



Eötvös Loránd Tudományegyetem

Informatikai kar

Algoritmusok és Alkalmazásaik Tanszék

Szakdolgozat

Konvex optimalizálási problémák párhuzamos megoldása grafikus processzorok felhasználásával

Témavezető:
Eichhardt Iván
Doktorandusz
ELTE-IK

Készítette:
Sipos Ágoston
prog. inf. BSc
ELTE-IK

Budapest, 2016

Tartalomjegyzék

1. Bevezetés	1
1.1. Motiváció	1
1.2. Az értekezés körvonalakban	1
1.3. Irodalmi áttekintés	1
2. Felhasználói dokumentáció	3
2.1. A szoftver célja	3
2.2. A szoftver használata	3
2.2.1. Rendszerkövetelmények	3
2.2.2. Telepítés	4
2.2.3. Vezérlés	4
2.2.4. Bemenet	5
2.2.5. Kimenet	6
2.2.6. Példák	7
3. Elméleti alapok	9
3.1. Matematikai alapfogalmak	9
3.2. Deriváltszámítási módszerek	9
3.2.1. Automatikus differenciálás	10
3.2.2. Automatikus differenciálás többváltozós esetben	11
3.3. Numerikus optimalizálási algoritmusok a számítógépes látásban	11
3.3.1. Gradiens módszer	12
3.3.2. Gauss-Newton módszer	12
3.3.3. Levenberg-Marquardt módszer	13
4. Fejlesztői dokumentáció	14
4.1. OpenCL alapok	14
4.2. Kódgenerálás	15
4.3. Párhuzamosítás	17
4.3.1. Gradiens módszer párhuzamos implementációja	17
4.3.2. Gauss-Newton módszer párhuzamos implementációja	18
4.3.3. Levenberg-Marquardt módszer párhuzamos implementációja	20

4.4. A szoftver felépítése	20
4.4.1. Statikus terv	20
4.4.2. Dinamikus terv	22
5. Tesztelés	24
5.1. Geometriai illesztések	24
5.1.1. Kör	24
5.1.2. Ellipszis	27
5.2. Síkhomográfia	33
6. Konklúzió	38
6.1. Összegzés	38
6.2. Fejlesztési lehetőségek	38
Köszönetnyilvánítás	39
A. DVD melléklet	40
Ábrák és algoritmusok jegyzéke	42
Irodalomjegyzék	43

1. Bevezetés

1.1. Motiváció

Számítógépes látási problémák esetén gyakran merül fel olyan feladat, amelynek csak numerikus megoldása képzelhető el. Ennek oka jellemzően a feladat magas dimenzionáltsága (egy perspektív kamerakalibráció például 11 paraméterrel írható le), illetve a felhasználandó adatok nagy száma. Ezeknek az algoritmusoknak azonban a számításigénye is magas, így érdemes különböző gyorsítási lehetőségeket keresni.

Dolgozatomban az ilyen algoritmusokat gyorsítani képes párhuzamosítási lehetőségeket fogom elemezni, illetve az implementációs lehetőségeiket grafikus processzorokra az OpenCL nyílt keretrendszeren keresztül. Ki fogok térni egy általam készített implementációra, és annak teljesítmény-elemzésére is.

1.2. Az értekezés körvonalakban

Ahogy címe is mutatja, a dolgozat témája konvex optimalizálási problémák párhuzamos megoldása. A témabejelentőben írottakat maradéktalanul teljesítettem; a főbb pontjait a következő bekezdésben emelem ki a társított fejezetekkel együtt.

A következő, *2. fejezetben* közlöm a program felhasználói dokumentációját. A *3. fejezetben* áttekintem a szükséges matematikai fogalmakat, a megvalósított algoritmusokat, a *4. fejezetben* vázolom a feladat implementációját, míg a *5. fejezetben* példákat mutatok be a szoftverem alkalmazásaira. A mellékelt DVD lemezen megtalálható a program, illetve a forráskódja, az *A. függelékben* olvasható a rajta levő állományok leírása.

1.3. Irodalmi áttekintés

A nemlineáris optimalizálás [1] egy alkalmazások szempontjából fontos téma. Számítógépes implementációkban a deriváltértékek numerikusan pontos számítása elengedhetetlen. Erre ad [2] egy automatikus differenciálási algoritmust első (és második) deriváltak meghatározására, és egy C++ implementációját a módszernek.

A párhuzamos számításokra alkalmas eszközök elterjedése óta egyre nagyobb az igény megfelelően párhuzamosított algoritmusok futtatására. Az OpenCL keretrendszer [3] lehetőséget biztosít párhuzamos algoritmusok különböző hardvereken (köztük GPU-kon) való futtatására.

A két megközelítés, azaz az automatikus differenciálással való pontos kiértékelés, és az OpenCL által biztosított párhuzamos futtatás ötvözésével azonban nem találkozom. OpenCL kernelek generálását már alkalmazzák speciális célokra ([4] például ritka mátrixokkal végzett lineáris algebrai műveletekre). Dolgozatomban azonban az automatikus differenciálás segítségével generált OpenCL kódokat fogok felhasználni a párhuzamos számítások futtatására.

A dolgozat témájához legközelebb álló legkorszerűbb (State of the Art) rendszer a Ceres Solver [5], mely egy nyílt forráskódú könyvtár nagy és komplikált optimalizációs problémák megoldására. Szintén használja [2] megközelítését.

2. Felhasználói dokumentáció

Az alábbiakban bemutatom a szoftver használatához szükséges legfontosabb tudnivalókat.

2.1. A szoftver célja

A szoftver egy numerikus optimalizálási feladatokra használható C++ könyvtár, a hozzá tartozó, és a tipikus felhasználási módokat lehetővé tevő tesztkörnyezettel együtt. Jelenleg háromféle speciális feladatra használható, de nyílt forráskódúsága miatt a hozzáértő felhasználók további problémák megoldására is alkalmassá tehetik.

A jelenleg elérhető feladatmegoldók:

- Körillesztés
- Ellipszisillesztés
- Homográfia (lásd: 5.2) illesztés

2.2. A szoftver használata

2.2.1. Rendszerkövetelmények

A program jelenleg Microsoft Windows operációs rendszeren elérhető, futtatható állomány formájában.

Rendszerkövetelmények:

	Várható működés	Tesztelt, garantált működés
Operációs rendszer	Microsoft Windows 7, vagy újabb	
Processzor	OpenCL-t támogató	Intel Core i5, vagy újabb
Videókártya	OpenCL-t támogató	NVidia GeForce GTX 650, vagy újabb

A két hardverkövetelmény közül elegendő az egyik teljesítése, de ekkor nem fogjuk tudni szabadon kiválasztani a számítások végzéséhez használt hardvert.

A program alkalmatlan hardver választása esetén az első futtatáskor hibajelzéssel leáll.

2.2.2. Telepítés

A DVD mellékleten (A.) található *build* mappában található fájlok a saját gép merevlemezére történő másolásával a szoftver futtatható.

Amennyiben a felhasználó magának szeretné a forráskódot lefordítani, akkor a *source* könyvtárban található meg a forráskódot Visual Studio projekt fájl formájában, C++ nyelven. A fordításhoz szükséges függőségek a *dependencies* könyvtárban vannak, ahhoz, hogy a fordító ezeket megtalálja, a mappában található *mount_t.bat* fájl futtatása szükséges. (Célszerű ezeket a könyvtárakat is a merevlemezre másolni előzőleg.)

2.2.3. Vezérlés

A szoftver jelenleg konzolos felületen keresztül parancsok megadásával vezérelhető. Az indítás a futtatható állományra való dupla kattintással, vagy egy parancs-sorablakban a megfelelő könyvtárban a futtatófájl (*start.bat*) nevének megadásával történhet. Ekkor megjelenik a szoftver kezelőfelülete.

A felhasználói felület nyelve angol.

A konzolba írható parancsok:

- *quit* - Kilépés.
- *help* - Kiír egy rövid segítséget az egyéb beírható parancsokról.
- *cpu, gpu* - Átállítja a számításokra alkalmazott hardvert a későbbiekben indított számításokban (a program állapotában tárolja).
- *circle <filename>* - elvégez egy körillesztést az adott fájlban levő inputra. A bemenet formátumát illetően lásd 2.2.4. Kapcsolóként lásd a táblázatot.
- *ellipse <filename>* - elvégez egy ellipszisillesztést az adott fájlban levő inputra. A bemenet formátumát illetően lásd 2.2.4. Kapcsolóként lásd a táblázatot.
- *homography <image1> <image2>* - homográfiaillesztést végez a két képfájlon. Képformátumokért lásd 2.2.4. Kapcsolóként lásd a táblázatot.

Kapcsoló	Hatás	Parancsok, amikre használható	Magyarázó hivatkozás
-grad, -gauss, -levenberg	A megfelelő algoritmussal végzi az illesztést	circle, ellipse, homography	3.3
-image <filename>	Vektorgrafikus kimenetfájl megadása	circle, ellipse	5.1, 5.4, 5.5
-matchesImg, -inputImg, -outputImg, -warpedImg	Kimeneti fájlok megadása a homográfiaillesztés szakaszaihoz	homography	5.13, 5.15, 5.16, 5.17
-distance, -mask	Beállítja a hibás pontok eldobásának módszerét	homography	5.2
-regular, -ransac, -lmeds	Beállítja a kezdeti homográfiabecslés algoritmusát	homography	5.2

Az implementációt befolyásoló kapcsolók részletes megértéséhez kövesd a mellékelt kereszthivatkozásokat.

2.2.4. Bemenet

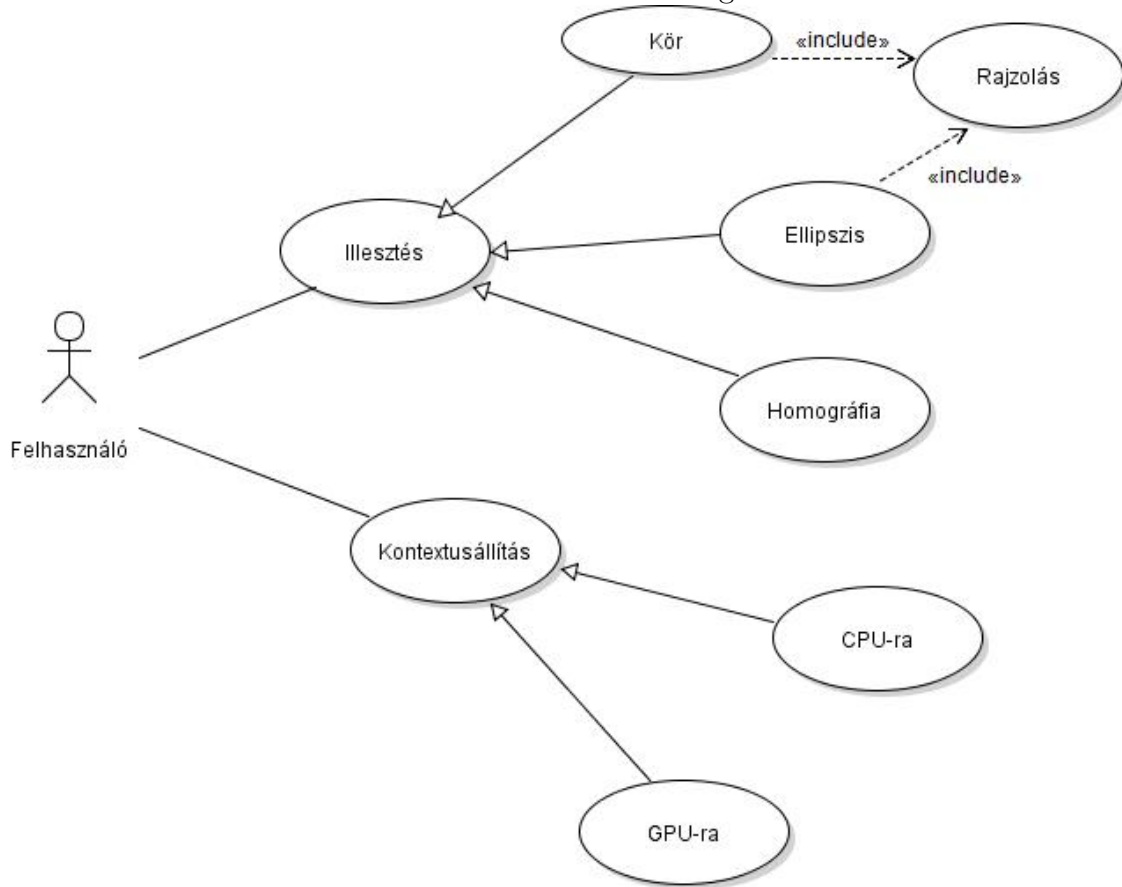
A bemenet a különböző feladatokra különböző:

