Szegedi Tudományegyetem

Informatikai Intézet

Aktigráfiás adatok mérését és analízisét támogató applikáció fejlesztése

Szakdolgozat

Készítette:

Vince Dániel

Mérnökinformatikus BSc szakos hallgató

Témavezető:

Vadai Gergely

tanársegéd

Szeged

2017

Feladatkiírás

Az emberi aktivitás vizsgálatát szolgáló aktigráfok (általában végtagon mért) 3 irányú gyorsulásjel alapján, bizonyos időközönként határozzák meg az aktivitási szintet. Az aktivitás mérése fontos szerephez jut az orvosi diagnosztikában, például az alvási ciklusok vizsgálata során, azonban lehetőséget biztosít az emberi mozgásmintázatok statisztikai vizsgálatára is. Ez utóbbi vizsgálata boltban vásárolható aktigráfok illetve létező okostelefonos alkalmazások esetén korlátokba ütközhet.

A szakdolgozó feladata egy a kutatásokat támogató, korábban elkészített, lokációs (GPS) adatokat mérő okostelefon-alkalmazás és adatbázis továbbfejlesztése aktigráfiás idősorok mérésére. A szakdolgozó feladata továbbá mérések végzése és a mért adatsorok alapvető statisztikai vizsgálata.

Tartalmi összefoglaló

- **A téma megnevezése:**

Lokációs adatokat mérő okostelefonos alkalmazás bővítése aktigráfiás jelek rögzítésével és feldolgozásával.

- **A megadott feladat megfogalmazása:**

Lakos Ferenc által elkészített Androidos alkalmazás továbbfejlesztése, alkalmassá tétele gyorsulásjelek adatbázisba történő mentésére, illetve későbbi lehetőség biztosítása az adatok elemzésére.

- **A megoldási mód:**

A fejlesztés a már definiált kliens-szerver architektúrára épített, melynek során a kliens szerepét továbbra is az Androidos alkalmazás, a szerver szerepét pedig egy új REST API töltötte be.

- **Alkalmazott eszközök, módszerek:**

A kliens alkalmazás fejlesztése a Google által támogatott, IntelliJ alapú Android Studio fejlesztői környezetben történt, Java nyelven. A szerver réteg egy ASP.NET MVC alkalmazás, melynek fejlesztése a Microsoft által készített Visual Studio 2017 fejlesztői környezetben történt, C#, Razor, JavaScript, HTML és CSS programozási nyelveken. Ez az alkalmazás magába foglalja a REST API-t és az adminisztrációs felületet is. Az alkalmazás mögötti adatbázis szervernek a Microsoft SQL Server-t választottam.

- **Elért eredmények:**

A kliens alkalmazást módosítottam, hogy alkalmas legyen az aktigráfiás jelek mérésére és tárolására, nagy hangsúlyt fektetve a hatékonyságra, mind akkumulátor idő, mind méret tekintetében. Az Androidos alkalmazás a hosszú távú mérések során nem bizonyult teljesen megfelelőnek. A szerver réteget képező alkalmazást újra írtam tervezési és biztonsági megfontolásokból és alkalmassá tettem az aktigráfiás jelek feldolgozására.

- **Kulcsszavak:**

Aktigráfia, mintavételezés, Android, ASP.NET, REST API, MS SQL

Tartalomjegyzék

1. Bevezetés

Napjainkban aktív kutatási terület az emberi aktivitás elemzése és megismerése. A mozgásmintázatok elemzése és az aktivitás hosszú távú figyelése potenciálisan hasznos lehet olyan kutatók és orvosok számára, akik például neurológiai rendellenességeket vizsgálnak[1][2][7][17] (pl. bipoláris zavar, depresszió). Az úgynevezett aktigráfok az emberi test mozgását mérik háromtengelyű gyorsulásmérővel és határoznak meg különböző aktivitás értékeket. A szenzorok magas pontosságú nyers gyorsulásjeleket szolgáltatnak meghatározott mintavételezési frekvenciával. Léteznek boltban kapható aktivitásmérők, melyeket a csuklóra erősítve folyamatosan figyelik az emberi aktivitást, amit okostelefonon keresztül bármikor ellenőrizhetünk. Mivel maga az aktivitás, mint matematikai mérőszám nem egységesen meghatározott, így számunkra fontos az, hogy több számítási módot összehasonlítsunk.

Lakos Ferenc az emberi mozgásmintázatokat vizsgálta[18] fix mintavételezés mellett. A kutatás a térbeli mozgásmintázatok megfigyelésére és elemzésére fókuszált, a célok között nem szerepelt az aktivitási szint meghatározása. Ezen eredmények mellett az időben fixen mintavételezett GPS adatok különbségéből adódó elmozdulással definiálható egyfajta aktivitásérték. A technika fejlődése lehetővé tette a téma pontosabb vizsgálatát, gyorsulásmérő szenzorokkal a kérdőívek helyett, viszont a céleszközök viselésének megkövetelése módosíthatja a vizsgált egyén viselkedését[8]. Az okostelefonok napjaink fontos részévé váltak, emellett minden okostelefon rendelkezik gyorsulásszenzorral, így alkalmasak lehetnek a feladatra. Az elemzés szempontjából fontos az aktivitás hosszú távú, fixen mintavételezett mérése, melyre a meglévő alkalmazás nem volt alkalmas.

A célom a meglévő alkalmazás optimalizálása, hibáinak javítása és bővítése volt, hogy lehetővé tegye a GPS adatokon túl nyers gyorsulásértékek mérését és tárolását meghatározott mintavételezési frekvenciával későbbi analízis céljából. A kutatás szempontjából fontos a GPS és a gyorsulásjelek egyenletes mérése hosszú távon, így a kettőt egymás függvényében is vizsgálhatjuk. Dolgozatom Lakos Ferenc „Hétköznapi mozgásmintázatok vizsgálatát támogató applikáció fejlesztése” című szakdolgozatára épül, így az ott kifejtett témákat nem tárgyalom újra. A következő fejezetekben bemutatásra kerülnek a rendszer egyes elemei, a bővítés és az újratervezés folyamata és motivációi. Ezen felül a fejlesztés és a tesztelés során felmerülő nehézségek és hibák leküzdésének módszerei és a különböző számítási módszerek összehasonlítása.

2. Irodalmi áttekintés

A fejezet visszatekintéssel kezd, melyben az emberi mozgásmintázatokat és azok vizsgálatát mutatom be nagyvonalakban a konzulensem kutatására és Ferenc megvalósítására alapozva, melyet jelen munkám bővít a lokációs elemzés és az aktivitás összehasonlításával, majd ismertetésre kerülnek az aktigráfia alapfogalmai, a használt számítási módszerek, valamint bemutatásra kerülnek az általam használt technológiák.

2.1 Az emberi mozgásmintázatok

Az élőlények mozgásának időbeli és térbeli mintázatait régóta vizsgálják, több tudományág számára fontos kutatási terület[18]. A mozgás időbeli vizsgálata az élőlények mozgását jellemző törvényszerűségeket definiálja, ezen felül hasznos az orvostudomány számára oly módon, hogy különféle betegségek megállapíthatók a mintázatokból.

A konzulensem kutatási területének egyik fő kérdése, hogy matematikailag milyen módon írható fel az élőlények mozgása, illetve milyen modellek alkalmazhatók a célra. Több élőlény mozgását vizsgálva statisztikai következtetéseket lehet levonni a kérdések megválaszolására. Ily módon a mozgást véletlenszerű lépések sorozataként lehet tekinteni, mely egy kétdimenziós bolyongás. A bolyongás legismertebb típusa a *Brown mozgás* (részecskék véletlen mozgása), viszont ez a fajta bolyongás nem áll fenn minden élőlénynél. Vízi ragadozók során megfigyelték, hogy kisebb csomópontokban keresnek táplálékot, majd egy idő után egy másik csomóponthoz úsznak. Ezt a mozgásmintázatot a nagy elmozdulásra való tekintettel *Lévy flight*-nak nevezi az irodalom, amely más élőlényeknél is fellelhető.

Aktív kutatási terület még az emberek mozgásmintázatának vizsgálata is. Vita tárgyát képezi, hogy tekinthető-e *Lévy flight* szerű mozgásnak az emberi bolyongás. A mért adatok több esetben is igen meggyőző eredményt mutatnak, viszont figyelembe kell venni a vizsgált emberek különböző viselkedését. Abban az esetben, ha feltételezzük, hogy az emberek nem teljesen véletlenszerűen bolyonganak az életterükben, léteznek úgynevezett kitüntetett helyek, melyek lehetnek például a munkahely, iskola, otthon stb. További vizsgálatokra ad lehetőséget, hogy ezeket a kitüntetett helyeket hányszor látogatják meg, illetve meddig tartózkodnak ott az egyes egyének. Az élőlények cirkadián ritmusának nevezik az olyan folyamatokat, amelyek ritmusszerűen ismétlődnek (pl. dolgozni járás).

Ferenc az ember lokációjának eloszlásait vizsgálta időtartományban, majd a kutatás tovább folyt frekvenciatérben, mely újszerű nézőpont. Aktigráfiás mérések során hasonló összefüggéseket véltek felfedezni, mely jelen kutatás tárgyát képezi. A vizsgálatok kivitelezéséhez a telefontársaságok cellainformációi alkalmasak lehetnek, viszont a pontosabb GPS által szolgáltatott koordinátákat szokás felhasználni. A bolyongások meghatározásához a GPS adatokat mintavételezetten szükséges vizsgálni.

2.2 Aktivitás és aktigráfok

Az emberi aktivitás elemzésében és megismerésében nagy potenciál lakozik. Több kutatás foglalkozik az emberi aktivitás vizsgálatával, mellyel különböző rendellenességek kimutathatók[1][2][7][17] (pl. bipoláris zavar, depresszió). A tanulmányok különböző módszerekkel definiálják az aktivitást, így matematikai mérőszámként nem egységes. Annak ellenére, hogy a tanulmányok különböző módszerrel számítják, többnél fellelhető, hogy aktív és passzív szakaszokra osztja azt[1][9][12], illetve a szakaszok hatványfüggvényszerinti eloszlást mutatnak. Az aktivitás nem egy pillanatnyi érték, hanem egy intervallumra vonatkozó mérőszám. A kutatók ezen a ponton is különböző módszerrel jártak el, van amelyik másodpercenként és van amelyik percenként határozott meg egy aktivitás szintet. Az eloszlások vizsgálata azt mutatta, hogy a percenkénti összegzés elegendő információt hordoz a mozgás struktúrájáról, Ferenc lokációs mérése is ezt a vonalat követte, így jelen implementáció is ezt a megvalósítást alkalmazza.

Az aktivitásmérő, vagy aktigráf egy – legtöbbször karon viselhető – céleszköz, amely képes meghatározott mintavételezési frekvenciával mérni és eltárolni háromtengelyű gyorsulásérzékelő által szolgáltatott adatokat hosszabb ideig, akár hetekig. Az eltárolt adatokat felhasználva képes aktivitás szintet számolni időegységekre lebontva, mellyel megállapítható a vizsgált egyén napi, vagy akár heti rutinja is. A hátránya a boltban kapható, „dobozos” aktivitásmérőknek, hogy a nyers adat feldolgozásánál egy – sokszor nem nyilvános – algoritmust használnak az aktivitásérték meghatározására. A vizsgálat eredményének szempontjából meghatározó lehet még az új eszköz viselése, mely szokatlansága miatt módosíthatja a vizsgált egyén viselkedését[8]. Ezekre a problémákra megoldást próbálnak nyújtani az egyre népszerűbb okostelefonon futó alkalmazások.

2.2.1 Okostelefonon futó alkalmazás, mint aktigráf

Az okostelefonok piacát az Apple által fejlesztett iOS és a Google által fejlesztett Android operációs rendszerrel ellátott eszközök uralják. Mindkét rendszernek számos felhasználója van, viszont az Android piac nyitottabb, így ezen a platformon futó alkalmazásokat tárgyalok ebben a fejezetben. Magával az operációs rendszerrel és a lehetőségeivel később fogok foglalkozni. Az alkalmazásokat böngészve a Play Store[20]-ban nagy választékot találhatunk aktivitásmérők terén. A Store-ban található alkalmazások nem azt a megközelítésmódot használják, mely az említett kutatás tárgyát képező tudományos vizsgálatokhoz kell. Azok közül, amik gyorsulásszenzorokat is használnak, két nagy csoportot különböztethetünk meg:

1. rövid ideig tartó aktivitás mérésére alkalmasak,
2. hosszú ideig tartó aktivitás mérésére alkalmasak.

A rövid ideig tartó aktivitás alatt sportolásra gondolhatunk, mint pl. futás, kerékpározás. Az ilyen mozgásokra kihegyezett alkalmazások intenzíven használják a telefon erőforrásait, így nem alkalmasak hosszú távú mérések kivitelezésére. Ilyen alkalmazások: Runtastic, Runkeeper, Endomondo - Running & Walking stb. Ezen alkalmazásoknak a közös tulajdonsága, hogy a mérés indítása előtt kiválaszthatjuk, milyen mozgásformát fogunk végezni, így az alkalmazás különféle modellek alapján elemzi a teljesítményt. Ez nem a kutatás szempontjából releváns aktivitásérték.

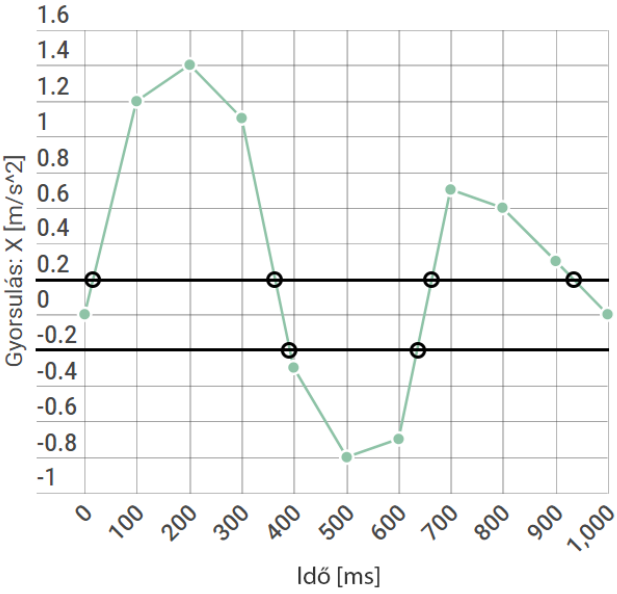
A hosszú ideig tartó aktivitásokat mérő alkalmazások nem kifejezetten különféle sportokra vannak kihegyezve, inkább a háttérben futnak és lépésszámlálóként működnek[17]. Ilyen alkalmazások: Step Counter - Pedometer Free & Calorie Counter, Pedometer & Weight Loss Coach, stb. A lépésszámlálók mellett vannak még úgynevezett *alváskövető* alkalmazások, melyek lényege, hogy elalvás előtt bekapcsoljuk az alkalmazást, az eszközt az ágyban magunk mellé (vagy párna alá) tesszük és egész éjjel figyeli az alvási szokásainkat a hangokból és a gyorsulásmérő által szolgáltatott jelekből, majd felkelés után megtekinthetjük az elkészített jelentést, hogy hogyan is aludtunk. Ezen alkalmazások kérhetik a töltőre való helyezést éjszakára, hogy az eszköz ne merüljön le, illetve a különféle optimalizációk ne gátolják a mérés lefolyását.

