Eötvös Loránd Tudományegyetem
Informatikai Kar

EKG jelek feldolgozása Hermite-függvények segítségével

BSc Szakdolgozat

Készítette: Dózsa Tamás
ELTE IK
Programtervező informatikus
BSc

Témavezető: Dr. Kovács Péter
Adjunktus
ELTE IK
Numerikus Analízis Tanszék

Budapest, 2016.11.16.

Tartalomjegyzék

1. fejezet

Bevezetés

Az információ ábrázolásának módja az informatika tudomány fontos kérdése. Természetesen annak eldöntése, hogy egy adott adat halmaz milyen módon kerül ábrázolásra erősen függ annak jellegétől. Az adatábrázolás felel az adatok hatékony felhasználhatóságáért (például egy internetes video hívásnál az adatokat gyorsan kell egymás után továbbítani), ugyanakkor biztosítania kell, hogy az adatokból kinyerhető információ nem veszik el (ha túl rossz mindőségű képeket továbbítunk, a fogadó fél nem tudja értelmezni azokat).

A szakdolgozat célja egy speciális adatábrázolás, nevezetesen EKG jelek egy ábrázolásának bemutatása. Mivel jellemzően ezeket az adatokat, későbbiekben *jelek*-et, általában nagyobb memória felhasználásával szokás ábrázolni, ezért a dolgozat az eljárásra EKG jelek *tömörítéseként* hivatkozik. EKG jelek esetén az ábrázolás minősége sok szemponttól függ. Mivel ezek a jelek fontos információkat hordoznak a szív állapotáról, különösen fontos, hogy a tömörítés során ne vesszen el fontos információ. Egy ilyen jel rögzítésekor azonban sok olyan adat is tárolásra kerül (például a végtagok mérés közbeni mozgatása miatt), amelyek nem hordoznak fontos információt. Az ilyen adatokra a dolgozat *zaj*-ként hivatkozik. Egy jó EKG ábrázolás sikeresen szűri a mérés során keletkezett zajt, miközben az orvosi szempontból fontosnak nevezhető információt megtartja. Mivel a dolgozatban egy tömörítési eljárás kerül bemutatásra, fontos szempont az EKG jelek memória takarékos ábrázolása. Gyakorlati szempontból minél kevesebb memórián történik meg a jelek ábrázolása, annál könnyebb azokat tárolni (hosszú mérések esetén fontos lehet), illetve egyszerűbb és biztonságosabb a jelek hálózaton történő továbbítása. EKG jeleknek egy igazán jónak nevezhető ábrázolása pedig az eddig említettek mellett az orvosok munkáját közvetlenül segítő információt is kódol magában. Ilyen lehet például egy olyan ábrázolás amely hatékony bemenetéül szolgál valamilyen osztályozó algoritmusnak, lehetővé téve az abnormális jelek automatikus felismerését.

A dolgozat három fő fejezetre tagolható. A Bevezetés című fejezetben található a dolgozatban bemutatott eljárás specifikációja, a jel reprezentáció matematikai modelljének ismertetése, valamint a bemutatott módszer egyéb jellemzőit ismertető alfejezetek. Ezek közé tartozik például a jel közelítésének optimalizációjához szükséges Nelder-Mead algoritmus elméleti bemutatása, illetve az EKG jel szegmentációját

elősegítő Matching-Pursuit algoritmust részletező alfejezet.

A dolgozat második fejezete az eljárás mellékelt implementációjának a fejlesztői dokumentációja. Ebben a fejezetben találhatóak a tömörítő eljárást implementáló c++ osztályok jellemzői, illetve az elérést segítő webes felület implementációjának részletes ismertetése. A fejlesztői dokumentáció fejezet tartalmazza továbbá a program logikai jellemzését elősegítő UML és egyéb osztálydiagrammokat. A fejezet igyekszik pragmatikusan és érthetően jellemezni a program felépítését, illetve kellően megindokolni az egyes implementációk mellett szóló döntéseket.

A Felhasználói dokumentáció fejezetbe, a program használatával kapcsolatos információk kerültek. Ebben a fejezetben található a felhasználói felület funkcióinak pontos ismertetése, valamint hasznos példák annak használatára. Bemutatásra kerülnek továbbá a program használatához szükséges előkészületi lépések, és az ismert rendszerkövetelmények.

A dolgozat utolsó része a függelék, melyben a Bevezetés fejezetben található matematikai állítások bizonyításai, egyéb kapcsolódó matematikai fogalmak leírásai, illetve felhasznált algoritmusok pszeudo-kódja található.

1.1. A feladat specifikációja

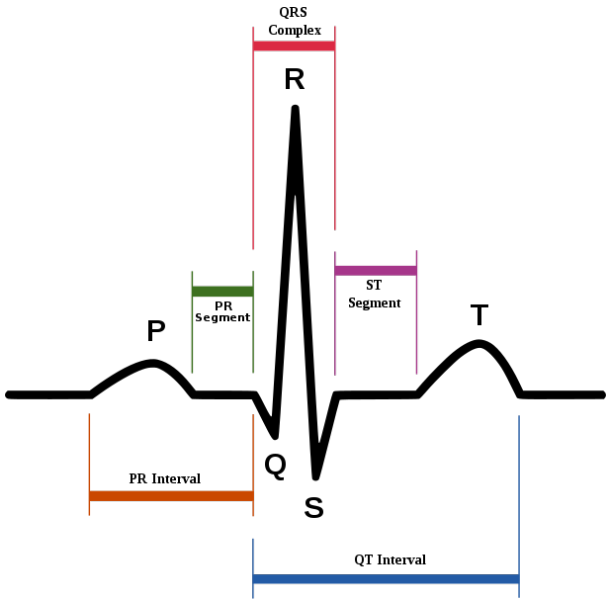
A dolgozat célja egy olyan tömörítési eljárás bemutatása, amely lehetővé teszi az EKG jelek hatékony (memória takarékos) ábrázolását, a mérések zaj szűrését, illetve az EKG hullámszegmenseinek szeparációját. Az utóbbi jellemző orvosi szempontból lehet hasznos, ugyanis sok kóros elváltozás kimutatásához szükséges az egyes hullámszegmensek széleinek ismerete. A dolgozatban bemutatott módszer hatékonysága, az irodalomban fellelhető más tömörítési eljárások [] hatékonyságának összehasonlításával igazolandó.

A dolgozatban bemutatott tömörítési eljárás implementációjának feladata, hogy az MIT-BIH adatbázisban található EKG jelek tömörítésére alkalmas legyen, ehhez pedig egy könnyen használható webes felületet biztosítson. Az implementációnak két féle bemente megengedett: tömörítés esetén az MIT-BIH adatbázisban megtalálható EKG jelek, illetve a tömörítés következtében létrejött számsorozat, melyet a tömörített jel helyreállításához használ.

1.2. A modell ismertetése

A modern orvostudományban nagy jelentőséggel bírnak a valamely élő szervezet által kibocsátott úgynevezett biológiaijelek. Ezek közé sorolható az Elektro Kardio Gram, vagy EKG, amely a szív állapotáról képes információt adni. Bár ennek a dolgozatnak nem célja az EKG jelek pontos elemzése, fontos néhány sorban ismertetni egy átlagos EKG jel meghatározó hullámait. Egyetlen szí vütés EKG reprezentációja három fő részre bontható: a szí vütés elején megjelenő P hullámra, az ezt követő QRS

komplexumra, és az ütés végén található T hullámra. Ezek rendre a pitvari összehűzódást, a kamrák depolarizációját és elektromos újratöltődését reprezentálják. Diagnosztikai szempontból a QRS komplexus a legfontosabb, ezért ezt nagy pontossággal kell tárolni. Általánosságban elmondható, hogy ezeknek a hullámoknak kezdő és végpontjai, valamint maximum és minimum értékei vesznek részt az orvosi diagnosztikában. Az említett paraméterek az 1.1 ábrán láthatóak.



1.1. ábra. Az EKG jel egy szívütése, illetve annak főbb diagnosztikai jellemzői.

Az irodalomban ismert tömörítő algoritmusokat [?] alapján három kategóriába sorolhatjuk: 1) egyszerű paraméteres becslések (pl.: interpoláció, különbségi kódolás, stb.), 2) direkt módszerek (pl.: csúcsok, meredekségek, stb. tárolása), 3) transzformációs eljárások. Az utóbbi osztály tartalmazza azokat az algoritmusokat, melyek a jelet egy előre adott függvényrendszer szerinti sorfejtéssel approximálják. Így az eredeti adatsorozat helyett csak az együtthatókat és a rendszer paramétereit kell tárolnunk. Ezen kategóriába sorolandó a dolgozatban bemutatott algoritmus is. Nevezetesen, az eredeti adatsorozatot speciális, Hermite-polinomok segítségével előállitott függvényrendszerrel fogjuk közelíti. A módszer alapját képező eljárás [?], jól ismert az irodalomban, mely nem csak a jelek tömörítéséhez, de azok modellezéséhez [?], illetve osztályozásához [?, ?] is alkalmazható. A dolgozatban az EKG jelekkel való hasonlóságuk miatt Hermite-függvényeket használunk az adatok reprezentálásához. Ezeket egy argumentum transzformáción keresztül szabad paraméterekkel egészítjük ki. Ennek köszönhetően az eredeti jelet egy adaptív bázisban írhatjuk fel. Az említett paraméterek megválasztásához a Nelder-Mead optimalizációs eljárást alkalmaztuk. Mivel az EKG jelek diszkrét adatsorozatok, ezért a módszert [?] alapján implementáltuk diszkrét ortogonális Hermite-polinomokra is. A dolgozatban különböző tesztekkel demonstráljuk az algoritmus hatékonyságát. Ehhez, több órányi, zajjal terhelt, valódi EKG felvételt használtunk. Ezen keresztül a bemutatott módszert összehasonlítottuk

több másik, az irodalomban jól ismert tömörítő algoritmussal is [?].

A tömörítő eljárást egy c++ nyelven megírt, objektum elvű alkalmazás implementálja, melyet egy webes felületen keresztül érhetünk el. A felület lehetőséget biztosít a dolgozatban jelölt tesztek újrafuttatására, valamint a teszteléskor felhasznált adatbázis további jeleinek a tömörítésére. Szintén a webes felületen keresztül nyílik alkamunk a már tömörített EKG jelek helyreállítására.

Az alkalmazás megetrvezésekor külön hangsúlyt kapott a kód újra felhasználhatósága. Ennek érdekében a felhasznált algoritmusok, illetve matematikai modellek a lehető legáltalánosabb formában lettek implementálva. Fontos szempontot jelentett továbbá a c++11-es nyelvszabvány által nyújtotta lehetőségek minél hatékonyabb kihasználása. Jó példa erre a lambda függvények alkalmazása az optimalizációs algoritmusok implementációja során. A hatékony működés mellett azonban a program igyekszik megfelelni a modern felhasználók igényeinek. Ennek érdekében a felhaszálói felület weboldalként lett implementálva. A rendszerfüggetlen, és installáció mentes elérés lehetővé teszi a gyors és egyszerű használatot, valamint az eredmények megosztását.

1.3. Matematikai háttér

1.3.1. Jelek approximációja

EKG jelek feldolgozásakor sok esetben szembesülünk gyakorlati kihívásokkal. Két sűrűn előforduló példa a hosszú mérések tárolása, valamint a zajjal terhelt mérések ábrázolása. Ezekre a nehézségekre egyszerre ad kielégítő megoldást, ha a jeleket valamely $\mathcal{H}$ Hilbert-tér sima függvényeiből álló $(\Phi_n, n \in \mathbb{N})$ ortogonális bázisában reprezentáljuk és a jelet véges sok $\Phi_0, \Phi_1, \cdots, \Phi_n$ bázisbeli elem lineáris kombinációjával közelítjük. Az $f \in \mathcal{H}$ jel legjobb közelítését a tér $\|\cdot\|$ normájában az

$$S_n f := \sum_{k=0}^{n} \langle f, \Phi_k \rangle \Phi_k$$

leképezés nyújtja, ahol $\langle \cdot, \cdot \rangle$ az $\mathcal{H}$ tér skaláris szorzatát jelöli. A jel és a közelítés eltérésének négyzete a

$$\|f - S_n f\|^2 = \|f\|^2 - \sum_{k=0}^{n} |\langle f, \Phi_k \rangle|^2$$

képplettel adható meg. Adott hibán belüli közelítést véve a jel helyett elég az $S_n f$ approximációt reprezentáló $\langle f, \Phi_k \rangle$ $(k = 0, 1, \cdots, n)$ Fourier-együtthatókat tárolni. Zajos jel esetén az ilyen típusú approximáció szűrőként is szolgál. A közelítés megvalósításához a klasszikus ortogonális rendszerek közül EKG görbék közelítésére az Hermite-féle függvények bizonyultak használhatónak. Ezt támasztják alá a [...] dolgozatok. Az Hermite függvények alkalmazása azzal is indikolható, hogy grafikonjuk hasonlít az EKG görbékre. Ezt a tulajdonságot a ?? ábra szemlélteti.

1.3.2. Hermite-függvények

A dolgozatban az $\mathbb{R}$ számegyenesen (Lebesgue-mérték szerint) négyzetesen integrálható függvények $\mathcal{H}$ Hilbert-tere helyett elegendő a szakaszonként folytonos, az $\mathbb{R}$-en négyzetesen integrálható függvények $\mathcal{F}$ euklideszi terét használni. Ebben a térben a skaláris szorzat és a norma a következő alakban írható fel:

$$\langle f, g\rangle := \int_{-\infty}^{\infty} f(t)g(t)\,dt, \quad \|f\| := \sqrt{\langle f, f\rangle} \quad (f, g \in \mathcal{F})\,. \tag{1.1}$$

Továbbá, a

$$\Phi_n(x) := H_n(x)e^{-x^2/2}/\sqrt{\pi^{1/2}2^n n!} \quad (n \in \mathbb{N})$$

normált Hermite-függvények (teljes) ortonormált rendszert alkotnak az $\mathcal{F}$ téren:

$$\langle \Phi_n, \Phi_m\rangle = \delta_{nm} \quad (m, n \in \mathbb{N}), \quad \|f - S_n f\| \to 0 \ (n \to \infty)\,.$$

Itt H_n $(n \in \mathbb{N})$ jelöli az Hermite-féle polinomokat.

Az Hermite-függvények alkalmazásának számos előnye van:

i) A Φ_n $(n \in \mathbb{N})$ rendszer zárt (teljes) az $\mathcal{F}$ téren.

ii) A $\Phi_n(x)$ függvények gyorsan tartanak 0-hoz, ha $|x| \to \infty$:

$$|\Phi_n(x)| \le M_n e^{-x^2/4} \le M_n \quad (x \in \mathbb{R}, n \in \mathbb{N}).$$

iii) A Φ_n függvények (stabil) másodrendű rekurzióval számíthatók:

$$\begin{aligned} &\Phi_0(x) := e^{-x^2/2}/\pi^{1/4}, \ \Phi_1(x) := \sqrt{2}\,xe^{-x^2/2}/\pi^{1/4} \\ &\Phi_n(x) = \sqrt{\frac{2}{n}}x\Phi_{n-1}(x) - \sqrt{\frac{n-1}{n}}\Phi_{n-2}(x) \quad (x \in \mathbb{R}, n \ge 2) \end{aligned} \tag{1.2}$$

iv) A Φ_n' deriváltak kifejezhetők a Φ_n, Φ_{n-1} függvényekkel:

$$\Phi_n'(x) = \sqrt{2n}\Phi_{n-1}(x) - x\Phi_n(x) \quad (x \in \mathbb{R}, n \in \mathbb{N}, \Phi_{-1} = 0) \tag{1.3}$$

1.3.3. Az approximáció optimalizálása

A jelek reprezentációja függ az időskála 0 pontjának és az egység megválasztásától. Ezeket a paramétereket a gyakorlatban önkényesen szoktuk megválasztani. Ezzel összefüggében felvethető a kérdés, hogyan lehet optimálisan megválaszthatani ezeket a paramétereket. Az approximáció pontossága javítható azonos együttható szám mellett, amennyiben az Hermite-függvények helyett azok

$$\Phi_n^{a,\lambda}(x) := \Phi_n(\lambda x + a) \quad (x, a \in \mathbb{R}, \lambda > 0) \tag{1.4}$$

affin transzformáltjait használjuk. A $\sqrt{\lambda}\Phi_n^{a,\lambda}$ ($n \in \mathbb{N}$) rendszer is ortonormált és teljes az $\mathcal{F}$ téren. Ebben az esetben az f legjobb approximációja az

$$S_n^{a,\lambda} f := \sum_{k=0}^{n} \langle f, \Phi_k^{a,\lambda} \rangle \Phi_k^{a,\lambda} \quad (n \in \mathbb{N}, a \in \mathbb{R}, \lambda > 0) \tag{1.5}$$

projekció és a közelítés hibája az a transzlációs és a λ dilatációs paraméter függvénye:

$$D_n^2(a, \lambda) := \|f\|^2 - \sum_{k=0}^{n} |\langle f, \Phi_k^{a,\lambda} \rangle|^2. \tag{1.6}$$

E két szabad paraméter optimalizálásával azonos együtthatószám mellett, az eredeti Hermite polinomokkal történő approximációhoz képest pontosabb közelítés érhető el. A D_n függvény minimalizálása ekvivalens az

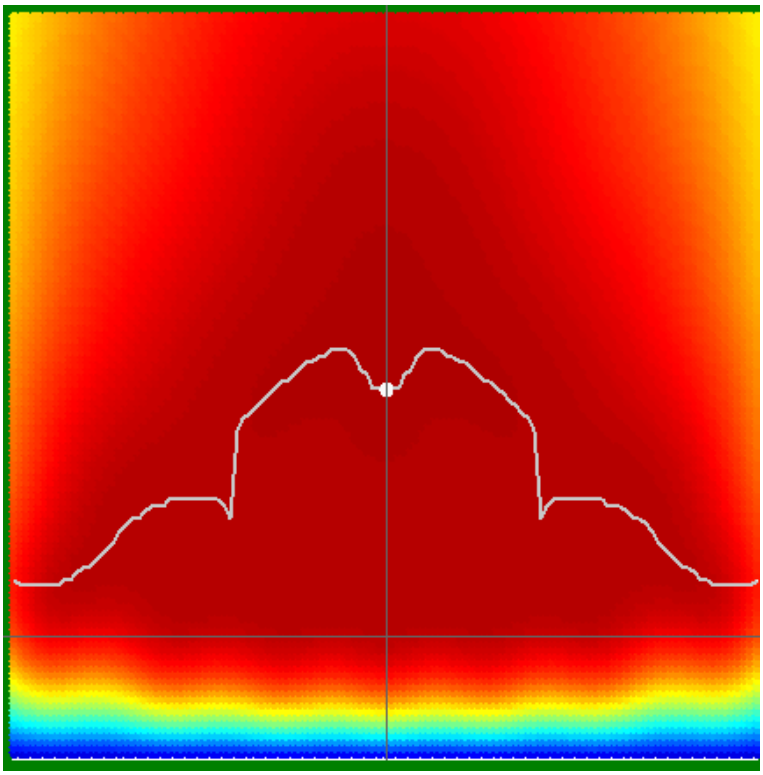
$$F_n(a, \lambda) := \sum_{k=0}^{n} |\langle f, \Phi_k^{a,\lambda} \rangle|^2$$

függvény maximumának meghatározásával. A paraméteres integrálok tulajdonságaiból következik, hogy az

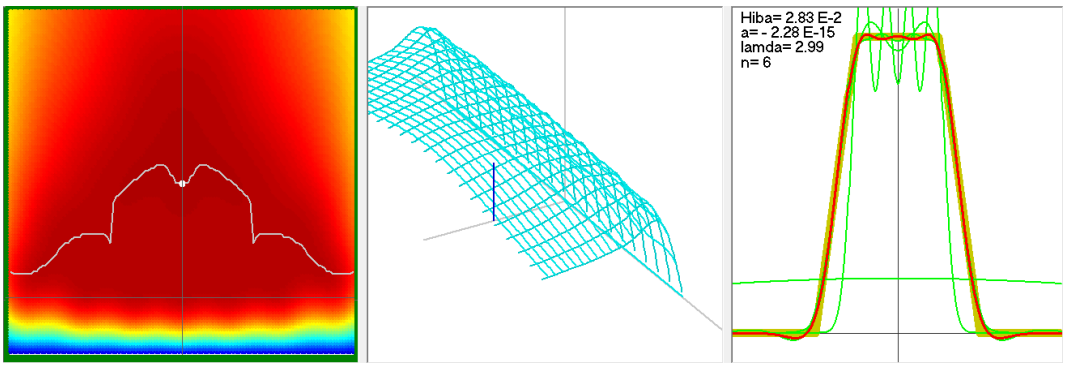
$$A_n(a, \lambda) := \langle f, \Phi_k^{a,\lambda} \rangle \quad ((a, \lambda) \in T := \mathbb{R} \times (0, \infty))$$

Fourier-együtthatók a T tartományon a paraméterek sima függvényei. Bebizonyítható, hogy $\lambda \to 0$ és $|a| + \lambda \to \infty$ esetén $F_k(a, \lambda) \to 0$, következésképpen az F_n függvénynek létezik a maximuma és a D_n függvénynek létezik a minimuma. A részleteket a Függelékben találhatók.

