EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR
KOMPUTERALGEBRA TANSZÉK

SZIMBOLIKUS SZÁMÍTÁSOK ALGEBRAI SZÁMOKKAL

Témavezető:
Burcsi Péter
Egyetemi docens

Készítette:
Uray Marcell János
Programtervező Informatikus MSc

Budapest, 2016

Tartalom

1. Bevezetés

A dolgozat bemutat egy új programkönyvtárat: a Symbolic Algebraic Numbers (SAN) könyvtárat, amely szimbolikusan számol algebrai számokkal. Célja, hogy hatékony programozási nyelven (C++-ban) numerikus számolás helyett egzaktul lehessen számolni, ezáltal némi futásiidő-növekedésért cserébe ki lehessen küszöbölni a kerekítési hibákat. Ennek megfelelően a könyvtár az algebrai műveleteken túl megvalósít szimbolikusan olyan, a valós számok esetén fontos műveleteket is, mint két szám összehasonlítása és az egészre kerekítés.

A dolgozat felépítése a következő:

A 2. fejezet áttekinti a témával kapcsolatos szakirodalmat, és bemutat különböző módszereket algebrai számok ábrázolására és az ezekkel való szimbolikus számolásra.

A 3. fejezet bemutatja a SAN könyvtár használatát.

A 4. fejezet a könyvtár működéséről szól: leírja, hogy a 2. fejezetben áttekintett lehetőségek közül a könyvtár mely ábrázolásokat és módszereket választotta, és konkrét algoritmusokon keresztül bemutatja, hogy ezeket hogyan alkalmazza, illetve milyen további ötletekkel javítja.

Az 5. fejezet a könyvtár tesztéléséből és alkalmazásából származó eredményeket mutatja be, az egyszerű futási időktől kezdve olyan bonyolultabb alkalmazásokig, mint az LLL-algoritmus.

Végül, a 6. fejezetben szó lesz a könyvtár néhány lehetséges továbbfejlesztési és gyorsítási lehetőségéről.

2. Algebrai számok ábrázolása

Ebben a fejezetben áttekintjük az algebrai számok néhány lehetséges ábrázolását.

Definíció: Egy $\alpha \in \mathbb{C}$ szám **algebrai szám**, ha van olyan $f \in \mathbb{Z}[x]$ nemnulla polinom, melynek gyöke: $f(\alpha) = 0$.

Jelölés: Az algebrai számok halmazának jele: $\mathbb{A}$.

A továbbiakban csak valós algebrai számokkal ($\mathbb{A} \cap \mathbb{R}$) foglalkozunk, mert komplex algebrai számok felírhatók két valós algebrai számmal:

Állítás: Egy $\alpha \in \mathbb{C}$ komplex számra $\alpha \in \mathbb{A} \iff \mathrm{Re}(\alpha), \mathrm{Im}(\alpha) \in \mathbb{A} \cap \mathbb{R}$.

Algebrai számokat szimbolikusan többféleképpen lehet ábrázolni. Ebben a fejezetben az alábbiakról lesz szó:

1. rekurzívan, alapműveletek és gyökvonás segítségével;
2. polinom gyökeként;
3. rögzített algebrai szám racionális függvényeként.

2.1. Rekurzív ábrázolás

Egyik lehetséges mód algebrai számok ábrázolására a kifejezésfa. Erre az algebrai számok alábbi tulajdonságai adnak lehetőséget:

Állítás: [2, 349-350. o.]

a) $\mathbb{Q} \subset \mathbb{A}$
b) $\alpha, \beta \in \mathbb{A} \Rightarrow \alpha + \beta \in \mathbb{A}$.
c) $\alpha \in \mathbb{A} \Rightarrow -\alpha \in \mathbb{A}$.
d) $\alpha, \beta \in \mathbb{A} \Rightarrow \alpha \cdot \beta \in \mathbb{A}$.
e) $\alpha \in \mathbb{A} \setminus \{0\} \Rightarrow \frac{1}{\alpha} \in \mathbb{A}$.
f) $\alpha \in \mathbb{A}, n \in \mathbb{N}^+ \Rightarrow \sqrt[n]{\alpha} \in \mathbb{A}$

A számokat a matematikai formulájuk (pl. $1+\sqrt{2}$) kifejezésfájával ábrázoljuk: olyan irányított fával, amelynek a leveleiben racionális számok vannak, a belső csúcsokban pedig algebrai műveletek ($+$, $-$, $\cdot$, $/$ és $\sqrt[n]{\ }$). Ebben a formában a műveletvégzés egyszerű: csak össze kell kapcsolni a két szám kifejezésfáját az új művelettel.

Az egyik probléma ezzel az ábrázolással, hogy nem egyértelmű: ugyanazt az algebrai számot többféleképpen is lehet ábrázolni (pl. $\sqrt{2} + 1 = \frac{1}{\sqrt{2}-1}$), ezért például az egyenlőségvizsgálat nehéz feladat. Másrészt a műveletvégzések során a kifejezésfa egyre nagyobb lesz, pedig lehet, hogy az eredményt sokkal egyszerűbb alakban is fel lehetne írni (pl. $\left(\sqrt{2}+1\right)\left(\sqrt{2}-1\right) = 1$). Ezért célszerű a műveletvégzések után

egyszerűsíteni a kifejezést, de ez már nem egyszerű feladat.

Az ábrázolás további problémája egy elvi korlát, amelyre a következő tétel mutat rá:

Tétel (Galois): Van olyan algebrai szám, amely nem írható fel véges sok racionális szám, alapművelet és gyökvonás segítségével. Példa: ilyenek az $x^5 - 4x + 2$ polinom gyökei. [2, 406. o.]

Ez azt jelenti, hogy ilyen módon nem lehet minden algebrai számot ábrázolni, csak egy részhalmazukat.

2.2. Ábrázolás polinom gyökeként

Ha semmilyen további megszorítást nem szeretnénk az algebrai számok halmazára tenni, akkor kézenfekvő ábrázolás maga a definíció: azzal a polinommal ábrázoljuk, amelynek gyöke a szám.

Ez az ábrázolás így nem egyértelmű: egy adott algebrai számra végtelen sok ilyen polinom létezik (elég megszorozni egy tetszőleges polinommal). Emiatt például az egyenlőségvizsgálat nehéz. Az egyértelműség az alábbi fogalommal biztosítható:

Definíció: Egy α algebrai szám **minimálpolinomja** a legalacsonyabbfokú $f \in \mathbb{Z}[x]$ nemnulla polinom, melyre $f(\alpha) = 0$.

Állítás: [2, 343. o.]

a) A minimálpolinom konstans szorzótól eltekintve egyértelműen létezik.
b) A minimálpolinom felbonthatatlan.

Definíció: Egy algebrai szám **fokszáma** a minimálpolinomjának fokszáma.

Ha tehát az ábrázoló polinomról megköveteljük, hogy felbonthatatlan legyen, vagyis a minimálpolinom, akkor ez így már egyértelmű (feltéve, hogy a konstans szorzóra is teszünk egy megszorítást), és ekkor az egyenlőségvizsgálat is egyszerű. Viszont ennek az az ára, hogy a felbonthatatlanságot minden művelet után biztosítani kell. Ehhez általában polinomfaktorizációra van szükség, amely időigényes művelet. Ennek ellenére érdemes ezt a megszorítást megtenni, mert különben a műveletvégzések során a fokszám exponenciálisan növekedhetne, miközben a minimálpolinom sokkal kisebb fokszámú is lehet.

Van egy másik probléma az ábrázolással: egy polinomnak általában több gyöke van.

Definíció: Algebrai számok **konjugáltak**, ha megegyezik a minimálpolinomjuk (konstans szorzótól eltekintve).

Ezért ha egy konkrét algebrai számot szeretnénk ábrázolni, akkor a polinom mellett még további adatokra van szükség, amelyek azonosítják, hogy a konjugáltak közül melyikről van szó. Erre több módszer van, ezekből néhányat mutatunk ebben a fejezetben.

Először azonban megnézzük általánosan, hogy hogyan lehet műveleteket végezni ebben az ábrázolásban.

2.2.1. Műveletvégzés

Polinomok gyökeként megadott számok közötti műveletvégzéshez fontos eszköz a rezultáns [3, 52-57. o.]:

Definíció: Legyen D integritási tartomány, $f, g \in D[x]$ n-ed-, illetve m-edfokú polinomok. Legyenek f gyökei $\alpha_1, \alpha_2, \ldots, \alpha_n$ (multiplicitással számolva), g gyökei pedig hasonlóan $\beta_1, \beta_2, \ldots, \beta_m$. Ekkor a két polinom **rezultánsa** (az x változó szerint):

$$\mathrm{res}_x(f, g) := f_n^m g_m^n \prod_{i=1}^{n} \prod_{j=1}^{m} (\alpha_i - \beta_j),$$

ahol f_n és g_m az f, ill. a g főegyütthatója.

Állítás:

a) $\mathrm{res}_x(f, g)$ az f és a g együtthatóiból összeadás, kivonás és szorzás segítségével kiszámítható. Következésképp $\mathrm{res}_x(f, g) \in D$.
b) $\mathrm{res}_x(f, g) = 0 \Leftrightarrow f$-nek és g-nek van közös gyöke.

A rezultáns segítségével az egyes műveletek eredményének polinomját az alábbiak szerint lehet kiszámítani [1, 159. o.]:

Állítás: Legyen $f(\alpha) = 0$ és $g(\beta) = 0$. Ekkor a táblázat első oszlopában álló kifejezés gyöke a második oszlopban lévő polinomnak. Ezen polinom fokszáma a harmadik oszlopban látható, a negyedik oszlopban pedig az, hogy ha f és g felbonthatatlan, akkor mit lehet mondani az eredménypolinomról.

Gyök	*Polinom*	*Fokszáma*	*Felbonthatatlanság*
$\alpha + \beta$	$\mathrm{res}_y(f(y), g(x - y))$	$\deg f \cdot \deg g$	
$\alpha - \beta$	$\mathrm{res}_y(f(y), g(x + y))$	$\deg f \cdot \deg g$	
$\alpha\beta$	$\mathrm{res}_y(f(y), y^{\deg g} \cdot g(x/y))$	$\deg f \cdot \deg g$	
α/β	$\mathrm{res}_y(f(y), g(xy))$	$\deg f \cdot \deg g$	
$-\alpha$	$f(-x)$	$\deg f$	felbonthatatlan
$1/\alpha$	$x^{\deg f} \cdot f(1/x)$	$\deg f$	felbonthatatlan
$\sqrt[n]{\alpha} \quad (n \in \mathbb{N}^+)$	$f(x^n)$	$n \cdot \deg f$	
$p(\alpha) \ \ (p \in \mathbb{Z}[x])$	$\mathrm{res}_y(f(y), x - p(y))$	$\deg f$	felbonthatatlan hatványa

Minimálpolinomos ábrázolás esetén fontos, hogy a kapott eredmény felbonthatatlan legyen, ezért azoknál a műveleteknél, ahol ez nem teljesül automatikusan, meg kell keresni a megfelelő felbonthatatlan faktort. Az utolsó műveletnél, a polinomba történő behelyettesítésnél tudjuk, hogy az eredmény egy felbonthatatlan polinom hatványa, ezért ott csak négyzetmentesíteni kell. Általában azonban a polinom faktorizációjára van szükség, és utána ki kell választani a megfelelő faktort. Ez viszont függ attól, hogy az eredeti polinomok melyik gyökéről van szó, így ehhez mindenképpen szükség van a polinom melletti további adatra, amely azonosítja a gyököt.

2.2.2. Gyök azonosítása

2.2.2.1. Izolációs intervallum

A konjugáltak közül a megfelelő gyök azonosítására a legkézenfekvőbb módszer az izolációs intervallum.

Definíció: Legyen $f \in \mathbb{Z}[x]$ polinom. Egy $[a, b]$ intervallumot, ahol $a, b \in \mathbb{Q}$, az f egy **izolációs intervallum**ának nevezzük, ha $\exists! \alpha \in [a, b] : f(\alpha) = 0$.

Ha tehát a polinom mellett tárolunk egy izolációs intervallumot is, akkor azzal egyértelműen definiáltuk az algebrai számot. Az egyértelműség azonban fordítva nem igaz: egy polinom egy gyökéhez több izolációs intervallumot is meg lehet adni. Sőt, a racionális számok sűrűsége miatt az intervallum tetszőlegesen szűkíthető. Ez utóbbi tulajdonság sok esetben jól használható, például annak eldöntéséhez, hogy két algebrai szám közül melyik a nagyobb.

Ahhoz, hogy egy intervallumról eldöntsük, hogy izolációs intervallum-e, a benne lévő gyökök számáról kell mondani valamit. Ehhez két tételt ismertetünk az alábbiakban: a Sturm-tételt és a Budan-Fourier-tételt.

Definíció: Egy $f \in \mathbb{Z}[x]$ polinom **kanonikus Sturm-sorozat**ának nevezzünk a következő polinomsorozatot:

- $f_0 := f$,
- $f_1 := f'$,
- $f_k := -\operatorname{rem}(f_{k-2}, f_{k-1}) \quad (k \geq 2)$,

ahol $\operatorname{rem}(f, g)$ az f-nek a g-vel való osztásából származó maradék. A rekurziót addig ismételjük, amíg $f_k \neq 0$.

Tétel (Sturm): Legyen $f \in \mathbb{Z}[x] \setminus \{0\}$ polinom, és kanonikus Sturm-sorozata legyen $f_0, f_1, \ldots, f_m$. Jelölje $\sigma_S(x)$ az $f_0(x), f_1(x), \ldots, f_m(x)$ sorozatban előforduló előjelváltások számát (elhagyva a 0-kat). Ekkor f-nek az $(a, b]$ intervallumban lévő különböző valós gyökeinek száma $\sigma_S(a) - \sigma_S(b)$, feltéve, hogy f-nek sem a, sem b nem többszörös gyöke.

Tétel (Budan–Fourier): Legyen $f \in \mathbb{Z}[x] \setminus \{0\}$ egy n-edfokú polinom. Jelölje $\sigma_F(x)$ az $f(x), f'(x), f''(x), \ldots, f^{(n)}(x)$ sorozatban előforduló előjelváltások számát (elhagyva a 0-kat). Ekkor ha f-nek az $(a, b]$ intervallumban multiplicitással számolva r gyöke van, akkor $r \leq \sigma_F(a) - \sigma_F(b)$, pontosabban $r = \sigma_F(a) - \sigma_F(b) - 2k$, ahol $k \in \mathbb{N}$. [7]

A Sturm-tétel segítségével minden esetben meg tudjuk mondani, hogy hány gyök esik egy adott intervallumba. A Budan–Fourier-tétellel könnyebb számolni (nem kell a polinomsorozatot kiszámítani, mert a Horner-algoritmus kiterjesztésével az összes deriváltat egyszerre ki tudjuk értékelni), viszont a gyökök számára általában csak egy felső becslést ad, és azon belül csak a paritását adja meg. Ez például akkor használható, ha $\sigma_F(a) - \sigma_F(b) = 1$, mert ekkor tudjuk, hogy pontosan egy gyök van az intervallumban (de a fordítottja nem igaz). Viszont speciális esetben pontos eredményt ad a tétel:

Állítás: Ha az f polinomnak minden gyöke valós, akkor a Budan–Fourier-tételben $k = 0$, azaz $r = \sigma_F(a) - \sigma_F(b)$.

