:                end while
13:                NewCandidates ← NewCandidates + [U];
14:                V ← T;
15:                while Vol(decr(V, i, j)) < Vol(V) do
16:                    V ← decr(V, i, j);
17:                end while
18:                NewCandidates ← NewCandidates + [V];
19:            end for
20:        end for
21:        Rendezzük NewCandidates elemeit növekvő sorrendbe méret szerint.
22:        if NewCandidates[1] < Candidates[1] then
23:            NoImpr ← 0;
24:        else
25:            NoImpr ← NoImpr + 1;
26:        end if
27:        Másoljuk NewCandidates legjobb CandNum darab elemét a Candidates tömbbe.
28:    end while
29:    return Candidates[1];
30: end function
```

4.3. A Φ függvény számítása

Az általánosított számrendszerekhez kapcsolódó algoritmusok megvalósítása során gyakran van szükségünk egy adott pont pályájának meghatározására. A az eldöntési és osztályozási feladatok megoldásához is szükségünk van a Φ függvény ismételt számítására, ezért egy olyan módszert szeretnénk megadni, amellyel Φ értéke hatékonyan, lehetőleg minél kevesebb számítás elvégzésével megkapható legyen.

Az 1.1.10. tétel következményeként kaptuk, hogy tetszőleges radix rendszer helyett vizsgálhatunk egy vele ekvivalens $(\mathbb{Z}^n, M, D)$ rendszert, ahol $M : \mathbb{Z}^n \to \mathbb{Z}^n$, valamint $D \subset \mathbb{Z}^n$. Így az általánosság megsértése nélkül a Φ függvény számítását csak $(\mathbb{Z}^n, M, D)$ rendszerek felett vizsgáljuk.

Az 1.2.1. pontban az alábbi módon definiáltuk a Φ függvényt.

Definíció. (Φ függvény) Legyen (Λ, M, D) radix rendszer. Definiáljuk a $\Phi : \Lambda \to \Lambda$ függvényt a következő módon

$$\Phi(z) = M^{-1}(z - a_i),$$

ahol $a_i \in D$ és $a_i \equiv z \pmod{M}$.

4.3.1. *Megjegyzés.* Ha $z_1, z_2 \in \mathbb{Z}^n$, valamint $M : \mathbb{Z}^n \to \mathbb{Z}^n$, akkor

$$z_1 \equiv z_2 \pmod{M} \iff M^{-1}(z_1 - z_2) \in \mathbb{Z}^n.$$

A továbbiakban legyen $t = |\det M|$, valamint $D = \{a_1, \ldots, a_t\}$ teljes *modulo* M maradékrendszer $\mathbb{Z}^n$ felett.

Egy tetszőleges $z \in \mathbb{Z}^n$ elem esetén $\Phi(z)$ számításához meg kell határoznunk melyik $a_i \in D$ számjegyre teljesül $a_i \equiv z \pmod{M}$. Az alábbiakban két módszert ismertetünk, amellyel meghatározhatjuk, hogy egy adott $z \in \mathbb{Z}^n$ elem melyik maradékosztályba tartozik.

4.3.1. Adjungált módszer

4.3.2. Tétel. Legyen $M \in \mathbb{Z}^{n\times n}$ invertálható mátrix, valamint legyenek $z_1, z_2 \in \mathbb{Z}^n$.

$$z_1 \equiv z_2 \pmod{M} \iff \forall i \in \{1, \ldots, n\} : (M^* z_1)_i \equiv (M^* z_2)_i \pmod{t}$$

Bizonyítás: Felhasználva az invertálható mátrix adjungáltjára vonatkozó állítást

$$M^{-1}(z_1 - z_2) \in \mathbb{Z}^n \iff \frac{M^*}{\det M}(z_1 - z_2) \in \mathbb{Z}^n \iff (tI)^{-1}(M^* z_1 - M^* z_2) \in \mathbb{Z}^n$$

vagyis $z_1 \equiv z_2 \pmod{M}$ akkor és csak akkor teljesül, ha $M^* z_1 \equiv M^* z_2 \pmod{tI}$. □

Számítsuk ki a következő halmazt, legyen

$$D_A = M^* D \pmod{tI} = \{b_1, \ldots, b_t\},$$

ahol $b_i \equiv M^* a_i \pmod{tI}$ úgy, hogy b_i egyes komponensei 0 és $t-1$ közötti értéket vesznek fel. Mivel D teljes maradékrendszer, így D_A minden $z \in \mathbb{Z}^n$ ponthoz tartalmaz egy megfelelő b_j elemet, melyre $M^* z \equiv b_j \pmod{tI}$.