Jelen kutatásban az aktivitásmérést a GPS jelek mintavételezett mérése mellett, folytonosan szeretnénk mérni hosszú időn át, így az említett kategória tagjai nem bizonyulnak megfelelőnek.

2.2.2 Számítási módszerek

A *2.* fejezet bevezetőjében szó esett arról, hogy az aktivitás, mint matematikai fogalom nem egységesen meghatározott. Ennek fényében felmerülhet a kérdés, hogy a mérések folyamán mely számítási módszer használata a legcélravezetőbb. Mivel erre a kérdésre nem született megfelelően indokolt válasz, így több módszer is ismertetésre kerül, hogy a mérés során különböző algoritmusok segítségével is számítható legyen az aktivitás. Ahhoz, hogy ez kivitelezhető legyen, az eszköznek nyers adatot kell szolgáltatnia, mely a későbbiek folyamán elérhető a kutatók számára.

Több tanulmány az emberek alvási ciklusának vizsgálata[2][9][12] során az úgynevezett *Zero-Crossing Method* (*ZCM*) módszert használt. A módszer számolja, hogy a gyorsulásmérő által szolgáltatott adatok hányszor metszik tengelyüket. Az így számolt aktivitás zajos mivolta miatt szükséges egy küszöbérték meghatározása[12], mellyel a zajszint csökkenthető. Egy mozdulat alkalmával, mely által generált gyorsulásérték a meghatározott küszöbértéket metszi, két metszéspont fog megjelenni. A [12] tanulmány a küszöbértéket egy másfajta megközelítésben is alkalmazza, ez pedig a *küszöbérték feletti idő* (*Time-Above-Threshold, TAT*). A *2.1* ábra szemlélteti az említett számítási módszerek működésének elvét *0.2* $\frac{m}{s^2}$ küszöbérték mellett. A *TAT* és a *ZCM* mellett a tanulmány egy harmadik módszert is alkalmaz az aktivitás számítására, ez pedig az *Integráló* módszer, mely a görbe alatti területet integrálva határozza meg az aktivitásértéket. Ezeket a módszereket analóg jelekre alkalmazták, még a jelen kutatás diszkrét jelekkel foglalkozik, így ezen módszerek némileg módosítva kerültek megvalósításra.



2.1 ábra: A ZCM számítási módszer működési elve.

Egy tanulmány[17] az emberek aktivitását vizsgálta egy rehabilitációs központban. A folyamban résztvevőket 7:00 – 17:00 között vizsgálták egy céleszközzel 128 Hz-es mintavételezési frekvenciával. A résztvevőknek különböző fizikailag megterhelő mozgásokat kellett csinálniuk, így az aktivitást modellekhez kötötték. A modell öt együtthatóból (b_0 – b_4) állt, amelyeket a nyers adatokból statisztikai számítások alapján határoztak meg. A modellek: nyugalmi állapot (fekvés, ülés és állás kombinációja), kerékpározás, lépcsőzés felfelé, gyaloglás (lassú és gyors gyaloglás, lépcsőzés lefelé és kocogás) és ismeretlen aktivitás. Az energiakibocsátást (*energy expenditure, EE*) a következő egyenlettel határozták meg:

$$EE = b_0 + b_1 * EEAC + b_2 * Kor * EEAC + b_3 * Magasság * EEAC + b_4 * Tömeg * EEAC \quad (2.1)$$

A (2.1) egyenletben az *EEAC* a nyers gyorsulásjelek átlagolása és simítása mozgóátlaggal, másodpercenként N adattal. A (2.1) egyenlet eredményeként kalóriát kapunk, amely a jelen kutatás számára szükségtelen, így a (2.2) egyenletet használtam fel az aktivitás számítás egy módjaként.

$$EEAC = smooth\left(\frac{1}{N}\sum_{i=1}^{N}\sqrt{a_{xi}^2 + a_{yi}^2 + a_{zi}^2}\right) \quad (2.2)$$

Ez a módszer jó alapot ad olyan mozgásoknál, ahol viszonylag nagyok az elmozdulások. A (2.2) egyenletben található összefüggést egy másik tanulmány[12] *Activity Countnak* (*AC*) nevezi, mely egyenlő a vektor hosszával:

$$AC = \sqrt{a_x^2 + a_y^2 + a_z^2} \quad (2.3)$$

Az *AC* mellett a [12] tanulmány még egy úgynevezett aktivitás indexet is vizsgált (*Activity Index, AI*), melyet nem a vektor hosszával definiáltak, hanem az adathalmaz szórásával.

$$AI_i(t;H) = \sqrt{max\left(\frac{1}{3}\left(\sum_{j=1}^{3}\sigma_{ij}^2(t;H) - \bar{\sigma}^2\right), \quad 0\right)} \quad (2.4)$$

A (2.4) egyenlet az AI-t időegységekre számítja ki mely *t* időpillanatban kezdődik és *H* ideig tart. Az egyenletben szereplő $\sigma_{ij}^2(t;H)$ a gyorsulásértékek szórása a *j* tengelyen (j = 1, 2, 3)

a megadott intervallumban. Ez az érték normalizálva van az adathalmaz teljes $\bar{\sigma}^2$ szórásával, mely előáll a $\bar{\sigma}^2 = \sigma_1^2 + \sigma_2^2 + \sigma_3^2$ egyenlet formájában.

A nyers jelek feldolgozása folyamán aktivitás értéket bizonyos időnként számolunk, ahogy ez a (2.4) egyenletben is megmutatkozik. Egyes tanulmányok másodpercenként teszik ezt meg, amíg mások percenként teszik ugyanezt[2][8][9][11][12][17]. A konzultációk során arra jutottunk, hogy a percenkénti összegzés megfelelő a kutatás számára. A későbbiekben kifejtett mintavételezési pontatlanság miatt minden számítás végeredményét osztja az adott percben található pillanatképek száma, mint ahogyan a *2.3* egyenlet is teszi.

2.3 Android operációs rendszer és lehetőségei

A Google tulajdonában álló Android egy Linux kernelt használó mobil operációs rendszer. Elsősorban érintőképernyős eszközökre tervezve, viszont ma már karóráktól kezdve, a tévéken át, gépjárművek fedélzeti számítógépeként is megjelenhet. A platformra történő fejlesztés Java nyelven történik. Több fejlesztői környezet volt fellelhető, viszont mára a cég által is ajánlott Android Studio maradt a piacon. A legújabb verzió *Oero* névre keresztelt, és API 26 a verziószáma.

Verzió	Kódnév	API	Eloszlás
2.3.3 – 2.3.7	Gingerbread	10	0.6%
4.0.3 – 4.0.4	Ice Cream Sandwich	15	0.6%
4.1.x	Jelly Bean	16	2.3%
4.2.x		17	3.3%
4.3		18	1.0%
4.4	KitKat	19	14.5%
5.0	Lollipop	21	6.7%
5.1		22	21.0%
6.0	Marshmallow	23	32.0%
7.0	Nougat	24	15.8%
7.1		25	2.0%
8.0	Oreo	26	0.2%

2.1 táblázat: Az operációs rendszer verziói és jelenlegi eloszlása a piacon. Forrás: [5]

2.3.1 Szenzorok

A legtöbb okostelefon rendelkezik beépített szenzorokkal, melyek érzékelik a mozgást, a forgást és különböző környezeti körülményeket. Ezek a szenzorok képesek nyers adatot biztosítani magas precizitással és pontossággal. Az adatok felhasználásával játékokat irányíthatunk (pl. autós játékok), de kutatási célra is felhasználhatók a pontosságból adódóan[18]. Az operációs rendszer egy keretrendszert biztosít a fejlesztőknek, mely segítségével el lehet érni az eszközön található szenzorokat, ezek specifikációját (mint pl. értéktartomány, gyártó, felbontás).

Az operációs rendszer három fő érzékelőkategóriával rendelkezik:

1. Mozgásérzékelők

   Ezek az érzékelők gyorsulásokat és forgásokat mérnek három tengely mentén. Ide tartoznak a gravitációs szenzorok, giroszkópok, forgási vektorérzékelők és gyorsulásmérők.

3. Környezetérzékelők

   Ezek az érzékelők különféle környezeti hatásokat mérnek, mint pl. hőmérséklet, nyomás, páratartalom. Ide sorolhatók a barométerek, termométerek és fényérzékeny szenzorok.

4. Helyzetmeghatározók

   Ezek az érzékelők az eszköz fizikai pozícióját mérik. Ide tartoznak a magnetométerek, orientáció érzékelők és a lokációs helymeghatározó szenzorok (GPS).

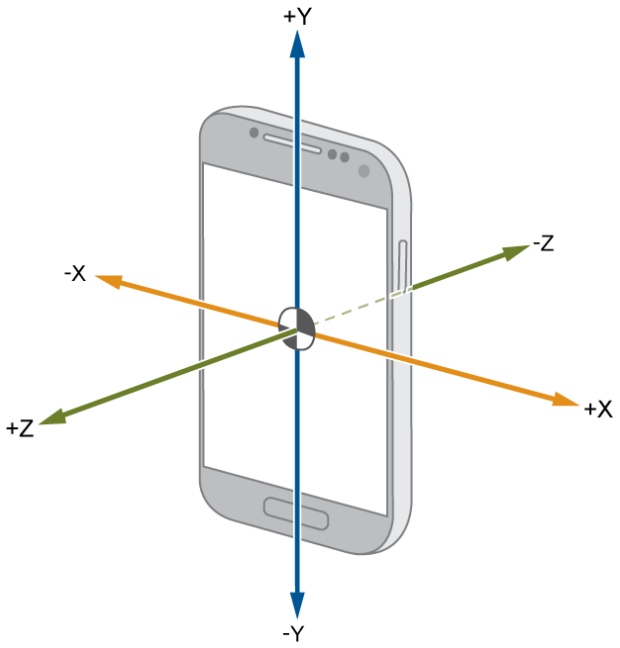
Maguk az érzékelők lehetnek hardveresek (*hardware-based*) és szoftveresek (*software-based*). A hardveres szenzorok fizikai részét képezik az eszközöknek és mért eredményeket szolgáltatnak a környezetükről (pl. gyorsulásmérő, mágneses térerősségszenzor). A szoftveres szenzorok nem találhatók meg fizikailag az eszközben, viszont utánozzák hardveres társaik viselkedését. Ezek egy vagy több hardveres szenzor feldolgozott értékét szolgáltatják (pl. lineáris gyorsulásmérő a hardveres gyorsulásmérő feldolgozott értékét szolgáltatja, kód szintjén ugyanúgy kell kezelni, mint egy hardveres szenzort, viszont fizikailag nem található meg az eszközben,).

A kutatásom kivitelezéséhez megfelelő szenzorok az említett gyorsulásmérő és lineáris gyorsulásmérő, melyeket a későbbiekben kifejtek. Az operációs rendszer nem követeli meg a gyártóktól, hogy bizonyos érzékelőket építsen az eszközeibe, így szenzorok

széles választékával találkozhatunk a piacon. Ezzel együtt a konfiguráció is változik. Ennek okán fontos olyan alkalmazásoknál, melyek funkcionalitása függ a megfelelő szenzor meglététől, futásidőben megbizonyosodni azok létezéséről és minőségéről. Két lehetőség áll a fejlesztők előtt, amellyel ezt megvalósítható:

1. Play szűrők alkalmazása, így csak a megfelelő szenzorokkal ellátott eszközökre lehet letölteni az alkalmazást.
2. Futásidőben detektálni az érzékelőket.

A keretrendszer az adatok előállításakor egy háromtengelyű koordinátarendszert használ, mely az eszközhöz van rögzítve. Fontos megjegyezni, hogy a koordinátarendszer tengelyei nem cserélődnek fel, amikor a telefon orientációja megváltozik.



2.2 ábra: Az eszközhöz rögzített koordináta rendszer. Forrás: [6]

Ahhoz, hogy elérjük az érzékelő által szolgáltatott adatokat, a *SensorEventListener* interfészt és annak két metódusát kell megvalósítani: *onAccuaryChanged* és *onSensorChanged*. Mindkettőt a rendszer hívja meg, az első esetben az érzékelő pontossági értéke változott, a második esetben pedig egy újonnan mért adattal szolgál. A paraméterül kapott objektum tartalmazza az új értéket, annak pontosságát és időbélyegét.

2.2.2 Mintavételezés

Egyik fontos kérdés, hogy milyen maximális mintavételezési frekvenciával tud adatot szolgáltatni egy szenzor. Ez meghatározható a keretrendszer getMinDelay függvényhívása által. Amelyik érzékelő nem nulla értékkel tér vissza, azt *streaming sensor*-nak nevezi az irodalom. Ebben az esetben bizonyos időközönként szolgáltat adatot az érzékelő, ezt már az Android 2.3-ban (API Level 9) bevezették. Abban az esetben, ha a getMinDelay nullával tér vissza, így az érzékelő csak akkor ad jelet, ha változik a mért fizikai mennyiség. Fontos megjegyezni, hogy a maximális mintavételezési frekvencia (ami a getMinDelay-ből számítható) nem feltétlenül az, amellyel a keretrendszer szolgáltatja az adatot az alkalmazásnak. Az operációs rendszer állapotától függhet a tényleges mintavételezés, így a fix frekvencia nincs garantálva az operációs rendszer által. Az ajánlás az, hogy több adat időbélyegéből számítsuk ki a mintavételezés pontos frekvenciáját. A Google ajánlása szerint a lehető legkisebb mintavételezési frekvenciát használjuk, amely még megfelel a specifikációnak. Minél kisebb mintavételezést választunk, annál kevésbé terheljük a processzort és kisebb az energiafelhasználás. Mielőtt egy érzékelőt szeretnénk használni, meg kell mondani a keretrendszernek, hogy milyen késleltetéssel küldjön adatokat. Vannak alapértelmezett értékek, amelyeket használhatunk és van lehetőség saját érték beállítására is.

Megnevezés	Érték [µs]
SENSOR_DELAY_NORMAL	200.000
SENSOR_DELAY_UI	60.000
SENSOR_DELAY_GAME	20.000
SENSOR_DELAY_FASTEST	0

2.2 táblázat: Alapértelmezett késleltetések és értékeik. Forrás: [5]

2.2.2 Gyorsulásérzékelő

Egy test gyorsulása a súlyos vagy tehetetlen tömeg miatt erőhatást is jelent, ami rugalmas anyag deformációját okozza. Ha egy testet gyorsítunk, akkor arra erővel hatunk, még a súlyt a gravitáció okozza. A gyorsulásszenzorokban a kapacitív elmozdulás-mérés elvét alkalmazzák, mikroméretű kivitelben. A szenzorok integrált áramkörben vannak, így az azt használóknak nem kell kis kapacitást mérni. A gyorsulás vektormennyiség, ennek

megfelelően szükséges lehet akár több független irány mentén mérni. A telefonokban található gyorsulásérzékelők három tengely mentén mérnek. A méréstartományuk általában a g gravitációs gyorsuláshoz van igazítva.