- A kör-, és ellipszisillesztés esetén egy szöveges fájl, amely
 - első sorában tartalmazza az illesztésben felhasználandó pontok számát
 - további soraiban egy-egy kétdimenziós pont két koordinátáját (tizedestört alakban, tizedesponttal)
- A homográfia illesztés esetén két képfájl
 - Támogatott formátumok: BMP, JPG, TIF, PNG
 - A homográfia két, ugyanazon sík felületről (pl. óriásplakát, közlekedési tábla) készült fotó alapján tud dolgozni.

A program az egy, illetve két fájl elérési útvonallal megadott neve alapján tud továbblépni, és elkezdni a számításokat. Amennyiben egy fájl betöltése sikertelen (nem létezik, vagy nem megfelelő formátumú), a program hibát jelez, és visszalép a vezérlőfelületre.

Néhány tesztbemenet található a DVD mellékleten a *tests* mappában.

2.1. ábra. Use-case-ek összefoglalása



2.2.5. Kimenet

A program a kimenetet alapvetően fájlokba írja, illetve amennyiben ez lehetséges, a konzolba is kiírja.

A **körillesztés** kimenete 3 lebegőpontos szám, amit a program kiír a képernyőre, illetve egy logfájlba is. A 3 szám jelentése: a kör középpontjának két koordinátája, illetve a kör sugara. A program emellett kimenetként elkészít egy vektorgrafikus (svg formátumú) ábrát is, melyen látható a kör, és az illesztésben felhasznált pontok elhelyezkedése.

Az **ellipszisillesztés** kimenete 5 lebegőpontos szám, amit hasonló módon kiír a program. Az 5 szám jelentése: az ellipszis középpontjának (azaz a két fókuszpont súlypontjának) két koordinátája, a vízszintes, és függőleges tengely, illetve az ettől az állástól vett elforgatása ívmértékben (radián). Az ellipszisillesztés esetén a körillesztéshez hasonló ábra készül.

A **homográfiaillesztés** szigorúan vett kimenete a 3×3 -as projektív transzformációs mátrix ami a képernyőre, illetve a logfájlba kerül. Itt azonban a program több képet is elkészít, amelyek a homográfia pontosságát, és alkalmazhatóságát mutatják be.

- *matchesImg* néven exportálható az a kép, amin megtalálható az összes detektált pontpár.
- *inputImg* néven kimenthető a kezdeti becslés homográfiájával keretezett képpár.
- *outputImg* néven menthető a finomítás utáni homográfiával keretezett képpár.
- *warpedImg* néven kapható meg az első kép homográfiával transzformált változata (ami remélhetőleg jól közelíti a második képet).

2.2.6. Példák

A program indítása:

```
T:\GPGPU\AutoDiff_GPU\AutoDiff>start.bat
T:\GPGPU\AutoDiff_GPU\AutoDiff>AutoDiff.exe 2>log.txt
> _
```

Körillesztés számítása:

```
> circle -levenberg circ.dat -image w.svg
Circle center: (3, -1.78132e-009), radius: 1
>
```

Ellipszisillesztés számítása:

```
> ellipse -gauss ellipse_rot.dat -image e.svg
Ellipse center: (2.99999, -6.89109e-006), axes: 1.90987, 0.829292, rotation: 3.47239
> _
```

Homográfia becslése:

```
> homography -levenberg 1.jpg 2.jpg -distance -outputImg o.jpg

[1.45493 0.0534038 -246.578 1
 [0.307606 1.26017 -453.456 1
 [0.00019672 -1.87444e-005 1 1
>
```

Az utolsó hívás eredménye:

2.2. A SZOFTVER HASZNÁLATA



3. Elméleti alapok

Ebben a fejezetben bemutatom a szükséges fogalmakat, matematikai tételeket, algoritmusokat.

3.1. Matematikai alapfogalmak

Az alábbiakban megemlítek néhány szükséges analízisbeli fogalmat, tételt.

3.1.1. Tétel. *Legyen f és g két differenciálható függvény. Ekkor $f + g$ is differenciálható, és $(f + g)' = f' + g'$.*

3.1.2. Tétel. *Legyen f egy $r > 0$ konvergenciasugarú hatványsor összegfüggvénye, $f(x) = \sum_{n=0}^{\infty} a_n(x - a)^n$, $(x \in \mathbb{R}, |x - a| < r)$.*

Ekkor: f differenciálható, és $f'(x) = \sum_{n=1}^{\infty} n a_n(x - a)^{n-1}$, $(x \in \mathbb{R}, |x - a| < r)$

Bizonyítások lásd: [6]

3.1.3. Tétel. *(másodrendű Taylor-formula)*

Legyen $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $\mathbf{x}, \delta \mathbf{x} \in \mathbb{R}^n$

Ekkor: $f(\mathbf{x} + \delta \mathbf{x}) = f(\mathbf{x}) + f'(\mathbf{x}) \cdot \delta \mathbf{x} + \frac{1}{2} \cdot \langle \delta \mathbf{x}, f''(\mathbf{x}) \cdot \delta \mathbf{x} \rangle + O(\delta \mathbf{x}^3)$

Megjegyzés: itt $f'(\mathbf{x}) \in \mathbb{R}^n$ gradiensvektor, és $f''(\mathbf{x}) \in \mathbb{R}^{n \times n}$ ún. Hesse-mátrix.

3.2. Deriváltszámítási módszerek

Három alapvető eljárás ismert deriváltak számítógépes kiértékelésére:

- Analitikus, vagy szimbolikus differenciálás.

- Ebben az esetben expliciten megadjuk a függvény deriváltját, majd csak ennek kiértékelését végezzük el megfelelő helyeken.
- Általános megoldása bonyolult.
- Mindazonáltal a megvalósítás során fogunk meríteni az ötletéből.
- Numerikus differenciálás
 - A deriváltat a definiálásához is felhasznált differenciahányadosokkal közelítjük.
 - Problémás, ugyanis a törtünk nevezője 0-hoz fog konvergálni, az osztás nagy numerikus hibával járhat
- Automatikus differenciálás
 - Ezt az alábbiakban fogom ismertetni

3.2.1. Automatikus differenciálás

Az algoritmus itt tárgyalt változata a duális számok elméletén alapul. Ez egy $\mathbb{R} \times \mathbb{R}$ -en értelmezett algebrai struktúra az alábbiak szerint¹:

$$(a, b) + (c, d) = (a + c, b + d) \quad (3.2.1.1)$$

$$(a, b) \cdot (c, d) = (a \cdot c, a \cdot d + b \cdot c) \quad (3.2.1.2)$$

A duális számokat a továbbiakban $\mathbb{D}$ -vel fogom jelölni.

3.2.1.3. Tétel. *A duális számok gyűrűt alkotnak, de nem alkotnak testet.*

Megjegyezzük, hogy az osztás nem mindig értelmezhetősége okozza ezt: ha $c = 0$, akkor az $(a, b)/(c, d)$ osztás nem végezhető el.

A továbbiakban legyen $f : \mathbb{R} \rightarrow \mathbb{R}$ analitikus függvény (hatványsor-összegfüggvény). Ekkor, mivel f minden tagja előállítható összeadások és szorzások véges sok lépésével, továbbá a végtelen összegzés könnyen definiálható a duális számokra, definiálni tudjuk azt a $\hat{f} : \mathbb{D} \rightarrow \mathbb{D}$ függvényt, ami az f -ével megegyező hatványsor által definiált, a duális számokon értelmezett analitikus függvény.

¹Lényegében a komplex számok egy olyan „módosításáról” van szó, ahol a képzetes egység négyzete -1 helyett 0.

3.2.1.4. Tétel. $\hat{f}((a, b)) = (f(a), b \cdot f'(a))$ (az előbbi jelölések mellett, $a, b \in \mathbb{R}$)

A tétel az 3.1.2 tétel felhasználásával könnyen belátható, azt kell csak felhasználni, hogy a képzetes egység négyzete 0.

Tehát: Ha meg tudjuk adni egy f valós-valós analitikus függvényhez az $\hat{f}$ függvényt, akkor ezzel egy lépésben számítható a függvényérték, és a derivált is, hiszen $\hat{f}((a, 1)) = (f(a), f'(a))$.

3.2.2. Automatikus differenciálás többváltozós esetben

Legyen most $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Ekkor minden $\mathbf{a} \in \mathbb{R}^n$ pontban megadható f *gradiense*, azaz az $f'(\mathbf{a}) \in \mathbb{R}^n$ vektor.

Ekkor célunk az $\hat{f}_i : \mathbb{D}^n \rightarrow \mathbb{D}$, ($i \in \{1, \dots, n\}$) függvények megadása, ahol az i . függvény az i . változó szerinti parciális deriváltat adja meg, azaz $\hat{f}_i(\mathbf{x}) = (f(\mathbf{a}), \partial_i f(\mathbf{a}))$. Az $\mathbf{x}$ vektort illetően:

$$\mathbf{x} = ((a_0, 0), \dots, (a_i, 1), \dots, (a_n, 0)) \quad (3.2.2.1)$$

3.3. Numerikus optimalizálási algoritmusok a számítógépes látásban

A numerikus optimalizálás célja egy valós értékű függvény minimumhelyének megtalálása. Célszerű okokból feltesszük, hogy a függvényünk n -változós valós függvény, és legalább egyszer differenciálható.

A látási problémák esetén az optimalizálási feladatok jellemzően a következő módon néznek ki:

- Néhány paraméterrel leírható a keresett transzformáció (paramétertér).
- Számos mérési adatunk van, amit konkrét képekből szereztünk. Ezek általában valamilyen detektáló eljárással megtalált pontmegfeleltetések.
- A minimalizálandó függvény (ún. költségfüggvény) általános alakja:

$$f(\mathbf{p}, \mathbf{x}) = \sum_{i=0}^n f_i(\mathbf{p}, \mathbf{x}_i) \quad (3.3.0.1)$$

, ahol $\mathbf{p}$ a paraméterek vektora, $\mathbf{x}_i$ az i . mérési adat vektora.