Az alábbi ábrák az $F_n(a, \lambda)$ függvényeket szemléltetik fényintenzitás és perspektivikus ábrázolást használva. A harmadik ablakon a jel közelí tését szemlélteti a maximum helynek megfelelő, ill. egyéb paraméter esetén.



1.2. ábra. Az F_n szí nkódos ábrázolása



1.3. ábra. Az F_n által meghatározott felület, és approximáció

1.4. A Nelder-Mead algoritmus

A $\mathsf{Nelder-Mead}$ szimplex algoritmust [] eredetileg 1965-ben fejlesztették ki azzal céllal, hogy létrehozzanak egy eljárást, amely képes meghatározni egy $f : \mathbb{R}^n \to \mathbb{R}$ nemlineáris függvény minimum (maximum) helyét a gradiens felhasználása nélkül, pusztán a függvényértékekre támszkodva. Mivel az algoritmust a bemutatott tömörítési eljárás során kétváltozós függvényekre kerül aklalmazásra, ezért ebben a speciális

esetben szemléltetjük, megjegyezve, hogy hasonló elvek szerint működik az általános n dimenziós eset is. Említendő továbbá, hogy Fejlesztői dokumentáció fejezetben bemutatott Nelder-Mead algoritmus implementációja képes kezelni az n-dimenziós esetet. A minimum meghatározásához az $f(x_1), f(x_2), f(x_3)$ adnak kiindulási pontot, melyek közül egyet lecserélendő az adott lépésben. Nevezetesen, az algoritmus

$$f(x_3) \leq f(x_2) \leq f(x_1)$$

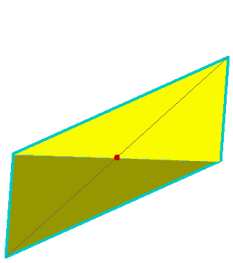
esetén olyan x' helyet keres, amelyre $f(x') \leq f(x_3) \leq f(x_2)$ teljesül és az x_3, x_2, x_1 ponthármasról az x_1-et elhagyva áttérünk az x', x_3, x_2 hármasra. Az x' pontot az előzőekből geometriai transzformációkkal származtatjuk, felhasználva az x_2x_3 szakasz $x = (x_2 + x_3)/2$ felezéspontját:

$$x' = x_1 + \alpha(x - x_1) \quad (\alpha \in \mathbb{R}) .$$

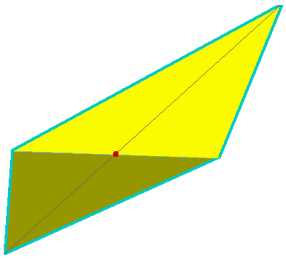
Az alábbi ábrák szemléltetik az algoritmusban használt transzformációkat. Látható, hogy $\alpha = 2$ esetén x' éppen az x_1 pont x-re vonatkozó középpontos tükrözése (T_1). Továbbá $\alpha > 2$ az eredeti háromszög tükrözése + nyújtása (T_2), az $1 < \alpha < 2$ pedig tükrözéses + zsugorításnak felel meg (T_3). A $-1 < \alpha < 0$ paraméterrel egyszerű zsugorítás adódik. Végül az 5. transzformáció (T_5) az x_3 pontból történő kicsinyítésnek feleltethető meg. Az x_1 képe ezekben a transzformációkban és a hozzá tartozó függvényértékek a következő alakban adottak:

$$x' := x_{3+i} = T_i(x_1) \quad (i = 1, 2, 3, 4), \qquad y_j = f(x_j) \quad (j = 1, 2, \cdots, 7) .$$

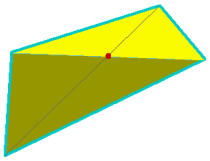
Azt, hogy mikor melyik transzformáció használandó, a Függelékben, illetve a Fejlesztői dokumentáció Nelder-Mead algoritmus implementációjával foglalkozó fejezetben található folyamatábrából látható. Az említett műveleteket a 1.4 ábra szemlélteti.



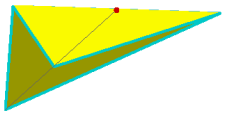
(a) Tükrözés ($T_1 : \alpha = 2$).



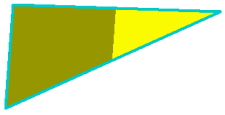
(b) T-Nyújtás ($T_2 : \alpha = 2.5$).



(c) T-Összehúzás ($T_3 : \alpha = 1.5$).



(d) Összehúzás ($T_4 : -1 < \alpha < 0$).



(e) Kicsinyítés x_3-ból (T_5).

1.4. ábra. A $\mathtt{Nelder-Mead}$ szimplex transzformációi.

Megjegyezendő, hogy a $\mathtt{Nelder-Mead}$ szimplex módszer egy determinisztikus algoritmus. Az algoritmus gyors, hiszen minden lépésben csak néhány függvénykiértékelést kell végezni. Továbbá, egyes módszerek, például a gradiens módszerrel ellentétben az olyan patologikus függvények optimumát is képes gyorsan megtalálni, mint amilyen a Rosenbrock függvény.

1.5. Kvadratúra formulák

Mivel a $\mathtt{Nelder-Mead}$ algoritmus kizárólag a függvényértékekre támaszkodik, a hibafüggvény értékeinek kiszámításához elegendő az Hermite-függvényeket valamilyen felosztás pontjaiban egyszer meghatatározni. Ez implementációs szempontból is fontos, hiszen nem kell minden $(\lambda, \mathfrak{a})$ paraméterhez kiszámolni az ettől függő, rekurzióval adott Hermite-féle függvényrendszer tagjait. Ehelyett elég az eredeti diszkrét adatsorozatot dilatálni, illetve eltolni, ami lényegesen kevesebb számítást igényel. Továbbá, az említett felosztás pontjainak valamely Hermite-függvény zérushelyeit választva, a hibafüggvényben szereplő integrálok kiszámításához a [] dolgozatban használt eljáráshoz hasonlóan kvadratúra formulát is alkalmazhatunk. Ennek előnye, hogy az alappontokban a módszer interpolál, így itt lehetséges a jel mintáinak pontos rekonstrukciója. Ugyanakkor, a kapott approximáció is pontosabb, amit a []-ben közölt numerikus tesztek is alátámasztanak.

Ahogyan ez a [] dolgozatban látható, további optimalizációs algoritmusok is alkalmazhatóak a probléma megoldásához. Egy ezek közül a leggyosrabb ereszkedés módszere, melynek az alkalmazásához szükségünk van a függvény parciális deriváltjaira. Ezek előállításával ilyen esetben külön kell foglalkozni. Az Hermite-függvények deriváltjaira vonatkozó formulák alapján a parciális deriváltakra a hibafüggvényhez hasonló ellőállítás adható. Ebben az esetben az (1.1) egyenletben szereplő integrálok kiszámításához ekvidisztans felosztása alkalmazandó.

1.6. A tömörítő eljárás jellemzése

A tömörítés kezdetekor az első feladat a teljes EKG mérést szívütésekre bontása. Ezt követően az egyes szívütések a tömörítő eljárás bemeneti paramétereként válnak értelmezhetővé. Inicializálni kell továbbá a tömörítő függvények által visszaadott, az aktuális szívütésre vonatkozó Fourier-együtthatókat, az approximáció hibáját, illetve az optimalizációs eljárások által meghatározott dilatációs és transzlációs paramétereket.

A tömörítés előkészítése nem ér véget a jel szívütésekre történő felbontásakor. A szívütéseket normalizálása is szükséges. Ez azt jelenti, hogy az első és utolsó helyen felvett értékeket összekötő egyenes, és az eredeti EKG különbségét tekinti az eljárás tömörítendő jelnek. Ezt az eljárást az irodalomban az alapvonal eliminálásnak nevezik. Ennek eredményeként a jel tartója a kezdő és a végpont által meghatározott intervallum. Végül, a szívütést normálásra kerül, vagyis az egyes értékeket elosztjuk az abszolút maximummal.

A tömörítés megkezdése előtt inicializálni kell a függvényrendszert, melynek során az egyes bázisfüggvények által felvett értékek egy mátrix soraiba kerülnek:

$$\Phi := [\Phi_n(\alpha_m)]_{0\le n<N,\, 0\le m<M} \cdot$$

A $\Phi \in \mathbb{R}^{N\times M}$ mátrix segítségével az $f \in \mathbb{R}^M$ diszkrét jel Fourier-együtthatói könnyen meghatározhatók:

$$c_n := \langle f, \Phi_n\rangle = \frac{1}{M}\Lambda^{-1}\Phi f \quad (0 \le n < N)\,,$$

ahol $\mathbb{R}^{N\times N} \ni \Lambda = \Phi\Phi^T$ Cristoffel-Darboux számokat tartalmazza. Az előállításban szereplő $\alpha_n \in \mathbb{R}\,(0 \le n < N)$ számok a ?? fejezetnek megfelelően kétféleképpen határozhatóak meg. Egyrészt a $\mathtt{Nelder-Mead}$ algoritmusban a Φ_N függvény gyökeit véve kvadratúra formulákat definiáltunk. Más optimalizációs algoritmusok, például a gradiens módszer esetén f tartóján egyenletes alapponterndszert alkalmazandó. Figyelmet érdemel, hogy az előbbi esetben a gyökök pontos meghatározása kritikus a feladat szempontjából. A probléma megoldásához a [?] könyv által javasolt numerikus eljárás található az implementációban. Nevezetesen, a H_n Hermite-polinom (5.3) rekurziójában szereplő együtthatókat egy tridiagonális mátrixba rendezzük. Könnyen belátható, hogy az α_n gyökök megegyeznek ezen tridiagonális mátrix sajátértékeivel.

Hátra van még a megfelelő (a, λ) paraméterek beállítása. Annak érdekében, hogy a reprezentáció minél adaptívabb legyen, több optimális transzláció, illetve dilatáció meghatározása szükséges. Az eredeti $f \in \mathcal{F}$ jelet tehát a következő alakban közelíti a módszer:

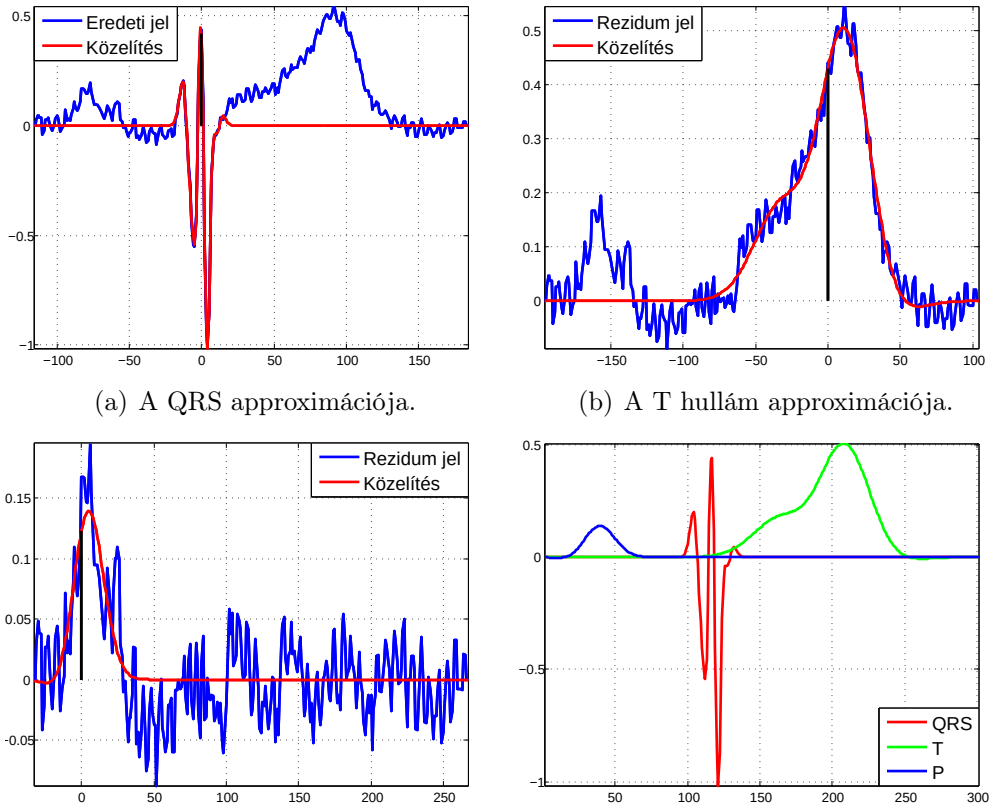
$$S_{\mathbf{n}}^{\mathbf{a},\lambda} f := \sum_{i=1}^{N}\sum_{k=0}^{n_i} \langle f^{a_i,\lambda_i}, \Phi_k\rangle \Phi_k \quad (a_k \in \mathbb{R}, \lambda_k > 0)\,,$$

ahol $\mathbf{a} = a_1, a_2, \ldots, a_N$ az alkalmazott transzlációk, $\boldsymbol{\lambda} = \lambda_1, \lambda_2, \ldots, \lambda_N$ pedig a dilatációk sorozata. Az egyes sorfejtésekhez tartozó együtthatók számát az $\mathbf{n} = n_1, n_2, \ldots, n_N$ vektor jelöli. Mivel az EKG jel alapvetően három fő hullámból áll ezért jelen probléma megoldásakor $N = 3$. Továbbá a [?] dolgozat eredményei alapján a QRS komplexumot egy heted, a T hullámot hatod, a P hullámot pedig egy másodfokú ortogonális rendszer segítségével approximálja az eljárás azaz $\mathbf{n} = 7, 6, 2$. Fontos, hogy a legjobb approximáció előállításához a (??) egyenlettel ellentétben már az eredeti függvény f^{a_i,λ_i} transzformáltja használatos. Ez nem jelent megszorítást az eredeti problémára nézve, implementációs szempontból viszont f^{a_i,λ_i} kiszámítása gyorsabb, mint a $\Phi_n^{a_i,\lambda_i}$ rendszer előállítása.

A (a_i, λ_i) paraméter párok optimalizációja egymástól függetlenül történik. Így azonban nem garantált, hogy az algoritmus a P, QRS, T hullámokat külön-külön approximálja. A problémát az irodalomban jól ismert ún. $\mathtt{Matching\ Pursuit}$ (MP) konstrukció [?] alkalmazásával oldja meg a módszer. Ez egy mohó algoritmus, mely minden lépésben a (??) egyenletben definiált $F_n(a_i, \lambda_i)$ függvény maximalizálására törekszik. Az iteráció i. lépése a következő alakban írható fel:

$$s^{(i)} = s^{(i-1)} + S_{n_i}^{a_i,\lambda_i} R^{(i-1)} \quad (1 \le i \le N)\,, \tag{1.7}$$

ahol $R^{(i)} = f - s^{(i)}$ a rezidum függvényt jelöli. Röviden tehát az $s^{(0)} = 0$, $R^{(0)} = f$ inicializálás után, az eljárás az i. lépésében megkeresi az $R^{(i-1)}$ függvény ℓ^2 norma szerinti legjobb közelítését, amit ki is von az említett rezidum vektorból. Ezt N iteráción keresztül ismétli az aktuális $R^{(i)}$ függvényre. Az MP módszer egy gyors algoritmus, mellyel megkonstruálható az $f \in \mathcal{F}$ jel ritka reprezentációja. Emellett lehetséges az EKG szívütéseinek automatikus szeparációja is, hiszen a jel különböző részeit eltérő ortogonális rendszerek segítségével közelítjük. Figyelmet érdemel, hogy az eredeti [?] módszerben ezt a lépést egy külön szegmentáló algoritmus végezte. Így a közelítés pontossága erősen függött a szegmentálás eredményétől (lsd. ?? fejezet). A kifejlesztett eljárásnál azonban ez a probléma nem áll fenn még zajos jelek esetén sem. A dolgozatban bemutatott módszer iterációs lépéseit, az $R^{(i)}$ rezidum függvények alakulását, illetve a szeparált EKG jelet az 1.5 ábra szemlélteti. Jól látható az is, hogy a fekete vonallal jelölt optimális transzláció általában nem az abszolút maximum helyén található. Ez indokolja, hogy a λ_i dilatáció mellett az a_i paraméter optimalizációjára is szükség van. Mivel ez utóbbi a kiindulásként használt [?] dolgozatból hiányzik, ezért a pontosság és tömörítési arány jelentős javulása várható el tőle.



(a) A QRS approximációja.

(b) A T hullám approximációja.

(c) A P hullám approximációja.

(d) Szeparált szívütés.

1.5. ábra. Az MP algoritmus lépései.

2. fejezet

Felhasználói dokumentáció

3. fejezet

Fejlesztői dokumentáció

Ebben a fejezetben található a Bevezetés-ben bemutatott eljárás implementációjának pontos (modul szintű) leírása, valamint az alkalmazott tesztek jellemzése illetve ezek eredményei. A tesztelés eredményeit a tapasztalok összegzése és néhány lehetséges jövőbeli fejlesztés jellemzése követi.

A fejlesztői dokumentáció első részében a tömörítési módszer implementációjának leírása található. Ez az alfejezet két fő részre tagolható: a webes felhasználói felület és az ehhez felhasznált könyvtárakat leíró, valamint a háttérben a konkrét tömörítést elvégző c++ program modulonkénti jellemzését. Mivel a webes felület megvalósítása rövid, és specializált scriptek összessége, ezért ezek jellemzését minden esetben egyedi módon érdemes kezelni. Ellentétben ezzel a szemlélettel, a c++ program nagyban épít a nyelv által támogatott objektum orientált megközelítésre. Ez lehetővé teszi és indokolja is egyben, hogy az implementációt jellemző dokumentáció is hasonlóan jól strukturált legyen. Az egyes modulok dokumentációja ugyanazon három szempont szerint közelíti meg a jellemzésüket: a modul feladatának, a modul interfészének (hogyan és mely egyéb modulokhoz kapcsolódik), valamint a modult felépítő osztályok implementációjának pontos megfogalmazása. A modulok leírásánál találhatóak továbbá a szemléltetést segítő UML és egyéb hasznos diagrammok.