Ahhoz, hogy egy tetszőleges $z \in \mathbb{Z}^n$ elemről eldöntsük, hogy melyik $a_i \in D$ számjeggyel kongruens *modulo* M, elegendő ellenőriznünk, hogy $M^* z$ melyik $b_i \in D_A$ elemmel egyezik meg komponensenként *modulo* t.

Ehhez el kell végeznünk egy szorzást M adjungáltjával, valamint t darab vektor összehasonlítást. Az M mátrix Smith-normálformája segítségével egy hatékonyabb módszert konstruálhatunk.

4.3.2. Smith-normálforma

4.3.3. Definíció. (Smith-normálforma) Ha $M \in \mathbb{Z}^{n \times n}$, akkor létezik olyan

$$UMV = S$$

felbontás, ahol $U, V \in \mathbb{Z}^{n \times n}$ unimoduláris mátrixok, valamint $S \in \mathbb{Z}^{n \times n}$ diagonális, s_i nemnegatív főátlóbeli elemekkel úgy, hogy

$$s_{i-1} | s_i,$$

ahol $i = 2, \ldots, n$, valamint

$$\prod_{i=1}^{n} s_i = |\det M|.$$

Az S mátrix az M mátrix Smith-normálformája.

4.3.4. Tétel. Legyen M invertálható mátrix, melynek Smith-normálformája $UMV = S$, valamint legyenek $z_1, z_2 \in \mathbb{Z}^n$.

$$z_1 \equiv z_2 \pmod{M} \iff \forall i \in \{1, \ldots, n\} : (Uz_1)_i \equiv (Uz_2)_i \pmod{s_i}$$

Bizonyítás: Ha $z_1 \equiv z_2 \pmod{M}$, akkor $z_1 \equiv z_2 \pmod{MV}$. Mivel $MV = U^{-1}S$, ezért $z_1 \equiv z_2 \pmod{U^{-1}S}$. Ebből viszont következik, hogy $Uz_1 \equiv Uz_2 \pmod{S}$ □

4.3.5. Állítás. Létezik $k \in \mathbb{N}$, melyre

$$\forall i \in \{1, \ldots, k\} : s_i = 1$$

4.3.6. Következmény. Tetszőleges $z \in \mathbb{Z}^n$ esetén

$$\forall i \in \{1, \ldots, k\} : (Uz)_i \equiv 0 \pmod{s_i}$$

Így elegendő az Uz szorzat $k+1, \ldots, n$. komponenseit meghatároznunk, azaz elegendő az U mátrix $k+1, \ldots, n$. soraival elvégezni a szorzást, amely mindösszesen csak $(n-k)n$ szorzást

igényel, szemben az *adjungált módszer*nél látott n^2 szorzással.

Legyen

$$D_S = UD \pmod S = \{b_1, \ldots, b_t\},$$

ahol $b_i \equiv Ua_i \pmod S$ úgy, hogy a b_i vektor i. komponense 0 és $s_i - 1$ közötti értéket vesz fel. Hasonlóan az *adjungált módszer* esetében kapott D_A halmazhoz, a D_S halmaz tartalmaz kell minden $z \in \mathbb{Z}^n$ ponthoz egy megfelelő b_j elemet, melyre $Uz \equiv b_j \pmod S$.

A Smith-normálforma használata esetén tudunk adni egy gyors módszert annak meghatározására, hogy egy tetszőleges $z \in \mathbb{Z}^n$ pont melyik $a_i \in D$ számjeggyel lesz kongruens. Ehhez definiáljuk a következő függvényt.

4.3.7. Definíció. (Hasítófüggvény) Legyen $h : \mathbb{Z}^n \to \mathbb{N}$,

$$h(z) = \sum_{i=k+1}^{n} ((Uz)_i \mod s_i) \prod_{j=k+1}^{i-1} s_i$$

4.3.8. Tétel. A h függvény a $\{0, 1, \ldots, t-1\}$ halmazba képez, valamint ha $z_1, z_2 \in \mathbb{Z}^n$, akkor

$$h(z_1) = h(z_2) \Longleftrightarrow z_1 \equiv z_2 \pmod M.$$

Bizonyítás: A h függvény valójában az $Uz \pmod S$ vektor $k+1., \ldots, n.$ komponenseinek $s_{k+1}, \ldots, s_d$ szerinti vegyes alapú felírása. □

4.3.3. A Φ függvény megvalósítása

A Φ függvény számítását a lehető leghatékonyabb módon kell megvalósítanunk. A hatékony implementációhoz az előző pontban közölt, Smith-normálformára vonatkozó eredményeket használjuk fel.