A szenzor az eszközre ható gyorsulásokat érzékeli, beleértve a gravitációs gyorsulást is. Ha az eszközt az asztalra helyezzük és nyugalmi állapotban van, akkor a szenzor által szolgáltatott érték a gravitációs gyorsulással lesz egyenlő ($g = 9.81\ m/s^2$) és nullával lesz egyenlő, ha az eszköz szabadon esik.

Szoftveres megoldásként létezik úgynevezett lineáris gyorsulásérzékelő is. A szenzor által szolgáltatott adatok nem tartalmazzák a gravitációs gyorsulást, melyet szoftveresen távolítanak el, így több energiát igényel, mint az egyszerű gyorsulásérzékelő. A szolgáltatott értéknek mindig van egy nullponti hibája, melyet el kell távolítani. A Google ajánlása egy kalibrációs lépés beiktatása az alkalmazás indulásakor: meg kell kérni a felhasználót, hogy tegye le az eszközt egy vízszintes felületre, így mindhárom tengely mentén kiolvasható az nullponti hiba. Ez a lépés olyan alkalmazásoknál kivitelezhető, melyek csak addig futnak, még a felhasználó közvetlen interakcióban van velük (pl. játékok).

A gravitációs gyorsulás kiszűrésére a Google ajánlása egy szűrővel:

```
public void onSensorChanged(SensorEvent event) {
  // In this example, alpha is calculated as t / (t + dT),
  // where t is the low-pass filter's time-constant and
  // dT is the event delivery rate.
  final float alpha = 0.8;
  gravity[0] = alpha * gravity[0] + (1 - alpha) * event.values[0];
  gravity[1] = alpha * gravity[1] + (1 - alpha) * event.values[1];
  gravity[2] = alpha * gravity[2] + (1 - alpha) * event.values[2];
  linear_acceleration[0] = event.values[0] - gravity[0];
  linear_acceleration[1] = event.values[1] - gravity[1];
  linear_acceleration[2] = event.values[2] - gravity[2];
}
```

2.3 ábra: Google ajánlása a gravitációs gyorsulás kiszűrésére. Forrás: [5]

A legtöbb eszköz rendelkezik gyorsulásmérővel, így jó választás az eszköz mozgásának követésére. Előnyként feltüntethető, hogy körülbelül tízszer kevesebb energiát igényel, mint a szoftveres mozgásérzékelők, hátránya viszont, hogy számítások szükségesek a gravitációs gyorsulás és a zaj szűrésére. A gyorsulásértékeket egy háromelemű vektorral reprezentálja. A vektor minden eleme a koordinátarendszer egy tengely menti gyorsulásértéke.

Adattag	Leírás
SensorEvent.values[0]	Gyorsulás x tengely mentén
SensorEvent.values[1]	Gyorsulás y tengely mentén
SensorEvent.values[2]	Gyorsulás z tengely mentén

2.3 táblázat: A gyorsulásmérő által szolgáltatott adatok struktúrája. Forrás: [5]

3 A rendszer felépítése és megvalósítása

Ahogy már említettem, Lakos Ferenc munkáját folytatom, így egy kész infrastruktúrát kaptam készhez, mely alkalmas volt lokációs adatok hosszú távú megfigyelésére. Ez az elosztott szoftverrendszer szerver és kliens alkalmazásokra volt osztható. A kliens a Human Mobility névre keresztelt Androidos alkalmazás volt, amely az adatgyűjtést végezte és továbbította a központi szervernek. A szerver három részből állt: REST szolgáltatás, adminisztrációs felület és adatbázis. Az adatbázis MySQL alapon nyugodott, még a szerver többi része PHP-ban készült.

A fejlesztés kezdete előtt elbeszélgethettem Ferenccel az általa kivitelezett rendszerről. A beszélgetés során fejlesztői szemmel végighaladtunk a projekt struktúráján és kifejtette a fejlesztés, illetve a tesztelés során felmerülő problémákat. Elmondása szerint az okostelefonokkal volt a legtöbb probléma: a mérések megszakadtak, nem küldtek fel adatok a szervernek, voltak pontatlan adatok. Azt is megtudtam, hogy a problémák többnyire olyan eszközökön jelentkeztek, amik a *Marshmallow* kódnevű 6.0-ás verziószámú operációs rendszert vagy újabbat futtattak. Emellett az is elhangzott, hogy Ferenc újra szerette volna tervezni az alkalmazást az elhangzott problémák javítására és a teljesítmény fokozására, viszont erre már sajnos nem volt ideje. Feladatom több részre tagolódott. Első körben tavasszal a már meglévő infrastruktúrával egy három hetes mérést kellett lebonyolítanom több résztvevővel. A kutatási munkálatokba negyvenkét önkéntes csatlakozott be. A toborzás a közösségi médiákon zajlott, illetve a közvetlen munkatársaimat és barátaimat kértem, hogy csatlakozzanak. Ezen mérés és az eredménye a konzulensem számára volt fontos, illetve számomra is belátást engedett a rendszer valódi működésébe azon felül, ami a beszélgetésen elhangzott.

Ezek után vált tényleges célkitűzéssé az alkalmazás továbbfejlesztése és alkalmassá tétele aktigráfiás jelek mérésére. Sokat vitatott téma volt az eltárolandó adat mivolta és struktúrája, illetve a mintavételezési frekvencia is, mivel a gyorsulásjeleket több Hertz

frekvenciával kell mérni ahhoz, hogy az eszköz mozgását rekonstruálni lehessen. Bár a feladatom elsősorban a bővítés volt, nagy hangsúlyt fektettem a rendszer komponenseinek újratervezésére is biztonsági, méretbeli és teljesítménybeli megfontolások alapján.

A következő fejezetekben kifejtem a rendszer szerkezet egységeit, tekintve az egységes egésztől egészen a komponensek részletezéséig. A felhasznált technológiák és a használatuk motivációja is itt kapott helyet. Nagy hangsúlyt fektetek arra, hogy kiemeljem mi Ferenc érdeme, saját munkámat és az eszközölt változtatásokat.

3.1 Általános felépítés

A rendszer végső állapotában megtartotta a kliens-szerver felépítést, melyet Ferenc alakított ki. A kliens oldali alkalmazás maradt az Android platformon, viszont a szerver réteg teljesen átrendeződött. A REST API-t és a statisztikai feladatokra szánt adminisztrációs felületet egyetlen alkalmazás váltotta fel, amely kielégíti mindkét feladatkört.

A kliens feladatai közé tartozik a fizikai mennyiségek mérése és küldése a központi szerver felé. A folyamatos kommunikáció erőforrásigényes az eszköz számára, így csak bizonyos időközönként küldi fel az adatokat. Ehhez szükséges egy lokális adattár kialakítása is, mely ebben az esetben egy adatbázis.

A szerver réteg alkalmazásának feladatai közé tartozik az adatok fogadása, validálása és központi adatbázisban tárolása a kliensektől, statisztikai számítások végrehajtása és exportálási lehetőség. A velem kapcsolatba került kutatók igényeihez igazodva szöveges fájlt kell exportálnia az alkalmazásnak, amit különböző szoftverek alkalmazásával fel tudnak használni analízisre. Ezen felül ellenőrzés és szemléltetés céljából grafikon exportálását is lehetővé tettem a feldolgozott aktivitásadatok gyors szemrevételezése érdekében.

Mind a kliens, mind a szerver réteg bővült egy úgynevezett hibabejelentő funkcióval. A felhasználó szövegesen be tudja jelenteni az észlelt hibát vagy észrevételt küldhet, melyet a szerveren az adminisztrátorok láthatnak. Ez a funkcionalitás a rendszer futásának hibáit hivatott monitorozni, mely végül a felhasználó felé is ki lett vezetve. Ezen a módon minden futásidejű hiba és felhasználói visszajelzés látszik az adminisztrációs felületen, így a fejlesztés és tesztelés során könnyen tudtam hibát diagnosztizálni.

Kliens	Szerver
Felhasználókezelés	Felhasználókezelés
Fizikai mennyiségek mérés és tárolása	Jogosultságkezelés
Kommunikáció a szerverrel	Adatok fogadása, validálása
Kommunikáció a felhasználóval	Validált adatok tárolása
	Elemző algoritmusok végrehajtása
	Exportálás

3.1 táblázat: A kliens és szerver feladatkörei.

3.2 A szerver

Ahogyan a koncepció eddig ki lett fejtve, körvonalazódott, hogy egy adatbázis alapú rendszert kellett létrehozni, illetve menedzselni.

A PHP-MySQL megvalósítást ASP.NET MVC és MS SQL Server váltotta fel. Több motiváló tényező is húzódik a változtatás hátterében. Ahogyan olvasható Ferenc dokumentációjában, minden *HTTP* kérésnél a fejléc tartalmazta a felhasználónév – jelszó párost, amelyet szerettem volna lecserélni. Továbbá szerepet játszott még a nagy adatmennyiség feldolgozásánál történő teljesítményjavulás, hiszen a *.NET*-es keretrendszer fordításidőben optimalizál a megírt kódon.

A Microsoft Azure szolgáltatást vettem igénybe, ahogyan előző évben Ferenc is tette. Az adatbiztonsági előírásoknak megfelelően egy Észak-Európában található adattárházban található a központi adatbázis. Az adattár tűzfallal védett és az Azure szolgáltatáson kívülről nem elérhető. A fejlesztés során helyi hálózaton (*localhost*) lehetőségem volt tesztelni, így a belső hálózatra kapcsolódott eszközökkel tudtam próbaméréseket végezni.

3.2.1 Adatbázis

Az adatbázis kezelése és menedzselése eddig kézzel történt, illetve némi redundancia is volt benne, így úgy döntöttem, hogy egy objektum-relációs leképző eszközt, az *Entity Framework*-öt[13] (*EF*) használom. Ezzel az eszközzel elérhetem az adatbázist anélkül, hogy egyetlen SQL szkriptet kelljen írnom. A felhasználó és jogosultságkezelés is már kész megoldásként kerül a fejlesztő keze közé, így terhet vett le a vállamról. Ahhoz, hogy ez használható legyen, egy referenciát kell hozzáadni a projekthez, melyben a fejlesztői

környezet nagy segítséget nyújt. Ezek után egy úgynevezett *code first*[10] megközelítési módot alkalmaztam.

A *code first* megközelítés lényege, hogy az adatbázisban eltárolt entitásokat objektum-orientált nyelven – jelen esetben C# – definiálhatjuk, melyet a keretrendszer fog SQL-re fordítani és lefuttatni az adatbázison. Az osztályokban az adattagok nevei mellett különböző megszorításokat (pl. külső kulcs, kötelezőség, értéktartomány) és indexeket definiálhatunk úgynevezett attribútumok által. Ha új entitással bővül a modell, akkor ezt jelezni kell az *EF* felé az *Add-Migration migrationName* paranccsal. Ez a parancs létrehoz egy új fájlt a projekt struktúrában a megadott névvel, ami tartalmazza az utolsó migráció óta történt változtatásokat. Ez legkönnyebben úgy értelmezhető, mint a verziókövetés analógiájában a *commit*. Ha létrejött a migráció, akkor az *Update-Database* paranccsal az adatbázist is felfrissíti a keretrendszer. A következő kódrészlet egy *Activity* táblát fog eredményezni az adatbázisban.

```
public class Activity : Common.EntityBase
{
    [Required]
    [ForeignKey("User")]
    [Index("UserAndSaveTime_Index", 1, IsUnique = true)]
    public string UserId { get; set; }

    [Required]
    [Index("UserAndSaveTime_Index", 2, IsUnique = true)]
    public DateTime SaveTime { get; set; }

    public float X { get; set; }
    public float Y { get; set; }
    public float Z { get; set; }

    public virtual ApplicationUser User { get; set; }
}
```

3.1 ábra: A *code-first* megközelítés szemléltetése.

Mivel minden táblának szükséges egy kulcs megadása, ezt kiszerverztem egy ősosztályba, emellett azt szeretném, ha egy felhasználó egy időbélyeggel csak egy adatsort tölthessen fel, ezért egy egyedi indexszel látom el ezt a két mezőt, melyet attribútummal végzek el. Az utolsó adattag virtuális és az első adattag hivatkozik rá, ami azt eredményezi, hogy a virtuális adattag nem fog megjelenni az adatbázis sémában, csak a kód szintjén lesz látható. Amikor egy adatsort olvas ki a keretrendszer, akkor a megfelelő felhasználó – akinek

az azonosítója megegyezik a rekord *UserId* értékével – is elérhető lesz a *User* adattagon keresztül.

A kódból a következőképpen érhetők el a különböző táblák:

```
// A Locations tábla azon soraival tér vissza, melyeknek UserId oszlop
// értéke megegyezik a userId paraméterrel és a SaveTime oszlop értéke
// a fromDate és toDate értékek között van.
IEnumerable<Location> locations = _context.Locations.Where(
    m => m.UserId == userId &&
         m.SaveTime >= fromDate && m.SaveTime <= toDate);

// A Locations táblába beszúr több adatot. A locations egy
// IEnumerable<Location> típusú változó.
_context.Locations.AddRange(locations);

// Az adatbázisban történt változtatásokat menti egy committal.
_context.SaveChanges();
```

3.2 ábra: Az adatbázis elérésének bemutatása kódból.

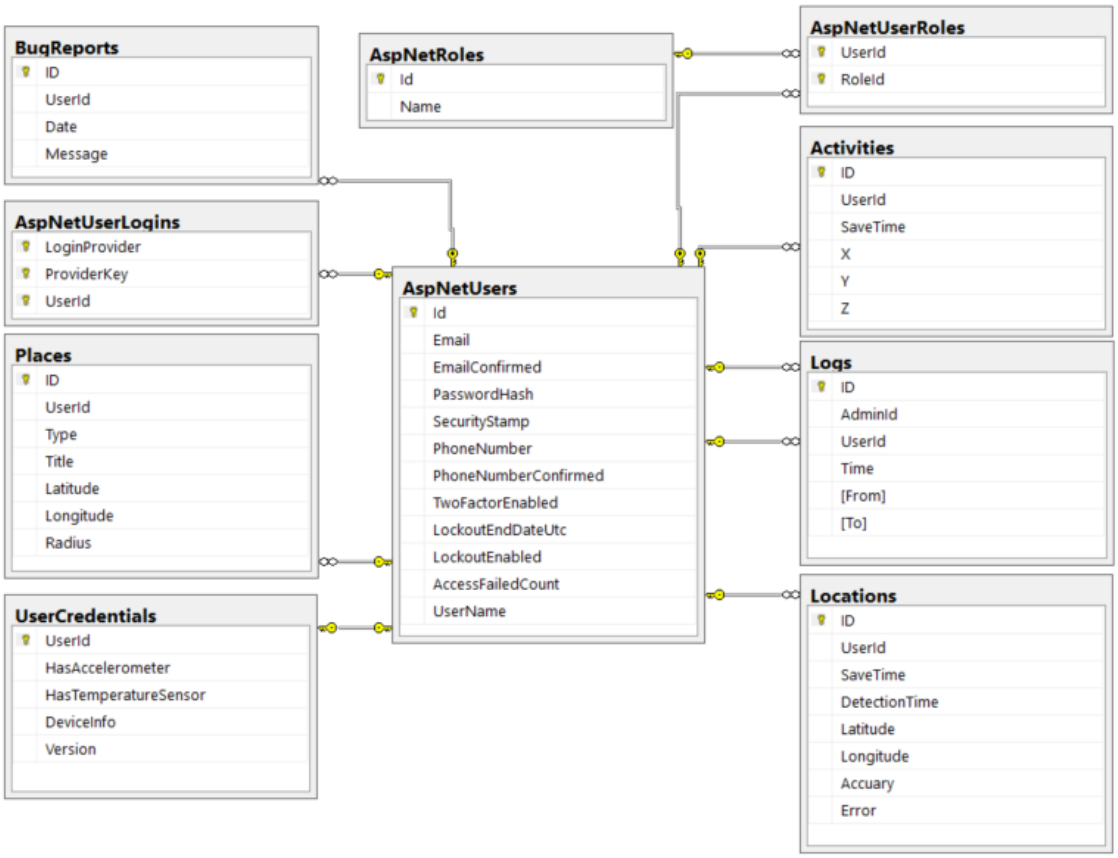
A megfelelő tervezés után a keretrendszer által generált migráció szerkezete az alábbi:

```
public override void Up()
{
    CreateTable(
        "dbo.Activities",
        c => new
            {
                ID = c.Long(nullable: false, identity: true),
                UserId = c.String(nullable: false, maxLength: 128),
                SaveTime = c.DateTime(nullable: false),
                X = c => c.Single(nullable: false),
                Y = c => c.Single(nullable: false),
                Z = c => c.Single(nullable: false),
            })
        .PrimaryKey(t => t.ID)
        .ForeignKey("dbo.AspNetUsers", t => t.UserId,
                                       cascadeDelete: true)
        .Index(t => new { t.UserId, t.SaveTime }, unique: true,
                        name: "Actigraphy_Index");
}
```

3.3 ábra: A generált migráció.