A teszteléssel foglalkozó alfejezet szintén több részre tagolható. Az első alfelejezetben található a webes felhasználói felület funkcióinak tesztelési terve, illetve az egyes tesztek eredményeinek leírása. Ezt követi a c++ program moduljainak részletes tesztelési terve. A modulok tesztelését minden esetben egy külön tesztfájl végezte. Ennek leírása, az eredmények értékelése, illetve a tesztállomány elérési útja szintén megtalálható az alfejezetben. A tesztelés alfejezet utolsó része a bemutatott tömörítési eljárást hivatott tesztelni. Ez több szempont alapján, egyéb EKG tömörítő módszerekkel történő összehasonlítás útján valósul meg.

3.1. A felhasználói felület implementációja

A fejezet fő célja a webes felhasználói felületet feladatainak specifikációja, és a felületet alkotó scriptek belső felépítésének jellemzése. A fejezet elején található a webes felület

specifikációja, amely pontosan meghatározza a felülettel szemben állított elvárásokat. Az implementációt tárgyaló egyes alfejezetek két típusra bonthatóak. Először is azoknak a .php kiterjetsztésű állományoknak a dokumentációja következik, amelyekkel a felhasználó közvetlenül érintkezik. Ez azt jelenti, hogy ezen állományok alkotják a programcsomag vizuális elemeit, illetve itt kerülnek meghatározásra a felhasználók által igénybe vehető bemeneti és kimeneti szolgáltatások. A fejezet második felében kerülnek tárgyalásra azok a scriptek, amelyekkel a felhasználó nem kerül közvetlen kapcsolatba. Ezek a rövid programok biztosítják a kapcsolatot a felhasználói felület és a jelfeldolgozást végző c++ alapú programcsomag között. A fejezet legvégén található a kakukktojásnak mondható `build_all.sh` script jellemzése, melynek segítségével egyszerűen megépíthető és felkonfigurálható a programcsomag.

3.1.1. A felhasználói felület specifikációja

A felhasználói felülettel szemben alkotott elvárásokat meg kell feleltetni a dolgozat elején, a **??** fejezetben található egész programcsomagra vonatkozó elvárásoknak. Ezt leszögezve azonban fontos rámutatni, hogy egy teljes, sok tízezer mintából álló mérés tömörítése nagyon időigényes művelet, a webes felhasználói felület elsődleges célja pedig, hogy a c++ nyelven írt fő program tömörítési, illetve kitömörítési funkcióinak meglétét, illetve helyességét igazolja. Ezt a célját a felület akkor is eléri, ha nem ad szabadkezet a felhasználónak abban, hogy mely adatbázisbeli jelek mely konkrét szívütéseit tömörítse, hanem a tömöríthető jeleket a **??** fejezetben található táblázat méréseinek első három szívütésében határozza meg. Természetesen a fő programcsomag alkamas az MIT-BIH adatbázisban található bármely teljes mérés tömörítésére is, azonban ezt a jelenlegi implementáció kizárólag manuális indítással teszi lehetővé **??**. Ez a fajta korlátozás lehetővé teszi, hogy a webes felhasználói felület által nyújtott proof of concept szolgáltatások élvezhetőek maradjanak a felhasználók számára.

Az eddig ismeretett fő irányelvek ismeretében összefoglalhatóak a legfontosabb elvárások a webes felhasználói felülettel szemben, melyek egyben annak a specifikációját is adják:

- A webes felület feladata, hogy vizuálisan ismertesse egy konkrét szívütés tömörítésének menetét.
- A webes felület feladata, hogy bemutassa a fő c++ program tömörítési képességét, vagyis az elérhető jelek bármelyikének első három szívütését tömörítse. A tömörítés során létrejött .cmp szöveges állományt (amely a szívütések tömörített alakját tartalmazza), letöltésre elérhetővé kell tennie a felhasználó számára.
- A felhasználói webes felületnek biztosítania kell, hogy a felhasználó a szerverre feltölthesse a birtokában lévő .cmp állományokat. Az ezekben található tömörített szívütéseket továbbítania kell a fő program részére, hogy az a kitömörítést elvégezhesse. A kitömörítés során a keletkezett jelek mintái egy szöveges állományba kerülnek, melynek elérhetővé tétele a webes felhasználói felület feladata.

- A webes felületnek rövid jellemzést kell adnia a bemutatott módszerről, illetve az elérhető szolgáltatásokról.

A továbbiakban a webes felület alkotó scriptek egyenkénti jellemzése következik. Az itt ismertett scriptek és állományok a `/www/szd` útvonalon érhetőek el.

3.1.2. Az about.php implementációja

Az `about.php` script a `/www/szd/about.php` útvonalon érhető el. A script feladata, hogy implementálja a megfelelő HTML elemeket ahhoz a weboldalhoz, amely a felhasználói felület, illetve a bemutatott jelfeldolgozási módszer rövid ismertetését tartalmazza. Az `about.php` állományban a HTML elemeken kívül megtalálható még a `hoverChangeImg(obj, path)` javascript függvény is, amely az oldal tetején található navigációs menü animációjáért felelős. Az oldal stílusjegyeinek definícióit a `/www/szd/css/about.css` állomány tartalmazza. A weboldal böngészőn keresztül a minden oldalon megtalálható navigációs menüsor "About" linkjére történő kattintással érhető el.

3.1.3. Az animation.php implementációja

Az `animation.php` script a `/www/szd/animation.php` útvonalon érhető el. A script feladata, hogy implementálja a megfelelő HTML elemeket, javascript függvényeket, illetve egyéb php scriptek hívását annak érdekében, hogy egy konkrét szívütés tömörítése valós időben megtekinthetővé váljon. Az `animation.php` által implementált weboldalhoz tartozó stílus definíciók a `/src/www/szd/css/animation.css` állományban találhatók. A weboldal böngészőn keresztül a minden oldalon megtalálható navigációs menüsor "Animation" linkjére történő kattintással érhető el.

A script első fontos feladata, hogy a szimultán történő tömörítés egyes fázisainak megjelenítéséhez szükséges HTML elemeket meghatározza. A weboldal tetején található a navigációs menüsor, melyhez az animációt a scriptben definiált `hoverChangeImg(obj, path)` javascript függvény biztosítja. A navigációs sor alatt található egy táblázat amelybe a tömörítés egyes állapotainak ábrázolásai, illetve a közelítést jellemző hiba (PRD), quality scroe (QS) és tömörítési arány (CR) értékek kerülnek. A táblázat legnagyobb cellájában mindig a tömörítés aktuális állapotának grafikonja található, a kisebb cellákban pedig rendre a QRS, T illetve P szívütés szegmensek közelítései, amennyiben ezek az adatok már rendelkezésre állnak.

A grafikonok megjelenítését az `animation.php` script `header` részében linkelt `plot_functions.js` javascript állomány végzi amely felhasználja a szintén itt linkelt jQuerry **??** illetve Google Charts **??** webes könyvtárakat. A jQuerry egy könnyen használható javascript könyvtár amely nagyban megkönnyíti a weboldalon található HTML elemek kezelését, a szerveren elérhető állományok elérését és egyebek mellett a `animation.php` scriptben szintén alkalmazott Ajax függvény hívásokat. A Google Charts könyvtár könnyen használható, sokféle grafikon implementációját tartalmazza. Mindkét könyvtár ingyenesen felhasználható. A `plot_functions.js` állomány a

/www/szd/javascript/plot_functions.js útvonalon érhető el. A program feladata, hogy a tömörítés (vagyis az Hermite függvények segítségével előállított approximáció) aktuális állapotát megjelenítse a weboldal megfelelő grafikonján. A főprogram animációs módban történő elindítása, az egyes szegmensek közelítéseinek az optimalizáció során létrejövő állapotait a /www/szd/results útvonalon található mappákban elhelyezett szöveges állományokba írását eredményezi. Ugyanez történik az aktuális approximációra jellemző PRD, quality score és compression ratio ?? értékekkel is. A /www/szd/results mappáiban található szöveges állományok tartalma folyamatosan változik attól függően, hogy a főprogram a szívütés mely szegmensének tömörítését végzi éppen. A plot_functions.js állomány feladata, hogy amennyiben ezekben a szöveges állományokban változást észlel, a tartalmukat a megfelelő grafikonon ábrázolja. Ennek érdekében a plot_functions.js script set_interval függvények segítségével folyamatosan monitorozza a megfelelő szöveges állományok tartalmát. A set_interval függvény egy olyan javascript metódus, amely bizonyos időközönként automatikusan újrahívódik. Minden egyes újrahívásnál a program ellenőrzi, hogy éppen zajlik-e tömörítés, (amennyiben nem, a megjelenítendő szöveges állományok egy speciális karaktersort kapnak értékül) illetve történt-e változás az egyes szöveges állományokban. Amennyiben igen, a script frissíti a megfelelő grafikon tartalmát a weboldalon. A tömörítés alatt végig a legnagyobb cellában látható az aktuális szegmens (QRS, T, P) közelítésének optimalizációja. Amikor az egyes szegmensek tömörítése véget ért, az approximáció és a reziduum jel grafikonjai a weboldalon található táblázat egy kisebb elemébe kerülnek. A tömörítés végén a táblázat legnagyobb cellájában egyszerre tekinthetőek meg az egyes szegmensek approximációi, illetve az eredeti szívütés grafikonja. A táblázat azon cellájában, amely tömörítés jellemzőit tartalmazza mindig a legnagyobb cellában található approximációhoz tartozó PRD, quality score és compression ratio értékek olvashatók (értelemszerűen pár másodperces csúszás előfordulhat amikor új szegmens kerül tömörítésre).

A animation.php script felel tehát azért, hogy egy példa szívütés tömörítése ábrázolva legyen. Az animációt a felhasználók a weboldalon található táblázat legnagyobb cellájára való kattintással tudják elindítani. Bár végső soron innen indul a főprogram animációs üzemmódban történő elindítása, mégsem az animation.php implementálja az ehhez megfelelő rendszerhívást. Ennek szükségessége abban rejlik, hogy amennyiben az animation.php script közvetlenül indítaná el a tömörítő programot, a böngészőnek meg kellene várnia annak lefutását, mielőtt a táblázatot megjelenítené. Azért, hogy ez a jelenség elkerülhetővé váljon, az animation.php egy úgynevezett Ajax hívást intéz a launch_animation.php állományhoz. Az Ajax egy olyan kliens oldali script amely lehetővé tesz a szerverrel történő kommunikációt a weboldal frissítése nélkül. Ezt alkalmazva az animation.php, a megfelelő grafikonra történő kattintás után meghívja a launch_animation.php scriptet amely így anélkül képes elindítani a tömörítő programot, hogy az animation.php által implementált weboldal elemeinek betöltése megakadna.

3.1.4. A compressionNormal.php implementációja

Az `compression_normal.php` script a `/www/szd/compression_normal.php` útvonalon érhető el. A script feladata, hogy implementálja azokat HTML elemeket, illetve javascript függvényeket amelyek lehetővé teszik egy tömöríteni kívánt jel kiválasztását. Ezen kívül a `compression_normal.php` feladata az is, hogy sikeres kiválasztás esetén a kiválasztott jel tömörítésének folyamatát elindítsa. Ez azt jelenti, hogy amennyiben a felhasználó sikeresen kiválaszt egyet a tömöríthető jelek közül a `compression_normal.php` script a böngészőt a `compression_loading.php` oldalra navigálja, valamint továbbítja a kiválasztott jel azonosítóját.

A kiválasztás menetét a `compression_normal.php` scriptben található javascript függvények implementálják. A weboldal betöltésekor a navigációs menü alatt egy üres táblázat található melynek csupán két cellája aktív. Az aktív cellákat kék színű hátterük, illetve a rajtuk található feliratok jelzik. A "Jelek" feliratú cellára kattintva a táblázat megtelik a választható mérések azonosítóival. Bár a beuszó effektust a `compression_normal.css` állományban található stílusdefiníciók adják, a cellák feltöltését a `compression_normal.php` scriptben található `showRecords()` javascript függvény végzi. A függvény két féle képpen lehet hatással a weboldalon található táblázat elemeire. Amennyiben a táblázat üres, úgy megtölti a tömöríthető mérések azonosítóival, az azonosítókat véletlenszerűen szétszórva a cellákban. Mivel összesen hét mérés áll rendelkezésre, ezért a véletlenszerű elrendezés még nem okoz túl nagy kellemetlenséget, ellenben érdekes stílust kölcsönöz a weboldal megjelenésének. Nem üres (mérés azonosítókat is tartalmazó) táblázat esetén viszont a `showRecords()` függvény meghívja a `hideRecords()` eljárást, amely kitörli a mérések neveit a táblázatból. Ennek eredménye képpen kétszer kattintva a "Jelek" cellára, először a táblázat celláinak feltöltését, majd kitörlését idézheti elő a felhasználó.

Amennyiben a kiválasztható mérések azonosítói láthatóak a táblázatban, ezek valamelyikére is kattinthat a felhasználó. Sikeres kattintás esetén meghívódik a `hideRecords()` függvény, amely paraméterül kapja a kiválasztott cella koordinátáját (sorindexét illetve oszlopindexét). A `hideRecords()` függvény működése úgy lett definiálva, hogy a táblázat minden elemét törölje, kivéve azt amelynek koordinátáját paraméterül kapta. Így egy mérés nevére kattintva eltünthető a többi mérés azonosítója. Ezzel véget ér a kiválasztás folyamata és a felhasználó megkezdheti a tömörítést a "Tömörítés" feliratú cellára kattintva.

A "Tömörítés" feliratú cellára történő kattintás meghívja a `beginCompression()` javascript függvényt. A függvény először ellenőrzi, hogy a mérések nevei láthatóak-e a táblázatban. Amennyiben nem, megjelenít egy erre figyelmeztető üzenetet a táblázat mellett, és visszatér. Ha a mérések azonosítói meg vannak jelenítva, a `beginCompression()` függvény megszámolja, hogy hány mérés látható. Egynél több mérés esetén nem egyértelmű, hogy a felhasználó melyiket kívánja tömöríteni, így a függvény ismét hibaüzenettel tér vissza. Abban az esetben, ha mindössze egyetlen mérés azonosítója látható, (vagyis a felhasználó sikeresen kiválasztott egy tömörítendő jelet), a függvény a `compression_loading.php` oldalra navigál, a `$_GET['sigID']` értékét pedig a kiválasztott jel azonosítójára állítja.

3.1.5. A compressionLoading.php implementációja

A `compression_loading.php` script a `/www/szd/compression_loading.php` útvonalon érhető el. A `compression_loading.php` scriptnek több feladata is van. Egyrészt implementálnia kell azokat a HTML elemeket amelyek a felhasználó a tömörítés alatt, illetve annak végeztével lát, valamint meg kell hívnia a főprogramot rendszerhíváson keresztül invokáló `launch_compression.php` scriptet. A `compression_loading.php` weboldal nem érhető el a navigációs menün keresztül, ide a felhasználó csak egy tömöríthető jel kiválasztásával juthat.

A `compression_loading.php` script először ellenőrzi a tömöríteni kívánt jel azonosítóját. Amennyiben az azonosító nem található az általa ismert tömöríthető mérések tömbjében, a script megjelenít egy erre figyelmeztető képet, és nem kezdi el a tömörítés folyamatát. Ha `$_GET['sigID']` változóban kapott azonosító benne van a listában, a `compression_loading.php` először megjeleníti a tömörítés időtartama alatt látható animációt, majd létrehoz egy Ajax hívást a `launch_compression.php` scripthez. Az Ajax hívás paramétereként a tömörítendő jel azonosítóját adja meg.

Amikor a tömörítés befejeződött, az Ajax hívásból sikeres visszatérés esetén a `compression_loading.php` script lecseréli a várakozást jelző animációt egy olyan képre amely a tömörítés befejezését jelzi. A képre kattintva a felhasználó letöltheti a tömörített jelet tartalmazó .cmp szöveges állományt. Ezt a folyamatot a `compression_loading.php` állományban definiált `createLinkForDownload()` javascript függvény végzi. Amennyiben az Ajax hívás valamilyen oknál fogva sikertelen volt (vagyis a jeltömörítés nem ért véget hiba nélkül), a `compression_loading.php` script ezt egy felugró hibaüzenettel jelzi a felhasználó felé.

3.1.6. A decompressionNormal.php implementációja

3.1.7. A decompressionloading.php implementációja

3.1.8. A launchanimation.php implementációja

3.1.9. A launchcompression.php implementációja

3.1.10. A launchdecompression.php implementációja

3.1.11. A webes felület működéséhez szükséges további mappák és tartalmuk

3.2. A tömörítő eljárás implementációja

A fejlesztői dokumentáció bevezetésében leírtakkal összhangban, a tömörítő eljárás implementációjával foglalkozó fejezet minden pontja három részre tagolható. Minden alfejezet az aktuális modul feladatának pontos megfogalmazásával kezdődik. Ezt

követi a modul interfészének definiálása, melynek segítségével átlátható a modul kapcsolata a program többi részével. Végül a modult alkotó osztályok és strukturák fő metódusainak, illetve adattagjainak leírása következik. A program robosztussága indokolja, hogy minden egyes metódus és függvény ne kerüljön itt jellemzésre, azonban a program forráskódjában minden metódus előtt megtalálható annak rövid jellemzése, elvárt bemenete, és esetleges visszaadott értékének típusa. Az egyes modulok dokumentációja tartalmazza ezen kívűl az ezeket leíró UML, és egyéb diagrammokat. Az egyes modulok imlpementációja külön .h, és .cpp kiterjesztésű állományokban található, ezért az egyes alfejezetek címei megegyeznek ezen állományok elnevezésével. A fejezet utolsó alfejezete leírást biztosít az implementáció által felhasznált külső könyvtárakhoz, illetve indokolja ezek választását.

3.2.1. A SigPrep modul implementációja

Az implementációt tartalmazó mappaszerkezet gyökerét /-vel jelölve, a SigPrep modult az `/src/headers/SigPrep.h`, illetve az `/src/SigPrep.cpp` állományok implementálják.

A modul feladata

A modul feladata, hogy egységes módszert biztosítson a bemeneti jelek kezeléséhez. Mivel az eljárás EKG jelek tömörítésével foglalkozik, illetve speciálisan az MIT-BIH adatbázisban megtalálható egyedi formátumú jelek feldolgozásával, ezért egy általános jel kezelő modul implementációja nem volt szükséges a módszer megvalósításához. Annak érdekében azonban, hogy hogy a bemutatott jel tömörítő eljárást jövőbeli vizsgálódások során könnyen ki lehessen próbálni egyéb jel típusokon, fontos az ehhez szükséges, általános metódusok definiálása. A SigPrep modul feladata tehát, hogy definiálja azokat az egységes függvényeket amelyek elengedhetetlenek a bemeneti jelek programon belül történő kezeléséhez. Ezen kezelési módszerek pontos megvalósítását a `/src/headers/SigPrep.h` állományban található SigPrep osztályból származtatatott osztályok, (a jelenlegi programcsomagban a `EcgSigPrep` osztály), írják le.