A $\Phi : \mathbb{Z}^n \to \mathbb{Z}^n$ függvényt a következő módon definiáltuk:

$$\Phi(z) = M^{-1}(z - a_i),$$

ahol $a_i \in D$ és $a_i \equiv z \pmod M$.

A Φ függvény számítása lényegében három lépésből áll: meg kell találnunk azt a $a_i \in D$ számjegyet, amely *modulo* M kongruens a bemenetként kapott vektorral, ezen felül el kell végeznünk egy vektor – vektor kivonást és végül egy mátrix – vektor szorzást.

A vektor – vektor kivonás műveletigénye a jegyrendszer elemeinek reprezentációjától függ. Amennyiben jegyrendszer elemei ritka vektorok, akkor a kivonást gyorsabban el tudjuk végezni, mint sűrű vektorok esetében.

Mivel M^{-1} általában nem egész elemekből áll, így ha egész számok felett szeretnénk számításokat végezni, akkor — kihasználva, hogy $M^{-1} = M^*/\det M$ — érdemesebb a szorzást a M adjungáltjával elvégezni, majd M determinánsával osztani az eredményt.

A Φ függvény számításának az implementáció szempontjából legösszetettebb része a bemenettel kongruens jegy meghatározása. Ehhez a Smith-normálformából származtatott, a 4.3.7. pontban definiált $h : \mathbb{Z}^n \to \mathbb{N}$ hasítófüggvény használatával felépített hasítótáblát használunk.

A $h : \mathbb{Z}^n \to \mathbb{N}$ hasítófüggvény implementációját a **GeNuSys::NumSys::SmithHash** osztály tartalmazza.

```
template<
  typename Type,
  template<typename Type> class MatrixType
>
class SmithHash { ... };
```

4.4. ábra. A **SmithHash** osztály

A továbbiakban legyen az M mátrix Smith-normálformája $UMV = S$. A korábbiakban láttuk, hogy létezik $k \in \mathbb{N}$, melyre

$$\forall i \in \{1, \ldots, k\} : s_i = 1,$$

illetve azt is láttuk, hogy a h függvény a $\{0, 1, \ldots, t-1\}$ halmazba képez, valamint ha $z_1, z_2 \in \mathbb{Z}^n$, akkor

$$h(z_1) = h(z_2) \iff z_1 \equiv z_2 \pmod{M}.$$

Mivel a h függvény számítása során az Uz szorzás eredményeként kapott vektornak csak a $k+1., \ldots, n.$ komponenseire van szükségünk, így a szorzást is elegendő elvégeznünk az U mátrix $k+1., \ldots, n.$ soraival, így valójában a teljes U mátrix helyett elegendő csak a mátrix megfelelő sorait tárolnunk. Ezen felül amennyiben a mátrix ritka, az elvégzendő szorzások számát tovább csökkenthetjük, ha U reprezentálására a **GeNuSys::LinAlg::SparseMatrix** adatszerkezetet használjuk. A MatrixType template paraméter lehetőséget ad U reprezentációjának megválasztására.

Mivel az Uz szorzás elvégzése potenciálisan memóriafoglalással jár, így abban az esetben, ha a függvényt több pontban is ki akarjuk értékelni, akkor függvény hatékonyságát tovább javíthatjuk, hogy ha az Uz szorzás eredményének előre foglalunk memóriát a createCache() függvény segítségével, és a számításaink alatt ezt a memóriaterületet használjuk újra.

A h függvény kiszámítása során elvégzendő szorzások számát tovább csökkenthetjük, ha ahelyett, hogy

$$\prod_{j=k+1}^{i-1} s_i$$

értékét minden függvényhíváskor kiszámítjuk, konstruálásakor miden megfelelő i esetére eltároljuk a szorzás eredményét.

Miután a h függvény megvalósítása már a rendelkezésre áll, elkészíthetjük a Φ függvény

implementációját. A függvény megvalósítását a **GeNuSys::NumSys::Phi** osztály tartalmazza.