Emellett a rendszer egy *Down* függvényt is generál, amely először eldobja a külső kulcsot, majd az indexet és végül a táblát. Ahhoz, hogy az adatbázishoz kapcsolódni tudjunk, olvasási, illetve írási műveleteket hajtsunk végre, egy *Context* objektumot kell létrehozni.

A megvalósított adatbázis szerkezete megtekinthető SQL Server Management Studio[22]-ban (*SSMS*):



3.4 ábra: Az elkészült adatbázis diagramja.

A keretrendszer előnyei számomra, hogy kódból, *linq*[19] segítségével érhetem el az adatbázist, a külső kulccsal hivatkozott entitásokat is tartalmazza a kézhez kapott objektum, viszont ennek a sokszínűségnek teljesítménybeli ára van. A kliens mintavételezési frekvenciájáról és a nyers adat eltárolásának motivációjáról a *3.3.2* fejezetben fogok beszélni, a szervernek első sorban csak a tárolás a feladata. Ha az eszköz 5 Hz frekvenciával mér, az percenként háromszáz sornyi adat, óránként tizennyolcezer és így tovább. Amikor az eszköz szinkronizálja a saját adatbázisát a központi szerverrel, akkor a szervernek egyszerre nagy mennyiségű adatot kell feldolgoznia. A keretrendszer által elvégzendő plusz feladatokkal ez a mennyiségű adat kezelése már nehézkes, egy lelkes fejlesztő kimérte saját infrastruktúrával[14] különböző módszerekkel és adatmennyiségekkel. A probléma körüljárásaként rátaláltam a C# nyelv saját megoldására, az *SqlBulkCopy*-ra, amely nagy mennyiségű adatsor egyszerre történő lementésére egy belső implementáció.

3.2.2 Adminisztrációs felület és REST szolgáltatás

Ahogyan már az előző fejezetekben említettem, az adminisztrációs felület és a REST szolgáltatás összeolvadt egy alkalmazásba. A .NET keretrendszerben található ASP.NET MVC alkalmazás, amely a böngészőben található weboldalakhoz sablon és külön ASP.NET Web API, amely pedig egy REST szolgáltatás kerete. Ezt a két alkalmazássablont egy egységes egésszé kellett alakítanom ahhoz, hogy egy alkalmazás ellássa mindkét feladatot.

A megvalósítás egy régebbi hitelesítést, az úgynevezett *Basic Authentication*-t használta az egyszerűsége miatt. Ez a fajta hitelesítés lényege, hogy minden *HTTP* kérés fejlécébe belekerül a kliens felhasználóneve és jelszava. A Web API sablon egy számomra elfogadható megoldást kínál erre, amelynek a neve *Token Based Auhentication*[24]-t. A módszer lényege, hogy amikor a felhasználó bejelentkezik a kliens alkalmazáson keresztül, akkor a szerver egy úgynevezett tókent generál, amely a felhasználót hitelesíti. Ez a tóken meghatározott ideig érvényes, melyet a konfigurációban állítani lehet. Ily módon a felhasználónév-jelszó párost csak az első bejelentkezésnél kell átküldeni a hálózaton. A *HTTP* kérések tesztelésére – hasonlóan Ferenchez – a *Postman* alkalmazást használtam, viszont már nem a Chrome bővítményeként, hanem különálló alkalmazásként.

A különböző kérések az alkalmazáshoz *JavaScript Object Notation* (JSON) formátumban érkeznek, melyet feldolgozva szintént egy JSON-t küld vissza válaszként a kliensnek. Ha egy kéréshez hitelesítésre van szükség, akkor feldolgozás törzsében ezt már nem kell ellenőrizni, hiszen a keretrendszer elutasítja mielőtt a kérést kielégítő kódrészlethez érne a futás. Mivel a *code first* megközelítés megköveteli a modellek részletes megalkotását kód szintjén, így egy *HTTP POST* esetében lehetőség van a beérkező modellt validálni. A *POST* kérésnél maradva a függvény paraméterlistájában megadható, hogy milyen objektumot reprezentáló JSON-t vár és a *ModelState.IsValid* tulajdonság ellenőrzésével könnyen meg lehet tudni, hogy a kérés összeállítása hibátlan volt-e, vagy el kell dobni.

```
[HttpPost]
[Route("Bug")]
public IHttpActionResult Bug(List<BugReportViewModel> reports)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState.Values.ToList().ToString());

    ...
}
```

3.5 ábra: Hibabejelentést végző *POST* kérés kezelése és a kérés validálása.

A REST szolgáltatáson felül az adminisztrációs felület csak a kutatók által használt oldal. A rendszer kialakításából adódóan a mérésben résztvevő felhasználók is be tudnak lépni, a kliens és szerver alkalmazások átjárhatók. Jövőbeli tervként szerepelt olyan elképzelés, hogy a mérésben résztvevő egyének a saját adataikat el tudják érni és exportálni, viszont a megvalósítás során még csak az adminisztrátorok férhetnek hozzá a központi adatbázishoz.

A weboldalon megtalálható az elérhetőségem, a felhasználási feltételek és egy tájékoztató arról, hogy mik a felhasználótól begyűjtött adatok. Ezek az információk a publikum számára is elérhetőek. A bejelentkezett adminisztrátoroknak lehetősége van a bejelentett hibák böngészésére, amely tartalmaz minden felhasználó kézzel írt visszajelzését és a rendszer bármely pontján történt kivétel (*exception*) üzenetét.

Ezen felül a lokációs és a gyorsulásmérő által szolgáltatott adatok exportálására van lehetőség feldolgozva, illetve nyersen. Az exportálás során legördülő menüből kiválasztható az exportálni kívánt felhasználó neve, a kívánt időintervallum és az adatfeldolgozás mivolta. A lokációs adatok feldolgozásának módszereit Ferenc már taglalta, így a gyorsulásmérő által szolgáltatott adatsorokat fejtem ki. Az adatsorokat lehet igényelni feldolgozás nélkül, ebben az esetben az adatbázisban található sorok, amelyek a kiválasztott felhasználóhoz tartoznak, módosítás nélkül kerülnek exportálásra.

Ezen felül különböző feldolgozási módszerek is implementálásra kerültek, mint pl. szórás meghatározása, különböző aktivitásértékek számítása. Az eredmény egy szöveges fájl, amelyben pontosvesszővel vannak elválasztva az adattagok egymástól és sorfolytonos. Aktivitás számítása esetén lehetőség van grafikon exportálására is. Az opciók kiválasztásának lehetőségeit egy *enum* tárolja, és abból generálódik a felhasználói felület. Az ASP.NET MVC által használt megjelenítő motort *Razor*[21]-nek hívják, amely lehetővé teszi *HTML* kódba C# nyelv beszúrását. Ily módon érhető el, hogy a felhasználói felületet dinamikusan generálja a rendszer az *enum* elemeiből. Bővítési igény esetén csak az *enum*-ot kell bővíteni, illetve a számítási módszert implementálni, így minimális költséggel bővíthető a rendszer.

3.3 A kliens

A szerver réteg befejezése után a kliens alkalmazás felé fordult a figyelmem. A beszámolók arról árulkodtak, hogy itt lesz a legtöbb dolgom. Ferenc egy jól megtervezett alkalmazást hagyott hátra, amely képes lokációs adatok mérésére viszonylag pontos mintavételezésére.

Feladatom az volt, hogy a lezajlott mérések során felmerülő problémákat orvosoljam, illetve bővítsem az alkalmazást a gyorsulásjelek kezelésének lehetőségével. Ferenc megvalósítása során egy olyan tervezési mintát használt, amely nagy és független rendszerek bővíthetőségét hivatott garantálni. A rendszer kliens oldala csak az Android platformra készült el, és a később kifejtett okok miatt valószínűleg ez így is marad, így ez a tervezési minta több terhet, mint hasznot hozott a fejlesztés során. Ezen felül Ferenc kitért még egy fontos részre, amely a munkám során sok bosszúságot okozott: az operációs rendszernek bármikor joga van leállítani a használt szolgáltatásokat.

A következő fejezetekben bemutatom a bővített alkalmazást, annak különböző komponenseit és a motivációkat. Az alkalmazás felhasználó felülete nagyszerűen van dokumentálva Ferenc szakdolgozatában, mellékesen minimális mennyiségű változtatás történt rajta, így arra nem fogok kitérni.

3.3.1 Mérések előkészítése

A bevezető részben szó esett az Android operációs rendszer által támogatott szenzorokról. Részletezésre került a gyorsulásérzékelő és a lineáris gyorsulásérzékelő. Könnyed megoldás lenne, ha a lineáris gyorsulásérzékelőt használni, viszont ez nem található meg minden eszközben. A fejlesztés során első körben saját okostelefonomat alkalmaztam teszt eszközként, ami hiányában szenved ennek a szenzornak – Huawei P8 Lite, 2015 Q1 – ezért a gravitációs gyorsulást is tartalmazó érzékelőt alkalmaztam.

A lokációs méréssel – nem gyorsan változó fizikai mennyiség volt mérve – ellentétben, a gyorsulásjeleket több Hertz frekvenciájú mintavételezéssel kell mérni. Ezt úgy lehet kivitelezni, hogy egy háttérben futó egység foglalkozik a jelek mérésével és tárolásával.

Az operációs rendszer olyan komponenst biztosít a fejlesztők számára, amellyel kivitelezhetők a hosszan tartó, háttérben futó műveletek is. Ezeket szervízeknek (*Services*[5]) nevezik. Különböző alkalmazáskomponensek indíthatnak szervizeket, melyek akkor is

tovább futnak, amikor a felhasználó másik alkalmazásra vált. A Google ajánlása a komponens használatára: zene lejátszás, hálózati kommunikáció, fájl műveletek, de a mérés lebonyolítására is tökéletesek lesznek. Amikor az alkalmazás elindít egy szervízt (pl. egy *Activity*), akkor az ugyanazon a processzen fog futni, mint maga az indító komponens. Ez gondot jelenthet, hiszen a processz megszűnése egyben a szervíz futásának végét is jelenti. Ezt az eseményt kiváltja, ha kilépnek az alkalmazásból, a felhasználó eltávolítja a legutóbbi alkalmazások listából vagy az operációs rendszer bizonyos okoknál fogva (pl. kevés memória) leállítja az alkalmazást.

Három fajta szervíz létezik:

1. *Foreground* (előtérben futó)

   Olyan műveleteket hajt végre, amelyek értesítik a felhasználót. Az értesítési sávra el kell helyeznie egy értesítést. Akkor is tovább fog futni, amikor a felhasználó már nem lép interakcióba az alkalmazással. Egy szervíz akkor lesz előtérben, ha egy komponens elindítja azt háttérben, majd maga az indított szervíz készít egy értesítést, és elindítja magát a *startForeground* metódushívással.

2. Background (háttérben futó)

   Olyan műveletek végrehajtására szolgál, amelyek nincsenek direkt kapcsolatban a felhasználóval (szerverrel történő szinkronizáció, fájlműveletek)

3. *Bound* (~kötött)

   Akkor lesz ez szervíz *bound*, ha egy komponens meghívja rá a *bindService* metódust. Ez a megoldás egy kliens-szerver interfészt kínál és lehetőséget nyújt a processzek közötti kommunikációra (*IPC – inter process communication*)

Ha egy szervízt külön szálon szeretnénk elindítani, akkor nem közvetlen, hanem *BroadcastReceiver*-en keresztül kell indítani. Az *AndroidManifest.xml* fájlban kell definiálni ezeket a komponenseket, és az *android::process* opcióval meg lehet adni a processzt, amelyen fusson.

```
<receiver android:name=
  "handlers.accelerometer.MeasurementRestartBroadcastReceiver"
  ...
  android:process=":actigraphyMeasurement">
  <intent-filter>
    <action android:name="humanmobility.RESTART_ACCELEROMETER" />
  </intent-filter>
</receiver>
```

3.6 ábra: BroadcastReceiver definiálása az AndroidManifest-ben.

Miután definiálva lett a *manifest* fájlban, a komponens lefut, ha az alkalmazásból meghívják a *sendBroadcast* metódust a *RESTART_ACCELEROMETER* intent segítségével. Amikor az üzenet eléri a komponenst, akkor a fejlesztőre van bízva a döntés, hogy mit csináljon vele. Az üzenet elkapásához az *onReceive* metódust kell felüldefiniálni, ami jelen esetben elindítja a gyorsulásértékek mérését. Mivel a *process* opció definiálva van, hogy az *actigraphyMeasurement* processzen fusson, így az indított szerviz is ezen fog futni. Ezzel a felhasználó kilépése után is tud az alkalmazás folyamatosan mérni, elméletileg.

Az operációs rendszer erőforrások hiányában először a szervízeket veszi célba, hiszen azok nincsenek közvetlen kapcsolatban a felhasználóval. Amikor egy videót néz a felhasználó, vagy épp közösségi médiát böngészi és az eszköz kifut az erőforrásokból, akkor nem azt az alkalmazást fogja leállítani, ami használatban van, hanem a háttérbe futókat. A tárgyalt elemek a megvalósítás szempontjából csak a háttérben futó és az előtérben futó szervízek. Az operációs rendszer először a háttérben futó szervízekhez fog nyúlni, így a megvalósítás során előtérben futó szervízt alkalmaztam.