A modul interfésze

A SigPrep modul közvetlenül nem kerül példányosításra a feladat megoldásának érdekében. Külső szolgáltatásokat a modul nem vesz igénybe a programcsomag egyéb elemeitől, azonban az általa definiált metódusokra erősen támaszkodik a `SigPrep` osztályból származtatott `EcgSigPrep` osztály. A SigPrep modul legfontosabb szolgáltatása a belőle származtatott osztályok felé, hogy pontosan meghatározza egy tömöríthető jel formáját és elemeit. További szolgáltatásként a modul biztosítja az átírható (virtuális) `virtual void SigPrep::setSignal()` metódust a jel éppen tömöríteni kívánt részének inicializásához. Az eddig felsoroltakon kívül a SigPrep modul getter függvényeket definiál a teljes tömörítendő jel (például sok órányi EKG jel), az

éppen tömörítendő jel részlet (egyetlen szívütés), illetve ennek bizonyos szignifikáns értékeihez.

A modul implementációja

A SigPrep modult kizárólag a `SigPrep` osztály alkotja. Összesen öt protected hozzáférhetőségű adattaggal rendelkezik. Ezek közül kettő `Eigen::MatrixXd` mátrix típusra mutató pointer, melyek feladata a tömörítendő jelhez való hozzáférés biztosítása. Az `entire_signal` azonosítójú mutatón keresztül lesz képes a `SigPrep` osztály elérni a teljes EKG mérést. A második mutató a `signal` azonosítóval van ellátva, és a teljes mérésen belül egy-egy szívütéshez való hozzáférést biztosítja. Ennek megadása fontos, mivel egy EKG mérés jellemzően sok szívütésből áll a bemutatott tömörítési eljárás pedig szívütésenként alkalmazható. A mutatókon kívül még három protected hozzáférhetőségű, `double` adattagot definiál a `SigPrep` osztály. Ezek rendre a `sig_first_val, sig_last_val`, illetve `sig_max_val` azonosítókkal vannak ellátva. Ezek az adattagok tárolják az aktuális, tömörítendő szívütés értékeit az eredeti jel első, utolsó és legnagyobb abszolút értékű pontjában. Erre az információra abban az esetben lehet szükség, ha a kitömörített jelet a méréssel azonos formára kell hozni, ugyanis magát a tömörítést egy "normált" jelen végzi a program. Ennek a "normálásnak" egy lehetséges megvalósítását implementálja a szintén protected hozzáférhetőségű `virtual void setSignal()` metódus. A metódusnak két fontos feladata van: az eredeti szívütésből kivonni az első és utolsó pontját összekötő egyenest, ezáltal eltávolítva a szívütésben található "baseline wandert" **??**, valamint elosztani a jel összes értékét a legnagyobb abszolút értékű elemével. Ez transzformáció lehetővé teszi a szívütések pontosabb közelítését olyan ortogonális függvényrendszerek segítségével (mint például az Hermite függvények), amelyek gyorsan tartanak a 0-hoz, ha $|x| \to \infty$. Természetesen a `setSignal()` definíciója nem jelent optimális előkészítést minden lehetséges tömörítési eljárás esetén, ezért a metódust `virtual`-ként deklarálja a `SigPrep` osztály. Ez lehetővé teszi, hogy a `SigPrep` osztályból származtatott jelkezelő osztályok felüldefiniálhassák a `setSignal()` metódust.

A `SigPrep` osztály konstruktorának egyetlen feladata, hogy a dinamikus memóriában helyet foglaljon a `signal` és az `entire_signal` pointerek által mutatott mátrixoknak. Az osztály destruktora ezeket a címeket szabadítja fel. Ezeken kívül összesen öt publikus hozzáférhetőségű függvényt definiál.
A `const Eigen::MatrixXd* getSignal()`, illetve a `const Eigen::MatrixXd* getEntireSignal()` konstans mutatótak adnak vissza, melyek rendre a `signal` és az `entire_signal` adattagokra mutatnak. Ezek a függvények lehetővé teszik az objektum által feldolgozni kívánt jel megismerését más modulok számára, azonban ezen a jelen változtatásokat kizárólag a `SigPrep`, illetve az ebből származtatott osztályok végezhetnek.

Az osztály további publikus függvényei konstans `double` értékeket adnak vissza, és a `const double getSigFirstVal()`, `const double getSigLastVal()` illetve `const double getSigMaxVal()` szignatúrák által azonosíthatók. Ezek a függvények a protected hozzáférhetőségű `virtual void setSignal()` függvény által megváltoztatott

jel értékek lekérdezésére képesek. Ugyan a programcsomag jelenleg nem használja fel őket, a helyreállított (kitömörített) jel eredeti formára hozásánál lenne alkalmazható az általuk biztosított szolgáltatás.

3.2.2. Az EcgSigPrep modul implementációja

Az implementációt tartalmazó mappaszerkezet gyökerét /-vel jelölve, az EcgSigPrep modult az `/src/headers/EcgSigPrep.h`, illetve az `/src/EcgSigPrep.cpp` állományok implementálják.

A modul feladata

A modul feladata, hogy definiálja azokat a függvényeket és metódusokat amelyek az MIT-BIH EKG jel adatbázisban található méréseket egy Hermite függvény rendszer segítségével tömöríthető állapotba hozzák. A modul célja, hogy a fennt említett adatbázisban szereplő bármely mérés azonosítóját kapva bemenetül, a mérésben szereplő, tömörítendő elvezetést egy mátrixos alakban adja meg. A feladat részét képezi továbbá, hogy a modul képes legyen az így megadott jel összehúzására és eltolására különböző dilatációs és transzlációs adatok esetén. Összefoglalva tehát a modul legfőbb feladata, hogy Hermite függvényekkel történő tömörítés esetén a bemeneti EKG jeleket kezelhető állapotba alakítsa, illetve szükség esetén manipulálja őket.

A modul interfésze

Az EcgSigPrep modult a `/src/headers/EcgSigPrep.h` állományban deklarált `EcgSigPrep` osztály alkotja. Az `EcgSigPrep` osztály a `SigPrep` osztályból lett származtatva, így birtokolja annak minden protected, illetve publikus hozzáférhetőségű adattagját és függvényét. Ez egyben úgy is értelmezhető, hogy az EcgSigPrep modul közvetlenül igénybe veszi a SigPrep modul szolgáltatásait. Az EcgSigPrep modul a feladatának megfelelően az MIT-BIH EKG adatbázisban található jelek ábrázolásához és manipulálásához biztosít szolgáltatásokat. Ezeket közvetlenül a MatchingPursuit modul veszi igénybe, a MatchingPursuit modulon keresztül pedig az EcgSigPrep osztályból példányosított objektumok által kiszámított jel értékekre támaszkodnak az OrtCompresser és NelderMead osztályok szolgáltatásai.

A modul implementációja

Az `EcgSigPrep` osztály a `SigPrep` osztálytól megörökölteken kívül további négy protected hozzáférhetőségű adattagot definiál. Az `std::queue<WFDB_Annotation> annotations` adattag tárolja az adatbázisból letöltött jelhez tartozó annotációkat (az EKG jel orvosok által megjelölt nevezetes pontjait), olyan sorrendben ahogyan azok a mérésben szerepelnek. A jelenlegi implementáció ezeket az annotációkat kizárólag az EKG mérés szívütésekre bontásához alkalmazza, azonban későbbi fejlesztések (például osztályozás implementációja) felhasználhatnák ezeket az adatokat ellenőrző értékek

gyanánt. A `curr_pos` egész típusú adattag hivatott tárolni a mérés tömörítésekor az éppen aktuális szívütés sorszámát. A `dilat`, illetve `trans` azonosítójú adattagok a jel összehúzásának illetve eltolásának mértékét tárolják. Ezek rendre `double`, illetve `int` típusúak. A transzlációs paramétert egész típusúnak definiálja az `EcgSigPrep` osztály, mivel az eltolást a jelet tartalmazó mátrix ($\mathtt{signal} \in \mathbb{R}^{\mathtt{l}\times\mathtt{n}}$) indexein definiálja. A `SigPrep` osztálytól megöröklt `virtual void SigPrep::setSignal()` metódus definícióján az `EcgSigPrep` osztály nem változtat, illetve ezen kívül egyéb protected hozzáférhetőségű függvényt nem definiál.

Az `EcgSigPrep` osztály konstruktora bemeneti paraméterként a feldolgozandó EKG mérés azonosítóját, a mérésben található elvezetések számát, illetve a mérést alkotó mintavételi pontok számát várja. Utóbbi paramétert önkényesen megválasztva lehetséges a teljes mérés helyett annak mindössze egy részét feldolgozni. A konstruktor két fő feladata, hogy a mérésekben szereplő értékeket a `signal` illetve `entire_signal` pointerek által mutatott memória területre másolja, illetve feltöltse az `annotations` sort az egymást követő QRS komplexumot jelölő minták sorszámaival. Ezen feladatok elvégzésének érdekében a konstruktor felhasználja a `wfdb`, EKG mérés kezelő külső könyvtár függvényeit és típusait. A könyvtár dokumentációjáról, illetve főbb jellemzőiről a **??** alfejezet értekezik. Miután az EKG mérés beolvasásra került, illetve az `annotations` sor feltöltődött a megfelelő pozíciókkal, a konstruktor meghívja az `EcgSigPrep::getNextSegment()` függvényt, ezzel biztosítva, hogy a `signal` pointer által mutatott területre a mérésben szereplő legelső szívütés minái kerülnek. Az `EcgSigPrep` osztály nem definiálja felül, illetve nem egészíti ki a `SigPrep` osztályban implementált destruktort.

A konstruktoron és a destruktoron, valamint a `SigPrep` osztálytól megörökölt publikus hozzáférhetőségű függvényeken kívül az `EcgSigPrep` osztály további négy publikus metódust definiál. Az első ezek közül a konstruktor által is igénybe vett `const Eigen::MatrixXd* getNextSegment()` szignatúrával elátott függvény. Ennek a függvénynek a feladata, hogy a feldolgozás során, a mérésben soron következő szívütés mintáit a `signal` pointer által mutatott memória területre másolja. A megvalósítás először ellenőrzi, hogy található-e még fel nem dolgozott szívütés az `entire_signal` pointer által mutatott, teljes mérést tartalmazó mátrixban. Amennyiben igen, az `annotations` sorból egy `pop()` művelet segítségével a következő R csúcsot tartalmazó annotáció kivételre kerül. A **??** tanulmányban írtak szerint a szívütés végét az így kapott R csúcstól számított következő 150 minta jelenti. Amennyiben ez a szám meghaladná a mérésben található minták számát,úgy a függvény a mérésben található utolsó mintát tekinti a szívütés végének. A függvény a `curr_pos`, és a szívütés vége között található mintákat az `entire_signal` által mutatott mátrixból a `signal` pointer által mutatott mátrixba másolja át. Ezt követően a függvény a jelenlegi mérésbeli pozíció, a `curr_pos` adattag értékét lecseréli a szívütés utolsó mintájának indexére. Végül a függvény meghívja a `SigPrep::setSignal()` **??** metódust, amely normalizálja a szívütést, majd visszaad egy a `signal` pointerrel megegyező konstans mutatót.

Az `EcgSigPrep` osztály publikus függvényeként definiált `Eigen::MatrixXd setDilatTrans(const double &l, const double &t, const Eigen::MatrixXd*`

alpha, Eigen::MatrixXd& sig) szignatúrával ellátott metódus felel a paraméterként kapott dilatáció illetve transzláció a szintén bemeneti paraméterként kapott szívütésen történő alkalmazásáért. A függvény alpha azonosítóval ellátott paramétere egy magas fokú Hermite függvény zérushelyeit tartalmazó mátrixra ($\mathtt{alpha} \in \mathbb{R}^{1*n}$) mutató pointer. A setDilatTans függvény először meggyőződik róla, hogy a paraméterként kapott dilatáció nullától eltérő értékkel bír. Amennyiben a dilatáció értéke nulla, a függvény ezt 1-re változtatja. Ezt követően a paraméterként kapott transzlációs értéket egész értékké alakítja. Miután kiszámította a végleges dilatációs és a transzlációs értékeket, a függvény definiálja az X,Y n elemű std::vector<double> típusú adatszerkezeteket, valamint az ArrayXd típusú domain azonosítóval ellátott, 0-ra szimmetrikus n elemű, egyenletes felosztású intervallumot, ahol n a szívütésben szereplő minták számát jelöli. Az X vektorba kerülnek a domain tömb dilatációval megszorzott elemei. Ez maga az "összehúzott" intervallum, amely fölött a függvény a jelet értelmezni fogja. Az Y vektor először a sig azonosítóval ellátott bemeneti jel elemeit kapja értékül. Az eltolás az Y vektor elemein alkalmazott std::rotate standard library-ban található függvény segítségével valósul meg. Ezt követően a setDilatTrans függvény a Spline.h külső állomány segítségével definiál egy spline-t, amely Y értékeit az X pontjaiban interpolálja. Az összehúzott és eltolt jel értékei megegyeznek az interpolációs spline értékeivel abban az esetben, ha annak az éppen vizsgált pontja beleesik a magas fokú Hermite függvény gyökei által kifeszített intervallumba.

Az EcgSigPrep osztály további két publikus függvénye a const double értékeket visszaadó getDilat(), és getTrans() függvények. Ezek rendre az aktuális szívütésre alkalmazott dilatációs és transzlációs értékeket adják vissza.

3.2.3. Az OrtFunSys modul implementációja

Az implementációt tartalmazó mappaszerkezet gyökerét /-vel jelölve, az OrtFunSys modult az /src/headers /OrtFunSys.h, illetve az /src/OrtFunSys.cpp állományok implementálják.

A modul feladata

A modul feladata, hogy definiálja azokat a konténereket illetve absztrakt metódusokat amelyek segítségével megadható egy a program által felhasználható függvényrendszer. A Bevezetés fejezetben leírt eljárás speciálisan az ott bemutatott Hermite függvényrendszert alkalmazza az EKG jelek tömörítéséhez. A bemutatott algoritmusokat azonban további, különböző tulajdonságú ortogonális (vagy akár csak lineárisan független) függvényrendszerek segítségével is érdemes lehet kipróbálni. Annak érdekében, hogy ez könnyedén implementálható legyen, fontos hogy az egyes függvényrendszereket kezelő metódusok absztrak módon legyenek definiálva. Az OrtFunSys modul feladata tehát, hogy megadja azokat az adatszerkezeteket és absztrakt metódusokat, amelyek segítségével a program képes függvényrendszerek kezelésére. Az egyes függvényrendszerek, példaul az Hermite függvények pontos implementációja a

`/src/headers/OrtFunSys.h` állományban definiált OrtFunSys osztályból származtatott osztályokban található.

A modul interfésze

A modul az absztrakt OrtFunSys osztályból áll, amely tartalmaz úgynevezett pure virtual (tisztán virtuális) függvényeket. Ennek következtében az osztály által definiált típusból objektumokat példányosítani nem lehet. A példányosítható objektumok hiányában az OrtFunSys modulhoz nem határozható meg konkrét interfész, melynek segítségével az OrtFunSys típusú objektumok szolgáltatásokat nyújthatnának a program többi részének. Megjegyezendő azonban, hogy az OrtFunSys osztályból származtatott osztályok objektumainak szolgáltatásait az OrtCompresser modul használja fel.

A modul implementációja

A modult alkotó OrtFunSys osztály definíciója a `/src/headers/OrtFunSys.h` állományban található. A bemutatott tömörítő eljárás egyik legköltségesebb művelete a felhasznált függvényrendszer értékeinek kiszámítása. Mivel a függvényértékek meghatározása bármely függvényrendszer esetében szükséges, ezért a kiszámított értékek tárolásának logikáját illetve a felhasznált adatszerkezeteket az OrtFunSys osztály határozza meg. Egyes függvényrendszerek (például az Hermite függvényrendszer) tulajdonságai lehetővé teszik, hogy a rendszert jellemző paraméterek (Hermite rendszer esetében a dilatáció illetve a transzláció) közvetlenül a jelben történő változtatását. Ilyen esetben a tömörítő eljárás során elegendő a felhasznált függvényrendszer értékeinek egyszeri meghatározása, hiszen a közelítést optimalizáló algoritmusok a rendszer paramétereit közvetlenül a bemeneti jelre alkalmazhatják. Ennek következményeként az OrtFunSys osztály a függvényrendszer értékeit a heap-en, dinamikus memóriában tárolja.

Az osztály négy protected hozzáférhetőségű adattaggal rendelkezik. A `rootNum` egész típusú adattag határozza meg a legmagasabb fokszámú függvény zérushelyeinek számát. Ezen zérushelyek kiszámítása fontos az 1.5 fejezetben bemutatott kvadratúra alkalmazásához. Ugyanez az adattag határozza meg az alkalmazott függvényrendszer fokszámát is. Az adattag elnevezése elfogadható, mivel egy n-ed fokú polimomnak amely egy ortogonális polinomrendszer tagja, pontosan n darab különböző és valós gyöke létezik.

Az osztály további három adattagja `Eigen::MatrixXd*` típusú, azaz nem rögzített méretű mátrixokra mutató pointer típusú. Az általuk mutatott mátrixokat a tömörítésért felelős modulok (Compresser, OrtCompresser) modulok többször használják fel, azonban értéküket elegendő szívütésenként egyszer kiszámolni. A `domain` pointer által mutatott mátrixba kerül az alkalmazott függvényrendszer felhasznált értelmezési tartománya, a `bigSys` és a `lambda` pointerek által mutatott mátrixok pedig magát a függvényrendszert, illetve a kvadratúra formula alkalmazásához szüksé-

ges Cristoffel-Darboux számokat tartalmazzák. Az osztály nem rendelkezik protected hozzáférhetőségű függvényekkel, illetve publikus adattagokkal.

Az OrtFunSys osztály a konstruktorán, és virtuális destruktorán kívül kizárólag tisztán virtuális (pure virtual) függvényekkel rendelkezik. Ezek azokat a funkciókat határozzák meg, melyek implementálására minden függvényrendszert megvalósító osztály kötelezve van. A legfontosabb ilyen függvény a konstans `Eigen::MatrixXd` értéket visszaadó `OrtSysGen`. Az OrtFunSys osztályból származtatott osztályok esetén az `OrtSysGen` hivatott az általa implementált konkrét függvényrendszer generálására. A további tisztán virtuális függvények biztonságos elérési (getter) felület biztosítanak az egyes protected elérhetőségű adattagokhoz.

A konstruktor, illetve a virtuális destruktor implementációi a `/src/OrtFunSys.cpp` állományban találhatóak. A konstruktornak egy egész típusú paramétere van amely a függvényrendszer fokszáma. Az OrtFunSys osztály konstruktora lefoglal a dinamikus memóriában két N*N, illetve egy 1*N dimenziós mátrixnak szükséges helyet, melyek címei lesznek a bigSys, lambda és domain pointerek értékei. A konstruktor pointerek által mutatott mátrixok minden elemét nullára inicializálja. Az osztály destruktora felszabadítja a lefoglalt memóriaterületeket.

3.2.4. A Hermite modul implementációja

Az implementációt tartalmazó mappaszerkezet gyökerét /-vel jelölve, az Hermite modult az `/src/headers/Hermite.h`, illetve az `/src/Hermite.cpp` állományok implementálják.

A modul feladata

Az Hermite modul feladata, hogy meghatározza a tömörítő eljárás által felhasznált Hermite függvényeket, illetve a felhasználásukhoz szükséges egyéb értékeket. Az Herimte modul a `/src/headers/Hermite.h` állományban található Hermite osztályból áll. Az Hermite osztály felelős az Hermite függvények értékeinek kiszámításáért. További feladata, hogy kiszámítsa az 1.5-ben alkalmazott kvadratúra formulához szükséges Cristoffel-Darboux számokat, valamint implementáljon a protected elérhetőségű adattagjaihoz biztonságos getter metódusokat.