Most nézzük, hogy az algebrai számok közötti műveletvégzés után hogyan lehet kiszámítani az eredmény izolációs intervallumát. Ennek az alapja az, hogy elvégezzük intervallum-aritmetikailag az adott műveletet a bemenő számok intervallumaira. Ez így önmagában azonban általában nem elég: bár a kívánt gyök benne lesz az új intervallumban, de nincs garancia arra, hogy nem kerül bele más gyök is. Ezt az intervallumok szűkítésével oldhatjuk meg: ha az új intervallumban több gyök is van, akkor az eredetieket szűkítjük, azokból újra kiszámítjuk az új intervallumot, és ezt addig ismételjük, amíg izolációs intervallumot nem kapunk. Az egyváltozós műveletek esetén (additív és multiplikatív inverz, illetve gyökvonás) ez a probléma elvileg nem jelentkezik: a kapott új intervallumban pontosan egy gyök marad. A gyökvonásnál mégis van egy probléma: bár a kapott intervallumban csak egy gyök lesz, de a végpontok nem feltétlenül racionális számok. Ilyenkor egy kicsit tágítjuk az intervallumot úgy, hogy racionális végpontjai legyenek, de mivel ezzel más gyök is kerülhet bele, ezután itt is a szűkítéses eljárást alkalmazzuk.

Ha megvan az eredmény izolációs intervalluma, akkor ezzel könnyen meg tudjuk válaszolni azt a kérdést is, hogy az összetett polinom melyik faktorának lesz gyöke a keresett algebrai szám: csak meg kell nézni, hogy melyik faktornak hány gyöke van az intervallumban (és mivel itt ez a szám csak 0 vagy 1 lehet, ezért ez egyszerű előjelvizsgálattal eldönthető).

2.2.2.2. Deriváltak előjelei

A gyök azonosítására más módszer is létezik, egy ilyet mutat Mishra és Pedersen [8]. Ők a polinom deriváltjainak előjeleinek segítségével különböztetik meg a konjugált gyököket. A módszer a következő tételen alapul:

Thom-lemma: Legyen $f \in \mathbb{Z}[x]$ n-edfokú polinom, és $s_0, s_1, \ldots, s_{n-1} \in \{-1, 0, 1\}$ előjelek. Ekkor az $\{x \in \mathbb{R} \mid \forall k \in \{0, 1, \ldots, n-1\} : \operatorname{sgn} f^{(k)}(x) = s_i\}$ halmaz vagy üres, vagy intervallum (akár véges, akár végtelen).

Következmény: Ha $f \in \mathbb{Z}[x] \setminus \{0\}$ n-edfokú polinom, $\alpha, \beta \in \mathbb{R}$ két gyöke, és $\forall k \in \{1, \ldots, n-1\} : \operatorname{sgn} f^{(k)}(\alpha) = \operatorname{sgn} f^{(k)}(\beta)$, akkor $\alpha = \beta$.

Tehát elegendő a polinom mellett ezeket az előjeleket tárolni, mert ezek egyértelműen meghatározzák, hogy melyik gyökről van szó.

2.2.3. Értékelés

A polinom gyökeként való ábrázolás kellően általános, hiszen minden algebrai számot lehet így ábrázolni. Viszont nem hatékony: a legegyszerűbb alapműveletekhez is, mint az összeadás és a szorzás, összetett polinomműveletekre van szükség: rezultánst kell számítani, és minimálpolinomos ábrázolás esetén utána faktorizálni is kell (vagy ellenkező esetben gyorsan nőnek a polinomok). Ez gyakorlatilag nem teszi lehetővé, hogy bonyolult algoritmusokban számtípusként ilyen ábrázolású algebrai számokat használjunk, például lebegőpontos számok helyett.

2.3. Ábrázolás algebrai számtestben

A hatékony műveletvégzést úgy fogjuk biztosítani, ha egy adott feladathoz leszűkítjük az algebrai számok körét.

Definíció: Egy F test **algebrai számtest**, ha véges dimenziós bővítése a racionális számtestnek ($\mathbb{Q}$-nak). Ekkor az F test **dimenziója** a $\mathbb{Q}$ feletti bővítés, mint lineáris tér fokszáma.

A továbbiakban egy rögzített algebrai számtestben fogunk számolni. A számtest, mivel véges dimenziós bővítés, felírható úgy, hogy a $\mathbb{Q}$-hoz hozzáveszünk véges sok algebrai számot: $\mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_k)$. A számtest elemeit ezen bővítő elemek segítségével fogjuk ábrázolni.

2.3.1. Egyszerű bővítés

A legegyszerűbb dolgunk akkor van, ha a számtest egyetlen bővítő elemmel van megadva.

Definíció: Az F test **egyszerű bővítése** $\mathbb{Q}$-nak, ha $\exists \alpha \in F : F = \mathbb{Q}(\alpha)$. Az ilyen α-t az F egy **primitív elem**ének nevezzük.

Az ilyen számtest elemeit a rögzített α segítségével könnyen fel tudjuk írni:

Állítás: Ha $F = \mathbb{Q}(\alpha)$, ahol α egy n-edfokú algebrai szám, akkor F dimenziója n, és egy bázisa $1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$. [2, 345. o.]

Vagyis a test elemei az α legfeljebb $n-1$-edfokú, racionális együtthatós polinomjai, és ez a felírás egyértelmű.

Ebben az ábrázolásban a számolás egyszerű. Összeadáshoz és kivonáshoz, illetve racionális számmal való szorzáshoz és osztáshoz egyszerűen csak együtthatónként végezzük el a megfelelő műveletet. Általános szorzásnál és osztásnál figyelembe kell venni, hogy $f(\alpha) = 0$, ahol f az α minimálpolinomja (melynek fokszáma n), így lényegében $\mathbb{Q}[x]/(f)$-ben kell számolni. Szorzásnál ez azt jelenti, hogy a két polinom szorzatának (amely lehet $n-1$-nél magasabb fokú) vesszük az f szerinti maradékát. Osztásnál multiplikatív inverzet számolunk, a bővített euklideszi algoritmus segítségével: egy g polinom inverzének meghatározásához megoldjuk a $sf + tg = 1$ polinomegyenletet a bővített euklideszi algoritmussal, és ekkor $tg \equiv 1(f)$ lesz, azaz a multiplikatív inverz t lesz.

2.3.2. Többszörös bővítés

Nehezebb a feladat, ha az algebrai számtestet egynél több bővítő elemmel adtuk meg: $F = \mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_k)$. Erre az esetre többféle megoldást mutatunk.

2.3.2.1. Többváltozós polinomokkal

Egyik megoldás, hogy általánosítjuk az egyszerű bővítés ábrázolását: egyváltozós helyett többváltozós polinomokat használunk, ahol az egyes változók az egyes bővítő elemeknek felelnek meg. Az ábrázolást rekurzívan építjük fel: egyenként adjuk hozzá a bővítő elemeket $\mathbb{Q}$-hoz, és a következő bővítéshez a már meglévő előző bővítést vesszük alapul. [4, 12-13. o.]

Ehhez először általánosítunk néhány fent bevezetett fogalmat:

Definíció: Legyen K test, és L a K testbővítése. Ekkor egy $\alpha \in L$ **algebrai elem a K test fölött**, ha van olyan $f \in K[x]$ nemnulla polinom, amelynek gyöke: $f(\alpha) = 0$.

Ennek speciális esete az algebrai szám, ha $K = \mathbb{Q}$, mert bár annak a definíciójában $\mathbb{Z}[x]$-beli polinomot követeltünk meg, de pontosan ugyanakkor van ugyanilyen $\mathbb{Q}[x]$ fölötti polinom is.

Definíció: Egy $\alpha \in L$ algebrai elem **minimálpolinomja a K test fölött** a legalacsonyabbfokú $f \in K[x]$ nemnulla polinom, melyre $f(\alpha) = 0$.

Adottak tehát az $\alpha_1, \alpha_2, \ldots, \alpha_k$ algebrai számok, és az $F = \mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_k)$ testet szeretnénk ábrázolni. Legyen $F_i := \mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_i)$ az i. közbülső bővítés ($F_0 := \mathbb{Q}$), és ezt bővítjük rekurzívan a következő elemmel: $F_{i+1} = F_i(\alpha_{i+1})$. Ha α_{i+1} algebrai szám, akkor algebrai elem az F_i fölött is, hiszen ha gyöke egy $\mathbb{Q}$ fölötti polinomnak, akkor az egyúttal F_i fölötti polinom is. Legyen α_{i+1}-nek az F_i fölötti minimálpolinomja f_{i+1}. Ez nem feltétlenül azonos a $\mathbb{Q}$ fölötti minimálpolinommal, mert az attól, hogy $\mathbb{Q}$ fölött felbonthatatlan, F_i fölött még nem feltétlenül az. Ha α_{i+1} a ($\mathbb{Q}$ fölötti) minimálpolinomjával van megadva, akkor azt először faktorizálni kell F_i fölött, és ki kell választani a megfelelő faktort (például behelyettesítéssel). Ha már megvan az F_i fölött minimálpolinom, akkor az ábrázolás ugyanúgy történik, mint a $\mathbb{Q}$ egyszerű bővítésénél: F_i fölötti legfeljebb ($\deg f_{i+1} - 1$)-edfokú polinomokat használunk, egészen pontosan az $F_i[x_{i+1}]/(f_{i+1})$ elemeivel ábrázoljuk F_{i+1} elemeit. Ha ezt rekurzívan alkalmazzuk, akkor a végeredmény egy k-változós $\mathbb{Q}$ fölöti polinom lesz, melynek minden x_i változójában korlátozott a fokszáma (legfeljebb $\deg f_i - 1$).

2.3.2.2. Primitív elem keresése

A többszörös bővítések ábrázolásának másik módja, hogy felírjuk a testet egyszerű bővítésként. Erre az alábbi tétel ad lehetőséget:

Tétel (primitív elem tétel): Minden F algebrai számtestben van $\alpha \in F$ primitív elem. [5, 556. o.]

Tehát minden $\mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_k)$ test felírható $\mathbb{Q}(\alpha)$ alakban. A kérdés az, hogy hogyan lehet egy ilyen α-t meghatározni az $\alpha_1, \alpha_2, \ldots, \alpha_k$ elemekből. Indukcióval visszavezethető a kérdés csupán két elemre: adottak a β és a γ algebrai számok, és keresünk olyan α-t, amelyre $\mathbb{Q}(\alpha) = \mathbb{Q}(\beta, \gamma)$.

A következő tétel választ ad arra a kérdésre, hogy milyen alakban kell keresni a primitív elemet:

Tétel: Legyen $\mathbb{Q}(\beta, \gamma)$ egy n-dimenziós algebrai számtest. Ekkor a $\beta + s\gamma$ szám

primitív eleme a testnek majdnem minden $s \in \mathbb{Z}$-re, kivéve véges sok (legfeljebb $n-1$) egész számot. [5, 563. o.]

A primitív elem keresése ezek után így néz ki: vesszük az $\alpha := \beta + s\gamma$ számot rendre $s = 1, 2, 3, \ldots$-ra, és megnézzük, hogy a kapott kifejezés primitív elem-e, azaz $\mathbb{Q}(\alpha)$ megegyezik-e $\mathbb{Q}(\beta, \gamma)$-val. A kérdés az, hogy ezt hogyan dönthetjük el. Legyen g és h a β, ill. a γ minimálpolinomja. Számítsuk ki a következő legnagyobb közös osztót:

$$\gcd\left(g(x), h\left(\frac{\alpha - x}{s}\right)\right)$$

A polinomok együtthatói a $\mathbb{Q}(\alpha)$ testben vannak, amelyben a fent leírt módon tudunk számolni. Mindkettőnek gyöke β, ezért a legnagyobb közös osztónak is gyöke lesz. Ha a legnagyobb közös osztó elsőfokú, akkor abból β közvetlenül kifejezhető az α segítségével, vagyis $\beta \in \mathbb{Q}(\alpha)$, továbbá $\gamma = \frac{\alpha-\beta}{s} \in \mathbb{Q}(\alpha)$. Ezekből már következik, hogy $\mathbb{Q}(\alpha) = \mathbb{Q}(\beta, \gamma)$, továbbá megkaptuk β és γ felírását az α primitív elem segítségével. A kérdés már csak az, hogy mit kezdünk azzal az esettel, ha a legnagyobb közös osztó magasabbfokú. Az egyik lehetőség, hogy faktorizáljuk az eredményt $\mathbb{Q}(\alpha)$ fölött, behelyettesítéssel kiválasztjuk a β-hoz tartozó faktort, és ekkor α pontosan akkor lesz primitív elem, ha ez a faktor elsőfokú. A másik, egyszerűbb megoldás, hogy ezzel az esettel nem foglalkozunk, hanem továbbmegyünk a következő s-re. Ezt megtehetjük, mert a következő állításból következik, hogy csak ritkán kapunk magasabbfokú eredményt:

Állítás: Csak véges sok olyan s egész szám létezik, melyre van olyan $\beta' \neq \beta$ gyöke g-nek és $\gamma' \neq \gamma$ gyöke h-nak, hogy $\alpha = \beta' + s\gamma'$.

Ugyanis a fenti legnagyobb közös osztónak pontosan az ilyen β'-k lesznek a gyökei (a β-n kívül), tehát pontosan akkor lesz magasabbfokú ez a polinom, ha β-n kívül van még ilyen gyöke g-nek. Az állítás szerint pedig csak véges sok s-re van.

3. A könyvtár használata

Ebben a fejezetben bemutatjuk a Symbolic Algebraic Numbers (SAN) könyvtár használatát. Ez egy C++-ban írt programkönyvtár, amely algebrai számtípusokat definiál. A fontosabb típusai:

Típus	*Leírás*	*Fejléc (`#include`)*
`san::integer`	egész szám ($\mathbb{Z}$)	`<san/integer.hpp>`
`san::rational`	racionális szám ($\mathbb{Q}$)	`<san/rational.hpp>`
`san::polynomial<type>`	polinom ($R[x]$)	`<san/polynomial.hpp>`
`san::algebraic`	valós algebrai szám ($\mathbb{A} \cap \mathbb{R}$)	`<san/algebraic.hpp>`
`san::algfield`	algebrai számtest	`<san/algfield.hpp>`
`san::algnumber`	algebrai számtest eleme	`<san/algnumber.hpp>`

Ezenkívül fontos fejléc még a `<san/error.hpp>`, amely hibatípusokat definiál (pl. `division_by_zero`).

A fentieket a fejezet későbbi részeiben részletesen tárgyaljuk. A könyvtár minden nevet (típust, függvényt, objektumot) a `san` névtérben definiál. A továbbiakban ezt nem jelezzük külön.