Algoritmusaink úgy fognak működni, hogy kiindulunk egy kezdőpontból, majd az ottani függvény- és deriváltérték figyelembe vételével lépünk tovább egy másik pontba. Ezt egy megállási feltétel eléréséig ismétljük.

3.1.1 alapján a 3.3.0.1 deriváltja is összegként számítható, ráadásul *ugyanazon függvény* különböző pontokban vett deriváltjai összegzésével. Ezt a körülményt a párhuzamosítás céljából ki is fogjuk használni.

Tekintsünk át néhány konkrét módszert [1].

3.3.1. Gradiens módszer

Választunk $\mathbf{x}_0 \in \mathbb{R}^n$ kezdőpontot, majd az alábbi iterációt végezzük:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha_i \cdot f'(\mathbf{x}_i), \quad (\alpha_i \in \mathbb{R}, \alpha_i > 0) \quad (3.3.1.1)$$

- Az α_i valós paraméter megválasztására a módszer nem ad konkrét, egyértelmű számítási módot, különböző stratégiák léteznek erre. Erősen feladatfüggő, hogy milyen stratégiát érdemes választani.
 - Lehet konstans, esetleg néhány lépésenként csökkentett konstans
 - Felhasználhatjuk a gradiens normálására, hiszen a gradiens nagysága változásában semmit sem mond a szélsőérték helyek távolságáról

3.3.2. Gauss-Newton módszer

A módszer célja jó közelítést találni az adott lépésben teendő lépésközzre.

A 3.1.3 tételből levezethető (a harmadrendű maradéktag elhanyagolásával), hogy:

$$\delta \mathbf{x} = -(f''(\mathbf{x}))^{-1} \cdot f'(\mathbf{x}) \quad (3.3.2.1)$$

lesz az a lépésköz, amellyel az $f(\mathbf{x} + \delta \mathbf{x})$ függvényérték minimális lesz.

Ehhez a költségfüggvényről fel kell tennünk, hogy kétszer deriválható, továbbá minden lépésben kell második deriváltat számítani. Ez azonban számításigénye miatt nem mindig elfogadható, ezért a második deriváltat közelíteni fogjuk.

Feltesszük, hogy $f = \sum f_i^2$ (ehhez annyi kell, hogy négyzetes hibát becsüljünk). Így $f' = \sum 2 \cdot f'_i \cdot f_i$ a deriválási szabályok alapján. Tehát az $F = [f_1, f_2, \dots, f_m]$ jelöléssel: $f'(\mathbf{x}) = 2 \cdot (F'(\mathbf{x}))^T \cdot F(\mathbf{x})$ (itt $F'(\mathbf{x})$ az F vektor-vektor függvény Jacobi-mátrixa).

Belátható továbbá, hogy (a másodrendű tagok elhanyagolásával): $f''(\mathbf{x}) \approx 2 \cdot (F'(\mathbf{x}))^T \cdot (F'(\mathbf{x}))$ a költségfüggvény Hesse-mátrixának becslése.

Tehát a 3.3.2.1 ezen közelítésével az alábbi iterációhoz jutunk²:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \left((F'(\mathbf{x}_i))^T \cdot (F'(\mathbf{x}_i)) \right)^{-1} \cdot (F'(\mathbf{x}_i))^T \cdot F(\mathbf{x}_i) \quad (3.3.2.2)$$

Az implementáció során természetesen az itt található számításokra (mátrixszorzás, lineáris egyenletrendszer megoldás³) figyelemmel kell lennünk.

3.3.3. Levenberg-Marquardt módszer

A Newton, illetve Gauss-Newton módszerekkel kapcsolatban felmerülhet az a probléma, hogy a lineáris egyenletrendszer mátrixa szinguláris, esetleg közel szinguláris. Ekkor numerikus pontatlanságból származó hibák keletkezhetnek. További felismerés lehet, hogy az optimumtól távoli értékekre az egyszerűbb gradiens módszer jobb (nagyobb, jó irányba tartó) lépéseket tud találni.

A Levenberg-Marquardt [7] algoritmus ötvözi a két módszer előnyös tulajdonságait, még hozzá úgy, hogy egy az iteráció során változó $\lambda > 0$ valós paraméter használatával a LER mátrixához hozzáad $\lambda \cdot E$ -t⁴. Tehát az iteráció:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \left((F'(\mathbf{x}_i))^T \cdot (F'(\mathbf{x}_i)) + \lambda \cdot E \right)^{-1} \cdot (F'(\mathbf{x}_i))^T \cdot F(\mathbf{x}_i) \quad (3.3.3.1)$$

A λ paraméter megválasztására alapvetően az a szabály, hogy ha a költségfüggvény értéke „nagy”, akkor sikertelen lépést tettünk, ezért nagyobb értékre állítjuk, így a gradiens módszer által elérhető gyorsabb konvergenciához kerülünk közelebb, míg ha „kicsi”, akkor sikeres volt a lépés, ezért csökkentjük, így a pontosabb Gauss-Newton módszerhez kerülünk közelebb.

²Megjegyzés: itt a mátrix invertálhatóságához szükséges $m > n$, azaz hogy a hibafüggvény komponenseinek száma legyen nagyobb, mint a változóinak a száma.

³Numerikus pontossággal kapcsolatos megfontolásokból az invertálást LER megoldással váltjuk ki.

⁴E a megfelelő méretű négyzetes egységmátrix

4. Fejlesztői dokumentáció

4.1. OpenCL alapok

Az OpenCL [3] egy *heterogén számításokat* támogató, nyílt, cross-platform framework. Egyaránt támogatja a processzorok (CPU), videókártyák (GPU), és egyéb alkalmas hardverek párhuzamos számításokkal történő kihasználását. Mivel nem hardverfüggetlen, több különböző, egy gépbe épített hardvert ki tud egyszerre használni.

Saját programozási nyelvvel rendelkezik (C-jellegű), amelyen az eszközön futó kódok (ún. kernelek) megírhatók. Számos nyelvhez (C++, Python, Java) elérhető API, amelyen keresztül az OpenCL kernelek vezérelhetők.

Megjegyzés: A felhasznált hardverek megnevezésével kapcsolatban meghonosodott terminológia szerint beszélhetünk *hosztoldalról* (itt vezéreljük a programot, jellemzően CPU oldal), illetve *device- vagy eszközoldalról* (itt futnak az OpenCL kernelek). Az inicializáláskor az OpenCL-nek megadható, hogy milyen eszközöket használjon fel (például csak CPU, vagy GPU hardvereket).

4.1.1. Program. *Példa OpenCL kernel, amely párhuzamosan összead két tömböt.*

```
__kernel void f(  
    __global const float* x0,  
    __global const float* x1,  
    __global float* R )  
{  
    int i = get_global_id(0);  
    R[i] = x[i] + y[i];  
}
```

Fontos megjegyezni, hogy a kernelek fordítását a programunkból az API-n keresztül végezhetjük, tehát futásidőben is változtathatunk a kódjukon.

4.2. Kódgenerálás

A célom az volt, hogy egy C++-ban megadott függvényhez (itt függvény alatt matematikai függvényt értek, lásd 3.3.0.1) generálni tudjam a deriváltját kiszámító OpenCL kódot. Ehhez az automatikus differenciálást használtam fel.

Definiáltam az alábbi osztályokat:

- Duális számok [2]:
 - Template osztály, mely két mezőt tartalmaz egy megadott típusból
 - Definiálva van rajta az összes C-ben lehetséges matematikai művelet és függvény a 3.2.1.1, 3.2.1.2 és 3.2.1.4 szerint. (Kihasználom, hogy ezek elemi függvények, a deriváltjaik ismertek.)

4.2.1. Program. *Példa függvénydefiníció a duális számokra.*

```
dualnumber<T> sin(dualnumber<T> x) {  
    return dualnumber<T>(sin(x.f0), x.f1*cos(x.f0));  
}
```

- Kódgeneráló típus:
 - Belső reprezentációja egy string
 - Definiálva vannak rajta ugyanezek a műveletek, kódgenerálást megvalósítva. Bináris műveleteknél összefűzi a két stringet és közéírja a műveleti jelet, valamint zárójelezi a kifejezést, míg függvényeknél a kifejezés köré írja a függvényhívást, azaz például $x + y = (x + y)$, $\sin(x) = \sin(x)$.
 - Elvégzek néhány optimalizációs lépést: 0-val szorzás esetén a kifejezés 0 lesz, 1-gyel szorzás esetén önmaga, 0-val osztás esetén generálási időben hibát jelzek.

4.2.2. Program. *Példa függvénydefiníció a kódgeneráló típusra.*

```
MathCodeGen pow(const MathCodeGen& x, const MathCodeGen& y) {  
    if (y.code == std::to_string(1)  
        || y.code == std::to_string(1.0) + "f") return x;  
    return MathCodeGen("pow("+x.code+", "+y.code+")");  
}
```

Meggondolható, hogy ha egy template függvényt, amelyben csak a megengedett matematikai műveleteket használjuk, *meghívunk egy kódgenerálóból álló duális számon* (a korábban említettek alapján (x , 1) bemenő értékkel), akkor a kimenő érték

a ($\langle \text{függvényérték kódja} \rangle$, $\langle \text{derivált kódja} \rangle$) pár lesz. Tehát megkapjuk azt a két kifejezést, amit kiértékelve a függvényérték, vagy a derivált kiszámítható.¹

Mivel ezt a kifejezést lehetőségünk van tetszőleges szintaxissal előállítani, ezért OpenCL kernelt is tudunk belőle generálni. Ezt az OpenCL lehetőségei miatt rögtön le is fordíthatjuk tárgykódra.

4.2.3. Program. Példa C++ kód, és a belőle generált deriváltkód

```
template <typename T>
T f(const std::array<T,3> t, const std::array<T,2> x) const
{
    return pow((x[0] - t[0])*(x[0] - t[0])
               + (x[1] - t[1])*(x[1] - t[1])
               - t[2] * t[2],
               2);
}

-----

__kernel void d1(const float a0, const float a1, const float a2,
    __global const float* x0,
    __global const float* x1,
    __global float* _R)
{
    int i = get_global_id(0);
    _R[i] =
        ((2.0f*(((x0[i]-a0)*(x0[i]-a0))+((x1[i]-a1)*(x1[i]-a1))))
        -(a2*a2))*(((x0[i]-a0)*(-1.0f))+((-1.0f)*(x0[i]-a0))));
}

-----

__kernel void d2(const float a0, const float a1, const float a2,
    __global const float* x0,
    __global const float* x1,
    __global float* _R)
{
    int i = get_global_id(0);
    _R[i] =
        ((2.0f*(((x0[i]-a0)*(x0[i]-a0))+((x1[i]-a1)*(x1[i]-a1))))
        -(a2*a2))*(((x1[i]-a1)*(-1.0f))+((-1.0f)*(x1[i]-a1))));
}
```

¹Többváltozós esetben persze a 3.2.2.1 szerint kell eljárni.

```
-----
__kernel void d3(const float a0, const float a1, const float a2,
    __global const float* x0,
    __global const float* x1,
    __global float* _R)
{
    int i = get_global_id(0);
    _R[i] = ((2.0f*(((x0[i]-a0)*(x0[i]-a0))
        +((x1[i]-a1)*(x1[i]-a1)))-(a2*a2)))*(-(a2+a2)));
}
```

Megjegyezhető, hogy a kódgenerálással voltaképpen megkapjuk a függvény deriváltjának *pontos* kiszámítását lehetővé tevő kódot, azaz az algoritmus tulajdonképpen szimbolikus deriválásra is használható lehetne.