Ahogy előzőleg említésre került, akkor lesz egy szerviz előtérben futó, ha értesítést helyez az értesítési sávra. Abban az esetben, ha csak a szerviz létrejöttekor egy értesítés jelzi annak futását, akkor a rendszer egy idő múlva inaktívnak érzékeli a komponenst, hiszen az nem kommunikált az elmúlt időszakban a felhasználóval, ily módon a rendszer leállíthatja a futást. Ennek kiküszöbölésére minden percben frissítem az értesítést, hogy a rendszer érzékelje a kommunikációs szándékot és hagyja futni a háttérben futó mérést. Ha már minden percben frissítem az értesítést, akkor jónak láttam itt jelezni a mérés kezdetétől számított időt. Ez visszajelzés értékű, így a kutatásban résztvevő felhasználók tudják, hogy mióta fut a mérés. Mindezek ellenére az operációs rendszernek joga van megállítani a szerviz futását, így megszakítani a kutatási folyamatot.

3.3.2 Gyorsulásjelek mérése

Ahhoz, hogy a gyorsulásjeleket mérni lehessen, egy – az előző fejezetben kifejtett – futó szervízre van szükség. Egy szenzor adatainak eléréséhez fel kell iratkozni az adott szenzorra, majd csak ezután fogja a rendszer az adatokat szállítani. A feliratkozáshoz a rendszer által biztosított *SensorEventListener* interfészt kell megvalósítani. Ez két metódust von maga után, az *onSensorChanged*-et és az *onAccuaryChanged*-et. Az *onSensorChanged* metódus akkor fog lefutni, amikor a szenzor új adatot szolgáltat, az *onAccuaryChanged* pedig ha a szenzor pontossága megváltozik. Ezt úgy is lehet tekinteni – mikrovezérlők

analógiáját használva – mint egy megszakítást. Az itt található kódnak minél gyorsabban le kell futnia. A mintavételezési frekvenciától is függ, hogy mekkora időrés áll rendelkezésre a következő eseményig. Mivel a kutatás céleszközei egyszerű okostelefonok és az operációs rendszer nem garantálja a fix mintavételezést, így két esemény között eltelt időt nem lenne szerencsés teljesen kitölteni, illetve biztosra venni a létezését. Ezen felül az energiafelhasználást is befolyásolja, hogy mi az, ami itt végrehajtódik.

A megvalósításom a *SensorEventListener* interfészt ugyanazon osztályban valósítja meg, mint a *Service* osztályt. Az alábbi kódrészlet szemlélteti a szenzoreseményekre való feliratkozást a szerviz létrejöttekor.

```
@Override
public void onCreate() {
    super.onCreate();
    ...
    sensorManager = (SensorManager) getSystemService(
          Context.SENSOR_SERVICE);
    accelerometer = sensorManager
          .getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

    if (accelerometer != null)
        sensorManager.registerListener(this, accelerometer, sampleRate);
    ...
}
```

3.7 ábra: Gyorsulásérzékelőre történő feliratkozás a szerviz létrejöttekor.

A *registerListener* metódus három paramétert vár: a *SensorEventListener*-t megvalósító objektumot, az érzékelőt, amire fel szeretnénk iratkozni, illetve a mintavételezési frekvenciát, amivel szolgáltassa az eseményeket a rendszer.

A mintavételezési frekvencia megválasztása hosszadalmas folyamat volt és többszöri konzultáció utána rögzült le. Minél magasabb a mintavételezési frekvencia, annál részletesebben állítható vissza az eszköz mozgása és annál kevesebb az adatvesztés. Ennek viszont a hátránya a nagyobb energiafelhasználás. A konzulensem ajánlása 10 Hz volt, viszont így a teszt eszközöm hét óra alatt teljesen lemerült mérsékelt használat mellett. Ezért 5 Hz-re csökkentettem a mintavételezés gyakoriságát, így már tartható volt a tizenkét – tizennégy óra készenléti idő mérsékelt használat mellett. A két frekvenciával végzett méréseim összehasonlítását a *4.* fejezetben fejtem ki.

Miután a *SensorEventListener* megvalósítása regisztrálva lett, az *onSensorChanged* metódusban elérhető a szenzor által szolgáltatott adat:

```
@Override
public void onSensorChanged(SensorEvent event) {
    if (!(event.sensor.getType() == Sensor.TYPE_ACCELEROMETER))
        return;

    float X = event.values[0];
    float Y = event.values[1];
    float Z = event.values[2];
}
```

3.8 ábra: A hasznos adat kinyerése a kapott *SensorEvent* objektumból.

Az operációs rendszer megállítja az adatok szállítását, amikor a felhasználó kikapcsolja az eszköz képernyőjét. Ez a mechanizmus olyan alkalmazásokra lett szabva, amelyek működéséhez bekapcsolt képernyő szükségeltetik (pl. játékok). A kutatás célja a folyamatos mérés biztosítása lehetőleg minden esetben, és nem kérhetjük meg a mérésben résztvevő önkénteseket, hogy állandóan hagyják bekapcsolva az eszközök képernyőjét.

BroadcastReceiver segítségével fel lehet iratkozni olyan rendszerszintű eseményekre, mint a képernyő bekapcsolása, illetve kikapcsolása, viszont ezt nem az *AndroidManifest* állományban lehet megtenni, hanem Java forráskódban, annak biztosítására, hogy az eseményről történő leiratkozás is megtörténjen.

```
private BroadcastReceiver screenOffReceiver = new BroadcastReceiver() {

  @Override
  public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals(Intent.ACTION_SCREEN_OFF)) {
        sensorManager.unregisterListener(SensorService.this);
        sensorManager.registerListener(
          SensorService.this,
          accelerometer,
          sampleRate);
  }
};
```

3.9 ábra: A képernyő lekapcsolásakor végrehajtódó kódrészlet.

Ha a képernyő lekapcsolása pillanatában a szenzor eseményekre újra feliratkozik a megvalósított *SensorEventListener*, akkor kikapcsolt képernyő mellett is mérhetők az érzékelő által szolgáltatott adatok. Ennek hátránya, hogy amikor ez a mechanizmus végrehajtódik, akkor a mintavételezési frekvencia kaotikussá válik egy pár másodpercig, majd visszaáll a beállított értékre.

3.3.3 Gyorsulásjelek tárolása

Ahogyan a szerver résznél már említettem, Ferenc saját megvalósítást alkalmazott az adatbázis menedzselésére. Ahogyan ott is, a kliens részt is lecseréltem egy *ORM* megvalósításra. Az ingyenesen használható *greenrobot* által kínált *GreenDAO*[16] eszközt használtam. Mint az *Entity Framework*, a *GreenDAO* is generálja az adatbázis sémát az osztályokban megalkotott modellekből. Amikor az alkalmazás elindul, akkor egy úgynevezett *DaoSession*-t kell létrehozni, amely ezután használható az alkalmazás teljes életciklusa alatt. Fontos megjegyezni, hogy a *DaoSession*-ból csak egy jöhet létre, így az adatbázis elérésére a *singleton* mintát használtam.

Törekedtem a felelősségi körök teljes elszigetelésére, így, ha az alkalmazás bármelyik komponense szeretne az adatbázissal kommunikációt folytatni, akkor csak a *DatabaseHandler* objektumon keresztül teheti meg azt.

```
// Referencia kérése egy Activityben vagy Serviceben.
databaseHandler = DatabaseHandler.getInstance(this);

// A singleton mód megvalósítása.
public static DatabaseHandler getInstance(final Context context) {
  if (instance == null)
    synchronized (DatabaseHandler.class) {
        if (instance == null)
           instance = new DatabaseHandler(context);
        }
  return instance;
}
```

3.10 ábra: Az adatbázis elérése az alkalmazásból és a singleton mód megvalósítása.

Abban az esetben, ha a háttérben fut a mérés és a felhasználó megnyitja az alkalmazást, akkor több processz egyszerre is használhatja az adatbázist. A *synchronized* kulcsszó segítségével elérhető, hogy különböző processzben ne próbáljanak létrehozni különböző *DaoSession* objektumpéldányokat.

Egy osztály akkor lesz látható a *GreenDAO* számára, ha ellátjuk az *Entity* annotációval. Ezután, ha a project újra fordításra kerül, akkor újra generálódik az adatbázis séma, illetve az azt létrehozó szkriptek. Abban az esetben, ha inkompatibilis változtatás történt az adatbázis sémában, akkor az alkalmazás nem fog frissülni az eszközökön. Ilyenkor a már meglévő táblából a rekordokat el kell dobni és újra létrehozni az adatbázist.

Mivel a *DatabaseHandler* osztálynak van csak privilégiuma az adatbázishoz hozzáférni, így ellenőrizni lehet minden tranzakciót mielőtt az lementésre kerülne. Ilyen például, hogy a hosszúsági és szélességi fok értékek maximum 90° értéket vehetnek fel. A validáláson felül a beszúrásra kapott adatsort módosítani lehet, illetve elindítani másik módosítást is. Ferenc a kliens megvalósítása során a felhasználóval napokra bontott statisztikával tudatta a mérési folyamat állapotát. Ez azt tartalmazta, hogy hány adatsort mentett adott nap az eszköz, ebből hány volt elfogadható pontosságú és hány nem. Ebben a megvalósításban minden megtekintésnél az egész adatbázist szűrte és futásidőben állította össze az eredményt. A futásidőben történő újraszámítást felváltotta egy olyan megoldás, ami statisztikai adatokat tárol napokra bontva a lokációs adatok helyességéről. Ez a megvalósítás tárigény tekintetében ront egy kicsit az alkalmazáson, viszont naponta egy sornyi megengedhető a futásidejű számítások kiváltására.

```
public boolean addLocation(Location location) {
  boolean hasError = false;

  if (location.getAccuracy() > 200)
      location.setError(true);

  if (location.getError())
      hasError = true;

  try {
      locationDao.insert(location);
      addOverviewElement(hasError);
      return true;
  } catch (Exception e) {
      addBugReport(e.getMessage());
      return false;
  }
}
```

3.11 ábra: Lokációs adatsor hozzáadásakor lefutó kódrészlet.

A fenti módosítást bemutatja a 3.11 kódrészlet. Amikor egy új lokációs adatot szeretnénk az adatbázishoz adni, akkor először meg kell nézni annak pontosságát, illetve az előre beállított *Error* tulajdonságát. Amennyiben a tárolni kívánt adat a kutatás szempontjából hibásnak mondható, akkor az *addOverviewElement* metódus lementi, hogy hibás adatot tárolunk, így a felhasználónak visszajelzés gyanánt csak ezt a táblát kell lekérdezni. Abban az esetben, ha bármilyen hiba merülne fel az adat tárolása közben, akkor a *catch* ágban az *addBugReport* metódus lementi a hiba üzenetét, amit a szerverrel történt szinkronizáció után az adminisztrátorok láthatnak.

A mintavételezési frekvencia mellett a másik fontos eldöntendő kérdés a tárolás módszere volt. Az egyik ötlet, ami felmerült, hogy az alkalmazás bizonyos intervallumokra – például percekre – tároljon valamilyen számított aktivitásértéket. Tárhelyigényileg ez megfizethetőbb volt, hiszen egy időbélyeg és egy aktivitásérték alkotott volna egy sort. Ilyen elképzelés szerint percenként összesítve tárolná az eredővektort, ahogyan a (2.2) egyenlet is számolja. Ebben az esetben az adatbázis szerkezete a következő lenne:

Dátum (long)	Összesített gyorsulásérték (float)
201709111745	20.2658
201709111746	15.5468

3.2 ábra: Az adatbázis szerkezete aktivitásérték tárolása esetén.

A tárolási mód esetén egy sor mérete *12 byte*. A módszer hátránya, hogy utólagos feldolgozásra, más számítási módszer futtatására nincs lehetőség. Az utólagos analízis biztosítása érdekében nyers adat tárolása vált szükségessé, amely viszont költségesebb. Ily módon az adatbázis szerkezete:

Dátum (long)	X (float)	Y (float)	Z (float)
201709111745021	1.4689564	8.287456	2.25685
201709111745221	0.9889545	9.598465	1.98568

3.3 ábra: Az adatbázis szerkezete nyers adat tárolása esetén.

Mivel az adatbázisban tárolandó entitásnak kell legyen egyedi azonosítója és az időbélyeg megfelel ennek a szerepnek, így a tengelymenti adatok mellett az időbélyeg kerül elmentésre *long* formátumban. Nyers adat tárolása esetén egy sor mérete *20 byte*. Az 5 Hz-es mintavételezés mellett ez naponta *8.64 MB*.

```
@Entity
public class ActivityLevel {

    @Id(autoincrement = false)
    private Long time;

    private float x;
    private float y;
    private float z;
    ...
}
```

3.12 ábra: Az osztálydefiníció nyers adat tárolása esetén.

Ahhoz, hogy az *ORM* a megfelelő formában generálja le a táblát, különböző annotációkkal kell ellátni. A fenti kódrészlet alapján a *time* adattag lesz a kulcs, mely a szenzor esemény időbélyeg értéke lesz. Mivel az idő csak előre halad és nem történhet ugyanabban a századmásodpercben két esemény, így ez kielégítő választás és egy plusz adattag tárigénye meg lett spórolva. A beillesztett részlet mellett még a Java világában ismert *getter* és *setter* függvények meglétére is támaszkodik a *GreenDAO*, így magában az osztálydefinícióban is különböző viselkedést lehet implementálni.

```
public static void createTable(Database db, boolean ifNotExists) {
  String constraint = ifNotExists? "IF NOT EXISTS ": "";
  db.execSQL("CREATE TABLE " + constraint + "\"ACTIVITY_LEVEL\" (" + //
          "\"TIME\" INTEGER PRIMARY KEY ," + // 0: time
          "\"X\" REAL NOT NULL ," +          // 1: x
          "\"Y\" REAL NOT NULL ," +          // 2: y
          "\"Z\" REAL NOT NULL );");         // 3: z
}
```

3.13 ábra: A 3.14 kódrészlet alapján generált SQL szkript.

3.3.4 Szinkronizáció a szerverrel

A szerverrel történő kommunikáció az *OkHttp*[23] segítségével történt, mint Ferenc megvalósításában. A szerver réteget érintő változtatások miatt szükséges volt a kliens oldalon is változtatni. Mivel a *HTTP* kérések hitelesítésénél a felhasználónév – jelszó páros felváltotta a *token* alapú hitelesítés, így indokolatlan lett a jelszó tárolása a kliens eszközökön. Emellett a kérések összeállítása során a sztring konkatenáció bizonyos méretek felett lassúvá vált.

```
String dataJson = "[";
    for (LocationInfo locationInfo : locationInfoList) {
        if (!dataJson.equals("[")) {
            dataJson += ",";
        }
        dataJson += locationInfo.toJson();
    }
dataJson += "]";
```

3.14 ábra: Konkatenációval összeállított JSON.

Az erőforrások kímélése érdekében a Google által fejlesztett *GSON*[15] csomagot használtam. Annak érdekében fejlesztették, hogy Java objektumok költséghatékonyan konvertálhatók legyenek a *JSON*-beli reprezentációjukra. A használatához fel kell paraméterezni a konvertálandó osztályt a *GSON* által definiált annotációkkal.

```
public class Location {
    @Expose
    @SerializedName("Accuary")
    private float accuracy;

    @Expose
    @SerializedName("Error")
    private boolean error;

    ...
}
```

3.15 ábra: *GSON* annotációk bemutatása.