3.1. Fordítás

A könyvtár fordításához bármely C++ fordító használható, amely megfelel a C++11 szabványnak, illetve biztosan használhatók az alábbi fordítók:

- GCC 4.6
- Microsoft Visual C++ 2015

A könyvtár lefordítását segítik a mellékelt segédállományok: Makefile (GCC-hez) és Makefile.msvc (Visual C++-hoz). Ezek segítségével parancssorból egyszerű a fordítás. GCC-hez a GNU Make segédprogramot használhatjuk:

```
make
```

Visual C++-hoz pedig a hozzá tartozó NMAKE parancsot:

```
nmake /F Makefile.msvc
```

A programot, amely használja a könyvtárat, olyan paraméterekkel kell lefordítani, hogy a fordító elérje a könyvtár fejlécállományait, és a szerkesztés során megtalálja a lefordított könyvtárat. Például ha a `san` könyvtár a program könyvtárához képest a `../san` elérési úton van, akkor a GCC-nek a következő kapcsolót kell megadni fordításhoz: `-I..`, és a következőt szerkesztéshez: `../san/san.a`. Visual C++-ban ez `/I..` és `..\san\san.lib`.

3.2. Közös számműveletek

A fejezet további részeiben az egyes típusok és fejlécek részletes dokumentációja következik.

A könyvtár négy számtípust definiál: `integer`, `rational`, `algebraic` és `algnumber`. Ezeknek a típusoknak sok közös műveletük van, ezért ezeket nem írjuk le mind a négynél külön-külön, hanem csak egyszer, ebben a részben. Az egyes típusokra itt a `type` névvel hivatkozunk, ezt értelemszerűen be kell helyettesíteni a négy típus valamelyikére.

Bármely típust ha paraméter nélkül létrehozzuk, értéke nulla lesz.

Aritmetika

Ezen négy típus objektumaival a beépített számokhoz hasonlóan lehet aritmetikai műveleteket végezni:

- négy alapművelet (`a + b`, `a - b`, `a * b`, `a / b`);
- ezek értékadásos változata (`a += b`, `a -= b`, `a *= b`, `a /= b`);
- előjelképzés (`+a`, `-a`).

Összehasonlítás

Ezek a típusok az elemi számokhoz hasonlóan támogatják a relációs műveleteket:

- egyenlőségvizsgálat (`a == b`, `a != b`);
- egyenlőtlenségek (`a < b`, `a > b`, `a <= b`, `a >= b`).

Ezenkívül van egy függvény, amely két számot összehasonlít, és háromféle eredményt adhat vissza:
`int compare (type const& a, type const& b);`
Ha `a == b`, akkor `0`-t, ha `a < b`, akkor `-1`-et, ha pedig `a > b`, akkor `+1`-et.

A fenti kétváltozós műveleteket különböző típusú argumentumokra is lehet használni, a négy típus közül bármely kettőre, kivéve az `algnumber` és az `algebraic` párosra.

Hatványozás

A szám önmagával való a szorzásának ismétlését az alábbi függvények hatékonyabban végzik el:
`type sqr (type const& x);` – a szám négyzete (x^2)
`type cube (type const& x);` – a szám köbe (x^3)
`type pow (type const& x, unsigned n);` – a szám `n`-edik hatványa (x^n)

Konvertálás

A számtípusok között az értékkészletüket tekintve matematikailag az alábbi relációk állnak fenn:

$$\texttt{integer} \subset \texttt{rational} \subset \texttt{algnumber} \subset \texttt{algebraic}$$

Ennek megfelelően lehet ezeket egymásba konvertálni: minden típus konvertálható a láncban tőle jobbra lévő bármelyik típussá. Ezek automatikusan (implicit módon) működnek, kivéve az `algnumber` → `algebraic` konverziót, amelyet hatékonysági okok miatt csak explicit módon lehet végrehajtani (ki kell írni a konverziót).

Emellett az alábbi további konvertálási lehetőségek vannak:

- Bármelyik fenti számtípus implicit módon konvertálható az elemi `int` típusból. Az `integer` ezenkívül az összes többi elemi egész típusból (`unsigned int`, `long` stb.) is konvertálható.
- Az `integer` explicit módon konvertálható bármely elemi egész számtípusba. Ha az érték nem fér bele az adott típusba, akkor a konvertálás eredménye nem definiált.
- A könyvtár bármely számtípusa explicit módon konvertálható `integer`-ré. Ez kerekítést jelent a nulla irányába (pozitív számokat lefelé, negatív számokat fölfelé). További egészre kerekítési lehetőségekről lejjebb írunk.
- A könyvtár bármely számtípusa explicit módon konvertálható lebegőpontos típusokká: `float`, `double` és `long double`. Ilyenkor közelítő értéket számol. Ugyanezt csinálja az `approx()` tagfüggvény is (lásd lejjebb).

Egészre kerekítés

A könyvtár számtípusait (az `integer` kivételével) lehet egészre kerekíteni, az alábbi négyféle módon:
`integer`-ré konvertálás – nulla felé kerekít: pozitív számokat lefelé, negatív számokat fölfelé.
`integer` **`floor`** `(type const&);` – lefelé kerekít.
`integer` **`ceil`** `(type const&);` – fölfelé kerekít.
`integer` **`round`** `(type const&);` – a legközelebbi egész szám felé kerekít. Ha két szomszédos egész szám között félúton van, akkor a nullától távolodva kerekít, azaz pozitív számokat fölfelé, negatív számokat pedig lefelé.

Lebegőpontos kiértékelés

`double type::`**`approx`**` () const;` – kiszámítja a szám lebegőpontos közelítését.

3.3. integer

Az `integer` egész számokat ($\mathbb{Z}$) ábrázoló típus.

Ellentétben a beépített elemi egész számokkal (`int`, `long` stb.), amelyek csak egy előre definiált tartományban lévő számokat képesek ábrázolni (pl. `-2147483648`-tól `+2147483647`-ig, de ez gépfüggő), az `integer` számok mérete korlátlan, pontosabban az egyetlen korlát a memória mérete és a számítási kapacitás.

Az `integer` osztály támogatja mindazokat a közös számműveleteket, amelyeket fent írtunk, illetve még az alábbiakat:

Maradékos osztás

Az `integer`-en – az elemi egész számokhoz hasonlóan – definiált a maradékos osztás: az `a / b` kiszámítja az osztás hányadosát, az `a % b` pedig a maradékot. Ezeknek használható az értékadásos változata is (`a /= b`, `a %= b`). Van továbbá egy függvény, amely a maradékos osztás két eredményét egyszerre számítja ki, ezáltal hatékonyabb, mintha a két műveletet használnánk külön-külön:

```
void divide (
    integer const& lhs, integer const& rhs,
    integer* quot = nullptr, integer* rem = nullptr
);
```
A függvény elosztja `lhs`-t `rhs`-sel, és a hányadost `*quot`-ba teszi, a maradékot pedig `*rem`-be (amelyiket megadjuk).

Legnagyobb közös osztó

`integer gcd (integer const& lhs, integer const& rhs);`
Kiszámítja a két szám legnagyobb közös osztóját. Mindig pozitív eredményt ad vissza, ha nem mindkét paraméter nulla. Itt `gcd(0, 0) == 0`.

3.4. rational

A `rational` típus racionális számokat ($\mathbb{Q}$) ábrázol.

Az osztály a fent leírt közös számműveleteken kívül az alábbiakat támogatja:

Létrehozás

`rational::rational (integer const& num, integer const& den);`
Létrehozza a `num / den` racionális számot. Feltétel: `den != 0`.

Egyszerű lekérdezések

`integer numer () const;`
`integer denom () const;`
Visszaadják a racionális szám számlálóját és nevezőjét a tört egyszerűsített alakjából. Ebben az alakban a nevező mindig pozitív.

3.5. polynomial

A `polynomial<type>` egyváltozós polinomokat ábrázol, ahol `type` az együtthatók típusa. Ez utóbbi tetszőleges olyan típus lehet, amelyeket a szokásos módon lehet összeadni, kivonni és szorozni (pontosabban az a feltétel, hogy integritási tartomány legyen). Ilyen bármely beépített egész vagy lebegőpontos típus, ilyen a könyvtár összes számtípusa, továbbá ilyen maga a `polynomial` is, ezáltal lehet rekurzívan többváltozós polinomokat is definiálni.

Létrehozás

`polynomial::polynomial ();` – létrehozza az azonosan nulla polinomot.
`polynomial::polynomial (type const&);` – létrehoz egy konstans polinomot.
`polynomial::polynomial (type const& c0, type const& c1);` – létrehoz egy legfeljebb elsőfokú polinomot: $\texttt{c1} \cdot x + \texttt{c0}$.
`polynomial::polynomial (type const& c0, type const& c1, type const& c2);` – létrehoz egy legfeljebb másodfokú polinomot: $\texttt{c2} \cdot x^2 + \texttt{c1} \cdot x + \texttt{c0}$.
`polynomial::polynomial (std::initializer_list<type> list);` – a megadott

együtthatókból létrehoz egy polinomot. A lista első eleme a konstans tag, a következő az elsőfokú tag, és így tovább. Példa: a `polynomial<integer>{-1, -3, 0, 1}` kifejezés létrehozza az $x^3 - 3x - 1$ polinomot.
`static polynomial::power (unsigned n);` – létrehozza az x^n polinomot.
`static polynomial::power (unsigned n, type const& c);` – létrehozza a `c`$\cdot x^n$ polinomot.

Konvertálás

A polinom konvertálható más együtthatótípusú polinommá, ha az együtthatók típusa konvertálható. Például a `polynomial<int>` konvertálható `polynomial<integer>`-ré.

Egyszerű műveletek

`int polynomial::degree () const;` – visszaadja a polinom fokszámát. Konstans nulla polinomra `-1`-et ad.
`type const& polynomial::coeff (unsigned k) const;` – visszaadja a `k`-adfokú tag együtthatóját. Feltétel: `k <= degree()`.
`type const& polynomial::lcoeff () const;` – visszaadja a főegyütthatót. Feltétel: a polinom nem azonosan nulla.
`type const& polynomial::tcoeff () const;` – visszaadja a konstans tagot. Feltétel: a polinom nem azonosan nulla.
`void polynomial::set_coeff (unsigned k, type const& cf);` – a `k`-adfokú tag együtthatóját beállítja `cf`-re. Itt `k` tetszőleges természetes szám lehet.

Aritmetika

A számokhoz hasonlóan a polinomokon is elvégezhetők az alapműveletek:

- összeadás (`a + b`), kivonás (`a - b`) és szorzás (`a * b`);
- maradékos osztás: az `a / b` kifejezés kiszámítja a hányadost, az `a % b` pedig a maradékot;
- ezek értékadásos változatai (`a += b`, `a -= b`, `a *= b`, `a /= b`, `a %= b`);
- előjelképzés (`+a`, `-a`).

A maradékos osztásnál ha az együtthatógyűrű nem test (pl. egész számok), akkor az osztás előtt az osztandót felszorozza annyival, hogy az osztás elvégezhető legyen.

Van továbbá egy függvény, amely a maradékos osztás két eredményét egyszerre számítja ki, ezáltal hatékonyabb, mint a két művelet (`a / b` és `a % b`) külön-külön:

```
void divide (
    polynomial<type> const& lhs, polynomial<type> const& rhs,
    polynomial<type>* quot = nullptr, polynomial<type>* rem = nullptr,
    type* denom = nullptr
);
```

A függvény elosztja `lhs`-t `rhs`-sel maradékosan, és a hányadost `*quot`-ba teszi, a maradékot pedig `*rem`-be (amelyiket megadjuk). Ha megadjuk a `denom`-ot is, akkor ide írja, hogy mennyivel kellett felszorozni `lhs`-t, hogy az osztás elvégezhető legyen (ha nem kellett, akkor 1 lesz).

Egyenlőségvizsgálat

A polinomokra a számokhoz hasonlóan működik az egyenlőség: `a == b`, `a != b`.

Egyéb

Behelyettesítés: a `p(x)` kifejezés kiszámítja a `p` polinom helyettesítési értékét az `x` helyen. Az `x` nemcsak a polinom együtthatótípusának megfelelő érték lehet, hanem bármi, amit az együtthatókkal össze lehet adni és szorozni, akár polinom is.

`polynomial polynomial::`**`derivative`**` (unsigned k = 1) const;` – kiszámítja a polinom deriváltját. Ha megadunk egy `k` természetes számot, akkor annyiadik deriváltját.

`type polynomial::`**`simplify`**` ();` – egyszerűsíti a polinomot. Test fölött ez azt jelenti, hogy leosztja a főegyütthatóval, egyébként pedig (pl. egészeknél) leosztja az együtthatók legnagyobb közös osztójával. Az egyszerűsítés után a főegyüttható egységnormált lesz (pl. egészek esetén pozitív). A visszatérési értékben megadja, hogy mennyivel osztotta le a polinomot. Konstans nulla esetén nem történik semmi, és 1-et ad vissza.

`type polynomial::`**`content`**` () const;` – visszaadja azt az értéket, amellyel a polinomot egyszerűsíteni lehet. Ugyanazt adja vissza, mint a `simplify()`, csak a polinomot nem módosítja.

Polinomműveletek

`polynomial`**` gcd`**` (polynomial const& lhs, polynomial const& rhs);`
Kiszámítja a két polinom legnagyobb közös osztóját. Az eredmény egyszerűsített (`simplify()`) lesz. Ha mindkét paraméter konstans nulla, akkor az eredmény is nulla lesz.

```
polynomial gcd_ex (
    polynomial const& lhs, polynomial const& rhs,
    polynomial* lb = nullptr, polynomial* rb = nullptr
);
```

Kiszámítja a két polinom legnagyobb közös osztóját és a Bézout-együtthatókat, vagyis azokat a legalacsonyabb fokú `lb` és `rb` polinomokat, melyekre `lb*lhs + rb*rbs == gcd(lhs, rhs)`. Amelyik eredményparamétert nem adjuk meg, azt nem számítja ki.

`type`**` resultant`**` (polynomial const& lhs, polynomial const& rhs);`
Kiszámítja a két polinom rezultánsát.

Polinomfaktorizáció

`polynomial::factorization polynomial::`**`factor`**` () const;`
Faktorizálja a polinomot, azaz felírja szorzatalakban: $f = cf_1^{k_1} f_2^{k_2} \dots f_m^{k_m}$, ahol az f_i-k páronként különböző felbonthatatlan, egyszerűsített (`simplify()`) polinomok, c pedig egy konstans.
A visszaadott `polynomial::factorization` objektum úgy használható, mint egy tömb (`size()`, `fact[i]`, `begin()`, `end()`), melynek elemei a különböző faktorok. Egy faktor egy `polynomial::factor` objektum, amely `polynomial`-ként viselkedik, de van egy `exp` adattagja is, a kitevő. A `factorization`-nek van továbbá egy `content` adattagja is, amely a c szorzó értékét tárolja.
A faktorizáció nem minden együtthatótípusra van definiálva, az itt tárgyaltak közül csak az `integer`-re és az `algnumber`-re.

3.6. algebraic

Az `algebraic` típus valós algebrai számokat ábrázol ($\mathbb{A} \cap \mathbb{R}$).

Ezzel a típussal a számolás lassú. Hatékonyabb, ha ilyen számokból létrehozunk egy algebrai számtestet, és annak az elemeivel számolunk (lásd az `algfield`-et és az `algnumber`-t később).