4.3. Párhuzamosítás

Korábban említettem (3.3.), hogy a költségfüggvények a vizsgált problémákban egy - akár nagyon sok tagú - szummával írhatók fel. Mivel ezt lehet tagonként differenciálni, ezért a kódgeneráláshoz elég egy tag függvénykénti megadása, és annak deriváltjának generálása, hiszen a többi tag kódja is ugyanaz, csak más adatpontokban számítandók.

Ez a párhuzamos programozási paradigma az SIMD (Single Instruction, Multiple Data). A modern hardverek hatékony támogatást biztosítanak hozzá (kiemelkedően a GPU-k). Ehhez viszont szükség van a memóriefoglalások pontos irányítására, hiszen az adatmozgatások mindig - a számításokhoz képest - nagy időt vesznek igénybe.

Az eszközoldali memóriakezelésre a VexCL [8, 9] könyvtárat használtam. Ez egy C++ template könyvtár, amely lehetővé teszi a lefoglalt memória STL-szerű, objektorientált kezelését. Például egy `vex::vector` az `std::vector`-hoz hasonló interface-szel rendelkezik, de megvalósítását tekintve a tárolt elemeket a felhasznált hardverek memóriájában tárolja.

4.3.1. Gradiens módszer párhuzamos implementációja

A gradiens módszernek minden lépésben pontosan egy deriválkiértékelésre van szüksége, e tekintetben tehát nincs kézenfekvő párhuzamosítás. A deriválkiértékelés

azonban hardveresen gyorsítható a szummás kifejezés tagonkénti kiértékelésével, majd az összegzés GPU oldali elvégzésével².

Mivel az egyes parciális deriváltak tagjait ugyanazzal a kóddal, ugyanazokkal a paraméterértékekkel, viszont különböző adatpontokban kell számítani, továbbá az adatpontok nem változnak az iteráció során, érdemes az adatpontokat előre *vec::vector*-okban eltárolni. Ezután az iterációs lépés során a paraméterek hozzárendelése mellett minden adatpontra kiszámítjuk a hozzá tartozó tagot párhuzamosan. Ezeket a tagokat is egy *vec::vector*-ba számítjuk. Ezt a konténert pedig a VexCL által biztosított ún. *Reductor* template segítségével eszközoldalon összegezzük, majd az eredményt visszaadjuk a C++ programnak. A megvalósítást vizualizáltam a 4.1 ábrán.

Az iteráció során a deriváltakkal ellentétes irányba kell lépnünk. A tapasztalatok alapján arra jutottam, hogy a gradiens „túl nagy” konstansokkal való szorzása esetén az algoritmus a nagy lépésekkel „elugrál” nagy távolságokra az optimumtól, míg a túl kicsi konstans nagyon lassúvá teszi a konvergenciát. Ennek oka, hogy a gradiensvektor euklideszi értelemben vett hossza valójában nem tartalmaz információt a legmegfelelőbb lépésközzől.

Ezért a kapott gradiensvektort leosztottam a 2-es normájával, majd megszoroztam egy az algoritmus során fokozatosan csökkenő lépésköz-konstanssal. Az így kapott vektort kivontam az aktuális argumentumvektorból, így léptem tovább.

4.3.2. Gauss-Newton módszer párhuzamos implementációja

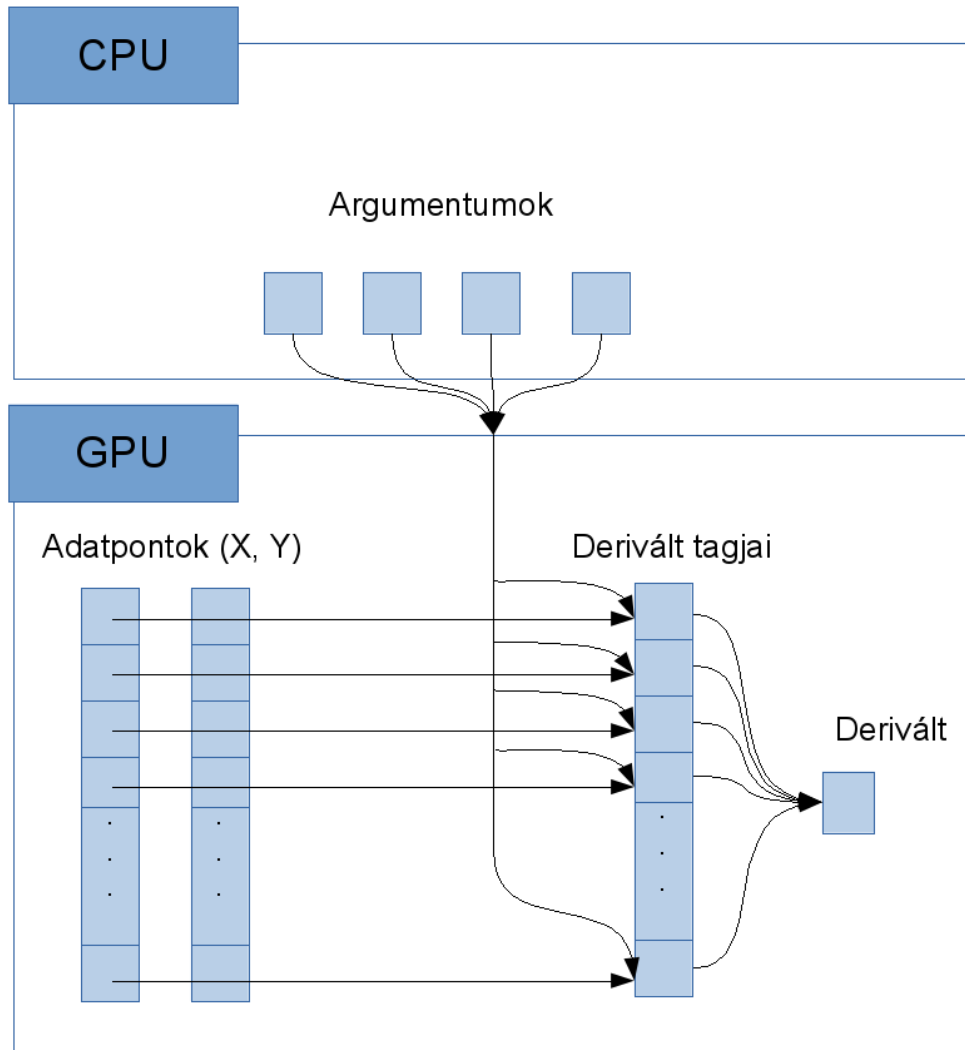
Emlékeztető: a végzendő iteráció:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \left((F'(\mathbf{x}))^T \cdot (F'(\mathbf{x})) \right)^{-1} \cdot (F'(\mathbf{x}))^T \cdot F(\mathbf{x}) \quad (4.3.2.1)$$

Itt $F'(x) \in \mathbb{R}^{n \times m}$ Jacobi-mátrix, melynek sorai az n . parciális derivált fentebb említett tagjai. $F(\mathbf{x}) \in \mathbb{R}^m$ vektor a függvényérték, mint összeg tagjai. A mátrixot és a vektort tehát az előző módszerrel meg tudjuk adni, de észrevehetjük, hogy erre nincs is szükségünk. Ugyanis az $(F'(\mathbf{x}))^T \cdot (F'(\mathbf{x})) \in \mathbb{R}^{n \times n}$ mátrix elemei kiszámíthatók ezen vektorok páronkénti skaláris szorzataiként. Hasonlóan, a $(F'(\mathbf{x}))^T \cdot F(\mathbf{x}) \in \mathbb{R}^n$ vektor elemei az $F(\mathbf{x})$ vektorral vett skaláris szorzatból számíthatók. Ezután már csak egy $n \times n$ -es mátrixú lineáris egyenletrendszer megoldása a feladat.

²Párhuzamos környezetben az összegzés $O(\log(n))$ időigénnyel elvégezhető

4.1. ábra. A parciális deriváltak számításának modellje.



A lineáris algebrai műveletekre a ViennaCL [10] könyvtárat használtam. Ez egy OpenCL-lel implementált mátrix-vektorműveleteket támogató könyvtár. Sajnos a VexCL-hez nincs tökéletes interfésze, ezért a program használ hosztoldali adatmozgatásokat.

A $(F'(\mathbf{x}))^T \cdot (F'(\mathbf{x}))$ mátrixot tehát a már megkapott vektorok skaláris szorzataiból számítom, ehhez a VexCL által támogatott elemenkénti szorzatot és összegzést használom. Az így kapott adatokkal inicializálok egy `viennacl::matrix`-ot. Hasonlóan számítom ki az egyenletrendszer jobb oldalán álló vektort. Ezután a ViennaCL lineáris egyenletrendszer megoldójával kapom meg az iterációs lépést.

4.3.3. Levenberg-Marquardt módszer párhuzamos implementációja

A Levenberg-Marquardt módszer iterációja:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \left((F'(x))^T \cdot (F'(x)) + \lambda \cdot E \right)^{-1} \cdot (F'(x))^T \cdot F(x) \quad (4.3.3.1)$$

A Gauss-Newton módszer kapcsán már levezettem a $(F'(\mathbf{x}))^T \cdot (F'(\mathbf{x})) \in \mathbb{R}^{n \times n}$ és a $(F'(\mathbf{x}))^T \cdot F(\mathbf{x}) \in \mathbb{R}^n$ kiszámítását. A mátrix diagonális elemeihez hozzá kell adni a λ paramétert, ezután a LER ugyanolyan módon megoldható, és a lépésköz felhasználható.

Fontos része még az algoritmus megvalósításának a λ paraméter megválasztása. Ehhez az alábbi eljárást alkalmaztam:

- Kezdetben 1-re állítottam.
- Majd minden iterációban megvizsgáltam, hogy hányszorosára változott a költségfüggvény értéke, és ennyivel szoroztam λ -t.

Így a paraméter mindig pozitív marad (hiszen a költségfüggvény értékei pozitívak), továbbá nagyobb hiba esetén a gradiens, kisebb hiba esetén a Gauss-Newton módszerhez közelíti az iterációt.

4.4. A szoftver felépítése

4.4.1. Statikus terv

Az alábbiakban felvázolom a programban alkalmazott főbb osztályokat, röviden ismertetve a funkciójukat, és implementációjukat.