```
template<
  typename Type,
  template<typename Type> class VectorType,
  template<typename Type> class MatrixType
>
class Phi {
  GeNuSys::LinAlg::Vector<Type> operator ()(
    const GeNuSys::LinAlg::Vector<Type>& z) const;

  GeNuSys::LinAlg::Vector<Type> createCache() const;

  const VectorType<Type>& operator ()(
    GeNuSys::LinAlg::Vector<Type>& z,
    GeNuSys::LinAlg::Vector<Type>& phiZ,
    GeNuSys::LinAlg::Vector<Type>& Uz) const;
};
```

4.5. ábra. A **Phi** osztály

A jegyrendszer megfelelő elemének megtalálásához egy hasítótáblát építünk a h hasítófüggvény segítségével. Mivel a h hasítófüggvény a $\{0, \ldots, t-1\}$ halmazba képez, így további számítások elvégzése nélkül, közvetlenül címezhetünk vele egy t hosszúságú tömböt.

A függvény konstruálásakor a hasítótáblát a jegyrendszer elemeivel töltjük fel. Mivel a jegyrendszer teljes maradékrendszert alkot, így a hasítótábla minden mezőjére pontosan egy számjegy jut.

A jegyrendszer elemei lehetnek ritka, illetve sűrű vektorok is, ezért az osztály mindkét adatszerkezetet támogatja. Ezen felül ugyanúgy lehetőséget ad a Smith-normálformából származtatott hasítófüggvénynél használt U mátrix reprezentációjának megválasztására is.

Az osztály két különböző megvalósítást ad a Φ függvényhez: az egyik megvalósítás a számítások — az Uz szorzás, a kongruens jegy kivonása, valamint az M^* mátrixszal való szorzás — elvégzése közben memóriafoglalást végez.

A másik megvalósítás a h függvény implementációjánál látott módon képes újrahasznosítani az Uz szorzás eredményének lefoglalt memóriát. A kongruens jegy kivonása helyben történik, az M^* mátrixszal való szorzás eredményének kiszámítása pedig szintén előre lefoglalt memóriaterületre történik. Így ennek a változatnak a használata nem jár semmilyen újabb memóriafoglalással.

A kongruens jegy kivonása azért történik helyben, hogy ritka jegyek esetén a kivonás elvégzése a lehető leghatékonyabb legyen, azonban ez megváltoztatja az eredeti bemeneti vektort.

4.4. Az osztályozási és az eldöntési algoritmus megvalósítása

Az osztályozási és az eldöntési feladatokkal korábban az 1.4 fejezetben foglalkoztunk. Az ott kapott eredményeket felhasználva az alábbi algoritmust adtunk az osztályozási feladat megoldására.

Algoritmus 19 Osztályozási algoritmus

```
1:  function CLASSIFY(Λ, M, D)
2:      finished ← { };
3:      P ← {};
4:      for all z ∈ K(Λ, M, D) do
5:          if z ∉ finished then
6:              orbit ← {};
7:              repeat
8:                  orbit ← orbit ∪ {z};
9:                  finished ← finished ∪ {z};
10:                 z ← Φ(z);
11:             until z ∉ finished and z ∈ K(Λ, M, D);
12:             if z ∈ orbit then
13:                 P ← P ∩ { GetCycle(z) };
14:             end if
15:         end if
16:     end for
17:     return P;
18: end function
```

A Φ függvény számításához az előző pontban látott implementációt használjuk. Mivel a Φ függvényt minden pontra kiszámítjuk, így a hatékony számításához a számítás során használt memóriaterületeket előre lefoglaljuk, és az algoritmus futása során újra felhasználjuk.

Az algoritmus végrehajtása során a $K(\Lambda, M, D)$ halmaz minden pontjáról vezetjük, hogy a pontot feldolgoztuk-e már valamely pályája számítása során. Mivel a halmaz elemeinek száma általában nagy, így egy hatékony módszert kell találnunk ennek megvalósítására.

Amennyiben a halmaz pontjait valamilyen módon sorszámozni tudjuk, akkor a feladatra használhatunk egy **bitvektor**t. A bitvektor minden indexhez 1 bit információt tárol — jelen esetben igaz / hamis értéket —, ezzel a lehető legkevesebb memória felhasználásával képesek vagyunk minden pontra követni, hogy az adott pontot feldolgoztuk-e vagy sem. Az algoritmus implementációja során a **GeNuSys::NumSys::BitVector** osztály által megvalósított bitvektor használjuk.