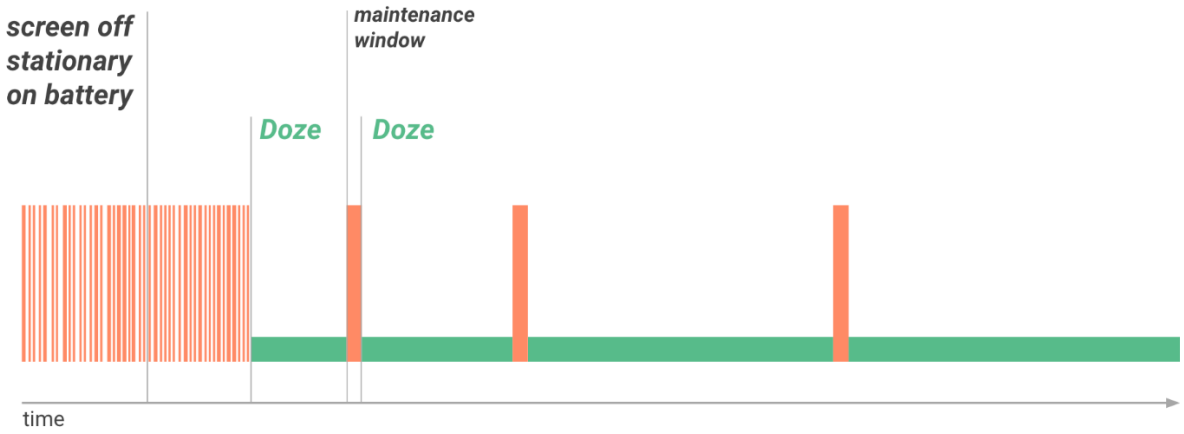
A *SerializedName* annotációval megadható, hogy a *JSON* formátumban milyen neve legyen az adott mezőnek, az *Expose* annotációval pedig elérhetjük, hogy az adattag és értéke szerepeljen a *JSON* objektumban. Ez külön konfigurációt igényel, viszont szükség volt rá, hiszen vannak olyan adattagok, melyek az alkalmazás rendeltetésszerű működéséhez elengedhetetlenek, viszont az adatfeldolgozás szempontjából feleslegesek, így a központi szerverrel nem szükséges szinkronizálni őket.

Saját eszközömön elvégeztem egy tesztet, mely egy *Long* és három *Float* típusú változót tartalmazó objektumhalmazt szerializál. A teszt során egy *100.000* elemből álló tömb volt a bemenet és egy *JSON* objektum a kimenet. Sztring konkatenáció (3.16 ábra) esetén a folyamat 1 perc 19 másodpercig tartott, a *GSON* csomagot használva viszont 14 másodpercig. Ezen felül a gyorsulásjelek szinkronizációja során a felgyülemlett sorokat nem egyszerre, hanem több kisebb csomagban küldi el az alkalmazás. Ezzel a módszerrel az alkalmazás memóriafelhasználása korlátozódik, hiszen nem egyszerre kell beolvasni a több tízezer sornyi adatot a memóriába.

3.3.5 Felmerülő problémák és kezelésük (Android 6.0 vagy újabb)

A Doze mód[5]-ot az Android 6.0 újításaként ismerhettük meg és az eszköz akkumulátoridejét hivatott növelni. Amikor Ferenc szakdolgozata készült, az Android 6.0 nem volt még elterjedt, csak 4.6%-át birtokolta a piacnak, viszont már akkor is foglalkozni kellett ezzel. Mára ez a szám 50%-ra emelkedett az azóta megjelent 7.0, 7.1 és 8.0 verziókat is beleszámolva. A mód korlátozza az alkalmazások processzorhoz való hozzáférését és megakadályozza a hálózat használatát. Ezen felül a háttérben futó alkalmazásoknak a futását leállítja, illetve a riasztásokat (*AlarmManager*) letiltja. Ha az eszköz nincs töltőhöz

csatlakoztatva és nincs használva hosszabb ideig, akkor belép a Doze módba. Bizonyos időközönként az operációs rendszer kilép a módból, hogy az alkalmazások befejezzék a félbehagyott munkáikat. Ezt karbantartási ablaknak (*maintenance window*) nevezi a szakirodalom.



3.16 ábra: Doze mód és a karbantartási ablakok. Forrás: [5]

Minden karbantartási ablak végén a rendszer újra belép a Doze módba, felfüggesztve minden tevékenységet. Az idő múlásával ezeket az ablakokat a rendszer egyre kevesebbszer ütemezi. Abban az esetben, ha a felhasználó megmozdítja az eszközt, bekapcsolja a képernyőt vagy töltőre csatlakoztatja, a rendszer kilép a módból és az alkalmazások visszatérnek a normál futásukhoz.

A Doze mód mellett még egy úgynevezett készenléti állapotba is kerülhet az alkalmazás, amikor is az eszköz ébren van, de a rendszer úgy érzékeli, hogy az alkalmazás már használaton kívül van. Ebben az esetben a futó alkalmazásra ugyanolyan megszorítások vonatkoznak, mint a Doze mód esetében. E két állapot meggátolhatja az alkalmazás futását, így a mintavételezés pontosságát rontja és meggátolhatja az egész mérési folyamat hasznosságát. Az operációs rendszer rendelkezik úgynevezett *Battery Optimizations whitelist*[5] lehetőséggel, melyhez az alkalmazást hozzáadva nem vonatkoznak rá a *Doze* mód megszorításai. Emellett a fejlesztői dokumentációban található egy leírás, hogyan kell a rendszert ébren tartani anélkül, hogy az alkalmazástól megvonná a futási jogokat. Több különböző módszert ismertet, amelyek közül az első az eszköz képernyőjének bekapcsolva tartása, ezt a játékok, médialejátszó és navigációs alkalmazások használják. Mivel a kutatás mivolta megköveteli a több órán, sőt napon át történő folyamatos mérést, így ez a lehetőség nem jöhet szóba. A második módszer az *AlarmManager* osztályt bővíti. Amikor egy feladat

ütemezése megtörténik, akkor lehetőség van megkérni a rendszert, hogy alvó állapotból felkelve elvégezze az ütemezett feladatot. Ez a *setAndAllowWhileIdle* metódushívással tehető meg, a rendszer tíz másodperc futásidőt biztosít az alkalmazásnak, mielőtt visszatér a *Doze* módba. Normál állapotban a rendszer maximum percenkénti ütemezést engedélyez, viszont ilyen állapotban ez 15 percre bővül. A harmadik módszer a rendszer által biztosított *PowerManager* osztály egy funkcióját használja, a *WakeLock*-ot. Ez lehetővé teszi bizonyos időkorlát erejéig, hogy az alkalmazás kontrollálja az eszköz állapotát. A megoldás használatához engedélyt kell kérni a felhasználótól, majd a *WakeLock*-ot inicializálni a következő módon:

```
PowerManager manager = (PowerManager) getSystemService(POWER_SERVICE);
wakeLock =
    manager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "ACTIGRAPHY");
wakeLock.acquire();
...
wakeLock.release();
```

3.17 ábra: *WakeLock* inicializálása és felszabadítása.

A fenti kódrészletben az *acquire* és a *release* metódushívások között a rendszer ébren fog maradni. Az utolsó megoldás a *WakefulBroadcastReceiver*-ek használata. Ez egy speciális *BroadcastReceiver*, mely automatikusan készít egy *PARTIAL_WAKE_LOCK*-ot az alkalmazás számára. Amikor a *receiver* elindítja a szervizt, akkor a zárolás életbe lép és addig ébren fogja tartani a rendszert, még a szerviz a munkája végeztével megáll.

A fenti megoldásokat implementáltam a mérést végző szervízbe, így az eszköz nem kerül bele a *Doze* módba. Ez az állapot csak egy meghatározott ideig tartható, utána a rendszer ignorálja a *WAKE_LOCK*-ot, amely a zárolást kérvényezte és alvó állapotba kerül. Méréseket végeztem, hogy a rendszer meddig hajlandó tartani a zárolást és a saját eszközömön ez átlagosan 1.5 óra volt. Az operációs rendszer megállíthatja a szerviz futását bármikor[5], amikor erőforráshiányban szenved. Amikor a szerviz indul, a prioritása nagy, viszont a hosszan futó szervizek esetében a rendszer a prioritást egyre csökkenti, így jelölve azt a megállításra. A dokumentáció szerint amikor a rendszer erőforrásai felszabadulnak, akkor újra indítja a szerviz futását, viszont ez a tesztelésem alatt nem történt meg. Amikor a rendszer megöli a szerviz futását, az hasonló módon történik, mint amikor a felhasználó „*Force-Stop”* gombra kattintva megállítja azt. Az ilyen módon megállított futást nem lehet detektálni, ellentétben az alkalmazásból történő egyszerű kilépéssel, amikor is a megfelelő *callback* metódusok meghívódnak. Sajnálatos módon a gyártó cégek, melyek módosított

rendszerrel látják el eszközeiket, felülírhatják saját megoldásaikkal a rendszer eredeti optimizációit. Példának okáért az általam birtokolt *Huawei P8 Lite* eszköz rendelkezik úgynevezett védett alkalmazások (*Protected apps*) energiatakarékossági móddal. Ez a módszer úgy takarít meg energiát, hogy a képernyő kikapcsolása után az alkalmazások futását megállítja, felülírva az operációs rendszer algoritmusát. Manuálisan lehet hozzáadni alkalmazásokat. Minden gyártó módosított rendszerét kódban kezelni szinte lehetetlen.

3.3.6 Adatvédelem és felhasználási feltételek

A mérések során az alkalmazás a gyorsulásjelek mérése mellett az eszköz pontos pozícióját is méri, amely során jogi és adatvédelmi kérdések merülhetnek fel. A felhasználóknak módjukban áll megtekinteni a felhasználási feltételeket a bejelentkezés felületén, illetve a belépéssel elfogadják azt. Az adatokhoz kizárólag csak a kutatók férnek hozzá elemzési célból, továbbá ezen adatokat nem adják tovább harmadik félnek semmilyen körülmények között. A kutatók az elemzésre szánt adatokat név nélkül kapják kézhez.

4 Adatsorok kezelése és elemzése

Munkám célja a méréseket végző kliens és az adattárolásra szolgáló szerver bővítése volt. Az idei év tavaszán egy három hetes mérést kellett menedzselnem Ferenc megvalósításával, amiben több, mint negyven önkéntes csatlakozott be. A mérés során az alkalmazás csak lokációs adatokat gyűjtött, a résztvevők felénél hosszabb, elemzéshez már felhasználható adatsorokat generált, míg *öt* önkéntes teljes adatsorral szolgáltatott. Az adathalmaz hibáinak kezelését Ferenc kifejtette a szakdolgozatában[18], így a fejezetben csak a gyorsulásjelekre fókuszálok. Továbbá az eredmények mélyebb kiértékelését konzulensem és *Antal András* hallgató végzik.

Az adatgyűjtés lefolyása után kezdődhetett az aktigráfiás adatgyűjtés folyamatának tervezése és kivitelezése. A *2.* fejezetben több aktivitásdefiníció is fellelhető, így a fejlesztés során a hangsúlyt a nyers adatok tárolása kapta, mely később a kifejtett módszerekkel feldolgozható és összehasonlítható.

Az idei év őszén egy újabb mérés indult tizenegy önkéntessel. A folyamat során az alkalmazás együtt mérte a lokációs adatokat a gyorsulásjelekkel. Ahogyan az előző mérés során, mérőszámokat percenként szerettünk volna, tehát lokációs adatot és aktivitást. Az utóbbihoz sűrűbben mintavételezett gyorsulásjel szükséges, ez *5 Hz* volt a korlátok miatt. Az első mérési ciklustól eltérően a folyamatos eszközhasználat mellett drasztikusan nőtt az akkumulátor merítése is, így az önkéntesek egy részénél már nem volt tartható az egy teljes napnyi folyamatos mérés sem. Az akkumulátor merítése mellett az operációs rendszer optimalizációi is gondot okozott, hiszen a folyamatos háttérben futást – ami mintavételezetten méri a gyorsulásértékeket – megállíthatja a *3.3.5* fejezetben kifejtettek alapján. Az eltelt bő fél évben az operációs rendszer verzióeloszlása is megváltozott, ennél fogva a piacon lévő eszközök csupán fele képes arra, hogy megközelítően pontos mintavételezés mellett mérjen hosszú időn át. Ennek okán az önkéntesek újabb csoportja esett ki, így maradt néhány eszköz, melyek pár óránál tovább voltak képesek az alkalmazást futtatni. Azon adatsorokon, melyek hosszabb időt ölelnek fel, felszínes vizsgálatot végezve látható, hogy a gyorsulásmérő fix mintavételezése nem pontosan kivitelezett. Vannak olyan intervallumok, melyekben a kitűzött mintavételezési frekvencia többszörösével mért az alkalmazás, és vannak olyanok is, melyekben egyáltalán nem (*3.3.5 fejezet*). A lokációs adatgyűjtés eredményei hasonlóak, hiszen az alkalmazás működését befolyásolta a gyorsulásmérés is. Ez megkérdőjelezheti azt, hogy az okostelefonos komplex aktivitásvizsgálat alkalmas-e kutatási szintű pontosság és folytonosság előállítására, amely

a célkitűzésben szerepel. A következő fejezetekben bemutatásra kerül az aktivitásszámítás módszereinek összehasonlítása, majd a mintavételezési frekvencia vizsgálata és validációja, majd az egy kiválasztott adatsorra, végül egy hosszabb adatsor vizsgálata.

4.1 Számítási módszerek összehasonlítása

Az aktivitás fogalma nincs egységesen meghatározva. Egy fogalom kialakulhat a fejünkben arról, hogy mi is az aktivitás, viszont a formális matematikai leírás már bonyolultabb. A *2.2.2* fejezetben több kutatás szemléletmódja bemutatásra került, kutatásonként különböző számítási módszerrel. Azon tanulmányok, melyek az alvási ciklusokat vizsgálták[2][9][12], nem tértek ki arra, hogy mely fizikai mennyiségekre végeztek vizsgálatot. Példának okáért a *TAT (2.2.2)* módszer egy főtengelyt használ, viszont a telefon orientációja változhat, így a három tengely gyorsulását külön kellene vizsgálni és összegezni vagy az eredő nagyságát? Más tanulmányok[3][4][7][17] az eredő vektor nagyságát használják fel az *AC* számításánál és a tengelyenként mért gyorsulás szórását külön – külön az *AI* számításánál. Ebből az inkonzisztenciából kifolyólag a számítási módszerek összehasonlítása a különböző fizikai mennyiségekre számított aktivitásértékkel történik. A *4.1 táblázatban* található értékek egy véletlenszerűen kiválasztott, egy percet felölelő adathalmazból számítottak a megadott algoritmus mellett. A módszerek külön tengelyenként (*X, Y, Z*) és az eredő vektorral is vizsgálva lettek. A *sum(X, Y, Z)* sor a tengelyeken számított aktivitásérték összegét tartalmazza.