A modul interfésze

Az Hermite modul interfésze két részre bontható. Az Hermite osztály bemenetét a tömöríteni kívánt jel méréseinek száma jelenti. Ez az információ az EcgSigPrep modul által ismert, és biztosított. Az Hermite osztályból létrehozott objektum adattagként a MatchingPursuit modul ugyanilyen nevű osztályában található. Szolgáltatásait a Matching Pursuit modulon kívül az OrtCompresser, és az EcgSigPrep modulok veszik igénybe. Az OrtCompresser modul a jel a függvényrendszer által kifeszített altérre vett ortogonális projekcióját hivatott kiszámítani, ehhez szükséges számára az

Hermite rendszer ismerete. Az EcgSigPrep modul pedig a jel összehúzása és eltolása során használja fel az Hermite modul által biztosított magas fokú Hermite függvény gyököket.

A modul implementációja

Az Hermite osztály kizárólag az OrtFunSys osztálytól megörökölt négy protected hozzáférhetőségű adattaggal rendelkezik. Az osztály definiál három protected hozzáférhetőségű függvényt, melyeket a konstruktorában hív meg. Ezek a függvények felelősek az Hermite függvényrendszer, és a kapcsolódó értékek (értelmezési tartomány felhasznált pontjai, Cristoffel-Darboux számok) kiszámításáért. Az első Hermite osztály által felhasznált protected hozzáférhetőségű függvény a `void` értéket visszaadó `ortfunsysroots`. A függvény egy a [**?**]-ben található eljárást implementál, amely képes meghatározni magas fokszámú ortogonális polinomok zérushelyeit. Az algoritmus első részében egy Jacobi mátrix konstrukciójára kerül sor. Az algoritmus úgy építi fel a mátrixot, hogy a konstrukció végén annak sajátértékei pont egy n-ed fokú Hermite függvény zérushelyei lesznek. A sajátértékek meghatározását az `Eigen` könyvtárban található `Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd>` típusú objektum végzi. A kiszámított sajátértékeket a függvény értékül adja a `domain` OrtFunSys osztályból megörökölt pointer által mutatott mátrixnak. Az hogy pontosan milyen fokszámú Hermite függvény gyökeit számolja ki a függvény, megegyezik a `rootNum` adattag értékével.

Az `ortfunsysgen` protected hozzáférhetőségű `void` értéket visszaadó függvény felel a tömörítéshez felhasznált Hermite rendszer kiszámításáért. Az egyes Hermite függvényeket a `domain` pointer által mutatott mátrix elemeiben számítja ki, és egy `PHI` azonisítójú, `Eigen::ArrayXXd` típusú többdimenziós, `double` típusú elemeket tartalmazó tömb soraiban tárolja. A függvény a **??** fejezetben bemutatott rekurziós formulát implementálja. Miután az összes (0 fokútól `rootNum - 1` fokúig) Hermite függvény kiszámításra került, a metódus normalizálja a `PHI` tömbböt. Ez a tömb összes elemének $\Pi^{(-1/4)}$-el való megszorzásával történik. A normalizálást követően, a `PHI` objektumot `Eigen::MatrixXd` típusúvá konvertálja a metódus, és az így kapott mátrixot értékül adja a `bigSys` pointer által mutatott mátrixnak.

Az `ortfunsyslamb` protected hozzáférhetőségű függvény feladata a kiszámított Hermite rendszerhez tartozó Crisstoffel-Darboux számok megadása. Szerencsére a [**?**] alapján ez könnyen kiszámítható. A megoldást, amely az Hermite rendszert tartalmazó `bigSys` pointer által mutatott mátrix saját maga transzponáltjával vett szorzata, a `lambda` pointer által mutatott mátrix kapja értékül.

A konstruktoron és a destruktoron kívül, az Hermite osztály publikus függvényei az OrtFunSys osztály által megadott tisztán virtuális függvényeket implementálják. A `getortfunlamb`, `getortfunsys`, és a `getortfunroots` getter metódusok rendre a `lambda`, `bigSys` illetve a `domain` pointerek konstans változatait adják vissza.

Az `const Eigen::MatrixXd` értéket visszaadó, `OrtSysGen` publikus függvény az Hermite függvényrendszer generálására képes megadott pontok felett. Erre a függvényre a tömörített jel helyreállítása során van szüksége az OrtCompresser modulnak.

A függvény az `ortfunsysgen` függvény működésével azonos módon működik, viszont a `rootNum` és `domain` paraméterek szerepét a bemeneti `Eigen::ArrayXd & x` és `int deg` paraméterek veszik át.

3.2.5. A Compresser modul implementációja

Az implementációt tartalmazó mappaszerkezet gyökerét /-vel jelölve, a Compresser modult az `/src/headers/Compresser.h` állomány implementálja.

A modul feladata

A Compresser modul feladata, hogy meghatározza azokat a metódusokat amelyek egy mátrix formában megadott jel tömörítéséhez szükségesek. Azok az osztályok amelyek egy konkrét jel tömörítését valósítják meg, (jelen implementáció szerint ez csak az OrtCompresser osztály) mind a Compresser modulban definiált Compresser osztályból vannak származtatva. A Compresser modul feladata továbbá, hogy `struct`-ok formájában definiálja azokat az adatszerkezeteket, amelyek segítségével a tömörített jel tárolásra kerül. A modul célja tehát, hogy megnevezze azokat a funkciókat amelyekkel minden jeltömrítő osztálynak rendelkeznie kell, illetve adatszerkezeteket biztosítson a tömörített jel tárolásához. Ez lehetőséget ad arra, hogy a dolgozatban bemutatott ortogonális függvényrendszereken alapuló tömörítési eljárás mellett egyéb tömörítő algoritmusokat (pl. Huffman kód) is alkalmazhasson a jövőben a program.

A modul interfésze

A modul nem használ szolgáltatásokat egyéb moduloktól. Mivel a Compresser osztály tisztán virtuális függvényeket tartalmaz, ezért példányosítani nem lehet. Ennek következtében Compresser típusú objektumok nem szerepelnek más modulokban, azonban az ebből származtatott OrtCompresser osztályból példányosított objektumok felelnek az egyes ortogonális projekciók kiszámításáért a Matching Pursuit modulban. Szintén a Matching Pursuit modul használja fel szolgáltatásként a Compresser modulban definiált `struct`-okat, hiszen az ezekből alkotott láncolt listában tárolja az egymást követő, már tömörített szívütés-szegmenseket.

A modul implementációja

A modul két `struct`-ból, és egy osztályból áll. A Compresser osztály nem tartalmaz protected elérhetőségű függvényeket, illetve adattagokat. Az osztály két tisztán virtuális publikus függvényt deklarál, melyek implementálása minden jeltömörítő osztálytól (az általa felhasznált algoritmustól függetlenül) elvárható. Az első a `compressBeat` függvény, amely egy `Eigen::MatrixXd` referenciát kap bemenetül, és a tömörítés elvégzése a feladata. A második függvény pedig a `decompress`, amely egy tömörített struktúrára mutató pointert kap bemenetül, és a helyreállított jelet hivatott mátrix formában visszaadni.

Mivel a jelenlegi implementáció kizárólag a dolgozatban bemutatott tömörítési eljárást valósítja meg, ezért összesen két olyan adatszerkezet definícióját tartalmazza a modul, amleyek képesek az egyes tömörített jelrészletek tárolására. Az első ilyen adatszerkezetet a `Compressed` struktúra definiálja. Két adattaggal rendelkezik: egy a következő tömörített jelrészletre mutató pointerrel, és egy `Eigen::MatrixXd` típusú, `compressedsig` azonosítóval ellátott konténerrel, amelyben a tömörítés eredménye található. A második, az `OrtCompressed` struktúra által definiált adatszerkezet azokat a szívütés szegmenseket hivatott tárolni, amelyek Hermite függvények segítségével kerültek tömörítésre. Ez az adatszerkezet a `Compressed` struktúrából lett származtatva, ám az előbb említett adattagok kiegészülnek a `double` típusú `dilat` illetve `trans` adattagokkal. Ezek hivatottak tárolni az egyes szívütés szegmensekhez meghatározott optimális dilatációs és transzlációs **??** értékeket.

3.2.6. Az OrtCompresser modul implementációja

Az implementációt tartalmazó mappaszerkezet gyökerét /-vel jelölve, az OrtCompresser modult az `/src/headers/OrtCompresser.h`, és a `/src/OrtCompresser.cpp` állományok implementálják.

A modul feladata

Az OrtCompresser modul feladata, hogy megvalósítsa azokat a függvényeket és eljárásokat, amelyek képesek kiszámítani egy mátrix alakban megadott bementi jel, egy Hermite függvényrendszer által kifeszített altérre vett ortogonális projekcióját. Az OrtCompresser modul felel továbbá a jelet helyreállító `decompress`, valamint a tömörítés minőségét jellemző különböző hiba függvények implementációjáért is.

A modul interfésze

Az OrtCompresser modul, amely az OrtCompresser osztályból áll, igénybe veszi az OrtFunSys, és az ebből származtatott Hermite modulok által nyújtott szolgáltatásokat. A program által, az adott szívütés tömörítéséhez kiszámított ortogonális függvényrendszer elemeit, valamint az ehhez kapcsolódó kvadratúra formulához szükséges számokat egy `OrtFunSys` típusú objektumra mutató pointeren keresztül éri el. Az osztály adattagjai között szerepel továbbá egy `Eigen::MatrixXd` típusú objektumra mutató pointer. Az osztályból származtatott objektumok, ennek a pointernek a segítségével képesek a jel helyreállításához szükséges Hermite függvényrendszerek elérésére. Az OrtCompresser osztály szolgáltatásait a MatchingPursuit modul veszi igénybe a közelítés optimalizálásának időtartama alatt, illetve az optimális dilatációs és transzlációs paraméterek megtalálása után, az utolsó tömörítéskor.

A modul implementációja

Az osztály konstruktora két paramétert vár bemenetnek. Az első egy OrtFunSys típusú objektum referenciája, amelyen keresztül elérhető az összes kiszámított Hermite függvény. A második paraméter az egész típusú `dim`, amely a tömörítéshez felhasználandó Hermite függvények számát adja meg. A konstruktor először helyet foglal a `Herm_sys` adattag által mutatott mátrixnak a dinamikus memóriában. Ezt követően, a konstruktor átmásolja a felhasználni kívánt első `dim` Hermite függvényt az OrtFunSys objektum által kiszámított mátrixból a protected hozzáférhetőségű `Herm_Sys` adattag által mutatott mátrixba. Az OrtCompresser osztály destruktora felszabadítja az `Herm_Sys` adattag által mutatott memória területet.

Az osztály négy publikus függvénnyel rendelkezik. Az első ezek közül az `OrtCompressed*` értéket visszaadó `rtCompresser::compressBeat`. A függvény bemeneti paramétere `Eigen::MatrixXd` referencia, melyet a `signal` azonosító jelöl. A függvény feladata, hogy kiszámítsa a `signal`-hoz rendelt Fourier-együtthatókat. A függvény először a dinamikus memóriában helyet foglal egy `OrtCompressed` típusú objektumnak, majd a **??** fejezetben található formula kerül alkalmazásra az együtthatók kiszámításához. A kiszámított Fourier együtthatók az `OrtCompressed` objektum `compressed_sig` adattagjában kerülnek eltárolásra. A transzlációs illetve dilatációs együtthatók kiszámítása nem a `OrtCompresser::compressBeat` feladata, ezek a bementül kapott jelre (`sig`), már alkalmazásra kerültek. A függvény által visszaadott `OrtCompressed` típusú objektumra mutató pointer felszabadítása a hívó metódus felelőssége.

A `OrtCompresser::decompress` függvény feladata, hogy egy `OrtCompressed` típusú objektumot felhasználva, helyreállítsa az eredeti jelet. Bemeneti paraméterként a függvény egy `OrtCompressed` típusú objektumra mutató pointert vár. A függvény egy `ArrayXd` típusú, `x` azonosítójú konténerbe tárolja a jel helyreállításához szükséges alappontokat. Az alappontok meghatározásához először egy egész értéket rendel a bementen szereplő transzlációs paraméterhez. *N/2*-vel jelölve az eredeti jelben szereplő pontok számának felét, az `OrtCompresser::decompress` függvény a *[-N/2, N/2]* intervallum *N* elemű egyenletes felosztását adja először értékül `x`-nek. Ezt követően a függvény `x` minden értékét a bemeneti paraméterben szereplő dilatációval megszorozza (az intervallum összehúzása), valamint minden értékéből az egész értékű transzlációt kivonja (az intervallum eltolása). Ezt követően az OrtCompresser osztály `big_ort_sys` adattagján keresztül elérhető `OrtSysGen` függvény segítségével, az `OrtCompresser::decompress` egy dilatált, és eltolt Hermite függvényrendszert generál az `x`-ben található pontok fölé. A függvény visszatérési értéke a kiszámított Hermite függvényrendszer és a bemeneti paraméterben található Fourier együtthatók szorzatának a transzponáltja, ami éppen az tömörített jel helyreállított alakja.

A fejezet elején látottak alapján OrtCompresser osztály felelőssége a tömörítés hibájának a meghatározása is. Ahhoz hogy ennek eleget tegyen az OrtCompresser osztály a túlterhelt `getPRD` függvényt használja. Mindkét implementáció a közelítés hibájának egy százalékos alakját adja meg, ám az egyik implementáció egy addicionális `string` típusú paramétert is kap, amely egy kimeneti fájl nevét tartalmazza, és amelybe beleírja az általa kiszámított hibát. Erre a felhasználói felületen elérhe-

tő animáció megjelenítésekor van szükség azért, hogy a hibát a tömörítés minden szakaszában kiolvashassa a felhasználó. A getPRD függvény mindkét implementációja bemenetként kap egy `Eigen::MatrixXd` típusú, `signal` azonosítóval ellátott referenciát, valamint egy `OrtCompressed` típusú objektumra mutató pointert. Első lépésként az utóbbi segítségével a függvény helyreállítja a jelet. Ezt követően az `OrtCompressed::getPRD` mindkét implementációja kiszámítja a helyreállított közelítés és az eredeti jel különbségét, továbbá az eredeti jel minden érékének és átlagos értékének a különbségét. Az így kapott vektorok kettes normáinak hányadosával, az úgy nevezett PRD-vel tér vissza. A PRD hiba részletesebb ismertetése a **??** fejezetben található.

3.2.7. Az Optimizer modul implementációja

Az implementációt tartalmazó mappaszerkezet gyökerét /-vel jelölve, a NelderMead modult az `/src/headers/Optimizer.h` állomány implementálja.

A modul feladata

Az Optimizer modul feladata, hogy definiálja az egyes optimalizáló algoritmusok által kötelezően implementálandó metódusokat. Ezen kívül a modul felelőssége, hogy pontos meghatározását adja azoknak a matematikai értelemben vett függvényeknek az osztályát amelyeken szélső érték keresést (optimalizációt) képes elvégezni a program. Mindent egybevetve az Optimizer modul feladata, hogy egy általános felépítést határozzon meg különböző optimalizációs eljárásokat implementáló osztályokhoz.

A modul interfésze

Az Optimizer modul más modulok szolgáltatásait nem veszi igénybe. Emiatt a modult felhasználó optimalizációs algoritmusok, (a jelenlegi programcsomagban a NelderMead modul által implementált optimalizációs eljárás), egy teljesen külön programban is felhasználható. Az Optimizer modul fő szolgáltatása, hogy általános interfészt biztosítson az egyes szélső érték keresésekhez. Nevezetesen a következő feltételeket határozza meg a szélső érték keresést implementáló modullal, illetve a függvénnyel szemben amelynek a szélső értékét meg szeretné határozni. A konkrét optimalizációt implementáló modul egy `std::function<double (Coord &)>`, típusú, valós értéket visszaadó és n dimenziós valós koordinátákon értelmezett függvényt vár bemenetként. Az egyes modulok ennek a bemeneti függvénynek a szélső érékeit hivatottak meghatározni. A bemeneti függvényekkel szemben az egyetlen jelenlegi elvárás, hogy $\mathtt{f} : \mathbb{R}^n \to \mathbb{R}$ alakúak legyenek. Egy lehetséges jövőbeli fejlesztése a programcsomagnak, hogy a megengedett függvények osztály, a komplex számokon értélemzett, illetve komplex értékű függvényekre is kiterjedjen.

A modul implementációja

Az előzőekben szemléltetett szolgáltatásokat az Optimizer modul két osztály implementációjával valósítja meg. Az első ezek közül a `Coord` osztály, amely a bemeneti függvények értelmezési tartományát hivatott implementálni. Annak érdekében, hogy az optimalizációs algoritmusokat ne kizárólag a dolgozatban bemutatott feladatra lehessen alkalmazni, szükséges volt egy általános koordináta típus bevezetése. Annak érdekében, hogy minél jobban újrahasznosítható legyen, az Optimizer modul nem használ a standard library-n kívül található metódusokat, illetve adatszerkezeteket. Ennek megfelelően a `Coord` osztály az `std::vector<double>` konténerből lett származtatva. Az osztályhoz két konstruktorral rendelkezik. Amennyiben egy `Coord` típusú objektum paraméterek nélkül kerül példányosításra, az osztály a vektort két eleműnek, vagyis az optimalizálandó függvényt $\mathbb{R}^2$-n értelmezettnek tekinti. Amennyiben az objektum konstruktora kap egy egész (`int`) típusú bemeneti paramétert, úgy a vektort ennyi eleműnek fogja újraméretezni. A konstruktorokon kívül az osztály túlterheli a `+,-,*,/` operátorokat, melyekq rendre egy n dimenziós valós elemű vektoron értelmezett összeadást, kivonást, skalárral történő szorzást, illetve egy skalárral történő osztást valósítsanak meg.

3.2.8. A NelderMead modul implementációja

Az implementációt tartalmazó mappaszerkezet gyökerét /-vel jelölve, a NelderMead modult az `/src/headers/OrtCompresser.h`, és a `/src/OrtCompresser.cpp` állományok implementálják.

A modul feladata

A NelderMead modul feladata, hogy a **??** fejezetben bemutatott Nelder-Mead optimalizációt implementálja. Ez a jelenlegi megvalósításban kizárólag az Hermite függvényrendszerek affin transzfolmáltjaihoz tartozó optimális dilatációs és transzlációs paraméterek meghatározására van felhasználva. Az osztály implementációja azonban lehetővé teszi, hogy egyéb optimalizációs feladatok megoldására is könnyen alkalmazható legyen.

A modul interfésze

A NelderMead modul az Optimizer osztályból származtatott osztályból áll, és igénybe veszi szolgáltatásként az Optimizer modulban definiált `Coord` struktúrát. A NelderMead osztályból példányosított objektumok szolgáltatásait a MatchingPursuit modul veszi igénybe, az aktuális tömörítendő szívütés szegmenshez tartozó optimális dilatációs és transzlációs paraméterek megtalálásának érdekében.

A modul implementációja

A NelderMead modult a NelderMead osztály alkotja, mely rendelkezik az Optimizer osztálytól megörökölt `const unsigned int generations`, illetve a `max_err` adattagokkal. A NelderMead osztály definiál egy további `std::multimap<double, Coord>` típusú, `population` azonosítójú protected hozzáférhetőségű adattagot. Ennek az adattagak a feladata, hogy az optimalizáció során a mindenkori szimplex egyes pointjaihoz tartozó aktuális hibát tárolja. Külön indokolta a `std::multimap` konténer alkalmazását, hogy az elemek ebben az asszociatív konténerben automatikusan kulcs szerint kerülnek rendezésre. Ez azt jelenti, hogy az algoritmus minden lépésében (amikor az aktuális szimplex egy régi pontját lecseréli egy új, alacsonyabb hibával rendelkező pontra), az új szimplex pontjainak rendezését nem volt szükséges külön implementálni.