Az `algebraic` támogatja a korábban tárgyalt közös számműveleteket, továbbá:

Létrehozás

```
explicit algebraic::algebraic (polynomial<integer> const&);
algebraic::algebraic (polynomial<integer> const&,
    infed<rational> const& a, infed<rational> const& b);
```

Létrehozáskor meg kell adni egy polinomot, amelynek gyöke az algebrai szám.
Megadhatunk továbbá egy zárt intervallumot (második változat), amely kijelöli a polinom megfelelő gyökét. A két végpontját kell megadni, amelyek vagy racionális számok (`rational`), vagy valamelyik lehet végtelen (`inf` vagy `-inf`). Feltétel, hogy ebben az intervallumban a polinomnak pontosan egy gyöke legyen.
Az első változatban a polinomnak pontosan egy valós gyöke lehet, vagy ha több is van, akkor csak egy pozitív valós gyöke lehet.
A megadott polinom nem kell, hogy felbonthatatlan legyen, és a gyöknek nem kell egyszeresnek lennie. Ha a polinomnak nincs gyöke a megadott intervallumban, akkor `algebraic::no_roots` típusú kivételt dob, ha pedig több (különböző) gyöke is van, akkor `algebraic::multiple_roots` kivételt kapunk.

Egyszerű lekérdezések

`polynomial<integer> algebraic::minpoly () const;` – minimálpolinom
`unsigned algebraic::degree () const;` – az algebrai szám fokszáma

Gyökvonás

`algebraic sqrt (algebraic const& x);` – négyzetgyök ($\sqrt{x}$)
`algebraic cbrt (algebraic const& x);` – köbgyök ($\sqrt[3]{x}$)
`algebraic root (algebraic const& x, unsigned n);` – `n`-edik gyök ($\sqrt[n]{x}$)

Behelyettesítés polinomba

```
algebraic algebraic::eval (polynomial<integer> const& pol) const;
```

Behelyettesíti az algebrai számot a megadott polinomba. Hatékonyabb, mint elvégezni a megfelelő szorzásokat és összeadásokat.

3.7. algfield

Az `algfield` egy algebrai számtest.

A testet egy vagy több algebrai számból (`algebraic`) lehet létrehozni. Elemei `algnumber` típusúak, amelyekkel sokkal hatékonyabb számolni, mint az `algebraic` típussal közvetlenül.

Az `algfield` speciális esetben lehet maga a racionális számtest is, ha nem adunk meg bővítő algebrai számot.

Létrehozás

`algfield::`**`algfield`**`();` – paraméter nélkül a racionális számtestet ($\mathbb{Q}$) hozza létre.
`explicit algfield::`**`algfield`**`(algebraic const&);` – a racionális számtestet egyetlen elemmel bővíti ($\mathbb{Q}(\alpha)$).
`algfield::`**`algfield`**`(unsigned k, algebraic const*);` – a $\mathbb{Q}$-t a megadott `k` elemmel bővíti ($\mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_k)$). A második paraméter egy tömb kezdőcíme, amelyben a `k` algebrai szám szerepel.

Szám létrehozása a testből

`algnumber algfield::`**`create`**`(integer const&) const;`
`algnumber algfield::`**`create`**`(rational const&) const;`
`algnumber algfield::`**`create`**`(algebraic const&) const;`
Létrehoznak az adott testből egy `algnumber`-t, amelynek az értéke a megadott egész szám, racionális szám vagy algebrai szám lesz. Az algebrai szám (`algebraic`) esetén ha a szám nincs benne a testben, akkor `algfield::not_in_field` kivételt dob.

3.8. algnumber

Az `algnumber` egy algebrai számtest (`algfield`) eleme.

Minden `algnumber` objektum egyértelműen valammelyik `algfield`-hez tartozik, és műveleteket végezni csak azonos testhez tartozó számok között lehet. Ezalól kivételt képeznek a racionális számok, mert az ilyen `algnumber` objektumoknak nem kell feltétlenül valamely `algfield`-hez tartozniuk, és bármely más `algnumber`-rel használhatók közös műveletben.

Az `algnumber` a fent leírt közös számműveleteket támogatja.

`algnumber`-t létrehozni az `algfield::create()`-tel lehet (azonkívül, hogy racionális számok konvertálhatók közvetlenül `algnumber`-ré).

3.9. error

A `<san/error.hpp>` definiálja azokat a kivételosztályokat, amelyeket hibák esetén dob a könyvtár:

- `error`: a hibaosztályok közös őse.
- `division_by_zero`: nullával osztás történt a könyvtár valamely típusánál.
- `negative_sqrt`: négyzetgyökvonás negatív számból. Például az `algebraic`-re vonatkozó `sqrt()` dobhat ilyet.
- `negative_root`: páros kitevőjű gyökvonás negatív számból. Ősosztálya a `negative_sqrt`-nek.

4. A könyvtár működése

Ebben a fejezetben a könyvtár működéséről lesz szó: az egyes típusok ábrázolásáról és a fontosabb műveletek megvalósításáról.

4.1. Segédtípusok

4.1.1. `integer`

Az egész számokhoz helyiértékes ábrázolást használunk:

$$a = (-1)^s \sum_{i=0}^{n-1} a_i B^i \in \mathbb{Z},$$

ahol $s \in \{0, 1\}$ az előjelbit, $a_i \in [\,0 \,..\, B-1\,]$-k a számjegyek, és B a számrendszer alapszáma. Ez utóbbi a gépi szó méretének megfelelő kettőhatvány, azaz akkora, hogy egy a_i éppen beleférjen egy gépi szóba. Nemnulla szám esetén a legnagyobb helyiértékű tárolt számjegy (a_{n-1}) nem nulla. A nullát nulla darab számjeggyel ábrázoljuk, és ekkor s tetszőleges (vagyis ebben az egy esetben az ábrázolás nem egyértelmű).

4.1.2. `rational`

A racionális számokat számlálóval és nevezővel ábrázoljuk:

$$q = \frac{a}{b} \in \mathbb{Q},$$

ahol $a, b \in \mathbb{Z}$, mindkettő `integer` típusú. A tört mindig egyszerűsített, és a nevező mindig pozitív. Ez biztosítja az egyértelműséget, és hogy minél kisebb számokat használjunk az ábrázoláshoz, viszont így minden művelet után egyszerűsíteni kell az eredményt (ehhez legtöbbször legnagyobb közös osztót kell számítani).

A továbbiakban a racionális számok ezen ábrázolás szerinti számlálóját és nevezőjét numer(q)-val és denom(q)-val jelöljük.

4.1.3. `polynomial`

A polinomokat együtthatók sorozatával ábrázoljuk, sűrű ábrázolással:

$$f = \sum_{i=0}^{n} a_i x^i \in R[x],$$

ahol $a_i \in R$ a tárolt együtthatók és $n = \deg f$. A konstans nulla polinomot nulla darab együtthatóval ábrázoljuk.

A `polynomial` fontosabb műveleteinek megvalósítása:

- Alapműveletek: a kézzel, írásban történő számolásnak megfelelő módon.
- Behelyettesítés a polinomba: Horner-módszerrel.
- Legnagyobb közös osztó: euklideszi algoritmussal.
- Rezultáns: a szubrezultáns algoritmussal. [1, 122. o.]
- Faktorizáció egészek fölött: Cantor–Zassenhaus-algoritmussal $\mathbb{Z}_p[x]$-ben, majd kiterjesztés Hensel-felvonással $\mathbb{Z}[x]$-be. [6, 368-376. o.]

4.2. `algebraic`

Az `algebraic` általános algebrai számot ábrázol, ezért a 2. fejezetben bemutatott ábrázolási módok közül a polinom gyökeként való ábrázolást alkalmazzuk. A gyök azonosítására a konjugáltak között alapvetően izolációs intervallumot használunk, de további feltételekkel és kiegészítésekkel.

Egészen pontosan egy $\alpha \in \mathbb{A} \cap \mathbb{R}$ algebrai számot az alábbiak szerint ábrázolunk:

$$\alpha \mapsto (f, [\,a, b\,], s),$$

ahol:

1. $f \in \mathbb{Z}[x]$ az α minimálpolinomja. Legyen $f = \sum_{i=0}^{n} a_i x^i$, ahol $a_i \in \mathbb{Z}$ az együtthatók és $n = \deg f \geq 1$. Az egyértelmű ábrázolás miatt (a konstans szorzót is figyelembe véve) megköveteljük, hogy primitív polinom legyen, azaz:
   (a) $\gcd(a_0, a_1, \ldots, a_n) = 1$, és
   (b) $a_n > 0$.
2. $[\,a, b\,] \subset \mathbb{R}$ izolációs intervallum, az alábbi további megszorításokkal:
   (a) $\forall k \in \{1, 2, \ldots, n\}, \forall x \in [\,a, b\,] : f^{(k)}(x) \neq 0$. Azaz nemcsak az f-nek nem lehet α-n kívül más gyöke az intervallumban, hanem a deriváltjainak sem lehet zérushelye ott. Minden izolációs intervallum leszűkíthető úgy, hogy ez teljesüljön, ugyanis a deriváltaknak csak véges sok gyöke lehet, és α nem lehet gyöke (mert f a legalacsonyabbfokú ilyen), következésképp α-nak van olyan környezete, amelyben semelyik deriváltnak sincs gyöke.
   (b) Az intervallum végpontjai bináris racionális számok, azaz olyan racionális számok, amelyeknek a nevezője kettőhatvány. Ezáltal hatékonyabb velük a számolás, mert nincs szükség legnagyobb közös osztók számítására. A két végpontot közös nevezővel tároljuk, azaz az intervallum $\dfrac{[\,a_0, b_0\,]}{2^k}$ alakú, ahol $a_0, b_0 \in \mathbb{Z}$ és $k \in \mathbb{N}$.
   (c) Ha α maga is bináris racionális szám, akkor pontintervallumot használunk: $[\,a, b\,] = [\,\alpha, \alpha\,]$.

3. $s = \langle s_1, s_2, \ldots, s_{n-1} \rangle \in \{-1, 1\}^{n-1}$ előjelsorozat, amely megadja a deriváltak előjeleit a gyökben: $s = \left\langle \operatorname{sgn} f'(\alpha), \operatorname{sgn} f''(\alpha), \ldots, \operatorname{sgn} f^{(n-1)}(\alpha) \right\rangle$. A 2.2.2.2. alfejezetben láttuk, hogy s önmagában is elegendő lenne α azonosítására a konjugáltak között, de itt csak kiegészítő adatként használjuk az izolációs intervallum mellett. Az előjelek, ellentétben az intervallummal, adott α esetén egyértelműek, így például az egyenlőségvizsgálat egyszerűbb velük.
   Az a feltétel, hogy a deriváltaknak nem lehet gyöke az intervallumban, azt biztosítja, hogy a deriváltak előjelei a teljes intervallumon megegyezzenek az s előjelekkel. Ez megkönnyít bizonyos műveleteket, például az α lebegőpontos közelítését (lásd a 4.2.5. alfejezetet).

Most lássuk az `algebraic` fontosabb műveleteinek a megvalósítását. Az itt szereplő algoritmusok a kódhoz képest leegyszerűsítettek és csak a lényeget mutatják be. A kódban ezek további ellenőrzésekkel, optimalizálásokkal egészülnek ki (például a racionális számokat külön esetként kezeljük).

4.2.1. Létrehozás

Az előző fejezetben láttuk, hogy az `algebraic` létrehozható egy egész együtthatós polinomból, amelynek gyöke, és megadható hozzá egy izolációs intervallum. Ezek azonban nem feltétlenül teljesítik a fenti feltételeket, például nemcsak felbonthatatlan polinomot lehet megadni, illetve az intervallum végpontjai nemcsak bináris racionális számok, hanem tetszőleges racionális számok lehetnek, vagy akár végtelenek is. Ezért létrehozáskor az adatokat a megfelelő alakúra hozzuk, a következő oldalon látható 1. algoritmussal (a kódban az `algebraic::init()` segédfüggvénnyel).

4.2.2. Alapműveletek

Az algebrai számok közötti alapműveletek elvégzésének két része van. Először kiszámítunk egy polinomot, amelynek gyöke a keresett érték. Aztán kiszámítjuk az eredmény további adatait (intervallumát és az előjeleket). Ha a kapott polinomról nem tudjuk biztosan, hogy felbonthatatlan, akkor a második részhez hozzátartozik az is, hogy kiválasztjuk a polinomnak azt a faktorát, amelynek gyöke a szám.

Az alapműveletek algoritmusát az összeadás példáján mutatjuk be. Az algoritmus első részét, a polinom előállítását az egyes műveletekre a 2.2.1. alfejezet táblázatában láthatjuk. A második rész, tehát az intervallum és az előjelek kiszámítása pedig a legtöbb műveletnél nagyon hasonló, vagy ahol a kapott polinom eleve felbonthatatlan, ott jóval egyszerűbb. Ez a második rész a kódban az `algebraic::isolate_factor()` segédfüggvényben található, az egyszerűbb (faktorizáció nélküli) változat pedig az `algebraic::isolate()`-ben.

Adottak a bemenő algebrai számok az ábrázoló adataikkal: $\beta \mapsto (f_\beta, [\, a_\beta, b_\beta \,], s_\beta)$ és $\gamma \mapsto (f_\gamma, [\, a_\gamma, b_\gamma \,], s_\gamma)$. Ezekből szeretnénk kiszámítani az $\alpha := \beta + \gamma$ szám ábrázolását: $\alpha \mapsto (f, [\, a, b \,], s)$. Az első részben kiszámítunk egy $\tilde{f}$ polinomot, amelynek gyöke α, az említett táblázatban lévő képlettel. A második részben ezt faktorizáljuk, és kiválasztjuk a megfelelő felbonthatatlan f faktort, közben pedig kiszámítjuk az $[\, a, b \,]$ izolációs invervallumot és az s előjeleket.

1. algoritmus: Létrehozás

Bemenet: $\tilde{f} \in \mathbb{Z}[x]$ és $[\,\tilde{a}, \tilde{b}\,]$, ahol $\tilde{a}, \tilde{b} \in \mathbb{Q} \cup \{-\infty, +\infty\}$
Kimenet: a feltételeknek megfelelő $(f, [\,a, b\,], s)$

Gyökszám ellenőrzése:

$S := \text{sturm\_sequence}(\tilde{f})$
$r := \text{sign\_changes}(S(\tilde{a})) - \text{sign\_changes}(S(\tilde{b}))$
if $r > 1$ **then**
  Hiba: több gyök van.
if $r = 0$ **then**
  Hiba: nincs gyök.