Az gyakorlati alkalmazás szempontjából a $K(\Lambda, M, D)$, azaz a rendszer periodikus pontjait garantáltan tartalmazó halmazként a $-H$ halmazának egy lefedését használjuk. Ennek az elő-

nye, hogy a lefedés elemeit könnyen fel tudjuk sorolni, illetve egy adott pontról hatékonyan el tudjuk dönteni, hogy a halmazban van vagy sem.

A lefedés előállítása során kapott $l, u \in \mathbb{R}^n$ korlátokat felhasználva definiáljuk a következő idx : $\mathbb{Z}^n \to \mathbb{N}$ bijektív függvényt, legyen

$$\text{idx}(v) = \sum_{i=0}^{n-1} (v_i + \lceil l_i \rceil) \prod_{j=0}^{i-1} (\lfloor u_i \rfloor - \lceil l_i \rceil + 1),$$

vagyis lényegében az adott komponensek lehetséges értékei alapján vett vegyes alapú felírását vesszük a $v - l$ vektornak.

Amennyiben a Horner módszert alkalmazzuk a függvény értékének számításához, akkor a

$$\prod_{j=0}^{i-1} (\lfloor u_i \rfloor - \lceil l_i \rceil + 1)$$

értékek kiszámítása nélkül, $\mathcal{O}(n)$ szorzás és ugyanennyi összeadás elvégzésével megkapjuk az eredményt.

Megkonstruálhatjuk az idx függvény inverzét is. Itt az egyes alapokkal kell sorban maradékosan osztanunk, illetve az l vektor megfelelő elemeit hozzáadnunk a maradékokhoz a vektor visszaállításához. Ez szintén $\mathcal{O}(n)$ maradékos osztást és kivonást jelent.

A **GeNuSys::NumSys::VolumeCodec** osztály biztosítja egy adott lefedéshez a vektorok indexének, illetve az adott indexhez tartozó vektornak az előállítását.

```
1 template<typename Type>
2 unsigned long long encode(const GeNuSys::LinAlg::Vector<Type>& z, bool&
    valid);
3 template<typename Type>
4 void decode(unsigned long long code, GeNuSys::LinAlg::Vector<Type>& z);
```

4.6. ábra. Vektor kódolása és dekódolás

A **valid** flag az index előállítása során állítódik be, amennyiben a vektor kódolható, azaz a $K(\Lambda, M, D)$ halmaz eleme, akkor az értéke igaz, különben hamis.

Az idx függvény a

$$\left\{0, \ldots, \prod_{i=0}^{n-1} (\lfloor u_i \rfloor - \lceil l_i \rceil + 1) - 1\right\}$$

halmazba képez. Így a bitvektor is közvetlenül indexelhetjük az idx függvénnyel, valamint a halmaz elemeinek bejárása is egyszerűvé válik.

Az implementáció szempontjából az utolsó megoldandó feladat a pálya követése. Mivel a halmazműveletek általában költségesek — a költségük a halmaz méretével, vagy annak logaritmusával arányos —, így halmaz helyett egy vermet használunk az adott körben vizsgált pálya

tárolására. Mivel a vektorok indexeit mindenképp kiszámítjuk, így a verem a pálya pontjainak indexét fogja tartalmazni.

Amikor a pálya számítása során elérünk egy olyan pontot, amelyet már feldolgoztunk, akkor elkezdjük a verem lebontását, amennyiben a verem lebontása során megtaláljuk a már feldolgozott pontot, ahol a számítás véget ért, akkor kört találtunk. Mivel a veremben egész számokat tárolunk, ezért itt sem lesz szükség vektorok összehasonlítására.

A vermet az egyes pálya számítások közt újra tudjuk használni, így újabb memóriafoglalásra sem lesz feltétlen szükség, csak akkor ha a verem mérete meghaladja az előre lefoglalt méretet. Mivel a veremnek lefoglalt memória mérete nem zsugorodik, és a tapasztalatok alapján a pályák hosszai között nincsenek nagyságrendi különbségek, így a verem mérete már az algoritmus futásának elején eléri a maximális értékét, és utána nem nő tovább.

Mivel minden pontot egyszer vizsgálunk meg, így az osztályozási algoritmus műveletigénye alapvetően a lefedő halmaz méretétől függ, ezért célszerű az algoritmus futása előtt a 4.2. pontban ismertetett módon javítani a lefedés méretét. Ezzel a módszerrel nagyságrendekkel csökkenthető a lefedés mérete, így az algoritmus futásideje is.