A NelderMead osztály rendelkezik egy protected hozzáférhetőségű, `std::vector <std::multimap<double, Coord>::reverse_iterator>` típusú értéket visszaadó `set_pointers` azonosítójú függvénnyel. Az algoritmus által használt szimplex pointjait a `population` adattag tárolja, azonban az elérésük a `set_pointers` függvény által visszaadott `std::vector` konténeren keresztül történik. Ez azért indokolt, mert ezzel a módszerrel a vektor első tagján keresztül elérhető a legkisebb hibájú, utolsó tagján keresztül pedig a legnagyobb hibájú pont, anélkül, hogy a hiba mértékének pontos ismerete szükséges lenne. Ennek eredménye képpen az aktuális szimplex egyes pontjai könnyen elérhetővé, és manipulálhatóvá válnak.

A NelderMead osztály konstruktora először ellenőrzi, hogy a bementként kapott `Coord` típusú objektumokat tartalmazó vektor mérete megegyezik-e hárommal. Ez azért fontos, mert a Nelder-Mead algoritmus egy hárompontú szimplex segítségével végzi az optimalizációt. Ezt követően feltölti a `population` adattagot a bement koordinátáival és "végtelen", pontosabban `std::numeric_limits<double>::max()` hibát rendel az egyes értékekhez. A NelderMead osztály default destruktort használ.

Az osztálynak egyetlen publikus hozzáférhetőségű függvénye van. Ez az Optimizer osztálytól megörökölt `Optimize` azonosítójú függvény, amely egy `Coord` típusú objektumot ad vissza. A program jelenleg két dimenziós koordináták (dilatáció, és transzláció párok) feldolgozására használja a NelderMead modult, azonban mind a NelderMead osztály, mind a Coord struktúra felépítése lehetővé teszi az ennél több dimenziós pontokon történő keresést. A NelderMead osztály `Optimize` függvény implementációja az alábbi, a **??** is megtalálható algoritmus pszeudokódja.

Algorithm 1 Nelder-Mead

```
1: function NELDER-MEAD
2:     if y_3 ≤ y_4 ∧ y_4 < y_2 then
3:         x_1 = x_4
4:     else if y_4 < y_3 then
5:         if y_5 < y_4 then
6:             x_1 = x_5
7:         else
8:             x_1 = x_4
9:         end if
10:    else if y_4 ≥ y_2 then
11:        if y_4 < y_1 then
12:            if y_6 ≤ y_4 then
13:                x_1 = x_6
14:            else
15:                IterLepes(5)
16:            end if
17:        else if y_4 ≥ y_1 then
18:            if y_7 < y_3 then
19:                x_1 = x_7
20:            else
21:                IterLepes(5)
22:            end if
23:        end if
24:    end if
25: end function
```

3.2.9. A MatchingPursuit modul implementációja

Az implementációt tartalmazó mappaszerkezet gyökerét /-vel jelölve, a NelderMead modult az `/src/headers/MatchingPursuit.h`, és a `/src/MatchingPursuit.cpp` állományok implementálják.

A modul feladata

A modul feladata, hogy megvalósítsa a **??** fejezetben leírt algoritmust, és ezáltal lehetővé tegye egy szívütés optimalizált tömörítését. A MatchingPursuit modul feladata továbbá, hogy a tömörített szívütést elérhetővé tegye a programcsomag egyéb moduljainak számára, valamint a tömörítés során keletkezett részeredményeket fájlokba írja. A webes felhasználói felület ezen részeredmények kiolvasásának segítségével képes a tömörítés menetének megjelenítésére.

A modul interfésze

A modul számos egyéb modul szolgáltatásait veszi igénybe a tömörítés megvalósításának érdekében. Az MP algoritmus minden lépésében szüksége van a közelítés dilatációs illetve transzlációs paramétereinek optimalizálására. Ennek érdekében a MatchingPursuit modul a NelderMead modul által biztosított szolgáltatásokat veszi igénybe. Az egyes tömörítések elvégzéséhez az OrtCompresser, és a Hermite modulok, a bemeneti jel kezeléséhez pedig az EcgSigPrep modul metódusait alkalmazza. A MatchingPursuit modul közvetlenül a `/src/main.cpp` állományon belül található `main` függvénynek nyújt szolgáltatást. A tömörített szívütés szegmensek itt kerülnek ugyanis feldolgozásra.

A modul implementációja

A MatchingPursuit modult a `/src/headers/MatchingPursuit.h` állományban deklarált `MatchingPursuit` osztály alkotja. Az osztálynak három privát hozzáférhetőségű adattagja van. Az első egy `EcgSigPrep` típusú objektumra mutató pointer, amely a feldolgozandó bemeneti jel kezeléséért felel. A két további adattag egy `std::map<std::string, std::string>` típusú, `files_dir` azonosítóval ellátott asszociatív tömb, és egy `std::function<double (Coord &)>` típusú, `costfun` azonosítójú c++ 11 szabványban bevezetett "függvénytípus". A típus lényege, hogy a hagyományos függvényekkel ellentétben egy eljárást mint objektumot képes kezelni a program. Ez azt jelenti, hogy a `costfun` adattag egyszerre felfogható úgy mint egy hívható függvény, és úgy mint egy azonosítóval ellátott adatszerkezet. Ennek következtében lehetőség nyílik ezen függvények ad-hoc módon történő definiálására, valamint paraméterként szerepelhetnek egy másik függvényben. Ezt az adattagot kapja a meg bemenetként az optimalizációs eljárás. Az osztály egyetlen privát hozzáférhetőségű `void set_costfun(std::function<double (Coord &)> cfun)` szignatúrával ellá-

tott metódust definiál. Ennek a metódusnak a feladata, hogy definícióval lássa el a `costfun` adattagot.

A `MatchingPursuit` osztályhoz két konstruktor és egy destruktor tartozik. Ezeken kívül csupán egy publikus hozzáférhetőségű függvényt definiál, amely `OrtCompressed*` értéket ad vissza és a `CompressBeat` azonosítóval van ellátva. A `CompressBeat` függvény valósítja meg a Matching Pursuit algoritmust. Bemeneti paraméterként egy `std::vector<double>` típusú vektort kap, melynek elemei az egyes EKG hullámszegmensek tömörítésénél felhasznált Hermite rendszerek dimenzióját határozzák meg. A bemutatott eljárás tesztelésének során ezek a fokszámok a 7, 6, 2 értékekben lettek megállapítva a **??**-ben található indoklás alapján. A függvény első lépésként betölti a tömörítendő szívütést. Ehhez a `sig_handler` adattag `getNextSegment()` metódusát használja fel. Az algoritmus inicializálásakor a `CompressBeat` függvény a jelet kért külön vektorban tárolja. Az első vektor `sig` azonosítóval van ellátva, és a Matching Pursuit algoritmus egyes lépései után hátramaradó reziduum jelet hivatott tárolni. A második vektor ezzel ellentétben végig az eredeti tömörítendő jelet tartalmazza, és `osig` vagyis original signal azonosítót kap. Ezt a tömörítés hibájának kiszámítására alkalmazza a függvény, az egyes approximációkhoz tartozó PRD értéket **??** ehhez a jelhez viszonyítva adja meg. A szívütést tartalmazó mátrixokon kívül a függvény szintén az algoritmus inicializálásának során deklarálja a `Herm` azonosítóval ellátott, `Hermite` típusú objektumot. Az objektum bemeneti paraméterkén a szívütésben szereplő mért minták számát kapja, ezáltal biztosítva, hogy a felhasznált Hermite függvényrendszer ugyanennyi ponton legyen értelmezve.

Az inicializáció után a függvény belép a Matching Pursuit algoritmus fő ciklusába. A ciklus lépéseinek számát a bemeneti paraméterként kapott `rounds_deg` vektor hossza határozza meg. A bemutatott módszer esetén az algoritmus minden alkalommal három körben végzi el a tömörítést, azonban a lépések számának ilyen meghatározása meghagyja a lehetőséget az esetleges jövőbeli kísérletezésnek.

Amennyiben a felhasználói felület részéről igény érkezik az animációs bemutatásra, a fő ciklus magjában a program először kimásolja az aktuális reziduum jel értékeit a felhasználói felület szempontjából megfelelő állományba. Ezt követően a függvény deklarálja az `OC` azonosítóval ellátott `OrtCompresser` típusú objektumot. Ez az objektum fogja a Matching Pursuit algoritmus aktuális lépésében az egyes tömörítéseket és hiba számításokat végrehajtani. Az `OC` objektum mellett a függvény a dinamikus memóriában lefoglalja az aktuális tömörített hullámszegmens tárolásához szükséges helyet. Ezt a területet az `OrtCompressed*` típusú, `p` azonosítóval ellátott mutató jelöli.

Az előző objektumok létrehozása után a függvény a `MatchingPursuit` osztály `set_costfun` metódusát hívja meg, paramétereként pedig definiál egy c++11-es szabványnak megfelelő anonim eljárást. Az itt definiált eljárásra a `MatchingPursuit` osztály `costfun` privát hozzáférhetőségű adattagján keresztül lehet majd a későbbiekben hivatkozni. Az eljárás definíciójában először is fel kell sorolni azokat az objektumokat, amelyek a külső (`CompressBeat`) függvényben kerültek deklarálásra, azonban az anonim függvény definíciójában is felhasználandóak. Mivel ennek a konkrét anonim függvénynek a feladata egyetlen approximáció kiszámítása, így az előbb említett

objektumok a már ismertetett reziduum jelet tartalmazó mártix, az ortogonális projekciót kiszámító objektum, illetve az Hermite rendszert kezelő objektum. Az anonim függvény számára ezen objektumok referencia szerint kerülnek átadásra. A definícióban felhasználásra kerülnek továbbá a `MatchingPursuit` osztály egyéb adattagjai is, így ezek elérésének érdekében szintén a bemeneti paraméter listára kerül a `this` mutató. A `costfun` lambda függvény hagyományos paraméter listáján csupán a `Coord` típusú `?? pos` azonosítóval ellátott elem szerepel. Ez a paraméter tartalmazza az optimalizációs algoritmus által éppen kiszámított dilatációs és transzlációs értékeket.

A `costfun` lambda függvény először lemásolja a reziduum jelet egy lokálisan deklarált mátrixba. Erre a mátrixra alkalmazza a paraméterül kapott dilatációt és transzlációt. Az eltolt és összehúzott jelhez kiszámítja az ortogonális projekcióval adódó Fourier-együtthatókat. Végül kiszámítja az így kapott tömörített jel hibáját, és visszaadja azt az optimalizációs algoritmus számára. Amennyiben a felhasználói felület animációs üzemmódban működik, szintén a `costfun` definíciójában található az aktuális részeredmény (approximáció) értékeinek fájlba írása.

Miután a `CompressBeat` függvény sikeresen definiálta az aktuális hullámszegmens optimális tömörítéséhez szükséges lambda függvényt, deklarálnia kell a tömörítés optimalizációjához felhasznált `NelderMead` típusú objektum bemeneti paramétereit. Mivel a Nelder Mead algoritmus egy szimplex alapú algoritmus, ezért a `CompressBeat` függvénynek először is meg kell határoznia három kezdeti dilatáció és transzláció párt. Az előbbiek kísérleti úton, önkényesen kerültek meghatározásra, még a kezdeti transzlációs paraméterek az aktuális reziduum jel mintáinak maximumát közrefogó intervallum szélei és maga a maximum hely indexe. Ezt követően deklarálásra kerül a `NelderMead` típusú, `opter` azonosítóval ellátott objektum. Az említett szimplexen kívül az objektum konstruktora további két paraméter megadását igényli. Az első paraméter határozza meg a Nelder Mead algoritmus által legfeljebb elvégezhető lépésszámot. Ezt, tekintettel a futási időre a `CompressBeat` függvény a 20 értékben határozza meg. Az `opter` objektum második paramétere a közelítés legmagasabb tolerálható hibáját jelöli, mely PRD `??` formában van megadva, és 20%-os értéket vesz fel. Ezután a függvény meghívja az `opter` objektum `Optimize` metódusát, a visszaadott optimális dilatációs és transzlációs értékeket pedig az `optimized_coords` lokális változó kapja értékül.

Az optimális dilatációs és transzlációs együtthatók ismeretében lehetőség nyílik a hullámszegmens végleges tömörítésére. A `sig_handler` adattag `setDilatTrans` metódusának segítségével a tömörítendő jelet a `CompressBeat` függvény eltolja illetve összehúzza `??`. Ezt követően a függvény az eltolt és összehúzott jelhez az `OC` lokális `OrtCompresser` típusú változó segítségével meghatátorozza a Fourier együtthatókat. Az `OC.compressBeat` metódus a kiszámított Fourier együtthatókat egy `OrtCompressed` típusú adatszerkezet `compressed_sig` adattagjába menti a dinamikus memóriában. A `MatchingPursuit::CompressBeat` függvény `p` azonosítójú lokális mutatóján keresztül éri el a tömörítés eredményét. Szintén `p`-n keresztül kerülnek beállításra a tömörített jelhez tartozó dilatációs és transzlációs együtthatók. Amennyiben a felhasználói felület számára az animációs adatok biztosítása szükséges, a `MatchingPursuit::CompressBeat` függvény az aktuális hullámszegmens tömörítés-

se után helyre állítja a tömörített jelet, és a minták értékeit a megfelelő állományba másolja. Végezetül a függvény kivonja a tömörítendő jelből az approximáció értékét, ezáltal a következő ciklusban az így kapott reziduum jel kerül tömörítésre. A függvény az utolsó ciklusban tömörített hullámszegmensre mutató pointerrel tér vissza. A tömörített hullámszegmensek egy láncolt listát alkotnak a dinamikus memóriában.

3.2.10. Felhasznált könyvtárak, és jellemzésük

Nagy terjedelmű szoftverek fejlesztése külső erőforrások igénybevétele nélkül a jelen korban nem csupán borzasztóan nehéz, hanem egyben felelőtlenségnek is tekinthető. Ennek oka az interneten legtöbbször ingyenesen elérhető és tetszés szerint felhasználható szolgáltatások kínálata. Sok esetben ezeket a könyvtárakat és szolgáltatásokat nagy cégek, illetve komoly kutatócsoportok fejlesztették ezáltal garantálva a megvalósított algoritmusok helyességét és hatékonyságát. A fejlesztői dokumentáció ezen fejezetében azok a könyvtárak és szolgáltatások kerülnek bemutatásra amelyeket közvetlenül felhasznál a bemutatott tömörítő eljárás implementációja. Minden egyes külső könyvtárhoz tartozó alfejezet ugyanazt a felépítést követi. Először a könyvtár dokumentációjának nagyon rövid összefoglalását tárgyalja, ez követően pedig bemutatja a tömörítő eljárás által felhasznált szolgáltatásokat. Az egyes könyvtárak részletes dokumentációját a könyvtárakat fejlesztő csoportok és cégek weboldalain lehet megtalálni. Ezen webhelyek elérhetőségei szintén megtalálhatóak az alfejezetekben.

Az Eigen könyvtár

Az Eigen könyvtár egy C++ sablon könyvtár amelyet lineáris algebrai feladatok megoldásához fejlesztettek ki. Gazdag választékát biztosítja mátrixok, vektorok és egyéb hasznos tömbön alapuló adatszerkezetek könnyen kezelhető reprezentációinak, illetve különböző numerikus feladatokat megoldó osztályoknak. A könyvtár fejlesztését önkéntesek biztosítják. A legtöbbet hozzájáruló önkénetesek nevei, illetve a könyvtár dokumentációja megtalálhatóak a projekt weboldalán, amely a http://eigen.tuxfamily.org címen érhető el.

A könyvtár legnagyobb erőssége, hogy a MATLAB nyelvben megszokott felületet biztosít mátrixok illetve vektorok kezeléséhez. Mivel a tömörítő eljárás megvalósításánál számos ilyen feladattal kell szembenézni, ezért ideális választásnak mondható a könyvtár alkalmazása. Az Eigen könyvtár által nyújtott szolgálatatásokat gyakorlatilag az implementáció minden modulja igénybe veszi. A program tömörítendő jelet az Eigen::MatrixXd dinamikusan változtatható méretű mátrixban tárolja, csakúgy mint a tömörítéshez felhasznált Hermite függvényrendszer értékeit. A pontokat melyek fölött az Hermite függvények értékeit kiszámítja a program az Eigen programcsomag egy saját érték probléma megoldó objektumának segítségével kerülnek meghatározásra. A tömörítő eljárás implementációja soran a könyvtár megbízható, gyors és könnyen alkalmazható tulajdonságairól tett tanúbizonyságot.

A wfdb könyvtár

A megvalósított programcsomag legfontosabb céljaként **??** az MIT-BIH PhysioBank adatbázisban található EKG mérések tömörítése. A wfdb (WaveForm DataBase) könyvtár az adatbázisban található mérések és a hozzájuk tartozó orvosi jelölések (annotációk) kezeléséhez biztosít függvényeket. Egészen pontosan a könyvtár a PhysioBank szerverein található mérések eléréséhez és feldolgozásához biztosít egy API felületet. A könyvtárban található függvények C illetve FortTran nyelveken érhetők el, de bizonyos funkcionalitások elérhetőek a népszerű MATLAB illetve python nyelveken is. A könyvtár fejlesztői interfészeket biztosítanak számos más programozási nyelv felé is, így gyakorlatilag bármely népszerű nyelvet felhasználva lehetőség nyílik az adatbázisban található jelek feldolgozására. Mivel a könyvtár lehetővé teszi a PhysioBank adatbázisban található mérések közvetlen (http, illetve FTP protokollokat felhasználó) elérését, ezért használatához szükséges a libcurl könyvtár feltelepítése is. A libcurl könyvtár képes az említett webes tartalmak böngésző nélküli eléréséhez. A könyvtár teljes dokumentációja a https://www.physionet.org/physiotools/wpg/wpg.pdf webcímen érhető el.

A bemutatott tömörítő eljárás jelenlegi implementációja az EcgSigPrep modulban használja fel a wfdb könyvtár szolgáltatásait. A modulban található `EcgSigPrep` osztály konstruktora nagy mértékben támaszkodik a könyvtár szolgáltatásaira annak érdekében, hogy a bemeneti paraméterül kapott mérés azonosítóhoz tartozó minták feldolgozásakor. Szintén a könyvtár függvényein keresztül képes az osztály az egyes szívütések különválasztására, mivel az ehhez szükséges R csúcs pozíciókat az eredeti méréshez tartozó annotációkból szűri ki. Az említett pozíciókat egy szintén a wfdb könyvtárban definiált típus a `WFDB_Annotation` segítségével tárolja az osztály. A wfdb könyvtár szolgáltatásait az `EcgSigPrep` osztály konstruktorán kívül a publikus hozzáférhetőségű `getNextSegment` függvénye is igénybe veszi a soron következő szívütés széleinek meghatározásához.