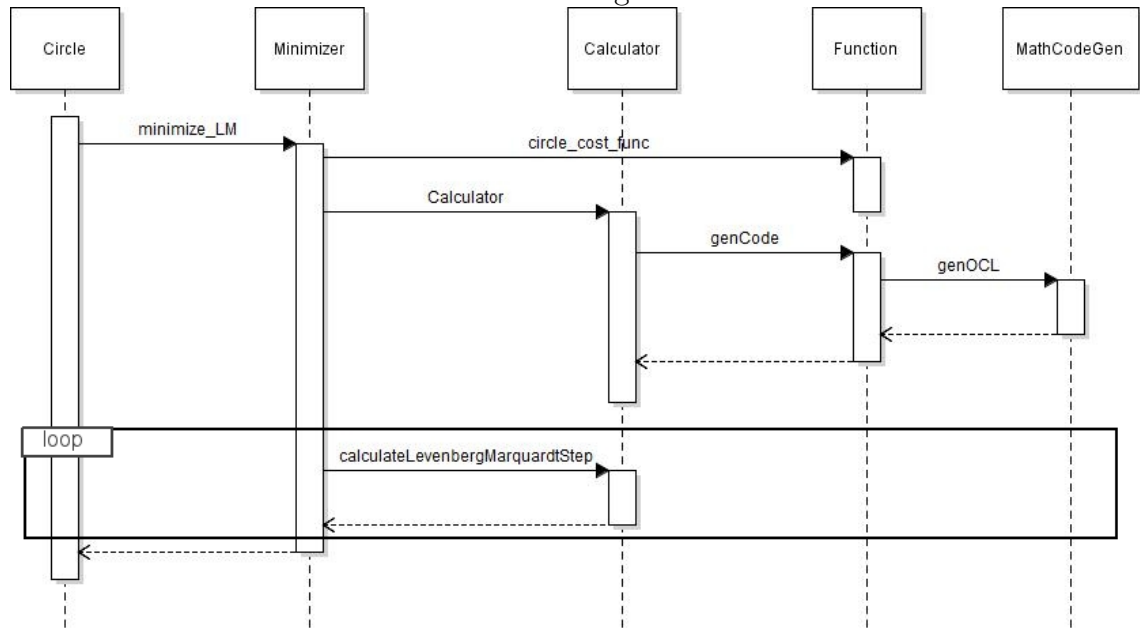
- **dualnumber**: A már ismertetett duális számok implementációja. Template osztály (1 típusparaméterrel)
 - Két T típusú mezőt tartalmaz.
 - Implementálva van rajta a duális számok algebrája (lásd 3.2.1). Az alapműveletek és a C nyelv matematikai könyvtárában megtalálható elemi függvények implementációja található meg benne.
- **MathCodeGen**: Matematikai függvények kódgenerálására alkalmas osztály.

- * `homography_cost_func`: egy homográfiaillesztés költségfüggvénye.
(Formális leírás: 5.2.0.1)
- A `genCode()` metódus elkészíti a függvény, illetve a deriváltak kiértékelésének kódját. Implementációjában használja a `dualnumber<MathCodeGen>` template specializációt.
- **Calculator**: A függvényérték, derivált, illetve különböző algoritmusok lépésközének számítására alkalmas osztály
 - A konstruktorban kapott függvény alapján készíti el az OpenCL kerneleket.
 - A kontextust a `context_type` alapján meghatározza és lekéri a globális kontextus-kezelőtől.
 - A megfelelő függvények kiértékelik a függvényértéket, gradienst, lépésközöket.
- **Minimizer**: Csak statikus függvényei vannak, a három implementált optimalizálási algoritmus.
 - Az algoritmusok első paramétere a kezdőpont, második (kimenő) paraméter az eredmény, harmadik az adatpontok vektorainak tömbje (azaz 2 dimenziós adattérnél egy 2-elemű, vektorokból álló tömb).
- **Circle, Ellipse**: A kör, illetve az ellipszis reprezentációját tároló osztályok. (A kör 3, illetve az ellipszis 5 paramétere a mezőik).
 - Közös tulajdonságaik ki vannak emelve a `Form` absztrakt őssztályba.
 - A `fit(...)` metódusok elvégzik a kör-, illetve ellipszisillesztést.
 - A `draw(...)` metódusok kirajzolják az eredményt egy vektorgrafikus ábrára.
- **Homography**: A homográfiaillesztést elvégző osztály.
 - Két kép van benne (ezeknek a típusa az OpenCV könyvtár által biztosított képkezeléssel van reprezentálva).
 - A projektív transzformációt egy 3×3 -as mátrixszal ábrázolja, amit egy tömbben tárol.

4.4.2. Dinamikus terv

Egy tipikus használat (körillesztés Levenberg-Marquardt algoritmussal) esetén a program működését mutatja be az alábbi szekvenciadiagram.

4.3. ábra. Szekvenciadiagram körillesztésre



A körillesztő meghívja a megfelelő minimalizáló függvényt. Ez a megfelelő függvénnyel elkészít egy Calculator objektumot. A Calculator a MathCodeGen segítségével legeneráltatja a függvény kódját. Ennek segítségével már képes a lépésközök kiszámítására, amit a Minimizer kérésére többször meg is tesz. Az iteráció végén a minimalizáló visszatér az eredménnyel.

5. Tesztelés

A teszteket az ELTE Informatikai Kar Számítógépes Grafika Laboratóriumában futtattam. A felhasznált számítógép hardverspecifikációja:

CPU	Intel Core i5-4570, 3.20 GHz
GPU	NVIDIA GeForce GTX 960
RAM	8 GB

5.1. Geometriai illesztések

Egy, a módszerek, implementációk tesztelésére jó problémaosztály a körök, ellipszisek mérési, vagy egyéb zajos adatpontokra történő illesztése.

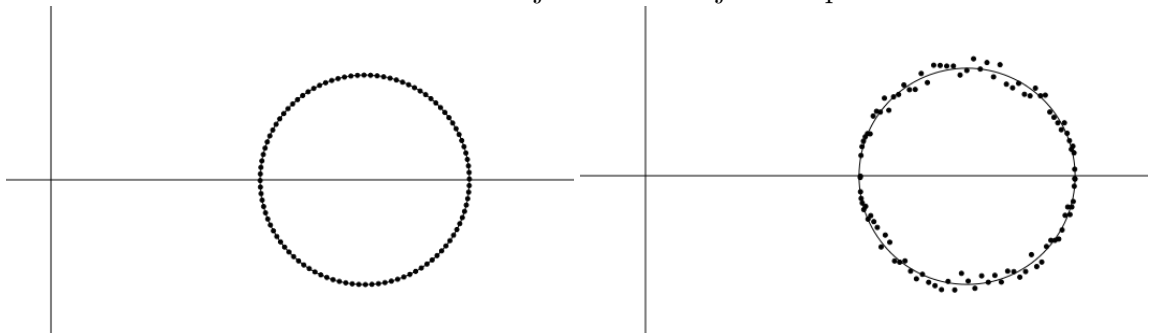
5.1.1. Kör

A körillesztés költségfüggvénye:

$$f(\mathbf{p}, \mathbf{x}) = \sum_{i=1}^n \left((x_i - c_x)^2 + (y_i - c_y)^2 - r^2 \right) \quad (5.1.1.1)$$

, ahol (c_x, c_y) a kör középpontja, r a kör sugara. (x_i, y_i) az i . pont két koordinátája. A kör tehát 3 paraméterrel írható le. $((x_i, y_i)$ az i . pont két koordinátája.).

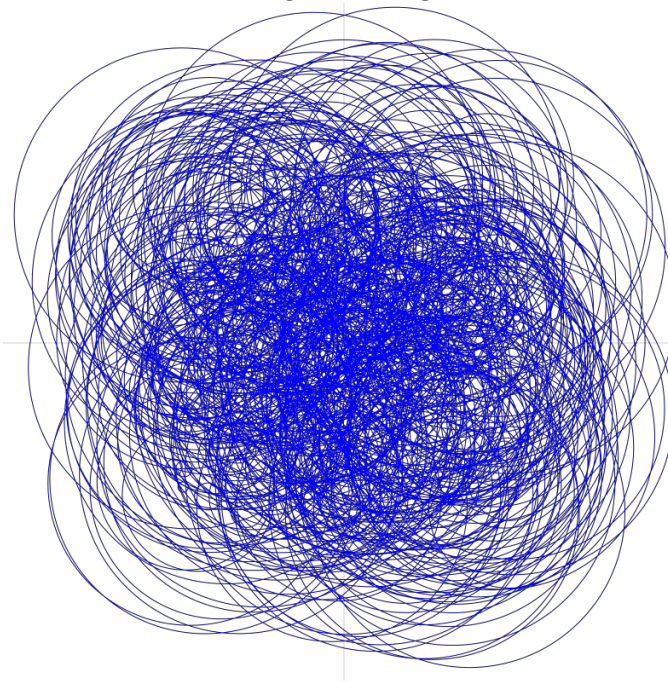
5.1. ábra. Körillesztés zajmentes és zajos adatpontokra



A körillesztés minden kiinduló körből indítva az iterációt konvergált az összes implementált módszerrel. Véletlenszerűen (egyenletes eloszlással) generáltam 100 kört a $[-10; 10]^3$ intervallumból. A köröket a 5.2 ábrán szemléltetem, aszerint színezve, hogy az iteráció végén mekkora hibával sikerült az illesztés.

<0.01	zöld
<0.1	kék
<1	fekete
>1	piros

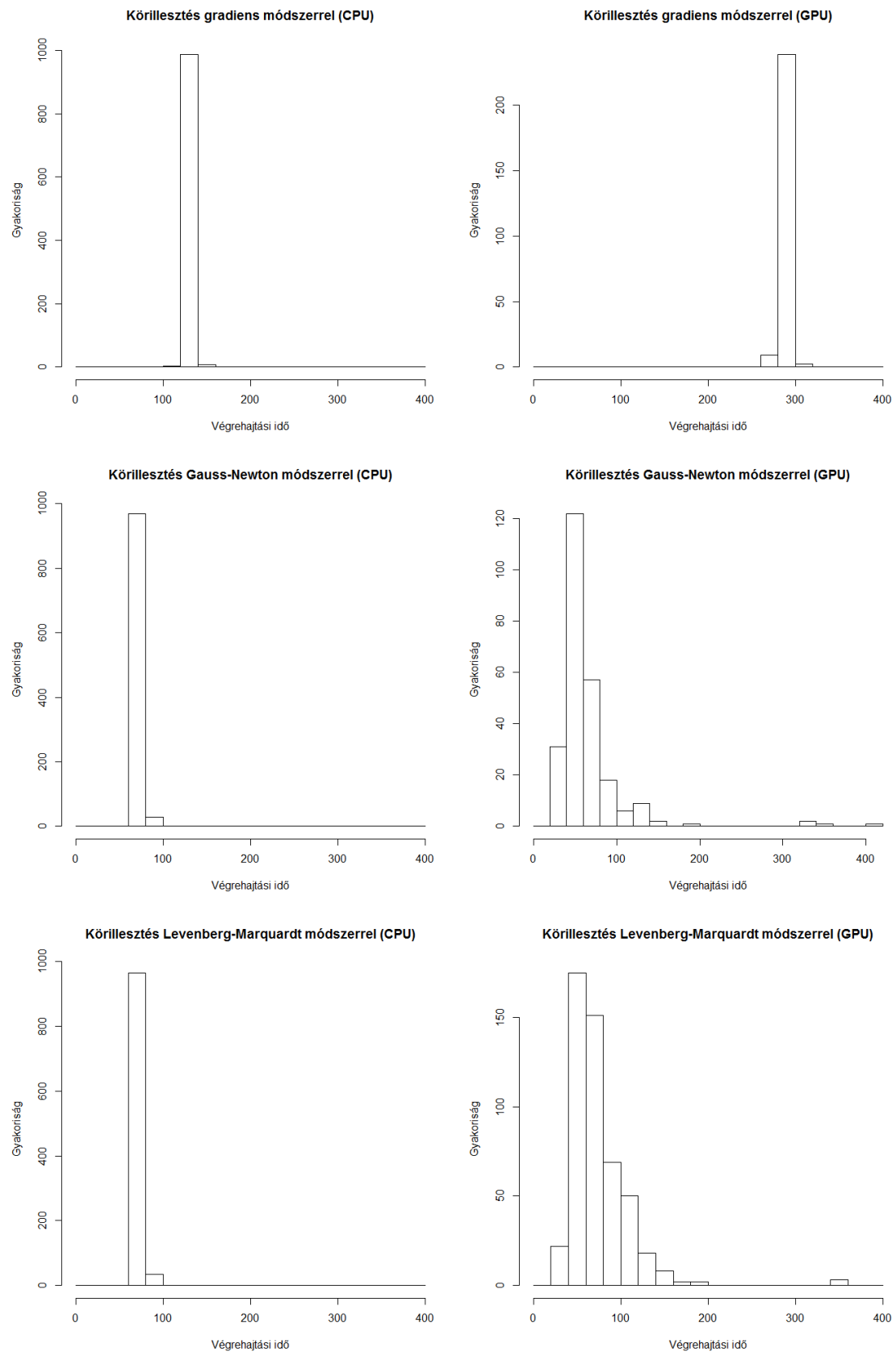
5.2. ábra. Körillesztés konvergenciavizsgálata 1000 kiindulópontból



Az iteráció mindenhol tökéletesen konvergált, ezért az összes kör egyszínű. (A zajmentes esetben zöld, míg a zajos esetben az 5.2 ábrán látható módon kék, ugyanis itt volt egy minimális, tovább nem csökkenthető hiba.)