/	ZCM	TAT	Integrál	(EE)AC	AI
X	0.52	0.91	1.07	/	11.76
Y	0.54	0.98	2.50	/	16.76
Z	0.55	0.96	1.89	/	13.85
sum(X, Y, Z)	**1.61**	**2.73**	**5.46**	/	**42.37**
Eredő vektor	/	1	/	**3.76**	21.69

4.1 táblázat: Egy véletlenszerűen választott másodperc pillanatképei.

A vastagon kiemelt értékeknek megfelelő módszerek segítségével hosszabb adatsorokat is megvizsgáltam a *4.2* és *4.3* fejezetekben. Az, hogy ezen módszerek eredményei mennyire korrelálnak egymással, azt a *4.3* fejezet fejti ki.

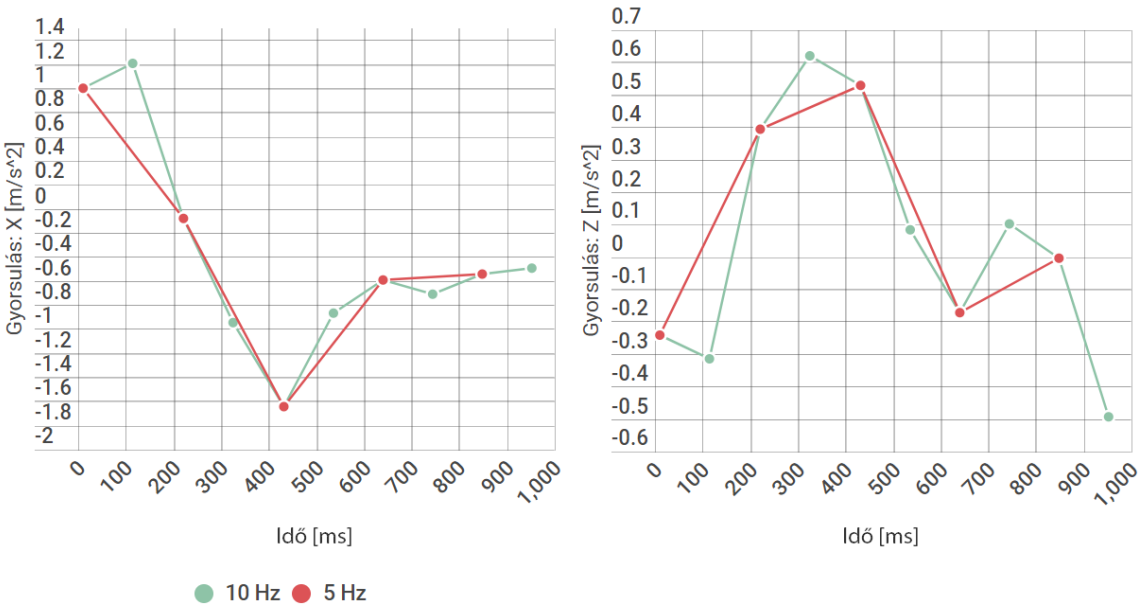
4.2 Mintavételezési frekvencia vizsgálata

A mintavételezési frekvencia korlátozása 5 Hz-re felvetette azt a kérdést, hogy ilyen mintavételezés mellett mennyi információ veszik el. Ezen felül az a fejlesztői dokumentációban[5] fellelhető, hogy a beállított mintavételezési idő az egy javasolt érték, melyet az operációs rendszer az állapotától függően módosíthat. Ennek tudatában próbaméréseket végeztem egy eszközön *10 Hz*-es mintavételezés mellett, majd véletlenszerűen kiválasztottam egy másodpercet. A kiválasztott szakasz tíz pillanatképet tartalmazott az abban a másodpercben történt mozgásból. Abban az esetben, ha ugyanezt a mozgást *5 Hz* frekvenciával mértem volna, akkor ezeknek a pillanatképeknek pontosan a fele állna rendelkezésemre. Ily módon a *10 Hz*-es mintavételezett adatsorból minden másodikat megtartva modelleztem a lassabb mintavételezést. Ha a két adathalmazt tengelyenként közös grafikonon szemléljük láthatóvá válik az adatvesztés mértéke, melyet a *4.1 – 4.2* ábra szemléltet.

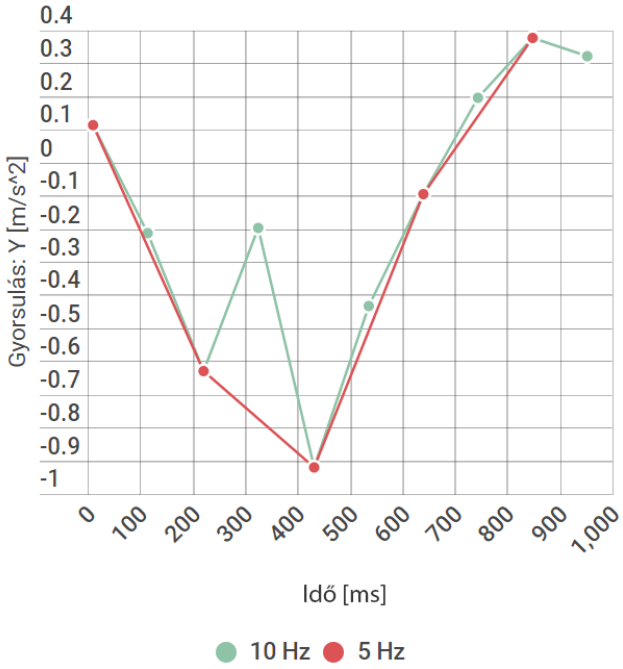
Időbélyeg	Gyorsulásv.: X	Gyorsulásv.: Y	Gyorsulásv.: Z
18:37:00.011	0.9993658	0.1127143	-0.2425938
18:37:00.115	1.202655	-0.2149243	-0.3139343
18:37:00.221	-0.08391881	-0.6295834	0.3917332
18:37:00.326	-0.9497336	-0.198571	0.6184826
18:37:00.431	-1.642385	-0.9215963	0.5274744
18:37:00.535	-0.867161	-0.4321811	0.08419514
18:37:00.641	-0.5956622	-0.09513044	-0.1723614
18:37:00.745	-0.716248	0.1963027	0.1018286
18:37:00.849	-0.5403095	0.3749678	-0.005706787
18:37:00.952	-0.4976253	0.3217669	-0.4948978

4.2 táblázat: Egy véletlenszerűen választott másodperc pillanatképei.

A *4.2* táblázat időbélyeg oszlopát megfigyelve látható a nem teljesen pontosan kivitelezett mintavételezése az operációs rendszernek. Pontos mintavételezés mellett a másodperc első időbélyege után a következő eseménynek *18:37:00.111*-kor kellene bekövetkeznie, majd *18:37:00.211*-kor és így tovább. Szemmel látható, hogy az esemény mindig késik, két esemény között eltelt idő átlagosan *104.55 ms* az elvárt *100.00 ms* helyett, a relatív hiba pedig *4.55%* a *4.1* táblázatban szemléltetett adathalmaz esetében. Az ábrákból látható, hogy bizonyos mértékű információvesztés fellelhető gyors mozgások esetén, külön kiemelve az Y tengely menti értékeket (4.2 ábra).



4.1 ábra: Mintavételezési frekvenciák összehasonlítása X és Z tengely szerint.



4.2 ábra: Mintavételezési frekvenciák összehasonlítása Y tengely szerint.

A szemléltetés segíti megérteni az információvesztés miértjét, viszont jelen kutatásban a mérvadó mérőszámok az aktivitásértékek, így az előző fejezetben számított értékek alapján három kiválasztott indexszel vizsgálom a mintavételezési frekvencia hatását a számított aktivitásértékre. A *4.2 táblázatban* ismertetett, majd a *4.1* és *4.2* ábrán bemutatott másodperc szemléltetéshez elégséges, viszont a további összehasonlításhoz öt másodpercnyi adatot használok fel.

Számítási módszer	10 Hz (aktivitás)	5 Hz (aktivitás)	Relatív hiba (%)
ZCM	1.31	1.75	33.59%
TAT	2.82	2.8	0.71%
Integrál	4.61	4.59	0.44%
EEAC	3.18	3.14	1.27%

4.3 táblázat: Aktivitásértékek különböző mintavételezés mellett.

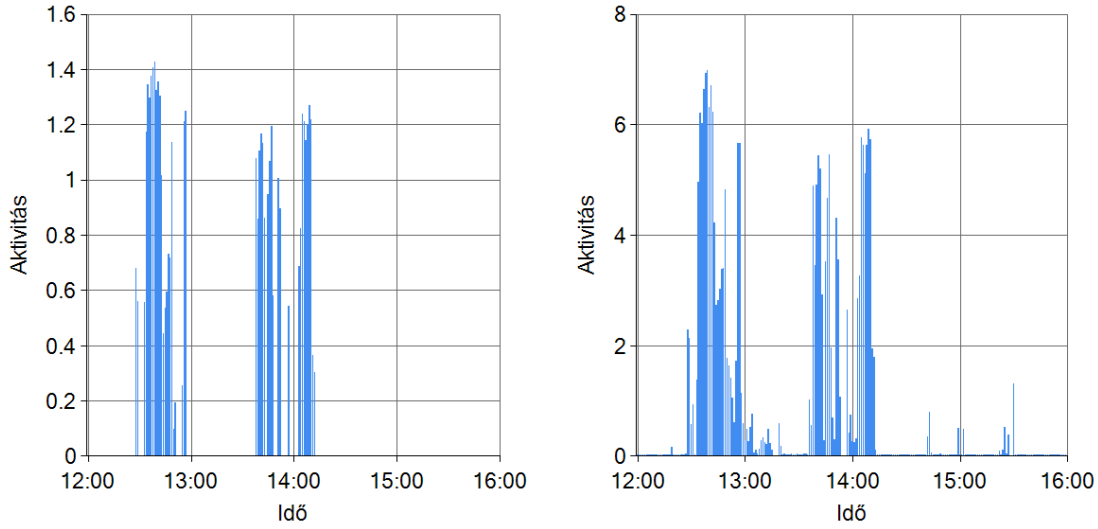
A hiba lassú mozgások esetén csökken, viszont gyors mozgások esetén növekedhet. Látható, hogy a különböző számítási módszerek más és más aktivitásértéket eredményeztek és különbözően függenek a pillanatképek hiányától.

4.4 Időbeli elemzés

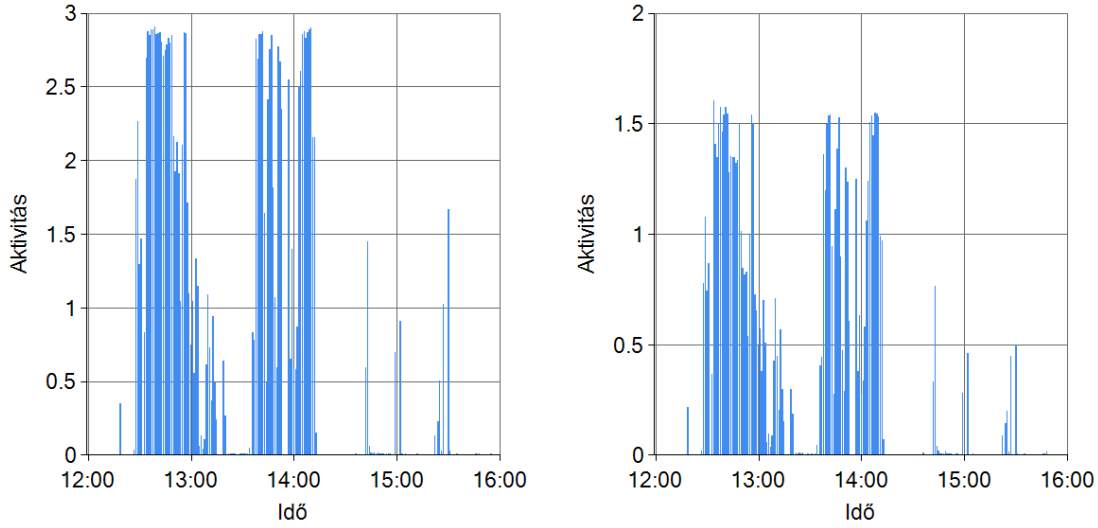
A mai világban az okostelefonok úgymond hozzánk nőttek, mindig a közelünkben vannak a zsebünkben, táskánkban, az asztalunkon. Ferenc kutatása arra a közelségre épített, ami az eszköz és tulajdonosa között van. A lokációs mérések esetében ez a közelség elegendő, hiszen egy piacon levő telefonnak a GPS modulja nem centiméter pontosságú, így a mért érték szempontjából mindegy, hogy hol helyezkedik el az eszköz. Az eddigi kutatás-fejlesztés arra a feltételezésre épült, hogy az ember felkel, leveszi a töltőről az eszközt, a napi teendői elvégzése közben használja, kiteszi az asztalra, sportolás közben az öltözőben hagyja, majd a nap végeztével visszateszi azt a töltőre. Ez a lokációs adatok mérése szempontjából közel tökéletes volt, hiszen ugyanazt a pozíciót fogja a GPS modul szolgáltani abban esetben is, ha az eszköz a zsebben van, és akkor is, ha az asztalon pár tíz centiméterre attól.

A gyorsulásértékek vizsgálatakor az eszköz közelsége nem elegendő a megfelelő következtetések levonásához. A jelenlegi kutatás mérési folyamata nagyban függ attól, hogy a résztvevő önkéntesek hogyan használják eszközeiket. Az elképzelt, laboratóriumi mérés akkor történik, amikor az önkéntes a telefonját egész álló nap egy helyen, pl. a zsebében tartja. Ekkor az eszköz azt aktivitást méri, ami a feladatom alapvető céljaként ki lett tűzve. Abban a pillanatban, ahogy az eszköz ettől a pozíciótól eltávolodik, az adathalmaz korrupt lesz, hiszen nem az egyén aktivitását méri, hanem nyugalmi helyzetben van. Az eszköz a méréssel járó terhelések hatására mérsékeltebb akkumulátorhasználatnak van kitéve, mely azt eredményezi, hogy minden este fel kell tenni tölteni, az éjszakai aktivitás mérését így lehetetlenné téve.

A fejezetben egy négy órás intervallum aktivitáseloszlása kerül bemutatásra, különböző számítási módszerek mellett a szemléltetés kedvéért. A *4.3 – 4.4* ábra bemutatja az a különböző módszerek által kiértékelt jelalakokat.



4.3 ábra: Aktivitáseloszlás négy órányi adaton. Balról *AI*, jobbról *Int* módszer.