A Spline.h állomány

Az implementációt tartalmazó mappaszerkezet gyökerét `/`-vel jelölve, a Spline.h állomány a `/src/headers/Spline.h` útvonalon található. Az állomány egy absztrakt harmadfokú spline modellt implementál. Az állományt Tino Kluge hozta létre, illetve tartja karban. A spline modellhez és az implementációhoz tartozó dokumentáció az alábbi weboldalon érhető el: http://kluge.in-chemnitz.de/opensource/spline/. A Spline.h előnyei közé tartozik, hogy nincsenek külső dependenciái, az `#include "Spline.h"` direktíva alkalmazása után az összes funkció elérhetővé válik. Ezen kívül az állomány használata egyszerű és gyors.

A tömörítő eljárás implementációja során egyetlen helyen szükséges spline interpoláció alkalmazása. Az eljárás robosztusságának a spline interpoláció csupán egy igen kis részét teszi ki. Ennek hatékony és pontos implementációja azonban önmagában is nagy feladatot jelenthet. Mindez indokolta egy külső spline modellt tartalmazó állomány igénybe vételét.

Az EKG tömörítés során spline interpolációra a bemeneti jel eltolásánál, azaz az aktuális transzlációs paraméter alkalmazásánál kerül sor. Ezt a műveletet az EcgSigPrep moduloban található `EcgSigPrep` osztály `setDilatTrans` függvénye végzi. Miután a jel értékeit az `std::rotate` függvény segítségével megfelelő számú pozícióval eltolta az eljárás, szükségessé válik az eltolt jel értékeinek kiszámítása a dilatált intervallum fölött. Ezen értékek kiszámítása spline interpolációval történik.

3.3. A felhasználói felület tesztelése

3.4. A tömörítő eljárás modul szintű tesztelése

A modul tesztek elvégzése elengedhetlen egy modern programcsomag működésének ellenőrzésekor. Az egyes modul tesztek által definiált tesztesetek hivatottak igazolni, hogy a modulok implementációja során meghatározott funkciókkal a modul valóban rendelkezik, illetve ezeket helyesen alkalmazza.

A fejlesztői dokumentáció jelen része tartalmazza a **??** fejezetben bemutatott modulokhoz tartozó teszteket és eredményeket. Az implementációt tartalmazó mappaszerkezet gyökerét /-vel jelölve, az egyes modulokhoz tartozó modul tesztek a `/src/mt/[az aktuális modul neve]` útvonal mentén érhetők el. Minden modul tesztet jelölő mappa tartalmazza azt a `.cpp` állományt amelyben a tesztesetek definiálva vannak, a tesztesetek bemenetét, illetve az egyes tesztesetek kimenetét. Ezen kívül megtalálhatóak a modult implementáló állományok is, valamint egy `make` állomány a modul teszt fordításához.

Minden modul teszt jellemzéséhez tartozik egy külön alfejezet. Az egyes modul tesztekhez tartozó alfelejezetek rövid jellemzését adják a tesztek legfontosabb céljainak. Ezen kívül táblázatot biztosítanak az egyes tesztesetek jellemzéséhez. A modulok szolgáltatásaitól függően a bemenetek és a kimenetek gyakran önmagukban is nagy terjedelműek, (például az EcgSigPrep modul egy teljes EKG felvételt vár bemenetül), ezért ezeknek csak a rövid jellemzése adott.

3.4.1. A SigPrep és EcgSigPrep modulok tesztjei

A SigPrep illetve az EcgSigPrep modulokhoz tartozó modul teszt a `/src/mt/sigPrepEcgSigPrep` mappában található. A modul teszthez tartozó forráskódot az `mt.cpp` állomomány tartalmazza. Az `mt.cpp` állomány először az elvégzendő tesztesetek paramétereit adja meg makrók segítségével, ezeket pedig a `main` függvény követi az egyes tesztesetek definícióival. Az egyes tesztesetek kimenetét a `/src/mt/sigPrepEcgSigPrep/output/mt_results.txt` szöveges állomány tartalmazza.

A modul teszt célja, hogy a SigPrep illetve az EcgSigPrep modulok publikus (és közvetetten a privát) hozzáférhetőségű metódusainak helyességét igazolja. Ez az `EcgSigPrep` osztályból példányosított objektum publikus függvényeinek meghívásával, és a visszaadott eredmények ellenőrzésével végezhető el. Az alábbi táblázat tartalmazza az elvégzett tesztesetek jelölését, rövid jellemzését, az eredményeket, illetve

az ellenőrzés módját. A pontos kimenetek megtalálhatóak az `mt_results.txt` állományban.

Jelölés	Teszteset	Eredmény	Ellenőrzés módja
TC1	Az EcgSigPrep típusú objektum konstruktora lefut	Ok	A standard output ellenőrzése. Helyes bemenet esetén az objektum létrejön, helytelen bemenet esetén hibaüzenet.
TC2	A beolvasott EKG mérés első illetve második szívütésének kiiratása	Ok	A kapott számsorozat grafikus ábrázolása. Ez összehasonlítható az ?? oldalon található ábrázolással
TC3	Dilatáció és transzláció alkalmazása a setDilatTrans metódus segítségével	Ok	A kapott számsor grafikus ábrázolása, összehasonlítása az eredeti szívütéssel.
TC4	Az objektum getter függvényeinek ellenőrzése	Ok	A kimenet összevetése az elvárt eredményekkel. Ahol szükséges, ott az eredmények grafikus ábrázolása.

3.4.2. Az OrtFunSys és Hermite modulok tesztjei

Az OrtFunSys illetve az Hermite modulokhoz tartozó modul teszt a `/src/mt/ortFunSys Hermite` mappában található. A modul teszthet tartozó forráskódot az `mt.cpp` állomomány tartalmazza. Az `mt.cpp` állomány először az elvégzendő tesztesetek paramétereit adja meg makrók segítségével, ezeket pedig a `main` függvény követi az egyes tesztesetek definícióival. Az egyes tesztesetek kimenetét a `/src/mt/ortFunSysHermite /output/mt_results.txt` szöveges állomány tartalmazza.

A modul teszt célja, hogy az OrtFunSys illetve az Hermite modulok metódusainak helyességét igazolja, bizonyítva ezzel, hogy valóban a **??** fejezetekben bemutatott szolgáltatásokat nyújtják. A metódusok helyességének belátása az `Hermite` osztályból példányosított objektum publikus függvényeinek meghívásával, és a visszaadott eredmények ellenőrzésével végezhető el. Az alábbi táblázat tartalmazza az elvégzett tesztesetek jelölését, rövid jellemzését, az eredményeket, illetve az ellenőrzés módját. A pontos kimenetek megtalálhatóak az `mt_results.txt` állományban.

Jelölés	Teszteset	Eredmény	Ellenőrzés módja
TC1	Az Hermite típusú objektum konstruktora lefut	Ok	A standard output ellenőrzése.
TC2	A 20 fokú Hermite függvény zérushelyeinek ellenőrzése	Ok	Az ortogonális polinomoknak megfelelően, 20 különböző értékű, egyszeres valós gyök az eredmény, amelyek szimmetrikusak az origóra.
TC3	A kiszámított Hermite függvények értékeinek kiiratása	Ok	A kapott számsor grafikus ábrázolása, kritikus értékek (pl.maximum) ellenőrzése.
TC4	A kiszámított Hermite függvényekhez tartozó Crisstoffel-Darboux számok kiiratása	Ok	A skaláris szorzás elvégzése két különböző, illetve két azonos Hermite függvénnyel. Az első esetben 0, második esetben 1 az eredmény.
TC5	Hermite függvényrendszer generálása ekvidisztans felosztású intervallumon	Ok	A kapott számsor grafikus ábrázolása, kritikus értékek (pl.maximum) ellenőrzése.

3.4.3. A Compresser és az OrtCompresser modulok tesztjei

Az Compresser illetve az OrtCompresser modulokhoz tartozó modul teszt a `/src/mt/compresserOrt compresser` mappában található. A modul teszthez tartozó forráskódot az `mt.cpp` állomomány tartalmazza. Az `mt.cpp` állomány először az elvégzendő tesztesetek paramétereit adja meg makrók segítségével, ezeket pedig a `main` függvény követi az egyes tesztesetek definícióival. Az egyes tesztesetek kimenetét a `/src/mt/compresserOrtCompresser /output/mt_results.txt` szöveges állomány tartalmazza.

A modul teszt feladata, hogy igazolja a **??** fejezetekben meghatározott funkciók létét, illetve az implementáció helyességét. Ennek érdekében a tesztesetek valós adatokkal teszik próbára az említett modulok által definiált `Compresser` és `OrtCompresser` osztályok szolgáltatásait. A metódusok helyességének belátása az `OrtCompresser` osztályból példányosított objektum publikus függvényeinek meghívásával, és a visszaadott eredmények ellenőrzésével végezhető el. Az alábbi táblázat tartalmazza az elvégzett tesztesetek jelölését, rövid jellemzését, az eredményeket, illetve az ellenőrzés módját. A pontos kimenetek megtalálhatóak az `mt_results.txt` állományban.

Jelölés	Teszteset	Eredmény	Ellenőrzés módja
TC1	Az OrtCompresser típusú objektum minden konstruktora lefut	Ok	A standard output ellenőrzése.
TC2	Egy szívütés tömörítése	Ok	Dilatáció, transzláció ellenőrzése tömörítés után. A Fourier együtthatók helyességének belátása a jel helyreállításával.
TC3	Egy tömörített szívütés helyreállítása	Ok	A kapott számsor grafikus ábrázolása, kritikus értékek (pl.maximum) ellenőrzése.
TC4	Mindhárom hibafüggvény (OrtCompresser::getPRD(...)) helyességének belátása	Ok	A hibafüggvények azonos, és hihető értékeket produkálnak a kipróbált inputokra.

3.4.4. Az Optimizer és a NelderMead modulok tesztjei

Az Optimizer illetve az NelderMead modulokhoz tartozó modul teszt a `/src/mt/optimizerNelderMead` mappában található. A modul teszthez tartozó forráskódot az `mt.cpp` állomány tartalmazza. Az `mt.cpp` állomány először az elvégzendő tesztesetek paramétereit adja meg makrók segítségével, ezeket pedig a `main` függvény követi az egyes tesztesetek definícióival. Az egyes tesztesetek kimenetét a `/src/mt/ortFunSysHermite/output/mt_results.txt` szöveges állomány tartalmazza.

A modul teszt feladata, hogy igazolja a **??** fejezetekben meghatározott funkciók létét, illetve az implementáció helyességét. Ennek érdekében a tesztesetek valós adatokkal teszik próbára az Optimizer modul által definiált `std::vector<double>` típusból származtatott `Coord` struktúrát és a NelderMead modulban implementált Nelder-Mead algoritmust. A metódusok helyességének belátása a `NelderMead` osztályból példányosított objektum publikus `NelderMead::Optimize()` függvényének meghívásával, és a visszaadott eredmények ellenőrzésével végezhető el. Az alábbi táblázat tartalmazza az elvégzett tesztesetek jelölését, rövid jellemzését, az eredményeket, illetve az ellenőrzés módját. A pontos kimenetek megtalálhatóak az `mt_results.txt` állományban.

Jelölés	Teszteset	Eredmény	Ellenőrzés módja
TC1	A NelderMead típusú objektum konstruktora lefut	Ok	A standard output ellenőrzése.
TC2	Az $f(\mathbf{x}, \mathbf{y}) := \mathbf{x}^2 + \mathbf{y}^2 \quad (\mathbf{x}, \mathbf{y} \in \mathbb{R})$ (1.1) függvény minimumhelyének kiszűrése a Nelder-Mead algoritmus segítségével. Különböző kezdeti paraméterek kipróbálásával.	Ok	A kiszámított minimum hely megtalálható a `mt_results.txt` állományban. Jól látható, hogy az algoritmus fenti célnak minél inkább megfelelő működéséhez, vagy a maximális elfogadható hibát csökkentve egyre közelebb kerül a kiszámított érték az origóhoz.

3.4.5. A MatchingPursuit modul tesztje

A MatchingPursuit modulhoz tartozó modul teszt a `/src/mt/matchingPursuit` mappában található. A modul teszthez tartozó forráskódot az `mt.cpp` állomomány tartalmazza. Az `mt.cpp` állomány először az elvégzendő tesztesetek paramétereit adja meg makrók segítségével, ezeket pedig a `main` függvény követi az egyes tesztesetek definícióival. Az egyes tesztesetek kimenetét a `/src/mt/matchingPursuit/output/mt_results.txt` szöveges állomány tartalmazza.

A modul teszt feladata, hogy igazolja a **??** fejezetekben meghatározott funkciók létét, illetve az implementáció helyességét. Ennek érdekében a tesztesetek valós adatokkal teszik próbára az említett modul által definiált `MatchingPursuit` osztály szolgáltatásait. A metódusok helyességének belátása az `MatchingPursuit` osztályból példányosított objektum publikus függvényeinek meghívásával, és a visszaadott eredmények ellenőrzésével végezhető el. Az alábbi táblázat tartalmazza az elvégzett tesztesetek jelölését, rövid jellemzését, az eredményeket, illetve az ellenőrzés módját. A pontos kimenetek megtalálhatóak az `mt_results.txt` állományban.

Jelölés	Teszteset	Eredmény	Ellenőrzés módja
TC1	A MatchingPursuit típusú objektum minden konstruktora lefut	Ok	A standard output ellenőrzése.
TC2	Három egymást követő szívütés optimalizált tömörítése MP algoritmussal. Az eredmény kiíratása az `mt_results.txt` állományba, illetve egy .cmp szöveges állományba	Ok	Az output kritikus értékeinek (dilatáció, transzláció) ellenőrzése.
TC3	A tömörített jelek helyreállítása a .cmp állományból.	Ok	A kapott számsor grafikus ábrázolása, kritikus értékek (pl.maximum) ellenőrzése.

3.5. A tömörítő eljárás hatékonyságának tesztelése

Ez a fejezet a bemutatott tömörítő eljárással kapcsolatos tesztesetek eredményeit tárgyalja. Ezek a tesztek implementációtól függetlenül hivatottak bemutatni a kidolgozott tömörítő eljárás helyességét, illetve eredményességét más eljárásokhoz képest.

Mivel a tömörítő eljárásról a **??** folyóiratban már korábban megjelent egy tanulmány, ebben a fejezetben az itt bemutatott eredmények szerepelnek. Az eredményeket a tömörítő algoritmus MATLAB nyelven íródott implementációjával sikerült elérni. A dolgozatban tárgyalt c++ nyelven implementált programmal az adatok nagy számossága miatt, az összes rekord újratömörítése nem volt szükségszerű. Természetesen a program alkalmas az itt bemutatott bármely mérés teljes tömörítésére, illetve a közelítés hibájának meghatározására. Ez a modul tesztelés során egyértelműen kiderült, hiszen a tesztesetek között szerepelt egy hosszú, összefüggő rekord tömörítése is.

A tömörítési eljárás a korábbi **??** dolgozatokban szereplő 117, 119-es rekordok mellett az MIT-BIH adatbázis egyéb felvételein is kipróbálásra került. Így összesen 3 órányi adaton, 12830 darab szívütést tömörített a módszer. Az összehasonlíthatóság érdekében az eredeti [?] algoritmus is elvégezte a tesztet. Mindkét esetben az egyes szegmensek $\mathbf{n} = 7, 6, 2$ együtthatóval lettek meghatározva. A futásidő csökkentésének érdekében az iterációk száma a 15 értékben került meghatározásra. Így egy 30 perces rekordon a Nelder-Mead (NM) szimplex módszer átlagosan 1400 másodpercig futott. Az eredményeket a pontosság és a tömörítési arány szempontjából is szükséges értékelni:

$$\mathrm{PRD} = \frac{\left\| S_{\mathbf{n}}^{\mathbf{a},\lambda} f - f \right\|_2}{\left\| f - \bar{f} \right\|_2} \times 100\,, \qquad \mathrm{CR} = \frac{\text{eredeti EKG mérete}}{\text{tömörített EKG mérete}} \times 100\,,$$

ahol $\bar{f}$ a jel átlaga. A pontosság mérésére az irodalomban az ún. percentage root mean square difference (PRD) terjedt el. Könnyen észrevehető, hogy ez az ℓ^2 normában mért relatív hibával egyenlő, ami független a jel átlagától. Az algoritmusok összehasonlításánál a CR-t és a PRD-t is figyelembe kell venni. Ez azonban megnehezíti a módszerek értékelését. Ezt kiküszöbölendő, a [?] cikkben bevezették az ún. Quality Score (QS) fogalmát, ami a CR és a PRD hányadosa. Így az az algoritmus tekinthető jobbnak, amelyik magasabb QS-el rendelkezik. Ennek megfelelően az eredményeket a 3.1 táblázatban foglaltuk össze.

	Hiba (PRD %)				**A tömörítés aránya (CR 1 : X)**				**Quality Score** (CR : PRD)			
Rec.	**Eredeti**	**NM**	**PSO**	**JPEG2**	**Eredeti**	**NM**	**PSO**	**JPEG2**	**Eredeti**	**NM**	**PSO**	**JPEG2**
101	11.20	11.10	11.13	11.07	29.71	27.22	27.22	18.94	**2.65**	2.45	2.47	1.71
117	13.20	11.81	17.66	11.66	36.07	33.06	33.06	23.66	2.73	**2.79**	1.87	2.02
118	19.83	17.79	16.65	17.72	24.34	22.30	22.30	32.91	1.22	1.25	1.33	**1.85**
119	14.27	8.76	10.20	8.80	27.89	25.55	25.55	23.85	1.95	**2.91**	2.51	2.71
201	13.51	12.17	12.17	12.14	28.21	25.35	25.35	13.15	**2.08**	**2.08**	**2.08**	1.08
213	19.92	18.28	17.60	18.29	17.08	15.64	15.64	35.23	0.85	0.85	0.88	**1.92**
Átlag	14.22	12.55	13.83	12.51	26.83	24.45	24.51	23.86	1.91	**2.06**	1.85	1.88

3.1. táblázat. A tömörítés összehasonlítása különböző módszerek esetén.

Általánosságban elmondható, hogy az összehasonlított módszerek közül, a dolgozatban bemutatott algoritmus volt a leghatékonyabb. Utóbbi PRD-je átlagosan 2%-al jobb, mint az eredeti módszeré. A tömörítési arány egy kicsit rosszabb, ami várható hiszen a transzlációs paramétereket is tárolni kell ellentétben [?]-el. Végeredményben viszont az átlagos QS alapján a dolgozatban bemutatott és a Nelder-Mead optimalizációval kombinált módszer bizonyult a leghatékonyabbnak. További érdekesség,

hogy a 119-es rekord esetén jelentős különbség van a két módszer között. A jelenség megértésében segít a **??** ábra, melyen a kérdéses jel első szívütése látható. Könnyen észrevehető, hogy az eredeti algoritmussal adott approximáció a T hullám közelében nagyon rossz. Ez egyrészt a szegmentáló algoritmus hibájának, másrészt a transzlációs paraméter hiányának köszönhető. Az eredeti algoritmus az EKG jelet a zöld pontokkal megjelölt szakaszokon közelíti, ahol az alkalmazott Hermite-függvények az intervallum középre vannak igazítva (lsd. fekete vonalak). Mivel a T hullám asszimetrikusan helyezkedik el az adott szegmensen, így még elég nagy dilatáció esetén sem lehet kompenzálni a hibát. A kidolgozott módszer azonban a (a_i, λ_i) szabad paramétereknek és az optimalizációnak köszönhetően képes megbírkózni a feladattal. Ennek következtében közel kétszer alacsonyabb PRD-vel állítja elő a jelet. Fontos megjegyezni, hogy ehhez mindkét reprezentációban ugyanannyi együttható felhasználására volt szükség. A dolgozatban bemutatott módszer tárolás szempontjából csupán abban különbözik, hogy szegmensenként szükség van plussz egy transzlációs paraméterre. A példából érthető az is, hogy a 3.1 táblázat bizonyos rekordjain az eredeti algoritmus miért teljesít majdnem olyan jól, mint a dolgozatban bemutatott módszer. Ha ugyanis normális szívütéseink vannak és a szegmentálás is pontos, akkor az intervallumok felezőpontja közel van az eltolás optimumához. Mivel ez valós jelek esetén egyáltalán nem garantált, így a dolgozat problémafelvetése jogos és az erre adott megoldás gyakorlati szempontból is értékelhető.