Megvizsgáltam továbbá az algoritmusok futásidejét 1000 tesztkezdőpontra, külön összevetve a CPU és GPU architektúrákon való futtatásokat, illetve mindhárom algoritmust.

5.3. ábra. Futásidők összevetése körillesztésre: első sor - gradiens módszer, második sor - Gauss-Newton, harmadik sor - Levenberg-Marquardt.



Összefoglalva azt láthatjuk, hogy jelentős javulás ennél a feladatnál a GPU-n való futtatással nem érhető el¹. A gradiens módszer esetén egyenesen romlik a futás-idő a kontextusváltások miatt. Érdekes továbbá, hogy GPU-n futtatva megnő az adatsorok szórása, ez még magyarázatra vár.

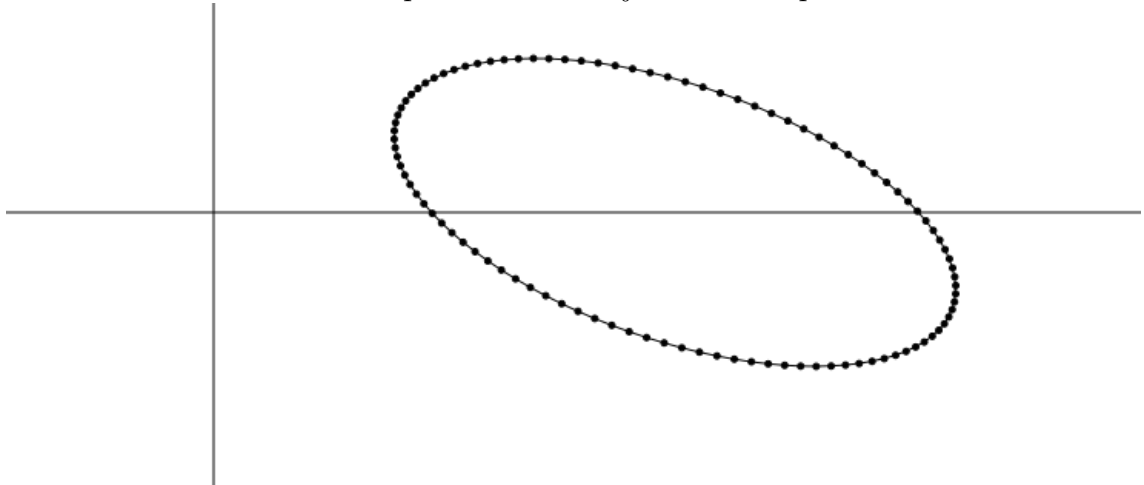
5.1.2. Ellipszis

Az ellipszisillesztésre felírható költségfüggvény²:

$$\sum_{i=1}^n \left(((x_i - c_x) \cos(A) + (y_i - c_y) \sin(A))^2 \frac{b}{a} + ((x_i - c_x) \sin(A) - (y_i - c_y) \cos(A))^2 \frac{a}{b} - ab \right), \quad (5.1.2.1)$$

ahol (c_x, c_y) az ellipszis középpontja, a, b az ellipszis vízszintes és függőleges tengelye, A a vízszintes álláshoz képesti elforgatás szöge. Ezzel az 5 paraméterrel leírható az általános helyzetű ellipszis. (x_i, y_i) az i . pont két koordinátája.

5.4. ábra. Ellipszis-illesztés zajmentes adatpontokra.



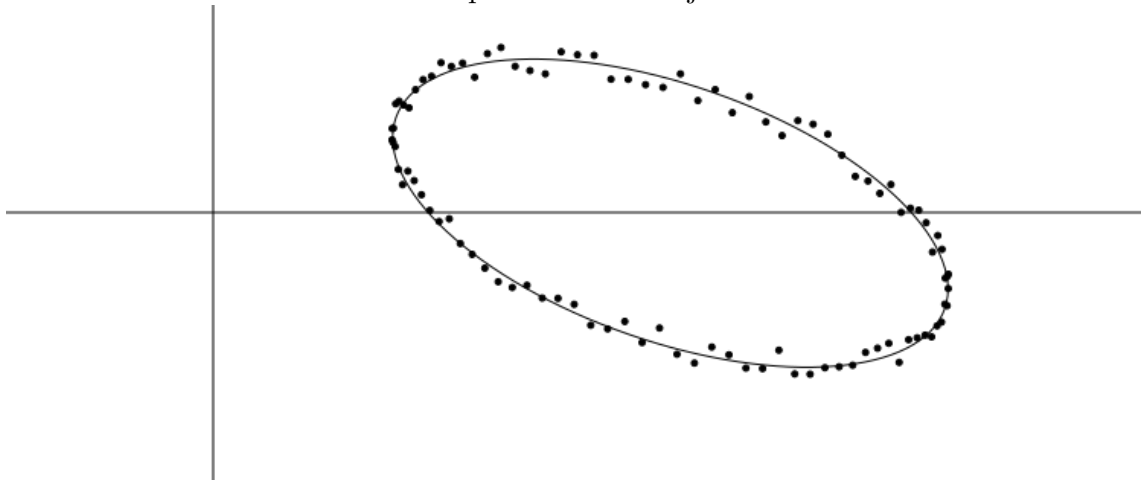
Az illesztések többnyire pontosak, a konvergenciát a 5.6 - 5.11 ábrákon elemzem. Általában a zajmentes esetben a hibafüggvény értéke numerikusan elenyésző, míg a zajos esetben jól simítja a pontokat.

Az ellipszisillesztésre a három módszer már mutat különbségeket a konvergenciára nézve. Külön vizsgáltam a zajmentes és zajos esetet 1000 véletlenszerűen (a

¹Ebben persze annak is szerepe van, hogy már CPU-n futtatva is párhuzamosan fut az algoritmus: a CPU is több magot tartalmaz

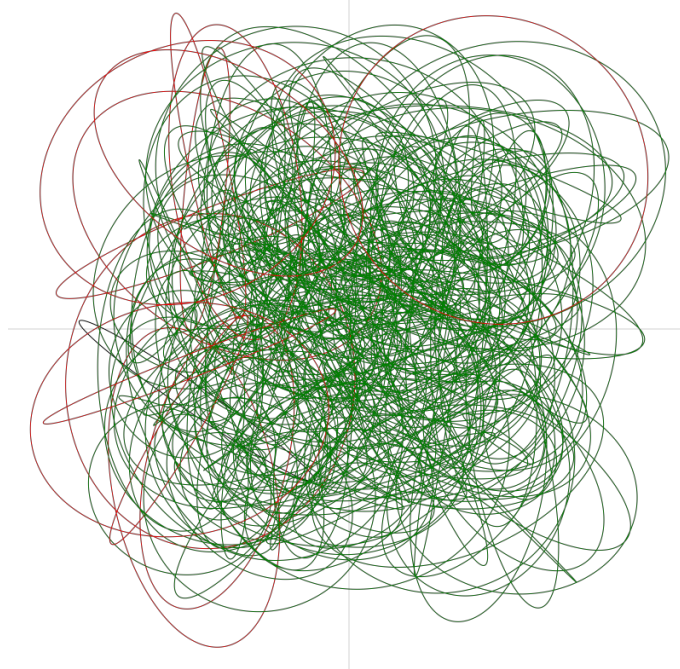
²Korábban a költségfüggvény ab -vel leosztott változatát használtam, de ez túl kis függvényértékeket, ezáltal lassú konvergenciát és pontatlanságokat eredményezett.

5.5. ábra. Ellipszis-illesztés zajos adatokra.



$[-10; 10]^5$ intervallumból) generált kezdőpontból. A kezdőpontok iterációpontosság szempontjából színezve az ábrákon láthatók.

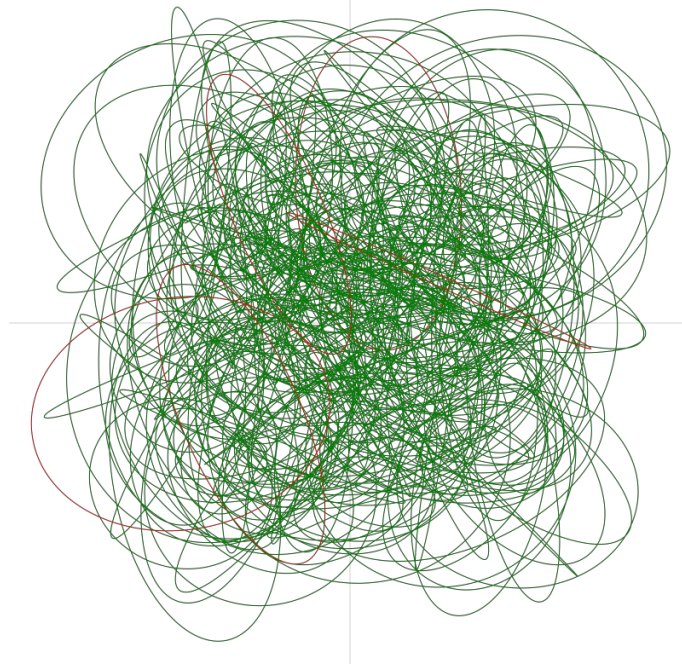
5.6. ábra. Zajmentes ellipszisillesztés pontossága - gradiens módszer



Látható, hogy a zajmentes esetben a módszerek az esetek túlnyomó részében konvergálnak (zöld szín), és csak néhány esetben nem (piros). Ezeknek száma a gradiens módszernél még jelentős, a Gauss-Newtonnál csak néhány van, míg a Levenberg-Marquardt esetén egyetlenegy.