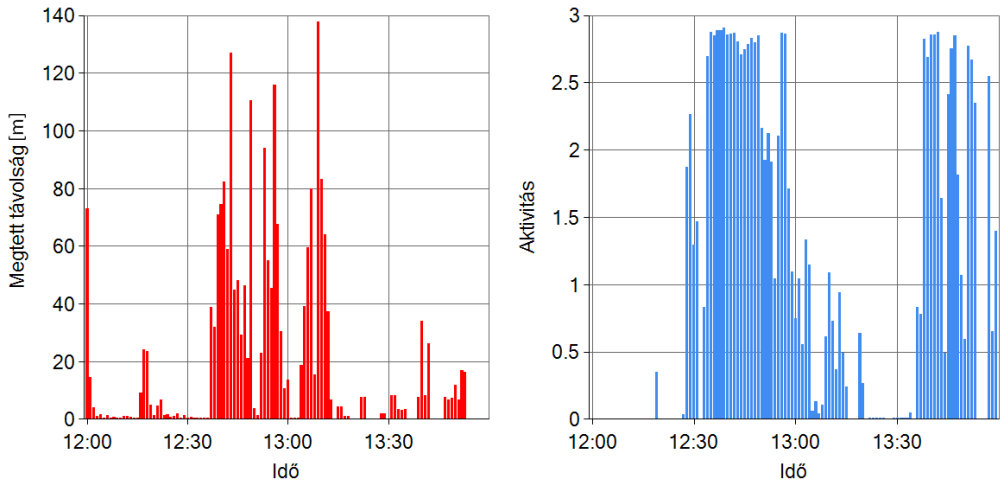
4.4 ábra: Aktivitáseloszlás négy órányi adaton. Balról *TAT*, jobbról *ZCM* módszer.

Az ábrákból látható, hogy a különböző módszerek különböző aktivitásértéket határoztak meg egy – egy percre, ami várható jelenség volt, a jelalak viszont hasonló. Ennek függvényében az adathalmazból számított aktivitásértékekre lineáris korrelációt meghatározva láthatóvá válik, hogy milyen mértékben függenek egymástól. A korreláció számításának eredménye 1, ha a két jel között erős pozitív lineáris összefüggés van, -1, ha erős negatív lineáris összefüggés van és 0, ha nincs a jelek között lineáris kapcsolat.

	ZCM	TAT	Integrál	AI
ZCM	1	0.98	0.88	0.77
TAT	0.98	1	0.87	0.74
Integrál	0.88	0.87	1	0.96
AI	0.77	0.74	0.96	1

4.4 táblázat: Korrelációk különböző számítási módok között.

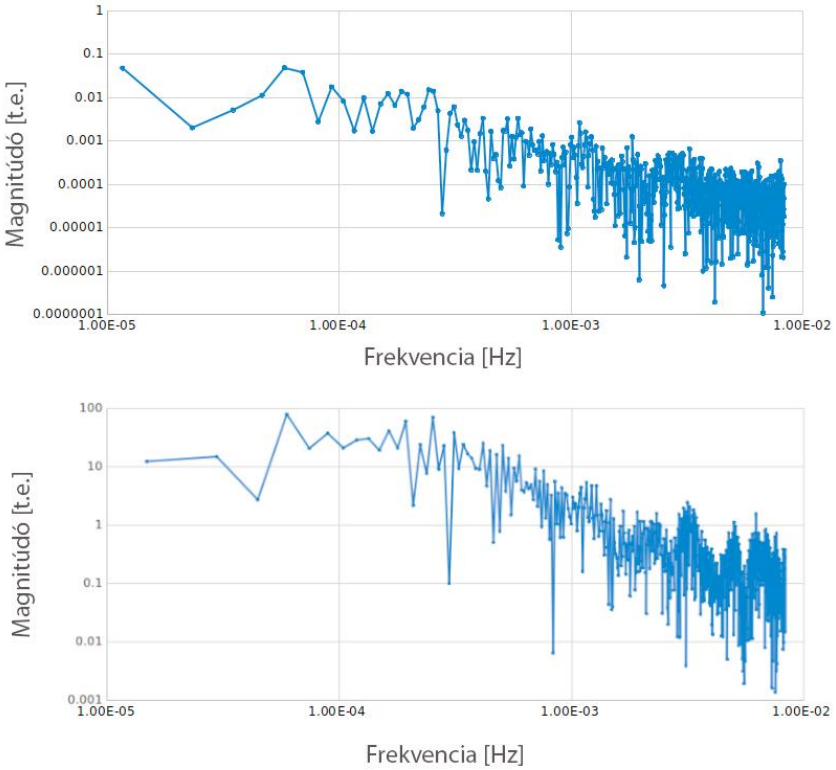
A *4.4* táblázatból látható, hogy a különböző számítási módszerek eredményei nem csak vizuálisan hasonlóak, erős lineáris összefüggés is van közöttük. Mivel ilyen mértékben korrelálnak a számítási módszerek, ezért a további vizsgálatok csak a *TAT* módszert használják. A fejezet eddig csak a gyorsulásszenzor által szolgáltatott adatok feldolgozását taglalta, viszont az alkalmazás lokációs adatokat is mért. A lokációs adatok feldolgozásával Ferenc foglalkozott, azóta pedig konzulensem és Antal András hallgató foglalkozik tüzetesebben. A fejezet csak demonstrációt tartalmaz, hogy milyen az elmozdulás és az aktivitás időbeli alakja közötti kapcsolat.



4.5 ábra: Balról a percenkénti megtett távolság, jobbról pedig az aktivitás látható.

A *4.5* ábrán látható a percenként mért GPS koordináták közötti távolság (melyet *András* bocsátott a rendelkezésemre), jobbról pedig az egyén aktivitása erre az időszakra. Látható, hogy a megtett távolság és az aktivitás közötti kapcsolat gyenge, amely értelmezhető, hiszen járművel való mozgás során a csuklónk mozdulatlan maradhat, illetve egyhelyben is végezhetünk intenzív kézmozgást. Hosszabb periódust tekintve mégis teremthet kapcsolatot a két jel között a napi mozgásmintázatunk, mely a jelek teljesítménysűrűség-spektruma alapján vizsgálható. A kutatás célja, melyhez a dolgozat

tárgyát képező szoftvert fejlesztettem, éppen ez volt. A *4.6* ábrán látható, hogy a két jel frekvenciatartománybeli képe igen hasonló, továbbá színes zaj jelenléte figyelhető meg. Ez az újszerű eredmény témavezetőm és András érdeme, mely előremutató a mozgásmintázatok kutatásában.



4.6 ábra: Felül az aktivitás, alul pedig az elmozdulások teljesítménysűrűség-spektruma.

A kutatás során kevés önkéntes eszköze produkált megfelelő, elemezhető adathalmazt, mely a kiértékelés használhatóságának rovására ment. Nehéz feladatnak bizonyult olyan hosszú ideig összefüggő adatsorokat találni, melyeknél a lokációs és a gyorsulásmérő adatai is rendeltetésszerűen mentésre kerültek. Azokon az eszközökön, melyeken Android *6.0*, vagy újabb operációs rendszer fut, legtöbb esetben sikeres a két fizikai mennyiség egyidejű mérése rövid távon, percekben, maximum órákban gondolkodva. Azon eszközök, melyek az említett verziószámnál régebbi operációs rendszert futtatnak, a mérés rendeltetésszerűen futhatott abban az esetben, ha az eszköz hardvere ezt megengedte. Abban az esetben, ha az eszköz kifut az erőforrásokból, először a szerverrel történő szinkronizációt, majd a GPS modult és végül a gyorsulásmérő adatmentését állítja le.

Fontos megjegyezni, hogy a fenti elemzések egy egyén mozgására készültek, amíg megbízható következtetéseket csak nagyszámú egyének vizsgálata esetén lehet levonni.

5 Összefoglalás és további lehetőségek

Célom egy meglévő szoftveres rendszer optimalizálása és bővítése volt, hogy lehetővé tegye a GPS adatokon túl nyers gyorsulásértékek mérését és tárolását meghatározott mintavételezési frekvenciával későbbi analízis céljából. Feladatom szempontjából fontos volt a GPS és a gyorsulásjelek egyenletes mérése hosszú távon, így a kettőt egymás függvényében is vizsgálhattuk.

Az Androidos kliens egyes részeit átírtam, még másokat érintetlenül hagytam azon folyamat alatt, még az alkalmazás alkalmassá vált a gyorsulásjelek mérésére és tárolására. Sok bosszúság ért fejlesztés közben az operációs rendszer új energiatakarékos módja, a *Doze* mód és a háttérben futó alkalmazásokra alkalmazott *készenléti állapot* miatt, hiszen ezek akadályozták a mérések futását. Az akkumulátor merítését a mintavételezési frekvencia korlátozásával sikerült mérsékelni. A rendszert támogató központi szervert teljesen lecseréltem biztonsági és teljesítménybeli motivációk alapján.

A szervert és a központi adatbázist felkészítettem nagy mennyiségű gyorsulásjel tárolására. Az Android operációs rendszerrel ellátott eszközök valósidejű mivoltának hiányában a fix mintavételezés nem bizonyult megvalósíthatónak. Az eszközök a felhasználót részesítik előnyben a háttérben futó folyamatokhoz képest, így erőforrások hiányában a teljes mérés futását is megállíthatja a rendszer. A felmerülő problémák ellenére két mérési ciklusban szakaszos adatgyűjtés történt, hosszú egybefüggő adatsor produkálása nem történt.

Az ennél pontosabb aktivitásméréshez már viselhető céleszközök szükségesek, amelyek több napos, akár hetes üzemidőre is képesek fix mintavételezés mellett. Ilyen lehet például a Műszaki Informatika Tanszék által fejlesztés alatt álló aktivitásmérő, amely csuklóra erősíthető. A megoldásom szerver rétege akár a céleszközök kivitelezésénél is felhasználható.

Az aktivitást, mint matematikai fogalmat több módszerrel definiáltam a források alapján, melyeket különböző mintavételezési frekvencián is megvizsgáltam. Végül összehasonlítottam egy kiválasztott időszakban a lokációs és az aktigráfiás időbeli jeleket.

Irodalomjegyzék

1. Abigail Ortiz, Benjamin Rusak, Kamil Bradler, Luiza Radu, Martin Alda: Exponential state transition dynamics in the rest–activity architecture of patients with bipolar disorder. 2015.
2. Aleksandra Domagalik, Dante R. Chialvo, Ewa Beldzik, Ewa Gudowska-Nowak, Jacek Tyburczyk, Jeremi K. Ochab, Jerzy Szwed, Halszka Oginska, Maciej A. Nowak, Magdalena Fafrowicz, Tadeusz Marek: Scale-Free Fluctuations in Behavioral Performance: Delineating Changes in Spontaneous Behavior of Humans with Induced Sleep Deficiency. 2014. http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0107542 (Utolsó megtekintés dátuma: 2017. november 11.)
3. Alex V. Rowlands, Charlotte L. Edwardson, Dale W. Esliger, James Sanders, Kamlesh Khunti, Kishan Bakrania, Melanie Davies, Sarah Bunnewell, Thomas Yates: Intensity Thresholds on Raw Acceleration Data: Euclidean Norm Minus One (ENMO) and Mean Amplitude Deviation (MAD) Approaches. 2016. http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0164045 (Utolsó megtekintés dátuma: 2017. november 11.)
4. Andrea Z. LaCroix, Ciprian M. Crainiceanu, Chongzhi Di, David M. Buchner, Jiawei Bai, Kelly R. Evenson, Luo Xiao: An Activity Index for Raw Accelerometry Data and Its Comparison with Other Activity Metrics. 2016 https://doi.org/10.1371/journal.pone.0160644 (Utolsó megtekintés dátuma: 2017. november 11.)
5. Android Developer: https://developer.android.com (Utolsó megtekintés dátuma: 2017. október 30.)
6. Android koordináta rendszer: https://www.mathworks.com/help/supportpkg/android/ref/simulinkandroidsupportpackage_galaxys4_accelerometer.png (Utolsó megtekintés dátuma: 2017. szeptember 09.)
7. Ann L. Sharpley, Alexis Economou, Aynur Gormez, Digby J. Quested, Franco De Crescenzo: Actigraphic features of bipolar disorder: A systematic review and meta-analysis. 2016.
8. Audrey de Nazelle PhD, David Donaire-Gonzalez MSc, Edmund Seto PhD, Mark J Nieuwenhuijsen PhD, Michael Jerrett PhD, Michelle Mendez PhD: Comparison of Physical Activity Measures Using Mobile Phone-Based CalFit and Actigraph. 2013. http://www.jmir.org/2013/6/e111 (Utolsó megtekintés dátuma: 2017. november 11.)
9. Barbara C. Galland, Barry J. Taylor, Edwin A. Mitchell, Gavin J. Kennedy: Algorithms for using an activity-based accelerometer for identification of infant sleep–wake states during nap studies. 2012.
10. Code First: https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application (Utolsó megtekintés dátuma: 2017. július 03.)
11. C. Anteneodoa, D. R. Chialvo: Unraveling the fluctuations of animal motor activity. 2009. https://arxiv.org/pdf/0904.2177.pdf (Utolsó megtekintés dátuma: 2017. november 11.)

12. Daniel F. Kripke, Girardin Jean-Louis, Jeffrey A. Elliott, Shawn D. Youngstedt, William J. Mason: Sleep estimation from wrist movement quantified by different actigraphic modalities. 2000.
13. Entity Framework: https://docs.microsoft.com/en-us/ef (Utolsó megtekintés dátuma: 2017. július 03.)
14. Fastest Way of Inserting in Entity Framework: https://stackoverflow.com/a/5942176/5436901 (Utolsó megtekintés dátuma: 2017. július 13.)
15. Google GSON: https://github.com/google/gson (Utolsó megtekintés dátuma: 2017. augusztus 19.)
16. GreenRobot GreenDao: http://greenrobot.org/greendao (Utolsó megtekintés dátuma: 2017. augusztus 14.)
17. Jens-Peter Gnam, Klaus Bös, Sascha Härtel, Simone Löffler: Estimation of energy expenditure using accelerometers and activity-based energy models – validation of a new device. 2010. https://link.springer.com/article/10.1007/s11556-010-0074-5 (Utolsó megtekintés dátuma: 2017. augusztus 28.)
18. Lakos Ferenc: Hétköznapi mozgásmintázatok vizsgálatát támogató applikáció fejlesztése, 2016.
19. Language-Integrated Query: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq (Utolsó megtekintés dátuma: 2017. október 28.)
20. Play Store: https://play.google.com/store (Utolsó megtekintés dátuma: 2017. november 03.)
21. Razor: https://docs.microsoft.com/en-us/aspnet/core/mvc/razor-pages/?tabs=visual-studio (Utolsó megtekintés dátuma: 2017. október 28.)
22. SQL Server Management Studio: https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms (Utolsó megtekintés dátuma: 2017. október 28.)
23. Square OkHttp: https://github.com/square/okhttp (Utolsó megtekintés dátuma: 2017. augusztus 21.)
24. Token Based Authentication: https://docs.microsoft.com/en-us/aspnet/web-api/overview/security/individual-accounts-in-web-api (Utolsó megtekintés dátuma: 2017. július 08.)

Nyilatkozat

Alulírott Vince Dániel mérnökinformatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Műszaki Informatika Tanszékén készítettem, mérnökinformatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

2017. november 30.

Aláírás

Köszönetnyilvánítás

Szeretnék köszönetet mondani témavezetőmnek, Vadai Gergelynek, aki észrevételeivel és hasznos tanácsaival segítette a munkámat, Antal Andrásnak, aki a jelfeldolgozásban nyújtott hatalmas segítséget, továbbá közvetlen kollégáimnak, akikhez a fejlesztés során technikai tanácsért fordulhattam.

Köszönöm a családomnak és élettársamnak, hogy szeretetükkel és türelmükkel támogattak egyetemi tanulmányaim alatt.