4. fejezet

Tapasztalatok és kitekintés

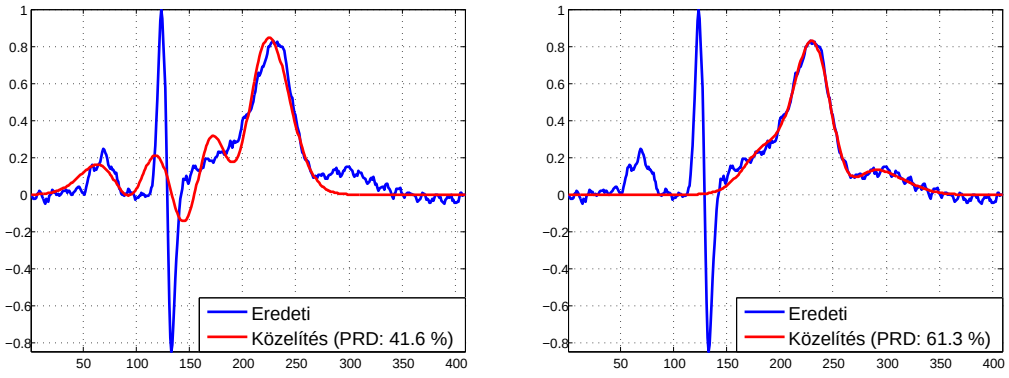
Ebben a fejezetben a bemutatott tömörítő eljárás tesztelése során megszerzett tapasztalatok, illetve az ezekből levont következtetések összegzése található. A fejezet első része a tömörítő eljárással kapcsolatos általános tapasztalatok és esetleges jövőbeli vizsgálódások vázlatát tartalmazza. A fejezet második felében található az eredmények összevetése a **??** fejezetben bemutatott specifikációval. Ennek következtében lehetőség nyílik a dolgozat elején bemutatott elvárások összehasonlítása a megvalósított eljárással. Szintén a Tapasztalatok és kitekintés fejezet végén található a köszönetnyílvánítás.

4.1. Az eljárással kapcsolatos tapasztalatok

A dolgozatban a [**?**, **?**] cikkekben bemutatott algoritmusokból kiindulva egy új EKG tömörítő eljárás konstrukciója került bemutatásra. Az módszer adaptív, amit a Hermite-függvények argumentum transzformációjával lett elérve. A reprezentáció szabad (α_i, λ_i) transzlációs és dilatációs paramétereit a Nelder Mead optimalizációs algoritmus állítja elő. Az említett optimum létezése formális bizonyítása is megtalálható a dolgozatban **??**. A tömörítő algoritmus hatékonysága a PhysioNet adatbázis valós EKG felvételein lett tesztelve. A numerikus vizsgálatokban összesen 3.5 órányi rekord került feldolgozásra, ami biztosította statisztikai kisérletekhez szükséges magas mintaelemszámot. Ezek alapján megállapítható, hogy a kiindulásként felhasznált [**?**, **?**] algoritmusok eredményeihez képest, sikerült egy hatékonyabb eljárás konstrukciója. Az eredmények között megtalálható a bemutatott algoritmus [**?**] cikkben ismertetett, JPEG 2000 alapú 2D-s EKG tömörítő eljárással vett összehasonlítása is. Ebben az esetben is a bemutatott tömörítő eljárás bizonyult hatékonyabbnak.

Megjegyzendő azonban, hogy a kidolgozott módszernek is vannak hátrányai. Mivel az iteratív approximáció során az MP megközelítés került alkalmazásra, így annak hibáit is örökölte a módszer. Előfordulhat, hogy az első lépésben nem a QRS komplexust közelíti, hanem a T hullámot. Így az eredetileg ide szánt 7 együtthatót a T hullámra "pazarolja". Ez azonban a helyes inicializálással, illetve a QRS hibájának súlyozásával kiküszöbölhető. Másik probléma, hogy az MP algoritmus egy mohó stratégiát követ

a (a_i, λ_i) paraméterek meghatározásához. Így előfordulhat, hogy egy iterációban a szívütés egynél több hullámát is közelíti, ha ez az ℓ^2 hiba szempontjából indokolt. Ezt szemlélteti a 4.1 ábra. Jól látható, hogy az első esetben a PRD kisebb, mint a második példában. Előbbinél azonban a rezidum függvény is jóval bonyolultabb. Tehát a végeredmény szempontjából előnyösebb, ha minden lépésben csak egy hullámot közelít a módszer. A jelenség az MP módszer és a mohó stratégia jól ismert hátránya, melyet a [?] dolgozat is részletesen tárgyal. Eszerint a probléma mérsékelhető, ha az mohó stratégiát más módszerekre cseréljük. Ennek a problémának egy lehetséges megoldása az iterációs megközelítés helyett, a 3 hullámhoz tartozó összesen 6 paramétert párhuzamosan történő optimalizálása. Más szóval a keresést az eredeti kettő helyett egy hat dimenziós problématérben is el lehet helyezni. Ezek a vizsgálatok jelenleg nem képezik a dolgozat tárgyát, de a fontos kiindulási pontot biztosítanak a további vizsgálatokhoz.



(a) Rossz eset. (b) Jó eset.

4.1. ábra. A mohó stratégia hátránya.

4.2. Eredmények a feladat specifikációjához képest

A ?? fejezetben bemutatásra került a feladat specifikációja, amely a dolgozatban bemutatott tömörítési eljárással, illetve az eljárás implementációjával kapcsolatban állított fel elvárásokat. A specifikáció első felében általános elvárások kerültek megfogalmazásra, melyek a tömörítő eljárást érintették. Ezek az elvárások röviden a következők:

- A tömörítés kimenete egy olyan számsor legyen, amely pontosan reprezentálja a bemenetként kapott EKG felvételt, azonban álljon a bemenetnél kevesebb mintából.
- A tömörítő eljárás szeparálja az egyes EKG felvételeken található szívütések szegmenseit.
- A tömörítő eljárás hatékonysága legyen összehasonlítható az idoralomban található egyéb eljárások hatékonyságával.

- A tömörítő eljárás szűrje az EKG felvételeken található zajt.

A **??** tesztek alapján elmondható, hogy a specifikációban megfogalmazott, tömörítő eljárással kapcsolatos elvárások közül mindegyik megvalósult. Az egyetlen kivételt a zajszűrés jelenti. A bemutatott algoritmus természetes módon szűri a bemenetkén kapott zajos felvételt, hiszen a mérést Hermite sima függvényekkel közelíti. A zajszűrés mértékének pontos meghatározása azonban további vizsgálatokat igényelne.

A dolgozat elején bemutatott specifikáció második fele a tömörítő eljárás implementációjával szemben állít elvárásokat:

- Az implementációnak képesnek kell lennie az MIT-BIH adatbázisban található EKG felvételek tömörítésére.
- Az implementáció bemenetként a fennt említett adatbázisban található felvételt, illetve egy korábbi tömörítés következtében létrejött EKG reprezentációt kell elfogadnia.

A **??** fejezetből látható, hogy az implementációval kapcsolatos legfontosabb elvárások közül mindegyiknek eleget tesz a tömörítő eljárás megvalósítása.

4.3. Köszönetnyilvánítás

5. fejezet

Függelék

A Függelék fejezet első része a felhasznát matematikai eszközök összefoglalását tartalmazza az [**?**, **?**] könyveket alapul véve. A matematikai eszközök bemutatásán kívül szintén ebben a fejezetben található a **??** fejezetben kimondott állítás bizonyítása. A Függelék fejezet végén a Nelder Mead optimalizációs algoritmus pszeudo kódja kerül ismertetésre.

5.1. Hermite polinomok, Hermite függvények

Stirling formula

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n, \binom{2n}{n} \sim \frac{2^{2n}}{\sqrt{\pi n}} \quad (n \to \infty)$$

Rodrigues formula

$$H_n(x) := e^{x^2}(-1)^n \left(\frac{d}{dx}\right)^n e^{-x^2} \quad (n \in \mathbb{N}, x \in \mathbb{R}) \tag{5.1}$$

Ortogonalitás, normált rendszer

$$\begin{aligned}
&\int_{-\infty}^{\infty} e^{-x^2} H_n(x) H_m(x)\, dx = \pi^{1/2} 2^n n!\, \delta_{mn} \quad (m, n \in \mathbb{N}) \\
&\Phi_n(x) := H_n(x) e^{-x^2/2} / \sqrt{\pi^{1/2} 2^n n!} = h_n(x) e^{-x^2/2} \quad (n \in \mathbb{N}, x \in \mathbb{R}) \\
&\int_{-\infty}^{\infty} \Phi_n(x) \Phi_m(x)\, dx = \delta_{mn} \quad (m, n \in \mathbb{N}) \\
&\int_{-\infty}^{\infty} e^{-x^2}\, dx = \pi^{1/2} < 2
\end{aligned} \tag{5.2}$$

Rekurziók

$$
\begin{aligned}
&H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x) \ (n \geq 2)\\
&H_{-1}(x) = 0, H_0(x) = 1, \quad H_1(x) = 2x\\
&\Phi_n(x) = \sqrt{\frac{2}{n}}x\Phi_{n-1} - \sqrt{\frac{n-1}{n}}\Phi_{n-2}(x) \ (n \geq 2)\\
&\Phi_0(x) = e^{-x^2/2}/\pi^{1/4}, \Phi_1(x) = \sqrt{2}xe^{-x^2/2}/\pi^{1/4} \quad (x \in \mathbb{R})\\
&H_n'(x) = 2nH_{n-1}(x) \ (n \in \mathbb{N})\\
&\Phi_n'(x) = \sqrt{2n}\Phi_{n-1}(x) - x\Phi_n(x) \quad (n \geq 0, x \in \mathbb{R})\\
&\frac{d}{dx}(x\Phi_n(x)) = (1-x^2)\Phi_n(x) + \sqrt{2n}x\Phi_{n-1}(x) \quad (n \geq 1, x \in \mathbb{R})
\end{aligned}
\tag{5.3}
$$

Függvényértékek a 0 helyen

$$
\begin{aligned}
&H_{2m+1}(0) = 0, \ H_{2m}(0) = (-1)^m\frac{(2m)!}{m!}\\
&H_{2m}'(0) = 0, \ H_{2m+1}'(0) = (-1)^m\frac{(2m+2)!}{(m+1)!}\\
&\Phi_{2m}(0) = (-1)^m 2^{-m}\sqrt{\binom{2m}{m}} \sim (2m)^{-1/4}\pi^{-1/2}\\
&\Phi_{2m+1}'(0) = (-1)^m 2^{-m}\sqrt{\binom{2m}{m}} \sim (8m)^{1/4}\pi^{-1/2}
\end{aligned}
\tag{5.4}
$$

Kvadratúra formula

$$\int_{-\infty}^{\infty} e^{-x^2} f(x)\,dx \backsimeq \sum_{i=1}^{n} \lambda_{in} f(x_{in})$$

$$H_n(x_{in}) = 0, \ \lambda_{in} = \frac{\pi^{1/2}2^{n+1}n!}{[H_n'(x_{in}]^2} \quad (n = 1, 2, \cdots, 1 \leq i \leq n)$$

5.2. Approximáció afffin transzformáltakkal

Jelölje a $\mathbb{R}$ számegyenesen szakaszonként folytonos, négyzetesen integrálható függvények terét $\mathcal{F}$. Az $\mathcal{F}$ téren jelentse

$$\langle f, g\rangle := \int_{-\infty}^{\infty} f(x)g(x)\,dx$$

a skaláris szorzatot.

Altértől vett távolság

A $\phi_n \in \mathcal{F}$ $(n \in \mathbb{N})$ függvényrendszer ortonormált, ha

$$\langle \phi_n, \phi_m \rangle := \delta_{mn} \quad (m, n \in \mathbb{N}).$$

Ebből a rendszerből az

$$\ell(x) = \ell(x, \lambda, a) = \lambda x + a \quad (x, a \in \mathbb{R}, \lambda > 0)$$

affin transzformációval származtatott

$$\phi_n^{a,\lambda}(x) := \sqrt{\lambda}\phi_n(\lambda x + a) = \sqrt{\lambda}\phi_n(\ell(x)) \ (x, a \in \mathbb{R}, \lambda > 0, n \in \mathbb{N})$$

rendszer szintén ortonormált. Valóban az $u = \ell(x) = \lambda x + a, dx = du/\lambda$ helyettesítéssel

$$\begin{aligned} \langle \phi_n^{a,\lambda}, \phi_m^{a,\lambda} \rangle &:= \int_{-\infty}^{\infty} \lambda \phi_n(\lambda x + a)\phi_m(\lambda x + a)\, dx = \\ &= \int_{-\infty}^{\infty} \phi_n(u)\phi_m(u)\, dx = \delta_{mn} \quad (m, n \in \mathbb{N}). \end{aligned} \tag{5.5}$$

Jelölje $X_n^{a,\lambda}$ a $\phi_0^{a,\lambda}, \cdots, \phi_n^{a,\lambda}$ függvények által kifeszített alteret. Adott $f \in \mathcal{F}$ függvényhez a legközelebb eső $X_n^{a,\lambda}$-beli függvényt az

$$S_n^{a,\lambda} f := \sum_{k=0}^{n} \langle f, \phi_n^{a,\lambda} \rangle \phi_n^{a,\lambda}$$

Fourier-projekcióval, az eltérés normájának a négyzete a

$$D_n^2(a, \lambda) := \|f - S_n^{a,\lambda} f\|^2 = \langle f, f \rangle - \sum_{k=0}^{n} |\langle f, \phi_k^{a,\lambda} \rangle|^2$$

függvénnyel adható meg. A $D_n(a, \lambda)$ minimumának meghatározása ekvivalens az

$$F_n(a, \lambda) := \sum_{k=0}^{n} |\langle f, \phi_k^{a,\lambda} \rangle|^2 \quad ((a, \lambda) \in T := \{(p, q) \in \mathbb{R}^2 : p \in \mathbb{R}, q > 0\}$$

függvény maximumának meghatározásával.

Becslések az F_n ***függvényre***

A továbbiakban a

$$\phi_n(x) = \Phi_n(x) = h_n(x)e^{-x^2/2} \quad (x \in \mathbb{R}, n \in \mathbb{N})$$

normált Hermite-függvényeket választjuk ortonormált rendszernek. Ekkor

$$\begin{aligned}
&|\Phi_n(x)| \le M_n e^{-x^2/4} \le M_n,\ |\Phi_n'(x)| \le N_n e^{-x^2/4} \le N_n\ (x \in \mathbb{R}, n \in \mathbb{N}) \\
&M_n := \max_{x\in\mathbb{R}} |h_n(x)| e^{-x^2/4},\ N_n := \max_{x\in\mathbb{R}} |h_n'(x) - x h_n(x)| e^{-x^2/4} \\
&\int_{-\infty}^{\infty} |\Phi_n(x)|\,dx \le M_n \int_{-\infty}^{\infty} e^{-x^2/4}\,dx < 4M_n
\end{aligned} \tag{5.6}$$

Ezeket felhasználva becsléseket adhatóak az

$$A_k(a,\lambda) := \langle f, \phi_k^{a,\lambda}\rangle = \sqrt{\lambda}\int_{-\infty}^{\infty} f(x)\phi_k(\lambda x + a)\,dx = \frac{1}{\sqrt{\lambda}}\int_{-\infty}^{\infty} f((u-a)/\lambda)\phi_k(u)\,du$$

Fourier-együtthatókra. A továbbiakban kompakt tartójú, korlátos függvényekből indulunk ki. Nem jelenti az általánosság megszorítását a következő feltételezés:

$$f(x) = 0,\ \text{ha } |x| \ge 1,\quad |f(x)| \le 1\ (x \in \mathbb{R}).$$

A paraméteres integrálokra vonatkozó tételből következik, hogy az

$$F_n(a,\lambda) := \sum_{k=0}^{n} |A_k(a,\lambda)|^2 \quad ((a,\lambda) \in T > 0)$$

függvény folytonosan differenciálható. A következőkben bemutatásra kerül, hogy az $F_n : T \to [0,\infty)$ folytonos függvénynek van maximuma a T (nem kompakt) halmazon.

A fenti egyenlőtlenségekből következik, hogy

$$\begin{aligned}
&|A_k(\lambda,a)| \le \sqrt{\lambda} M_k \int_{-\infty}^{\infty} |f(x)|\,dx \le 2M_k\sqrt{\lambda}\ \ (\lambda \le 1) \\
&|A_k(\lambda,a)| \le \frac{1}{\sqrt{\lambda}}\int_{-\infty}^{\infty} |\Phi_k(u)|\,du \le \frac{4M_k}{\sqrt{\lambda}}\ (\lambda \ge 1),
\end{aligned} \tag{5.7}$$

továbbá $\lambda \ge 1,\ a \ge 2\lambda$ esetén

$$\begin{aligned}
&|A_k(\lambda,a)| \le \frac{1}{\sqrt{\lambda}}\int_{a-\lambda}^{a+\lambda} |\Phi_k(u)|\,du \le \frac{1}{\sqrt{\lambda}}\int_{a-\lambda}^{\infty} |\Phi_k(u)|\,du \le \\
&\le \frac{M_k}{\sqrt{\lambda}}\int_{\lambda}^{\infty} e^{-u^2/4}\,du < \frac{M_k}{\sqrt{\lambda}}\int_{\lambda}^{\infty} u e^{-u^2/4}\,du = 2M_k\sqrt{\lambda}e^{-\lambda^2/4} < \frac{8M_k}{\sqrt{\lambda}}
\end{aligned} \tag{5.8}$$

Hasonlóan egyenlőtlenséget ad az $a < -2\lambda$ eset. Innen következik, hogy az

$$T_s := \{(p,q) : -2s \le p \le 2s, 1/s \le p \le s\}$$

téglalapon kívül érvényes a

$$\sup_{(a,\lambda)\notin T_s} |A_k(a,\lambda)| \le \frac{8M_k}{\sqrt{s}}\quad (s > 1)$$

becslés. Ennek alapján nyilvánvaló, hogy az F_n függvénynek létezik a maximuma a T téglelapon.

5.3. Algoritmusok

Az indexek rendezésénél $y_3 \leq y_2 \leq y_1$ teljesül. Az IterLepes(5) rutin az (x_1, x_2, x_3) hármast a T_5 transzformációval kapott hármasra cseréli.

Algorithm 2 Nelder-Mead

```
1: function NELDER-MEAD
2:     if y3 ≤ y4 ∧ y4 < y2 then
3:         x1 = x4
4:     else if y4 < y3 then
5:         if y5 < y4 then
6:             x1 = x5
7:         else
8:             x1 = x4
9:         end if
10:    else if y4 ≥ y2 then
11:        if y4 < y1 then
12:            if y6 ≤ y4 then
13:                x1 = x6
14:            else
15:                IterLepes(5)
16:            end if
17:        else if y4 ≥ y1 then
18:            if y7 < y3 then
19:                x1 = x7
20:            else
21:                IterLepes(5)
22:            end if
23:        end if
24:    end if
25: end function
```