Megfelelő faktor kiválasztása:

$c f_1^{k_1} f_2^{k_2} \dots f_m^{k_m} := \text{factor}(\tilde{f})$
for $i := 1$ **to** m **do**
  if $\operatorname{sgn} f_i(\tilde{a}) \neq \operatorname{sgn} f_i(\tilde{b})$ **then**
    $f := f_i$
    break

Végpontok bináris racionális számok legyenek:

if $\tilde{b} = +\infty$ **then**
  $\tilde{b} := 2$
  while $\operatorname{sgn} f(\tilde{b}) \neq +1$ **do**
    $\tilde{b} := \tilde{b}^2$

if $\tilde{a} = -\infty$ **then**
  $\tilde{a} := -2$
  while $\operatorname{sgn} f(\tilde{a}) \neq (-1)^{\deg f}$ **do**
    $\tilde{a} := -\tilde{a}^2$

if $\log_2 \operatorname{denom}(\tilde{a}) \in \mathbb{Z}$ **and** $\log_2 \operatorname{denom}(\tilde{b}) \in \mathbb{Z}$ **then**
  $[\,a, b\,] := [\,\tilde{a}, \tilde{b}\,]$
else
  $d := 1$
  $[\,a, b\,] := \left[\,\lceil \tilde{a} \rceil, \lfloor \tilde{b} \rfloor\,\right]$
  while $[\,a, b\,] = \emptyset$ **or** $\operatorname{sgn} f(a) = \operatorname{sgn} f(b)$ **do**
    $d := 2d$
    $[\,a, b\,] := \left[\,\dfrac{\lceil \tilde{a} d \rceil}{d}, \dfrac{\lfloor \tilde{b} d \rfloor}{d}\,\right]$

Előjelek kiszámítása és az intervallum előjelfeltétele:

$s_a := \left\langle \operatorname{sgn} f'(a), \operatorname{sgn} f''(a), \dots, \operatorname{sgn} f^{(n-1)}(a) \right\rangle$
$s_b := \left\langle \operatorname{sgn} f'(b), \operatorname{sgn} f''(b), \dots, \operatorname{sgn} f^{(n-1)}(b) \right\rangle$
while $s_a \neq s_b$ **do**
  $c := \dfrac{a+b}{2}$
  $s_c := \left\langle \operatorname{sgn} f'(c), \operatorname{sgn} f''(c), \dots, \operatorname{sgn} f^{(n-1)}(c) \right\rangle$
  if $\operatorname{sgn} f(c) \neq \operatorname{sgn} f(a)$ **then**
    $b := c, \quad s_b := s_c$
  else
    $a := c, \quad s_a := s_c$
$s := s_a$

Az $[a, b]$-t a bemenő két intervallumból, $[a_\beta, b_\beta]$-ból és $[a_\gamma, b_\gamma]$-ból számítjuk ki: $[a, b] := [a_\beta + a_\gamma, b_\beta + b_\gamma]$. (Más műveleteknél ez a sor módosul.) Az így kapott $[a, b]$ azonban még nem feltétlenül izolációs intervallum, vagy ha igen, nem feltétlenül teljesíti az `algebraic` ábrázolásához megkövetelt feltételeket. Ha $[a, b]$ még nem megfelelő, akkor szűkítjük a bemenő intervallumokat, majd azokból újra kiszámítjuk $[a, b]$-t. Ezt addig folytatjuk, amíg a feltételeknek megfelelő izolációs intervallumot nem kapunk.

Az algoritmus először faktorizálja az $\tilde{f}$ polinomot, majd végigmegy a különböző faktorokon fokszám szerint növekvő sorrendben. Minden f_i faktornál addig szűkíti az $[a, b]$ intervallumot a fenti módon, amíg abban legfeljebb csak egy gyöke marad f_i-nek, és amíg az nem teljesíti az előjelfeltételt, azaz a-ban és b-ben a deriváltak előjelei meg nem egyeznek.

A gyökök számának megállapításához a legkézenfekvőbb eszköz a Sturm-tétel lenne (lásd a 2.2.2.1. alfejezetben). Ehelyett egy olyan módszert használunk, amelyhez nem kell kiszámítani segédpolinomokat: eleve csak a második feltételt, a deriváltak előjeleit vizsgáljuk, mert ha azok a két végpontban megegyeznek, akkor f_i-nek legfeljebb csak egy gyöke lehet $[a, b]$-ben (lásd a Budan–Fourier-tételt ugyanott). A kérdés az, hogy ez a módszer megáll-e előbb-utóbb. Akkor nem áll meg, ha $[a, b]$-ben már legfeljebb csak egyetlen gyök van, de bármennyire is szűkítjük $[a, b]$-t, a deriváltelőjelek nem fognak megegyezni a két végpontban. Ez akkor fordulhat elő, ha α gyöke f_i valamelyik deriváltjának. Mivel azonban fokszám szerint növekvő sorrendben haladunk, ezért ha semelyik korábbi faktornak nincs gyöke $[a, b]$-ben, akkor α minimálpolinomjának fokszáma legalább $\deg f_i$, vagyis a deriváltjainak nem lehet gyöke α.

Miután addig szűkítettük az $[a, b]$-t, hogy f_i-nek már legfeljebb csak egy gyöke maradt benne, és teljesíti az előjelfeltételt, azután egy egyszerű előjelvizsgálattal eldönthető, hogy ténylegesen van-e itt gyöke. Ha nincs, akkor egyszerűen megyünk tovább f_{i+1}-re. Ha van, akkor még nem biztos, hogy ez tényleg α, mert lehet, hogy az α majd egy későbbi faktornak lesz gyöke, és csak bennmaradt az intervallumban f_i-nek egy gyöke, amely további szűkítésre kiesne. Ezt azonban itt még nem tudjuk eldönteni, ezért eltároljuk az f_i faktort f-be, és így megyünk tovább f_{i+1}-re. Ha később nem találunk más jó faktort, akkor az eltárolt f-nek tényleg az α volt a gyöke.

Ha f-be már eltároltunk egy faktort, akkor az ezután következő f_i-knél az $[a, b]$-t szigorúbb feltételekkel szűkítjük: azt is vizsgáljuk, hogy az f és az f_i közül csak az egyiknek maradjon gyöke $[a, b]$-ben. Amelyiknek marad, azt tároljuk el f-ben. Ilyenkor azonban van egy probléma a szűkítés fent vázolt módjával: most van f_i-t megelőző faktor, amelynek gyöke lehet α, így előfordulhat, hogy α gyöke f_i valamelyik deriváltjának. Ilyen esetben lehet, hogy sosem fognak megegyezni a deriváltak előjelei a két végpontban. Ezért ebben a speciális esetben néhány lépés után mégis áttérünk a Sturm-tételre, és azzal már pontosan tudni fogjuk, hogy hány gyök van az intervallumban.

2. algoritmus: Két algebrai szám összege

Bemenet: $(f_\beta, [\,a_\beta, b_\beta\,], s_\beta), (f_\gamma, [\,a_\gamma, b_\gamma\,], s_\gamma)$
Kimenet: $(f, [\,a, b\,], s)$
$\tilde{f}(x) := \mathrm{res}_y(f_\beta(y), f_\gamma(x-y))$
$cf_1^{k_1} f_2^{k_2} \dots f_m^{k_m} := \mathrm{factor}(\tilde{f})$
$\mathrm{sort}(\langle f_1, f_2, \dots, f_m\rangle, \deg)$ *(rendezés fokszám szerint növekvő sorrendbe)*
$f :=$ **null**
for $i := 1$ **to** m **do**
  $n := \deg f_i$
  $[\,a, b\,] := [\,a_\beta + a_\gamma, b_\beta + b_\gamma\,]$
  $s_a := \left\langle \mathrm{sgn}\, f_i'(a), \mathrm{sgn}\, f_i''(a), \dots, \mathrm{sgn}\, f_i^{(n-1)}(a) \right\rangle$
  $s_b := \left\langle \mathrm{sgn}\, f_i'(b), \mathrm{sgn}\, f_i''(b), \dots, \mathrm{sgn}\, f_i^{(n-1)}(b) \right\rangle$
  $S :=$ **null** *(Sturm-sorozat, ha majd kell)*
  if $s_a = s_b$ **then**
    $r :=$ (**if** $\mathrm{sgn}\, f_i(a) \neq \mathrm{sgn}\, f_i(b)$ **then** 1 **else** 0)
  else
    $r := \infty$ *(több gyök lehet, nem tudjuk pontosan)*
  $r_f :=$ (**if is** f **and** $\mathrm{sgn}\, f(a) \neq \mathrm{sgn}\, f(b)$ **then** 1 **else** 0)
  $j := 0$
  while $r + r_f > 1$ **do**
    if is f **and** $r = \infty$ **and** $j \geq 2$ **then**
      $k := \max\{\, i = 1 \,..\, n-1 \mid (s_a)_i \neq (s_b)_i \,\}$
      if $f \mid f_i^{(k)}$ **then**
        $S := \mathrm{sturm\_sequence}(f_i)$
    $\mathrm{tighten\_intervals}((f_\beta, [\,a_\beta, b_\beta\,], s_\beta), (f_\gamma, [\,a_\gamma, b_\gamma\,], s_\gamma))$
    $[\,a, b\,] := [\,a_\beta + a_\gamma, b_\beta + b_\gamma\,]$
    $s_a := \left\langle \mathrm{sgn}\, f_i'(a), \mathrm{sgn}\, f_i''(a), \dots, \mathrm{sgn}\, f_i^{(n-1)}(a) \right\rangle$
    $s_b := \left\langle \mathrm{sgn}\, f_i'(b), \mathrm{sgn}\, f_i''(b), \dots, \mathrm{sgn}\, f_i^{(n-1)}(b) \right\rangle$
    if $s_a = s_b$ **then**
      $r :=$ (**if** $\mathrm{sgn}\, f_i(a) \neq \mathrm{sgn}\, f_i(b)$ **then** 1 **else** 0)
    else
      if is S **then**
        $r := \mathrm{sign\_changes}(S(a)) - \mathrm{sign\_changes}(S(b))$
      else
        $r := \infty$ *(több gyök lehet, nem tudjuk pontosan)*
    if is f **then**
      $r_f :=$ (**if** $\mathrm{sgn}\, f(a) \neq \mathrm{sgn}\, f(b)$) **then** 1 **else** 0)
      if $r_f = 0$ **then**
        $f :=$ **null**, $S :=$ **null**
    $j := j + 1$
  if $r = 1$ **then**
    $f := f_i$
    $s := s_a$

4.2.3. Összehasonlítás

Algebrai számok egyenlősége a polinom és az előjelek összehasonlításával könnyen eldönthető. Különböző algebrai számok esetén az intervallumok szűkítésével tudjuk eldönteni, hogy melyik a nagyobb.

Az alábbi algoritmus összehasonlít két algebrai számot. A kódban ez a `compare()` függvényben található.

3. algoritmus: Két algebrai szám összehasonlítása

Bemenet: $(f_\alpha, [\,a_\alpha, b_\alpha\,], s_\alpha), (f_\beta, [\,a_\beta, b_\beta\,], s_\beta)$
Kimenet: az összehasonlítás eredménye: $\operatorname{sgn}(\alpha - \beta) \in \{-1, 0, +1\}$
if $f_\alpha = f_\beta$ **and** $s_\alpha = s_\beta$ **then**
  return 0
else
  repeat
    if $b_\alpha \leq a_\beta$ **then**
      return -1
    if $a_\alpha \geq b_\beta$ **then**
      return $+1$
    tighten_intervals$((f_\alpha, [\,a_\alpha, b_\alpha\,], s_\alpha), (f_\beta, [\,a_\beta, b_\beta\,], s_\beta))$

4.2.4. Egészre kerekítés

Algebrai szám egészre kerekítését ugyancsak az intervallumok szűkítésével oldjuk meg. Az alábbi algoritmus a lefelé kerekítést mutatja (a többi hasonlóan működik). A kódban a kerekítések közös megvalósítása az `algebraic::to_integer_impl()` segédfüggvényben található.

4. algoritmus: Algebrai szám egészre kerekítése (lefelé)

Bemenet: $(f, [\,a, b\,], s)$
Kimenet: $x \in \mathbb{Z}$
while $\lceil b \rceil - \lfloor a \rfloor > 1$ **do**
  tighten_interval$(f, [\,a, b\,], s)$
$x := \lfloor a \rfloor$

4.2.5. Közelítő érték

A szám lebegőpontos közelítését a következő oldalon látható algoritmussal számítjuk ki (a kódban ez az `algebraic::approx()` függvényben szerepel).

Általános esetben a Newton-iterációt használjuk. Kihasználjuk, hogy az izolációs intervallumban a deriváltak előjelei végig azonosak. A Newton-iteráció monoton konvergenciatétele miatt ilyenkor az egyik (megfelelő) végpontból indítva konvergens lesz az iteráció [9, 334.o.].

Speciális esetként az első- és másodfokú esetre egyszerűbb megoldást adunk. A másodfokú egyenlet megoldóképletét kicsit átalakítottuk: ahol két közeli számot vonnánk ki egymásból, ott a numerikus stabilitás miatt inkább gyöktelenítettük a számlálót.

5. algoritmus: Algebrai szám lebegőpontos közelítése

Bemenet: $(f, [a, b], s)$
Kimenet: $x \in \mathbb{R}$ közelítés
if $\deg f = 1$ **then**
  $x := -f_0/f_1$
else if $\deg f = 2$ **then**
  if $f_1 < 0$ **then**
    $q := -f_1 + \sqrt{f_1^2 - 4f_0f_2}$
    $x :=$ (**if** $s_1 = -1$ **then** $2f_0/q$ **else** $q/2f_2$)
  else
    $q := -f_1 - \sqrt{f_1^2 - 4f_0f_2}$
    $x :=$ (**if** $s_1 = -1$ **then** $q/2f_2$ **else** $2f_0/q$)
else
  $x :=$ (**if** $s_1 = s_2$ **then** b **else** a)
  do
    $\tilde{x} := x$
    $x := x - \dfrac{f(x)}{f'(x)}$
  while $|x - \tilde{x}| > \epsilon$

4.3. `algfield`

Az algebrai számtest ábrázolása primitív elemmel történik. A primitív elem egy `algebraic` típusú algebrai szám, amelyre az `algfield`-ben megszorításokat teszünk.

Legyen $F = \mathbb{Q}(\alpha)$ egy n-edfokú algebrai számtest, és legyen az α primitív elem minimálpolinomja $f = \sum_{i=0}^{n} a_i x^i$. Ekkor α-ra az alábbi megszorításokat tesszük:

1. Algebrai egész, azaz $a_n = 1$.
2. Egyszerűsített algebrai egész, azaz bármely $c \in \mathbb{Z}, c \geq 2$-re α/c nem algebrai egész.
3. $\alpha \geq 0$.
4. $a_{n-1} = 0$.

Minden algebrai számtestben tetszőleges primitív elemből kiindulva találunk ilyen tulajdonságú primitív elemet is: az első három feltételt racionális számmal való szorzással lehet teljesíteni, a negyediket pedig racionális szám hozzáadásával. Az a céljuk ezeknek a feltételeknek, hogy a számot minél egyszerűbb alakra hozzák. Például $n = 2$ esetben $\alpha = \sqrt{d}$, ahol d egy négyzetmentes pozitív egész szám, $n = 1$ esetben pedig (azaz ha $F = \mathbb{Q}$) $\alpha = 0$. Az első feltételnek, az algebrai egésznek további előnye van: megkönnyíti az f-re vett moduláris számolást, például az elemek szorzásánál vagy az inverz kiszámításánál. A kódban az `algfield::normalize()` segédfüggvény hozza a fenti feltételeknek megfelelő alakra a primitív elemet.

4.3.1. Primitív elem kiszámítása

Ha több bővítő elemet adunk meg, akkor ki kell számítani egy primitív elemet.

Az alábbi algoritmus két bővítő elemből határoz meg egy primitív elemet. A kódban ez az `algfield::add_algebraic()` segédfüggvényben található. Több bővítő elem esetén ismételjük az algoritmust.