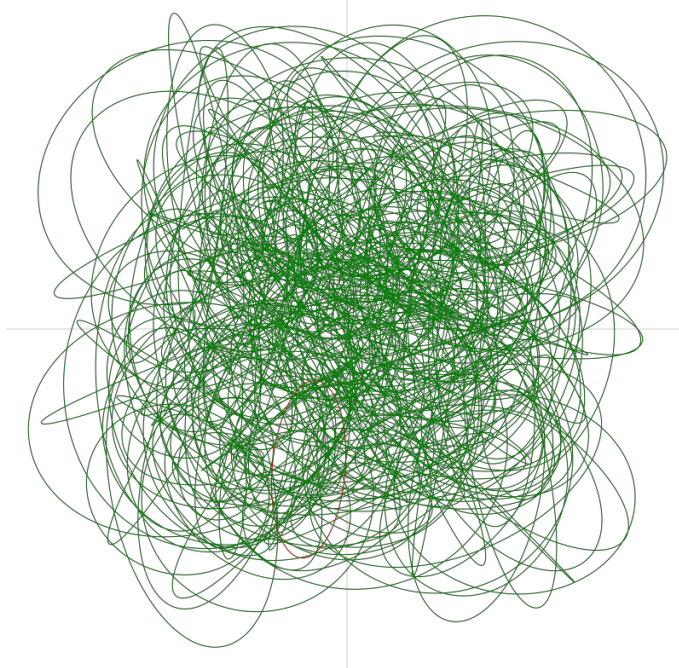
A zajos esetben annyival romlik a helyzet, hogy a hiba már nem tud 0.1 alá csökkenni (ezért a fekete szín), de ez az adatpontok elhelyezkedésének következménye. Némiképp több olyan kezdőpont van, ahonnan nem konvergál az iteráció, de a számuk legalábbis a Levenberg-módszernél még mindig nem jelentős.

5.7. ábra. Zajmentes ellipszisillesztés pontossága - Gauss-Newton módszer

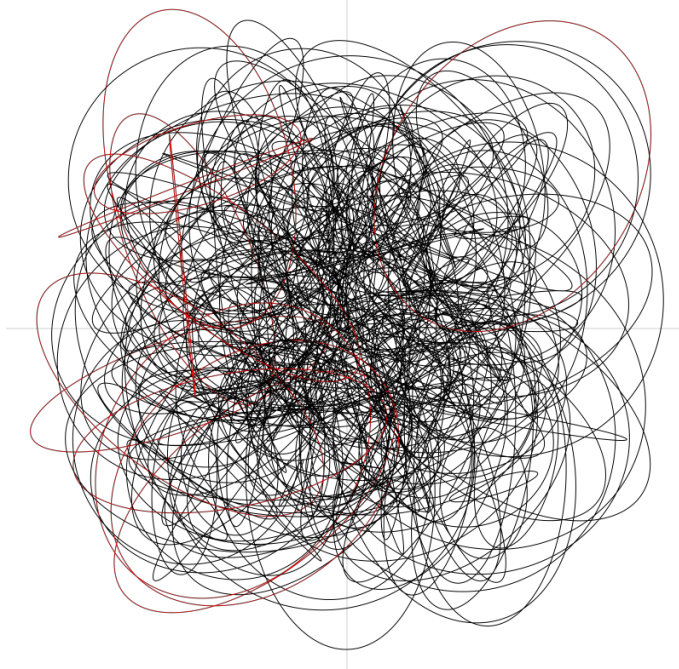


A futásidőket a körillesztéshez hasonlóan mértem. Mindhárom módszerre összevettem a CPU-n és GPU-n futtatott változatokat.

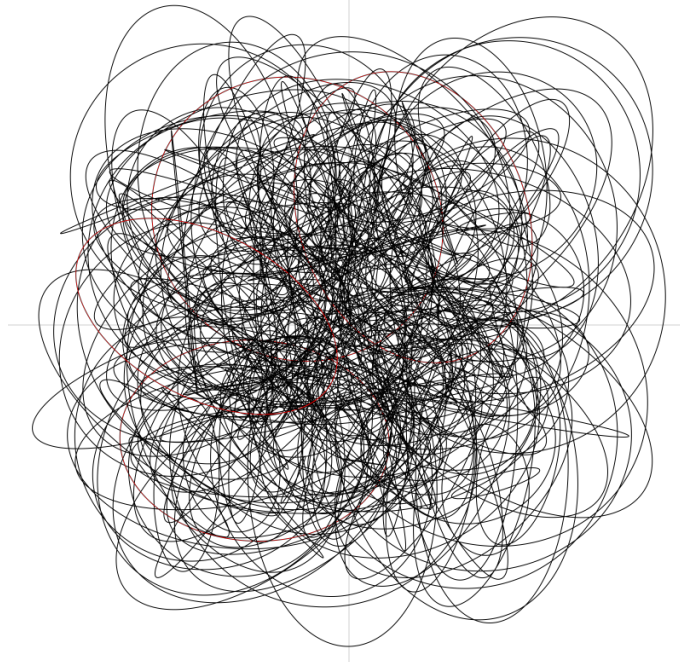
5.8. ábra. Zajmentes ellipszisillesztés pontossága - Levenberg-Marquardt módszer



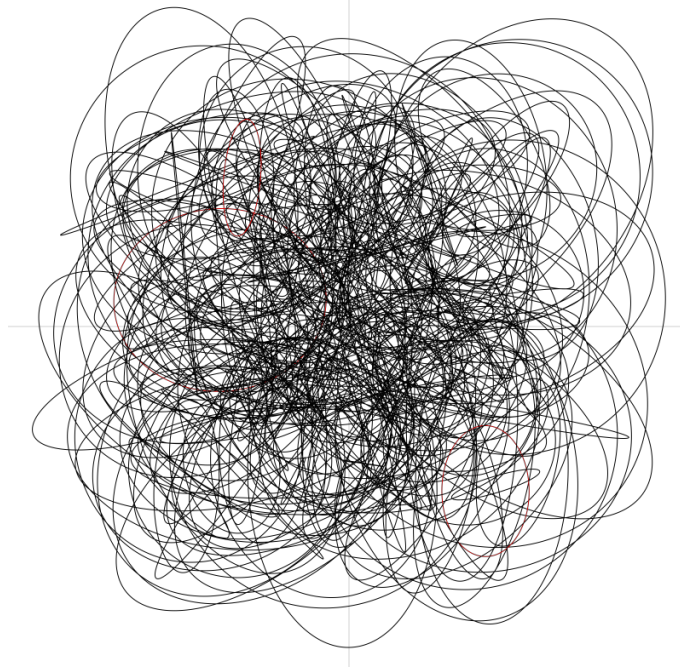
5.9. ábra. Zajos ellipszisillesztés pontossága - gradiens módszer



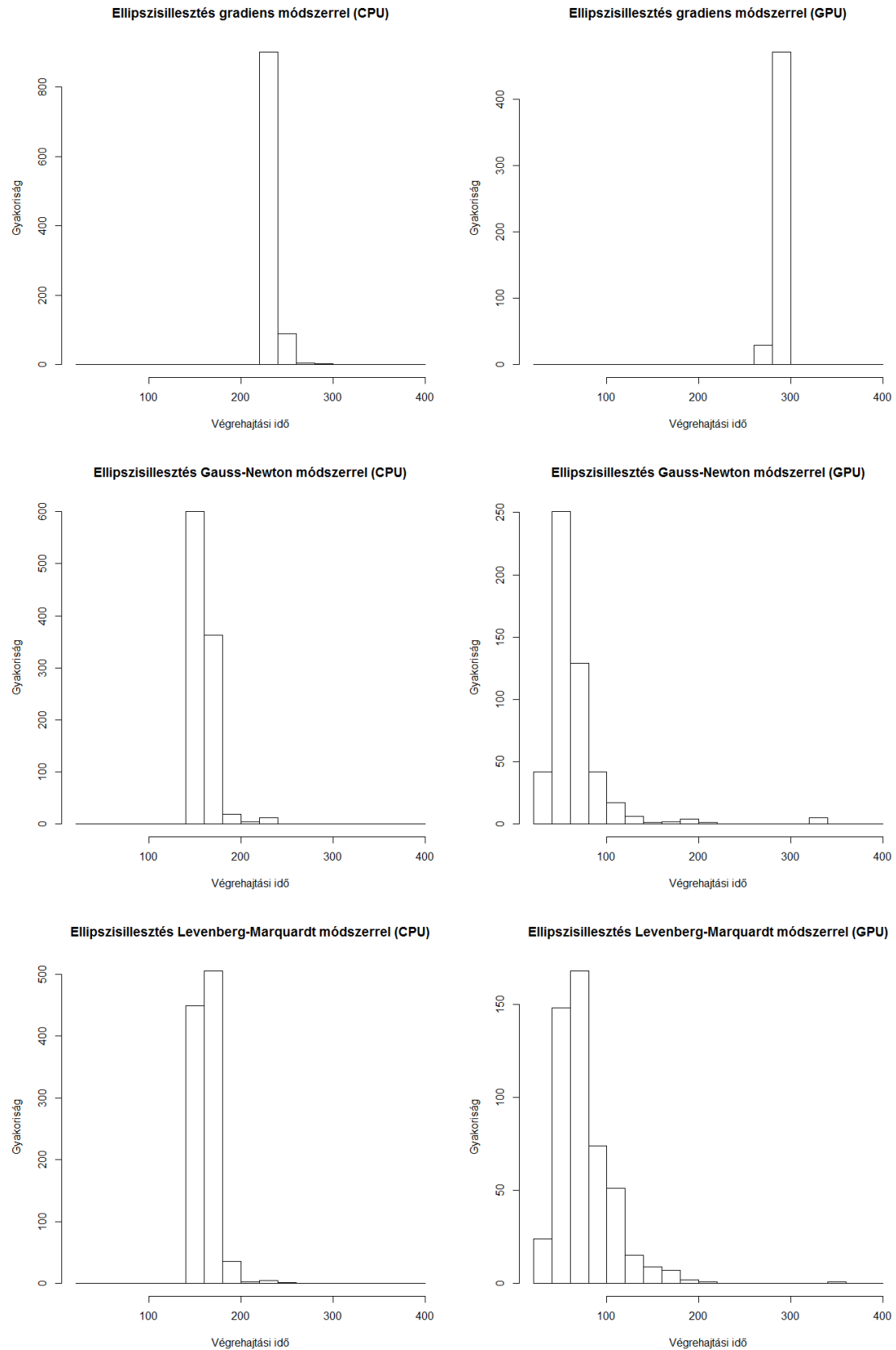
5.10. ábra. Zajos ellipszisillesztés pontossága - Gauss-Newton módszer



5.11. ábra. Zajos ellipszisillesztés pontossága - Levenberg-Marquardt módszer



5.12. ábra. Futásidők összevetése ellipszisillesztésre: első sor - gradiens módszer, második sor - Gauss-Newton, harmadik sor - Levenberg-Marquardt.



Átlagos futásidők:

grad. - CPU	grad. - GPU	GN - CPU	GN - GPU	LM - CPU	LM - GPU
235.6 ms	287.3 ms	161.7 ms	64.7 ms	163.4 ms	75.6 ms

Látható, hogy itt már legalábbis a kifinomultabb módszereknél jól mérhető gyorsulást okozott a párhuzamosítás. Megjegyezhető, hogy a GPU-n futtatott tesztben a kontextusinitializálásnak a mérésekben egy egyszeri, 2383 ms költsége volt.

A Levenberg-Marquardt módszert itt sebességben felülmúlja az egyszerűbb Gauss-Newton, de ezt a megbízhatóságbeli előnye ellensúlyozza.