Az eldöntési algoritmus megvalósítása

Az eldöntési algoritmus megvalósítás szempontjából lényegében megegyezik az osztályozási algoritmussal. Az egyetlen különbség, hogy az első megtalált, nem 0 periodikus pont esetén megállíthatjuk az algoritmus futását, és kijelenthetjük, hogy a bemenetként kapott (Λ, M, D) hármas nem számrendszer. Amennyiben nem találunk ilyen kört a $K(\Lambda, M, D)$ halmaz bejárása során, akkor a bemenet számrendszer.

Az implementáció során az osztályozási algoritmusnál látott adatszerkezeteket és módszereket használjuk.

Mivel az algoritmus futásidejét az határozza meg, hogy milyen hamar találunk nem 0 periodikus pontot, így ha egy adott rendszer esetén sejtjük merre lehetnek periodikus pontok, akkor érdemes a $K(\Lambda, M, D)$ átvizsgálását azon a helyen kezdeni. Bár általános állításként nem mondhatjuk ki, de a tapasztalatok azt mutatják, hogy bizonyos esetekben érdemes a keresést a 0 közeli pontok átvizsgálásával kezdeni.

Az ebben a fejezetben leírt módszereket a **GeNuSys::NumSys::NumberSystem** osztály valósítja meg. A **GeNuSys::NumSys::NumberSystem** osztály implementációját a **number_system.h** és **number_system.hpp** fájlok tartalmazzák.

4.5. Szimultán rendszerek

4.5.1. Definíció. (Szimultán rendszer) Legyen $n, k \in \mathbb{N}^+$, valamint vegyük az $M_1, M_2, \ldots, M_k \in \mathbb{Z}^{n\times n}$ mátrixokat, melyekre teljesül az 1.1.6. tétel 2 és 3 pontja. Továbbá legyen $D' \subset \mathbb{Z}^n$. Ezekből

$$\Lambda = \mathbb{Z}^{kn}, \quad M = \text{diag}(M_1, \ldots, M_k), \quad D = \{(a_i, \ldots, a_i) : a_i \in D'\}.$$

Az ilyen módon konstruált rendszereket *szimultán rendszer*eknek nevezzük.

4.5.2. *Megjegyzés.* Könnyen látható, hogy

$$M^{-1} = \text{diag}(M_1^{-1}, \ldots, M_k^{-1}), \quad \text{valamint} \quad \det M = \prod_{i=1}^{k} \det M_i.$$

Mivel a szimultán rendszerhez tartozó D jegyrendszer elemeinek alakja speciális, így a korábban ismertetett módszerekkel nem tudunk megfelelő jegyrendszert előállítani.

A következőkben egy olyan módszert ismertetünk, amellyel kis n értékek esetén hatékonyan tudunk egy adott $\|.\|$ norma szerint sűrű jegyrendszert keresni.

4.5.1. Sűrű jegyrendszer keresése

4.5.3. Definíció. (Sűrű jegyrendszerek) Azon $D \subset \mathbb{Z}^{kn}$ jegyrendszereket nevezzük *sűrű jegyrendszerek*nek, amelyekre minden $a_i \in D$ jegy normája minimális.

Az alábbiakban $n = 2$ esetben elvégzett kereséshez adunk algoritmust. A módszerünk a következő: az $\{(a, b, \ldots, a, b) : a, b \in \mathbb{Z}\}$ hipersík elemeit járjuk be $\|.\|_\infty$ norma szerint növekvő sorrendben.

A korábban definiált, a Smith-normálformából származtatott h hasítófüggvényt használjuk az egyes elemek maradékosztályának meghatározásához.

Az algoritmus futása során minden maradékosztályhoz vezetjük a minimális normájú eddig megtalált jelöltet. Amennyiben találunk egy olyan elemet az adott maradékosztályból, amelynek a normája kisebb az eddig megtalált minimális normájú elem normájánál, akkor a jelöltet lecseréljük az újonnan vizsgált elemre.

A keresést legalább addig kell folytatnunk ameddig nem találunk minden osztályhoz egy jelöltet.

Az origótól indítjuk a keresést, és minden körben eggyel nagyobb $\|.\|_\infty$ normájú elemeket vizsgálunk. Így ha már minden maradékosztályból találtunk elemet és az adott körben minden megvizsgált jelölt normája nagyobb a legnagyobb normájú megtalált elem normájánál, akkor a keresést leállíthatjuk, a következő körökben már nem találhatunk kisebb normájú jelölteket.