6. algoritmus: Primitív elem meghatározása két bővítő elemből

Bemenet: $\beta, \gamma \in \mathbb{A} \cap \mathbb{R}$, melyeknek minimálpolinomja g és h.
Kimenet: $\alpha \in \mathbb{A} \cap \mathbb{R}$, melyre $\mathbb{Q}(\alpha) = \mathbb{Q}(\beta, \gamma)$
$s := 0$
do
  $s := s + 1$
  $\alpha := \beta + s\gamma$
  $f(x) := \gcd\left(g(x), h\left(\frac{\alpha - x}{s}\right)\right)$
while $\deg f > 1$

A 2.3.2.2. alfejezetben láttuk, hogy a ciklus véges sok lépésben véget ér. Sőt, gyakran az első s-re jó eredményt kapunk.

Az algoritmusban a legnagyobb közös osztó számítása $\mathbb{Q}(\alpha)[x]$-ben történik, vagyis az éppen létrehozandó új `algfield` elemeiből készített, `polynomial<algnumber>` típusú polinomokkal. Az eredményül kapott f polinomnak gyöke lesz a β. Mivel leálláskor az f elsőfokú lesz, ezért az $f(\beta) = 0$ egyenletből β könnyen kifejezhető, vagyis felírható a $\mathbb{Q}(\alpha)$ testben, `algnumber`-ként (és ebből a $\gamma = \frac{\alpha - \beta}{s}$ is felírható).

4.3.2. Algebrai szám felírása a testben

Adott egy $\mathbb{Q}(\alpha)$ algebrai számtest, és egy $\beta \in \mathbb{A} \cap \mathbb{R}$ algebrai szám. Szeretnénk eldönteni, hogy $\beta \overset{?}{\in} \mathbb{Q}(\alpha)$, és ha igen, akkor írjuk fel β-t az α függvényében.

Az alapmódszer a következő: vesszük a β minimálpolinomját (f_β), és faktorizáljuk $\mathbb{Q}(\alpha)$ fölött. A faktorok között lesz pontosan egy, amelynek gyöke β: a β $\mathbb{Q}(\alpha)$ fölötti minimálpolinomja ($f_{\beta,\mathbb{Q}(\alpha)}$). Ha ez a faktor elsőfokú, akkor $\beta \in \mathbb{Q}(\alpha)$, és könnyen kifejezhető a α-val. Ha nem elsőfokú, akkor a β $\mathbb{Q}(\alpha)$ fölötti foka 1-nél magasabb, tehát nincs benne a testben.

A módszert gyorsítjuk egy ötlettel, amely hasonlít a primitív elem kiszámításához: előállítjuk a $\gamma := s\alpha + \beta$ számot (minimálpolinomja f_γ), és tekintjük az $f_\gamma(x + s\alpha)$ polinomot. Ez egy $\mathbb{Q}(\alpha)$ fölötti polinom, melynek gyöke β, vagyis ennek is faktora lesz a keresett $f_{\beta,\mathbb{Q}(\alpha)}$. Vegyük ennek és f_β-nak a legnagyobb közös osztóját, ezáltal várhatóan egy f_β-nál jóval alacsonyabb fokú polinomot kapunk, amelynek szintén faktora $f_{\beta,\mathbb{Q}(\alpha)}$. Sőt, sokszor rögtön magát $f_{\beta,\mathbb{Q}(\alpha)}$-t kapjuk, és nem is kell faktorizálni. A módszer megismételhető több különböző s-re.

Ezt az alábbi algoritmusban láthatjuk. Itt csak egy s-re, az $s := 1$-re alkalmazzuk a fenti ötletet.

7. algoritmus: Algebrai szám felírása a testben

Bemenet: $\alpha, \beta \in \mathbb{A} \cap \mathbb{R}$, minimálpolinomjuk f_α és f_β
Kimenet: β felírása $\mathbb{Q}(\alpha)$-ban

```
f := f_β
if deg f > 1 then
    γ := α + β
    f(x) := gcd(f(x), f_γ(x + α))
if deg f > 1 then
    c f_1^{k_1} f_2^{k_2} ... f_m^{k_m} := factor_{Q(α)}(f)
    for i := 1 to m do
        if deg f_i = 1 and f_i(β) = 0 then
            f := f_i
            break
if deg f = 1 then
    return −f_0/f_1
else
    Hiba: nincs benne a testben.
```

Az algoritmus a kódban az `algfield::to_algnumber()` segédfüggvényben látható. Ott még további egyszerű gyorsítási ötleteket alkalmazunk. Egyrészt ha $\beta \in \mathbb{Q}(\alpha)$, akkor β fokszáma osztja $\mathbb{Q}(\alpha)$ dimenzióját, tehát ezzel az oszthatósági vizsgálattal sok $\beta \notin \mathbb{Q}(\alpha)$ eleve kizárható. Másrészt az `algfield`-ben eltárolunk néhány már ismert algebrai számot, és azok felírását a testben. Az eltárolt számokat először egyszerűsítjük az `algfield::normalize()` segédfüggvénnyel (ugyanezt használjuk a primitív elem egyszerűsítésére is). Így ha legközelebb egy olyan számot kérdezünk le, amelynek az egyszerűsített alakját már eltároltuk, akkor arra nem kell újra végrehajtani a fenti algoritmust. Az eltárolt számok között eleve szerepelnek a létrehozáskor megadott bővítő elemek is, amelyeknek a felírását automatikusan megkapjuk a primitív elem kiszámítása közben.

4.4. `algnumber`

Legyen $F = \mathbb{Q}(\alpha)$ egy n-edfokú algebrai számtest, és legyen α minimálpolinomja f. Ekkor a test elemeit az α primitív elem függvényében írjuk fel:

$$\beta = b_0 + b_1\alpha + b_2\alpha^2 + \cdots + b_{n-1}\alpha^{n-1},$$

ahol $b_0, b_1, \ldots, b_{n-1} \in \mathbb{Q}$. A hatékonyabb számolás érdekében azonban az `algnumber` nem egy az egyben így, racionális együtthatókkal ábrázolja a számot, hanem közös nevezőre hozva a törteket, egész együtthatókkal:

$$\beta = \frac{a_0 + a_1\alpha + a_2\alpha^2 + \cdots + a_{n-1}\alpha^{n-1}}{d},$$

ahol $a_0, a_1, \ldots, a_{n-1}, d \in \mathbb{Z}$. Az egyértelműség kedvéért a törtet mindig egyszerűsítve tároljuk, azaz $\gcd(a_0, a_1, \ldots, a_{n-1}, d) = 1$, és $d > 0$.

4.4.1. Alapműveletek

Az összeadás és a kivonás az `algnumber` esetén – ellentétben az `algebraic`-kel – nagyon egyszerű: csak együtthatónként el kell végezni a műveletet. A közös nevezőre hozott ábrázolás miatt ráadásul nem is n pár racionális számot kell összeadni n egyszerűsítéssel (azaz legnagyobb közös osztó számítással), hanem csak egyszer kell a végén egyszerűsíteni. A kódban az összeadás és a kivonás közös megvalósítása az `algnumber::addition()` segédfüggvényben szerepel.

A szorzás az ábrázoló polinomok szorzását jelenti, majd maradékképzést a test primitív elemének f minimálpolinomjára. Optimalizálásként a test létrehozásakor kiszámítjuk az $x^n, x^{n+1}, \ldots, x^{2n-2}$ monomok maradékát az n-edfokú f polinomra. Ezek azok az x-hatványok, amelyek a szorzás során előállnak, de a maradékképzés után eltűnnek. A szorzás során a polinomok szorzatát együtthatónként állítjuk elő, növekvő fokszám szerint. A legfeljebb $n-1$-edfokú együtthatókat közvetlenül írjuk be az eredménybe, a magasabbfokú tagokra pedig az előre kiszámolt maradékok megfelelő konstansszorosát adjuk hozzá az eredményhez. A kódban az általános szorzás az `algnumber::general_multiply()` segédfüggvényben található. Arra a speciális esetre, amikor az egyik paraméter racionális szám, az egyszerűbb és hatékonyabb `algnumber::rational_multiply()`-t használjuk.

Az osztást multiplikatív inverzzel való szorzással valósítjuk meg, az inverzet pedig a 2.3.1. alfejezetben leírtak alapján a bővített euklideszi algoritmussal számítjuk ki. Ezt a kódban az `algnumber::general_inverse()` segédfüggvény valósítja meg. Két speciális esetben egyszerűbb módszert használunk: ha a szám racionális, illetve ha a test másodfokú. Ez utóbbi esetben kihasználjuk, hogy a primitív elem az egyszerűsített alak miatt $\sqrt{d}$ alakú.

4.4.2. Összehasonlítás

Az `algnumber` számok közötti egyenlőségvizsgálat az ábrázolás egyértelműsége miatt csak az ábrázoló polinomok egyenlőségvizsgálatát jelenti.

Különböző számok összehasonlítását intervallum-aritmetika segítségével oldjuk meg, visszavezetve arra, hogy a test primitív elemét izolációs intervallummal ábrázoljuk, és azt tudjuk igény szerint szűkíteni. Lásd a 8. algoritmust a következő oldalon (és a kódban a `compare()` függvényt).

4.4.3. Egészre kerekítés

Az `algnumber` egészre kerekítését szintén intervallum-aritmetika segítségével oldjuk meg. A következő oldalon látható a lefelé kerekítés algoritmusa (9. algoritmus). A kódban a kerekítések közös megvalósítása az `algnumber::to_integer_impl()` segédfüggvényben található.

8. algoritmus: Két `algnumber` összehasonlítása

Bemenet: $\mathbb{Q}(\alpha)$ test az α ábrázolásával: $(f, [\,a, b\,], s)$, illetve $g(\alpha)$ és $h(\alpha)$, az összehasonlítandó számok, ahol $g, h \in \mathbb{Q}[x]$

Kimenet: az összehasonlítás eredménye: $\operatorname{sgn}(g(\alpha) - h(\alpha)) \in \{-1, 0, +1\}$

if $g = h$ **then**
  return 0

repeat
  (Intervallum-aritmetikai behelyettesítés Horner-algoritmussal)
  $[\,a_1, b_1\,] := g([\,a, b\,])$
  $[\,a_2, b_2\,] := h([\,a, b\,])$
  (Elkülönülnek az intervallumok?)
  if $b_1 \leq a_2$ **then**
    return -1
  if $a_1 \geq b_2$ **then**
    return $+1$
  (Eredeti intervallum szűkítése és ismétlés)
  tighten_interval$(f, [\,a, b\,], s)$

9. algoritmus: `algnumber` egészre kerekítése (lefelé)

Bemenet: $\mathbb{Q}(\alpha)$ test az α ábrázolásával: $(f, [\,a, b\,], s)$, illetve $g(\alpha)$, a bemenő szám, ahol $g \in \mathbb{Q}[x]$

Kimenet: $x \in \mathbb{Z}$

$[\,a', b'\,] := g([\,a, b\,])$

while $\lceil b' \rceil - \lfloor a' \rfloor > 1$ **do**
  tighten_interval$(f, [\,a, b\,], s)$
  $[\,a', b'\,] := g([\,a, b\,])$

$x := \lfloor a' \rfloor$

5. Tesztelés és alkalmazás

5.1. Futási idők

Megvizsgáltuk néhány alapvető művelet sebességét: az összeadását, a szorzását és a kisebb-nagyobb összehasonlítását a könyvtár mindkét algebrai számtípusán (`algebraic` és `algnumber`).

A műveletek mért futási ideje az alábbi két táblázatban látható másodpercben (s), ezredmásodpercben (ms) vagy milliomodmásodpercben (µs).

`algebraic`

Fokszám:	3	4	5	6	7	8	9	10
`(+)`	0,38 ms	2,5 ms	0,020 s	0,15 s	1,2 s	5,1 s	27,3 s	109,5 s
`(*)`	0,38 ms	2,5 ms	0,021 s	0,18 s	1,3 s	6,5 s	25,2 s	82,9 s
`(<)`	1,2 µs	2,0 µs	2,5 µs	4,0 µs	5,1 µs	8,3 µs	10,5 µs	15,4 µs

`algnumber`

Fokszám:	4	6	8	10	15	20	25	30
`(+)`	0,6 µs	0,8 µs	1,0 µs	1,2 µs	1,7 µs	2,2 µs	2,7 µs	3,2 µs
`(*)`	1,5 µs	3,5 µs	6,2 µs	9,7 µs	21,5 µs	41,3 µs	68,2 µs	108,7 µs
`(<)`	1,3 µs	2,2 µs	3,1 µs	4,5 µs	8,9 µs	14,2 µs	20,3 µs	26,6 µs

A méréseket a `test/speed.cpp` programmal végeztük 3 GHz-es gépen, GCC 4.6-os fordítóval, annak `-O2` kapcsolójával lefordítva.

A másodpercnél jóval gyorsabb műveleteket sokszori ismétlés átlagából számítottuk. A bemeneti adatokat véletlenszerűen generáltuk, és a generálás idejét külön lemérve levontuk az eredményből. A generálás paraméterei:

- `algebraic`: a minimálpolinom együtthatói -10 és $+10$ között vannak;
- `algnumber`: a primitív elemmel való felírásban az együtthatók -1000 és $+1000$ közöttiek;
- `algnumber`: a primitív elem minimálpolinomjának együtthatói -100 és $+100$ között vannak.

Az eredményekből látszik, hogy az `algnumber` aritmetikája sokkal gyorsabb, mint az `algebraic`-é. Ez nem meglepő, hiszen az `algebric` bonyolult polinomműveleteket használ, mint a rezultáns és a faktorizáció, míg az `algnumber` lényegében csak összeadja és összeszorozza a megfelelő polinomokat. Az adatokból kiolvasható a nagyságrend is: az `algnumber` esetén – a várakozásoknak megfelelően – az összeadás lineáris és a szorzás négyzetes, az `algebraic`-nél pedig mindkettő exponenciális. Részletesebb méréssel az is kiderül, hogy az `algebraic` műveleti idejének nagy részét a polinomfaktorizáció adja.

Az összehasonlításnál (<) a két típus között nincs nagyságrendi eltérés. Ez nem is meglepő, hiszen mindkettő egyszerű intervallum-aritmetikát használ (lásd a 4.2.3. és a 4.4.2. alfejezeteket). Viszont itt van még egy szempont a fokszámon kívül, amelytől jelentősen függ a futási idő: nevezetesen hogy a két szám milyen közel van egymáshoz. A közel lévő számoknál ugyanis sokszor kell szűkíteni az intervallumokat, mire azok teljesen elkülönülnek egymástól.

Nézzünk példát közeli számok összehasonlításának futási idejére. Közeli számokat úgy állítottunk elő, hogy kerestünk egy testet, amelynek van kicsi abszolút értékű, viszonylag egyszerű δ eleme, és olyan számpárokat vettünk, melyeknek különbségei δ, δ^2 és δ^3. Legyen α a következő egyenlet egyetlen valós gyöke (kb. 0,724):

$$\alpha^5 + 10\alpha^3 + 4\alpha^2 + 4\alpha - 9 = 0$$

A $\mathbb{Q}(\alpha)$ testben az alábbi δ szám kicsi abszolút értékű:

$$\delta := \alpha^4 + \alpha - 1 \approx 9{,}25 \cdot 10^{-6}$$

Az alábbi táblázatban láthatók a számpárok és az eredmények. A méréseket a `test/compare.cpp`-vel végeztük.

Pontos értékek	*Közelítő értékek*	*Eltérés kb.*	*Futási idő*	
			Első	*Átlag*
$2\alpha^3 - 2\alpha + 2$	1,31157543491666906	$9{,}245 \cdot 10^{-6}$	46,4 µs	1,9 µs
$\alpha^4 + 2\alpha^3 - \alpha + 1$	1,31158468001313568			
$-42\alpha^4 + 11\alpha^3 - 50\alpha^2 + 27\alpha + 15$	0,92823107376485003	$8{,}547 \cdot 10^{-11}$	150,5 µs	3,9 µs
$52\alpha^4 + 80\alpha^3 - \alpha^2 - 57\alpha - 2$	0,92823107385032183			
$-14674\alpha^3 - 1543\alpha^2 + 5235\alpha + 2599$	1,54551317570726950	$7{,}902 \cdot 10^{-16}$	273,1 µs	5,3 µs
$8216\alpha^4 + 6733\alpha^2 + 1507\alpha - 6888$	1,54551317570727029			

A táblázatban két oszlopa is van a futási időnek: az *Átlag* azt jelenti, hogy egy rögzített `algfield`-ben ugyanazt a két `algnumber`-t sokszor összehasonlítjuk, és ezek futási idejének átlagát vesszük; az *Első* pedig azt, hogy egy új `algfield`-ben mennyi a futási ideje a legelső összehasonlításnak. A kettő között azért van különbség, mert az összehasonlító algoritmus szűkíti a primitív elem izolációs intervallumát, és ezt a szűkítést a test megőrzi, ezáltal legközelebb már gyorsabb lesz ugyanaz az összehasonlítás. Látható, hogy különbség jelentős, akár ötvenszeres is lehet. Ha tehát egy testben sok összehasonlítást végzünk, akkor kezdetben az első néhány lassabb lesz, de később már kellően szűk lesz az intervallum, és már csak nagyon ritkán kell tovább szűkíteni, tehát gyorsabb lesz.

Mindkét oszlopban teljesül, hogy minél közelebb vannak egymáshoz a számok, annál lassabb az összehasonlítás. Az *Első* esetén ezt vártuk, hiszen közelebbi számok esetén többször kell szűkíteni az intervallumot. Az *Átlag* esetén viszont ez már elhanyagolható szempont, mert csak az első összehasonlításkor van szűkítés; itt ennek inkább az lehet az oka, hogy a szűkebb intervallumnak általában nagyobbak lesznek az együtthatói, és ezekkel lassabb számolni.

Megjegyzés: az utolsó számpár egyúttal arra is példa, hogy két szám olyan közel van egymáshoz, hogy ha numerikusan kiértékeljük őket duplapontosságú lebegőpontos számokkal, akkor azokat összehasonlítva a kerekítési hibák miatt rossz eredményt kapunk.

5.2. Geometriai alkalmazás

Az egzakt számolás hasznos lehet olyan esetekben, amikor az eredmény szerkezete, logikája függ attól, hogy bizonyos értékek megegyeznek-e vagy sem.

Például tekintsünk egy síkgeometriai szerkesztőprogramot: a felhasználó elhelyez pontokat a síkon, ezeken keresztül egyeneseket és köröket rajzol, és a program megjelöli ezek metszéspontjait. A szerkesztés eredményének szerkezetét befolyásolja, hogy bizonyos pontok egybeesnek-e vagy sem, illetve hogy például egy kör és egy egyenes éppen érinti-e egymást, vagy két metszéspontjuk van, vagy egy sem. A szerkesztőprogramtól azt szeretnénk, ha ezeket az illeszkedéseket észlelné, és ennek megfelelően rajzolná ki az eredményt.

Nézzük egy egyszerű példát: egy háromszög beírt körének középpontját szeretnénk megszerkeszteni. Elemi geometriából ismeretes, hogy a háromszög belső szögfelezői egy ponton mennek keresztül, és ez a pont a beírt kör középpontja. Ha a háromszög csúcsai (x_1, y_1), (x_2, y_2), (x_3, y_3), és az oldalainak hossza $l_{1,2}$, $l_{2,3}$ és $l_{3,1}$, akkor könnyen levezethető, hogy az f_i szögfelezők egyenlete a következő:

$$f_i: \quad a_i x + b_i y = c_i,$$

ahol:

$$a_i := \frac{y_j - y_i}{l_{i,j}} + \frac{y_k - y_i}{l_{k,i}}, \quad b_i := \frac{x_i - x_j}{l_{i,j}} + \frac{x_i - x_k}{l_{k,i}},$$

$$c_i := a_i x_i + b_i y_i,$$

$$l_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2},$$

$$(i, j, k) \in \{(1, 2, 3), (2, 3, 1), (3, 1, 2)\}.$$

Jelöljük továbbá $M_{i,j}$-vel az f_i és f_j szögfelezők metszéspontját.

Konkrét számpéldaként tekintsük az alábbi háromszöget:

$$(x_1, y_1) = (6, 2), \quad (x_2, y_2) = (7, 2), \quad (x_3, y_3) = (5, 6).$$

A felhasználó elhelyezi ezeket a pontokat a szerkesztőprogramban, és megszerkeszti az általuk meghatározott háromszög szögfelezőit. A program kiszámítja ezen három egyenes metszéspontjait. Ha lebegőpontosan számol, például dupla pontosságú számokkal, akkor a fenti képletekből az alábbi metszéspontok jönnek ki:

```
M12: (6.3254848353090400, 2.4168732977062453)
M23: (6.3254848353090418, 2.4168732977062448)
M31: (6.3254848353090409, 2.4168732977062461)
```

Vagyis numerikusan három különböző metszéspont keletkezik a kerekítési hibák miatt. Ebből a program nem veszi észre, hogy a három egyenes valójában egy ponton megy keresztül. Elvész tehát ez a logikai információ, amely matematikailag is igazolható. Ráadásul ha a felhasználó valamelyik metszéspont körül megszerkesztené a beírt kört, akkor az nem pontosan érintené az összes oldalt, hanem némelyikkel két metszéspontja is lenne, némelyiket pedig esetleg nem metszené.

Numerikus számolás esetén ezen a problémán lehet javítani, de teljesen megoldani nem. A szokásos megoldás az, hogy nem pontos egyenlőséget vizsgálunk, hanem hibahatáron belüli egyezést. Azonban bonyolultabb, többlépéses szerkesztéseknél a

kerekítési hibák halmozódnak, és átléphetik az előre rögzített hibahatárt. Ha pedig a küszöböt nagyobbnak választjuk, akkor esetleg egyenlőnek vehetne különböző, de egymáshoz nagyon közel lévő értékeket.

Ha azonban a program numerikus számolás helyett szimbolikusan számol, például a SAN könyvtár egzakt algebrai számtípusaival, akkor ez a probléma megoldódik. A könyvtár használatával a következő eredmények jönnek ki:

```
M12: ((13 + 10g - g^3)/2, (42 + 11g - 3g^2 - g^3)/6)
M23: ((13 + 10g - g^3)/2, (42 + 11g - 3g^2 - g^3)/6)
M31: ((13 + 10g - g^3)/2, (42 + 11g - 3g^2 - g^3)/6)
g: root of x^4 - 11x^2 + 9 (++++) in [3,15/4]: 3.17959
```

Az eredmény pontos értelmezése nélkül is rögtön látható, hogy a három kiszámított metszéspont megegyezik. Ebből tehát a program fel tudja ismerni az eredmény matematikailag is igazolható szerkezetét, hogy a három egyenes egy ponton megy keresztül. A beírt kör megszerkesztésekor pedig ehhez hasonlóan észlelni fogja, hogy az valóban pontosan egy pontban érinti bármelyik oldalt, és ennek megfelelően tudja jelölni minden oldalon az egyetlen érintési pontot.

A kétféle számítás a `test/incircle.cpp` tesztprogramban megtekinthető. A kódban egy közös, általános függvény számítja ki a fenti képleteket, és azt hívjuk meg először lebegőpontos számokkal, majd algebrai számokkal. Az utóbbi esetben a bemenő adatokból először felépítünk egy algebrai számtestet (`algfield`), majd annak `algnumber` elemeivel hívjuk meg a függvényt. Mivel azonban egy algebrai számtestben négyzetgyököt vonni általában nem lehet, ezért még a test létrehozása előtt kiszámítjuk a háromszögek $l_{i,j}$ oldalhosszúságait is (amelyek jelen esetben 1, $2\sqrt{5}$ és $\sqrt{17}$), és ezeket is bemenő adatként vesszük hozzá a testhez.

Most értelmezzük, hogy mit jelent pontosan a kiírt eredmény. A könyvtár a számokat az ábrázolásnak megfelelő formában írja ki: az `algnumber` elemeket az algebrai számtest `g` primitív elemének függvényében, az `algebraic`-et pedig (mint a `g`) a minimálpolinomjával, egy izolációs intervallumával, a deriváltak előjeleivel a gyök helyén, illetve egy közelítő értékkel.

Ha ezek alapján könnyebben érthető alakban szeretnénk felírni az eredményt, akkor a kiírt adatokból kiszámíthatjuk a primitív elemet:

$$g = \frac{\sqrt{5} + \sqrt{17}}{2},$$

és ha ezt behelyettesítjük a kiírt eredménybe, akkor ezt kapjuk:

$$M_{1,2} = M_{2,3} = M_{3,1} = \left(\frac{13 - 2\sqrt{5} + \sqrt{17}}{2}, \ \frac{17 - \sqrt{5} + \sqrt{17} - \sqrt{85}}{4} \right)$$

A könyvtár ugyan egyelőre nem tudja ilyen alakban kiírni az eredményt, de enélkül is teljesíti azt a célt, amit a fentiekben elvártunk tőle: egzaktul el tudja dönteni, hogy adott értékek egyenlőek-e vagy sem.

5.3. Lineáris egyenletrendszer

A szimbolikus számolás különösen hasznos lehet olyankor, amikor egy probléma numerikusan nem stabil.

Például tekintsük a következő 8x8-as lineáris egyenletrendszert:

$$\begin{bmatrix} 1 & 2 & 4 & \dots & 128 \\ 1 & 7/2-\sqrt{2} & \left(7/2-\sqrt{2}\right)^2 & \dots & (7/2-\sqrt{2})^7 \\ 1 & 5\sqrt{2}-5 & \left(5\sqrt{2}-5\right)^2 & \dots & \left(5\sqrt{2}-5\right)^7 \\ 1 & 11\sqrt{2}-27/2 & \left(11\sqrt{2}-27/2\right)^2 & \dots & \left(11\sqrt{2}-27/2\right)^7 \\ 1 & 23/3-4\sqrt{2} & \left(23/3-4\sqrt{2}\right)^2 & \dots & \left(23/3-4\sqrt{2}\right)^7 \\ 1 & 21/2-6\sqrt{2} & \left(21/2-6\sqrt{2}\right)^2 & \dots & \left(21/2-6\sqrt{2}\right)^7 \\ 1 & 10\sqrt{2}/7 & 200/49 & \dots & 80000000\sqrt{2}/7^7 \\ 1 & 2\sqrt{2}-4/5 & \left(2\sqrt{2}-4/5\right)^2 & \dots & \left(2\sqrt{2}-4/5\right)^7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} 1538 \\ \left(37636049-26431018\sqrt{2}\right)/128 \\ 86284280\sqrt{2}-122022480 \\ \left(5380398776126\sqrt{2}-7609032685421\right)/128 \\ \left(609637214276-431076155088\sqrt{2}\right)/2187 \\ \left(417093876363-294929762556\sqrt{2}\right)/128 \\ \left(56800400+96226070\sqrt{2}\right)/117649 \\ \left(1340230090\sqrt{2}-1763989908\right)/78125 \end{bmatrix}$$

Ugyanez közelítő értékekkel:

$$\begin{bmatrix} 1{,}000 & 2{,}000 & 4{,}000 & 8{,}000 & 16{,}000 & 32{,}000 & 64{,}000 & 128{,}000 \\ 1{,}000 & 2{,}086 & 4{,}351 & 9{,}074 & 18{,}927 & 39{,}478 & 82{,}342 & 171{,}747 \\ 1{,}000 & 2{,}071 & 4{,}289 & 8{,}883 & 18{,}398 & 38{,}104 & 78{,}916 & 163{,}441 \\ 1{,}000 & 2{,}056 & 4{,}229 & 8{,}695 & 17{,}881 & 36{,}769 & 75{,}610 & 155{,}481 \\ 1{,}000 & 2{,}010 & 4{,}039 & 8{,}118 & 16{,}316 & 32{,}793 & 65{,}907 & 132{,}461 \\ 1{,}000 & 2{,}015 & 4{,}059 & 8{,}178 & 16{,}476 & 33{,}195 & 66{,}879 & 134{,}741 \\ 1{,}000 & 2{,}020 & 4{,}082 & 8{,}246 & 16{,}660 & 33{,}658 & 67{,}999 & 137{,}378 \\ 1{,}000 & 2{,}028 & 4{,}115 & 8{,}346 & 16{,}929 & 34{,}340 & 69{,}656 & 141{,}291 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} 1538{,}000 \\ 2007{,}382 \\ 1918{,}996 \\ 1834{,}000 \\ 1586{,}344 \\ 1611{,}006 \\ 1639{,}492 \\ 1681{,}685 \end{bmatrix}$$

Az egyenletrendszer mátrixa egy Vandermonde-mátrix, amelynek az alappontjai közel vannak egymáshoz. Az ilyen mátrix nagyon rosszul kondícionált: ennek például a kondíciószáma ($||\cdot||_2$ szerint) kb. $8{,}02 \cdot 10^{17}$. Ennek következtében az egyenletrendszer numerikus megoldása sokszorosára felnagyíthatja a bemenő adatok hibáját.

Ezt az egyenletrendszert kétféleképpen oldottuk meg (lásd a `test/lineqs.cpp`-t): egyrészt numerikusan, duplapontosságú lebegőpontos számokkal, másrészt pedig a SAN könyvtár egzakt algebrai számaival. Mindkét esetben ugyanazt az algoritmust használtuk, a Gauss-eliminációt. Bal oldalon láthatjuk a numerikus számításból származó megoldást, jobb oldalon az egzaktot, alattuk pedig az egyenletmegoldás futási idejét:

$$\begin{bmatrix} -16{,}607 \\ 58{,}036 \\ -81{,}950 \\ 71{,}644 \\ -29{,}676 \\ 14{,}912 \\ 4{,}379 \\ 7{,}114 \end{bmatrix} \qquad \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{bmatrix}$$

0,77 µs 290 µs

Látható, hogy a numerikus megoldás teljesen rossz, pedig csak annyi volt a bemenő adatok hibája, hogy duplapontosságú lebegőpontos számra kerekítettük az eredeti algebrai számokat.

Ilyen esetekben numerikusan csak annyit tudnánk javítani, hogy nagyobb pontosságú típussal számolunk. De ha ehhez hasonló szerkezetű nagyobb mátrixunk van, akkor annak még nagyobb lesz a kondíciószáma, így a kisebb hibát is nagyra nagyíthatja. Szimbolikus számolással azonban – ha a bemenő adatok egzaktul rendelkezésre állnak – az efféle numerikusan instabil problémák is pontosan, kerekítési hibák nélkül oldhatók meg – megnövekedett futási időért cserébe.