Az algoritmus minimális átalakítással $n > 2$ esetekre is működik, azonban ahogy n értékét növeljük, a bejárandó tér mértéke exponenciálisan növekszik.

Algoritmus 20 Sűrű jegyrendszer keresése

```
1: function FINDDENSE(‖.‖)
2:     numFound ← 0;
3:     norm_max ← 0;
4:     digits ← [NULL, ..., NULL];
5:
6:     TRYCANDIDATE(0, 0);
7:     i ← 1;
8:     while numFound ≤ t or flag do
9:         flag ← FALSE;
10:        for j from −i + 1 to i − 1 do
11:            TRYCANDIDATE(i, j);
12:            TRYCANDIDATE(i, −j);
13:            TRYCANDIDATE(−i, j);
14:            TRYCANDIDATE(−i, −j);
15:        end for
16:        TRYCANDIDATE(i, i);
17:        TRYCANDIDATE(i, −i);
18:        TRYCANDIDATE(−i, i);
19:        TRYCANDIDATE(−i, −i);
20:    end while
21:    return digits;
22: end function
```

Algoritmus 21 Jelölt vizsgálata

```
1: function TRYCANDIDATE(i, j)
2:     v ← (i, j, ..., i, j);
3:     if ‖v‖ ≤ norm_max then
4:         flag ← TRUE;
5:     end if
6:     hash ← h(v);
7:     if found[hash] = NULL then
8:         numFound ← numFound + 1;
9:         digits[hash] ← v;
10:        if ‖v‖ > norm_max then
11:            norm_max ← ‖v‖
12:        end if
13:    else if ‖v‖ < ‖digits[hash]‖ then
14:        digits[hash] ← v;
15:        if ‖v‖ > norm_max then
16:            norm_max ← ‖v‖
17:        end if
18:    end if
19: end function
```

Összefoglalás

A dolgozatban bemutattuk az általánosított számrendszerek vizsgálatával kapcsolatos főbb elméleti eredményeket és hatékony algoritmusokat adtunk az eldöntési és osztályozási feladatok megoldásához.

Először hatékony implementációt adtunk a számításaink során felhasznált lineáris algebrai adatszerkezetekhez, valamint a velük végzendő műveletekhez, illetve megvalósítottuk az általánosított számrendszerek vizsgálatához szükséges lineáris algebrai algoritmusokat is.

Bemutattunk két módszert egy adott radix rendszer periodikus pontjainak behatárolására: először a konstruált operátornorma alapján meghatározott origó középpontú gömb sugarának kiszámítását, illetve később a $-H$ halmaz lefedésének előállítását. Továbbá ismertettünk egy módszert, amellyel a $-H$ halmaz lefedésének méretét, így az eldöntési, illetve osztályozási feladat során átvizsgálandó pontok számát nagyságrendekkel tudtuk csökkenteni.

Bemutattunk és megvalósítottunk egy módszert tetszőleges $\mathbb{Z}^n$ feletti pont maradékosztályának meghatározására, illetve a Φ függvény hatékony számítására. Végül ezen módszerek alapján elkészítettük az eldöntési, valamint az osztályozási feladat hatékony implementációját.

Ezen felül eljárásokat adtunk különböző speciális jegyrendszerek meghatározására, illetve bemutattuk szimultán rendszerek konstrukcióját és algoritmust adtunk jegyrendszer előállítására szimultán rendszerekhez is.

Bízunk benne, hogy a jövőben a könyvtár által biztosított hatékony algoritmusokat felhasználva az általánosított számrendszerekkel kapcsolatos új eredményekhez jutunk.

Irodalomjegyzék

[1] Burcsi, P.: Algorithmic aspects of generalized number systems. *Eötvös Loránd Tudományegyetem, Budapest*, Ph.D. értekezés, 2008, 1–56.

[2] Kovács, A.: Radix expansion in lattices. *Eötvös Loránd Tudományegyetem, Budapest*, Ph.D. értekezés, 2001, 1–98.

[3] Matula, D. W.: Basic digit sets for radix representation. *Journal of the ACM (JACM). 29*, 1982, 1131–1143.

[4] Vince, A.: Replicating Tessellations. *Siam J. Discrete Math. 6(3)*, 1993, 501–521.