5.4. LLL-algoritmus

A Lenstra–Lenstra–Lovász-algoritmus (LLL-algoritmus) egy rács bázisát redukálja, azaz előállít belőle egy „majdnem" merőleges bázist. [10, 65-72. o.]

Hasonlít a Gram–Schmidt-ortogonalizációra, amely egy vektortér bázisából csinál merőleges bázist, a következőképpen:

Állítás: Ha $b_1, b_2, \ldots, b_n \in \mathbb{R}^n$ bázis, akkor az alábbi $\tilde{b}_1, \tilde{b}_2, \ldots, \tilde{b}_n$ vektorok merőleges bázist alkotnak:

$$\tilde{b}_i := b_i - \sum_{j=1}^{i-1} \mu_{i,j} \tilde{b}_j \quad (i \in [\,1 \,..\, n\,]),$$

ahol:

$$\mu_{i,j} := \frac{\langle b_i, \tilde{b}_j \rangle}{\langle \tilde{b}_j, \tilde{b}_j \rangle} \quad (j \in [\,1 \,..\, i-1\,]).$$

Az LLL-algoritmus ehhez hasonlót csinál, csak rácson.

Definíció: Adott egy $b_1, b_2, \ldots, b_n \in \mathbb{R}^n$ bázis. Ekkor **rács**nak nevezzük a következő halmazt:

$$\mathcal{L} := \left\{ \sum_{i=1}^{n} x_i b_i \mid x_i \in \mathbb{Z} \right\}$$

Rácson tehát csak egész együtthatós lineáris kombinációk megengedettek. Ennek következtében itt általában nem lehet merőlegességet biztosítani, csak „majdnem" merőlegességet, amennyire az egész együtthatók engedik. Pontosabban:

Definíció: Az $\mathcal{L}$ rács egy $b_1, b_2, \ldots, b_n \in \mathbb{R}^n$ bázisa **LLL-redukált**, ha (a fenti Gram–Schmidt-ortogonalizáció jelöléseit használva):

1. $|\mu_{i,j}| \leq 1/2 \quad (i, j \in [\,1 \,..\, n\,], j < i)$, és
2. $||\tilde{b}_{i+1}||^2 \geq (\delta - \mu_{i+1,i}^2)||\tilde{b}_i||^2 \quad (i \in [\,1 \,..\, n-1\,])$, ahol $\delta \in (1/4, 1)$ rögzített, tipikusan $\delta = 3/4$.

Az első feltétel biztosítja a közelítőleg való merőlegességet: a pontos merőlegességhez tartozó együtthatót egészre kerekítjük, amely által a hiba legfeljebb 1/2 lehet (ha a bázis pontosan merőleges lenne, akkor $\mu_{i,j} = 0$ lenne). A második feltétel a vektorok méretét korlátozza: a következő vektor ne lehessen sokkal kisebb az előzőnél.

Próbaképpen lefuttattuk az LLL-algoritmust a SAN könyvtár algebrai számaival, és összehasonlításként lebegőpontos számokkal is (`test/lll.cpp`). Az algoritmus egy olyan alkalmazása a könyvtárnak, amely használja az összehasonlítás és az egészre kerekítés műveleteit. Bemenő adatként az előző (5.3.) részben lévő egyenletrendszer mátrixát használtuk, annak sorai a bemenő bázisvektorok.

Az alábbiakban láthatjuk az LLL-algoritmus eredményét. A mátrixok sorai az egyes bázisvektorok. Az első mátrix a numerikus számítás eredménye, a második pedig a szimbolikus eredmény numerikusan közelítve. (Az egzakt szimbolikus eredmény a `test/lll.cpp` kimenetében látható, de numerikusan jobban összevethetők az értékek.)

Numerikus eredmény

$$10^{-7} \cdot \begin{bmatrix} 0{,}000 & 1{,}885 & 2{,}752 & 0{,}936 & 1{,}919 & -3{,}943 & 0{,}493 & 2{,}444 \\ 0{,}000 & 1{,}355 & -3{,}589 & -5{,}510 & -1{,}674 & -1{,}578 & 2{,}135 & 3{,}001 \\ 0{,}000 & -1{,}624 & -0{,}354 & -0{,}499 & 0{,}462 & 2{,}435 & -1{,}441 & 6{,}484 \\ 0{,}000 & -0{,}464 & 0{,}497 & 4{,}723 & -4{,}718 & 0{,}336 & 2{,}269 & -0{,}170 \\ 0{,}000 & -1{,}652 & 0{,}268 & -4{,}262 & -3{,}427 & -0{,}508 & -5{,}174 & -0{,}164 \\ 0{,}000 & -2{,}478 & 2{,}655 & -3{,}644 & 1{,}512 & 1{,}945 & 3{,}755 & 3{,}570 \\ 0{,}000 & -8{,}378 & -2{,}063 & 1{,}502 & -2{,}775 & -1{,}303 & -1{,}399 & 1{,}710 \\ 10^7 & -1{,}402 & -0{,}867 & 3{,}943 & 3{,}791 & -0{,}828 & 0{,}227 & -0{,}292 \end{bmatrix}$$

Futási idő: 0,422 ms

Szimbolikus eredmény, közelítve

$$10^{-7} \cdot \begin{bmatrix} 0{,}000 & -1{,}782 & 2{,}019 & 1{,}011 & 1{,}442 & 0{,}085 & -3{,}712 & 0{,}862 \\ 0{,}000 & -0{,}248 & -1{,}742 & -0{,}656 & 1{,}993 & 4{,}962 & 2{,}500 & 1{,}786 \\ 0{,}000 & 2{,}909 & 3{,}819 & -1{,}267 & -1{,}560 & -0{,}564 & 0{,}220 & -1{,}839 \\ 0{,}000 & -0{,}737 & 0{,}133 & -0{,}340 & 0{,}983 & -5{,}206 & 1{,}341 & 3{,}298 \\ 0{,}000 & -4{,}543 & -1{,}202 & -4{,}537 & -3{,}495 & -2{,}058 & 2{,}025 & 0{,}118 \\ 0{,}000 & 5{,}425 & -0{,}611 & -0{,}192 & -2{,}258 & 1{,}728 & -0{,}071 & 6{,}377 \\ 0{,}000 & -3{,}157 & 1{,}896 & 7{,}832 & -2{,}186 & -1{,}423 & 0{,}550 & -2{,}184 \\ 10^7 & 0{,}429 & 0{,}561 & 2{,}426 & -3{,}169 & 1{,}070 & -0{,}579 & -0{,}828 \end{bmatrix}$$

Futási idő: 6,19 s

A két eredmény bár szerkezetében és nagyságrendjében hasonló, de a konkrét értékek teljesen mások. Ez is egy példa olyan feladatra, ahol a numerikus számolás rosszul viselkedik. Ugyan itt is javítani lehet a helyzeten a numerikus pontosság növelésével, de teljesen megoldani szimbolikus számítással lehet.

6. Összefoglalás és továbbfejlesztési lehetőségek

A dolgozat az algebrai számokkal való szimbolikus számolásról szólt. Áttekintettünk néhány ezzel kapcsolatos módszert a szakirodalomból, majd bemutattuk egy új könyvtár, a Symbolic Algebraic Numbers (SAN) használatát és működését, végül pedig ennek néhány lehetséges alkalmazásáról volt szó.

Az alábbiakban felvázolunk néhány lehetőséget a könyvtár továbbfejlesztésére és hatékonyabbá tételére.

Összehasonlítás: lebegőpontos optimalizálás

Bizonyos műveleteket gyorsíthatunk azzal, ha lebegőpontosan kiértékeljük a bemenő számokat, és azokon végezzük el a műveletet. Tipikus példa erre az összehasonlítás.

Mivel egzaktul számolunk, nem mindig támaszkodhatunk lebegőpontos számolásra a korlátozott pontosság miatt. Az 5.1. alfejezet végén láttunk példát olyan közeli számokra, amelyeket ha dupla pontosságú lebegőpontos számokkal kiértékelünk, akkor azok összehasonlítása hibás eredményt ad. Ez azonban nagyon kivételes eset, és legtöbbször bőven elegendő a dupla pontosság. Ezért gyorsíthatjuk az összehasonlítást azzal, ha lebegőpontos számolást használunk olyan esetben, amikor az bizonyíthatóan helyes eredményt ad. Ehhez egy egzakt felső becslést kell adni a közelítés pontosságára, és ha az értékek elég távol vannak egymástól (eltérésük nagyobb, mint a két hibahatár összege), akkor a lebegőpontos összehasonlítás pontos lesz. Ellenkező esetben pedig – várhatóan nagyon ritkán – visszatérünk az eredeti algoritmusra.

Hasonlóan lehetne gyorsítani az egészre kerekítés műveletét is.

Minél egyszerűbb primitív elem keresése

Az algebrai számtestben való számolást gyorsítani lehet azzal, ha minél egyszerűbb szerkezetű primitív elemet használunk.

Például tekintsük a $\mathbb{Q}(\sqrt{2}, \sqrt[3]{2}, \sqrt[3]{3})$ testet. A könyvtár ebből az $\alpha := \sqrt{2} + \sqrt[3]{2} + \sqrt[3]{3}$ primitív elemet állítja elő, amelynek minimálpolinomja:

$$x^{18} - 18x^{16} - 30x^{15} + 144x^{14} + 180x^{13} - 621x^{12} + 360x^{11} - 4860x^{10} - 3640x^{9} + 28260x^{8} +$$
$$+\ 3600x^{7} - 19305x^{6} - 226080x^{5} + 41310x^{4} + 53430x^{3} + 341748x^{2} - 42300x + 10097$$

Ezzel a primitív elemmel így lehet felírni például a $\sqrt{2}$-t:

$$\begin{aligned}\sqrt{2} = (&-2736584319330927000\alpha^{17} + 1422846532640549355 3\alpha^{16} + 5234008970537188608 0\alpha^{15} - \\ &- 1702289661183185190 48\alpha^{14} - 8650739038403045964 60\alpha^{13} + 1416596998923379830336\alpha^{12} + \\ &+ 4403139815993832838080\alpha^{11} - 9437007954344310615366\alpha^{10} + 1794120361390805805448 0\alpha^{9} - \\ &- 5574316078426115540941 2\alpha^{8} - 1444219700456717618077 80\alpha^{7} + 3530144312012062228342 56\alpha^{6} + \\ &+ 1239118033571677763244 00\alpha^{5} + 3574356285821489272366 47\alpha^{4} - 3245428534782736684733520\alpha^{3} + \\ &+ 7953340033079453614352 4\alpha^{2} - 5026901984523696514311 0\alpha + 2481499577247256384601692) \, / \\ &/ 1761548336347250716752250\end{aligned}$$

Ugyanakkor ebben a testben primitív elem a $\beta := \sqrt[6]{2}\sqrt[3]{\sqrt[3]{2}-1}$ is, amelynek minimálpolinomja:

$$x^{18} - 6x^{12} + 60x^{6} - 8,$$

és ezzel felírva a $\sqrt{2}$-t:

$$\sqrt{2} = (3\beta^{15} - 16\beta^{9} + 156\beta^{3})/40$$

Ilyen együtthatókkal sokkal gyorsabb számolni, mint a fentiekkel. Hasznos fejlesztés lenne tehát keresni egy minél egyszerűbb primitív elemet a testben.

Erre az LLL-algoritmust tudjuk felhasználni, a következő eljárással [1, 168-173. o.]: határozzunk meg egy bázist a test algebrai egészeiben, majd redukáljuk ezt az LLL-algoritmussal (egy speciális norma szerint). Ezáltal kapunk egy bázist olyan elemekkel, amelynek a minimálpolinomjai várhatóan jóval kisebb együtthatókkal rendelkeznek, mint az eredeti primitív elemé. Ezek között lehetnek alacsonyabbfokú elemek is (amelyek így nem primitív elemei a testnek), de a többi közül válasszuk azt, amelyiknek a legegyszerűbb a minimálpolinomja.

Algebrai szám felírása gyökjelekkel

A beírt kör középpontjának megszerkesztésénél (az 5.2. alfejezetben) láttuk, hogy a könyvtár az eredményt az ábrázolásnak megfelelő alakban írja ki, amely sokszor nehezen értelmezhető. Hasznosabb lenne, ha egy egyszerű matematikai formulával írná ki az eredményt, pl.: $\sqrt{10+2\sqrt{5}}$.

Ehhez a Galois-elmélet nyújt segítséget [2, 339-419. o.]. Az elmélet főtétele szerint egy algebrai számtest résztestjei és a test Galois-csoportjának (a szimmetriáiból álló csoportnak) a részcsoportjai között szoros kapcsolat van. Ez lehetőséget ad arra, hogy egy algebrai számtest szerkezetét visszavezessük egy jóval egyszerűbb struktúrára, egy véges csoportra. A Galois-elmélet fontos eredménye, hogy azt a kérdést, hogy egy polinom gyökei felírhatók-e gyökjelek segítségével, csoportelméleti kérdésre vezeti vissza: pontosan akkor írható fel, ha a polinom gyökeivel megadott test Galois-csoportja feloldható, azaz felépíthető Abel-csoportokból csoportbővítés segítségével.

Ha egy adott algebrai számot szeretnénk gyökjelekkel fölírni, akkor először elő kell állítani az általa generált algebrai számtest Galois-csoportját, majd arról el kell dönteni, hogy feloldható-e. Ha igen, akkor a feladat az, hogy a csoport szerkezetéből előállítsuk a szám gyökös felírását. Ha a csoport nem feloldható, akkor nem lehet a számot gyökjelekkel felírni, ezért ilyenkor visszatérünk az eredeti felírásra (pl. minimálpolinommal).

Megjegyzés: geometriai szerkesztésből adódó algebrai számok mindig felírhatók gyökjelekkel, sőt, elegendő csak a négyzetgyökvonás [2, 391-401. o.].

Hivatkozások

[1] Cohen, H.: A Course in Computational Algebraic Number Theory. *Springer-Verlag Berlin Heidelberg*, 1996

[2] Kiss E.: Bevezetés az algebrába. *Typotex*, 2007

[3] Iványi A.: Informatikai Algoritmusok I. *ELTE Eötvös Kiadó*, 2004

[4] Smedley, T. J.: Symbolic Computation with Algebraic Numbers and Functions. *Research report, University of Waterloo*, 1987

[5] Yokoyama, K., Noro, M., Takeshima, T.: Computing Primitive Elements of Extension Fields. *Journal of Symbolic Computation* 8. (1989), 553-580. o.

[6] Geddes, K. O., Czapor, S. R., Labahn, G.: Algorithms for Computer Algebra. *Kluwer Academic Publishers*, 1992

[7] Conkwright, N. B.: An Elementary Proof of the Budan-Fourier Theorem. *The American Mathematical Monthly* 50.10 (1943), 603-605. o.

[8] Mishra, B., Pedersen, P.: Arithmetic with real algebraic numbers is in NC. *Proceedings of the International Symposium on Symbolic and Algebraic Computation* (1990), 120-126. o.

[9] Stoyan G., Takó G.: Numerikus módszerek 1. *Typotex*, 2005

[10] Smart, N. P.: The Algoritmic Resolution of Diophantine Equations. *Cambridge University Press*, 1998