5.2. Síkhomográfia

A síkhomográfia egy sík két képe közötti projektív térbeli lineáris leképezés. Természetes megközelítésből úgy adódik, hogy két különböző orientációjú kamerával lefényképezünk egy jól felismerhető mintázattal rendelkező síklapot³, majd a feladatunk meghatározni azt a perspektív transzformációt, amely az egyik síklapot a másikba viszi át.

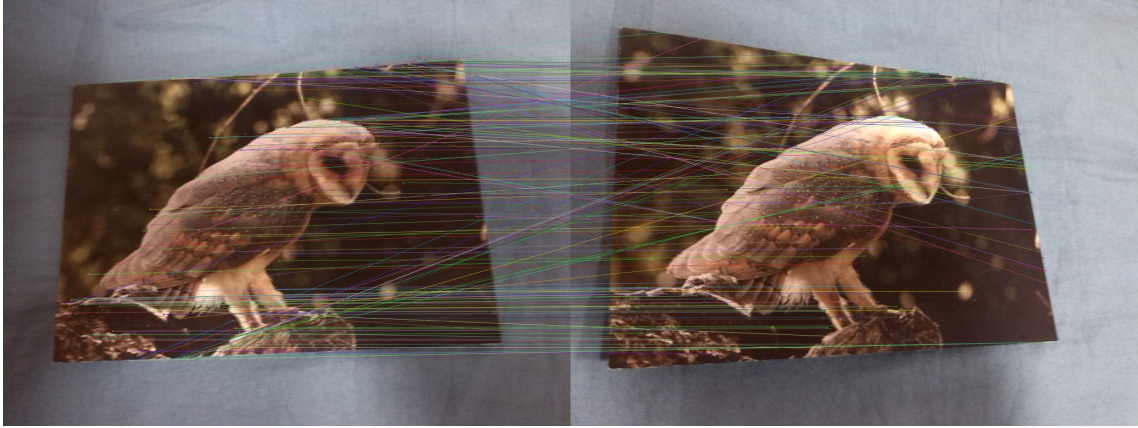
A perspektív transzformációkat leghatékonyabban a homogén koordinátarendszerek segítségével írhatjuk le. Egy kétdimenziós, (u, v) pont homogén koordinátarendszerbeli alakja a $(\lambda u, \lambda v, \lambda)$, $\lambda \in \mathbb{R}$ vektorcsalád tetszőleges eleme. A pontok ilyen felírásával a perspektív transzformációk 3×3 -as mátrixokkal megadhatók. Meggondolható, hogy mivel a homogén koordináták valós számmal szorozva is ugyanazt a pontot határozzák meg, ezért a transzformációs mátrix sem lesz érzékeny a skálázásra, azaz 8 paraméterrel leírható.

A függvényoptimalizációhoz szükségünk van adatpontokra is. Meg kell adnunk olyan pontpárokat a két képen, hogy az egyik pont homográfia szerinti képe a másik. Erre léteznek különböző megvalósítások, én az OpenCV [11] könyvtár által biztosított *Feature Detection* algoritmusokat használtam fel [12]. Ez nagyszámú pont meghatározására képes, azonban sajnos természetesen hibás felismeréseket is elkövet.

Az OpenCV háromféle algoritmust biztosít arra, hogy kezdeti becslést adjon a homográfiára: az egyszerű illesztést, melyben minden pontpárt figyelembe vesz, a RANSAC [13, 14] algoritmust, amely nemdeterminisztikus kísérletezéssel próbál megszabadulni a hibás párosításoktól, illetve a Least Median [15] módszert, amely csak a legjobb néhány pontot veszi figyelembe. Mindhárom módszerrel adtam kezdeti homográfiabecsléseket, majd ezeket használtam az iterációs módszerek kezdőpontjaként.

³Például sakktábla, plakát

5.13. ábra. Detektált pontpárok két képen szűrés nélkül



Kétféle lehetőséget biztosítok a rossz párosítások eldobására: az OpenCV becslő algoritmusát, illetve egy geometriai távolságon alapulót, amely a pontpárt akkor tartja meg, ha a becsült homográfiával transzformált és a tényleges pont közötti euklideszi távolság kisebb, mint egy korlát.

5.14. ábra. Szűrt pontpárok



A költségfüggvény ekkor az alábbiak szerint írható fel. Legyen $\mathbf{x}_i = (x_{i,1}, x_{i,2})$ és $\mathbf{y}_i = (y_{i,1}, y_{i,2})$ az i . felismert pontpár. Legyen továbbá $H \in R^{3 \times 3}$ az a homográfia, amely $\mathbf{x}_i$ -t $\mathbf{y}_i$ -be viszi (homogén koordináták alakban)

$$f(\mathbf{p}, \mathbf{x}, \mathbf{y}) = \sum_{i=1}^n \left\| \text{trf}(H, \mathbf{x}_i) - \mathbf{y}_i \right\|_2^2 \quad (5.2.0.1)$$

Itt a $\text{trf}(H, \mathbf{x}_i)$ a perspektív transzformáció elvégzését jelenti, azaz:

- $\mathbf{x}_i$ kiegészítését egy 1-essel,
- ennek H -val való megszorzását,
- majd a harmadik koordinátával való leosztást (homogén osztás)

Tehát a költségfüggvény megadja a transzformált és tényleges pontok euklideszi távolságának négyzetösszegét, ennek minimalizálása logikus célkitűzés.

A becslések minőségét a jobb oldali képre illesztett zöld kerettel szemléltetem, ami a bal oldali kép sarokpontjainak a számított homográfiával való transzformálásából adódik.

A másik szemléltetés az átalakított képek megjelenítése lesz, az OpenCV lehetőséget ad arra, hogy egy adott homográfiamátrixszal transzformáljon egy képet. Így összevethető a második fotó az elsőből transzformált ábrával.

Amikor az OpenCV-re hagyatkoztam a pontok eldobásában is, akkor az a furcsa eredmény lépett fel, hogy a finomítás során nem tudott jó lépést tenni egyik algoritmus sem. Ez egyelőre tisztázatlan, hogy hogyan történhet, hiszen a kezdeti becslést tevő algoritmusok elméletileg csak lineáris becslést végeznek. Ezért inkább a saját ponteldobó módszeremmel mutatom be az eredményeket.

A RANSAC algoritmus egy nagyon jó kezdeti becslést adott (5.15 ábra). A hiba-függvény kezdeti értéke RMSE felírásban 13.7122 volt (ez lényegében az átlagos, pixelben mért távolság a transzformált és a valódi pontok között).

5.15. ábra. Kezdeti homográfiabecslés RANSAC algoritmussal



A kezdeti hibát a Levenberg-módszerrel 13.6625-re sikerült csökkenteni. Az eredmény a 5.16 ábrán látható.

A különbség alig látható, de közelről megvizsgálva az ábrákat az látható, hogy a bal felső saroknál némi javulás történt, míg a jobb alsónál némi romlás a sarok illesztésének pontosságát tekintve. Ennek oka az, hogy a detektált pontok a képen nem egyenletesen helyezkednek el, így mivel a jobb alsó sarok közelében nincsenek pontok, az ottani becslés pontossága kevésbé hangsúlyos.⁴

⁴Megpróbáltam keresni olyan képpárt, amin a detektált pontok egyenletes eloszlásúak, de ennél jobbat nem találtam.

5.16. ábra. Levenberg-Marquardt finomítás RANSAC-cal adott kezdeti becslésre



Az eredeti és a transzformált kép összehasonlítása a 5.17 ábrán látható. A két kép meglehetősen jól illeszkedik egymásra (a környezetükre pedig ezt nem is garantálja a homográfia).

5.17. ábra. Eredeti és transzformált kép összehasonlítása



Ennek a módszerpárnak a futásideje:

	CPU	GPU
Teljes futás	3220 ms	2969 ms
Ebből optimalizációs lépés	270 ms	23 ms

Látható tehát, hogy a futásidő túlnyomó részét az OpenCV különböző algoritmusai tették ki, a finomítás ehhez képest (különösen párhuzamos futtatás esetén) csekély többletet jelent.

A többi módszerrel kapcsolatban:

- A gradiens módszer kifejezetten instabillá vált, és a már a kezdeti becslésben is meglevő jó eredményeket is elrontotta. Ezért részletesebben nem tárgyalnám.
- A Gauss-Newton teljesen hasonló eredményeket adott, mint a Levenberg-Marquardt, és a sebességbeli különbségek is hibahatáron belül vannak.

- Az egyszerű illesztés annyira rossz kezdeti becslést adott, hogy el kellett dobni a jó pontok nagy részét is, majd a fennmaradó 5 db, egyenetlen eloszlású pontpárra való optimalizálás hiába vitte le a hibafüggvény értékét 75-ről akár 5-re, az eredmény nagyon pontatlan volt. Lásd 5.18 ábra.
- A Least Median módszer egy kevésbé adott rosszabb becslést, mint a RAN-SAC, és ezt ahhoz hasonlóan is sikerült „javítani”, azaz a több pontot tartalmazó részeket fejleszteni, a nem szignifikáns részeket nem figyelembe véve.

5.18. ábra. Egyszerű illesztés



Mindazonáltal ez nem eredményezett szemmel érzékelhető javulást, melynek egyik fő oka feltehetően a képeken detektált pontok nem egyenletes eloszlása. Az eredmények javulása érdekében jövőbeli munkaként mindenféleképpen érdemes felhasználni az adatnormalizálás lehetőségét és technikákat az outlier-adatpontok szűrésére.

6. Konklúzió

6.1. Összegzés

Elkészítettem egy OpenCL backendet felhasználó numerikus optimalizáló rendszert. A rendszerben elegendő csupán a minimalizálandó a költségfüggvényeket deriváltak nélkül megadni, mely lehetővé teszi a probléma gyors felvázolását (fast prototyping), jelentősen csökkentve a fejlesztésre fordított időt. Emellett, a GPU-kban rejlő teljesítmény kiaknázásával futásidő-csökkenésre is számíthatunk.

A programot háromféle gyakorlati problémán futtattam, ebből a kör-, illetve ellipszisillesztést pontos eredményekkel megoldotta, míg a homográfiaillesztésben részleges sikert ért el.

Az elkészített programnak természetesen vannak korlátai is: csak olyan függvények megadását támogatja, ami a C nyelv könyvéraiban adott matematikai függvények véges számú alkalmazásával megadható, egy függvénytárral, vagy más iterációval megadott nem elemi függvény megadása problémákat jelent.

6.2. Fejlesztési lehetőségek

Célom a program továbbfejlesztése további számítógépes látással kapcsolatos problémák megoldására. A legközelebbi ilyen probléma a kamerakalibráció becslése lesz. További célok az implementáció hatékonyságának további növelése elsősorban az iterációs algoritmusok jobb heurisztikákkal való támogatása kapcsán, illetve más típusú költségfüggvényekkel történő optimalizálás.

Köszönetnyilvánítás

Szeretném megköszönni témavezetőmnek, Eichhardt Ivánnak a remek témafelvetést, és a sok segítséget, amit a dolgozat elkészítése során nyújtott.

Szintén hálával tartozom az Eötvös Loránd Tudományegyetem Informatikai Karának, hogy az itt töltött félévek alatt elsajátíthattam az elengedhetetlenül szükséges matematikai és informatikai alapokat.

A. DVD melléklet

Az alábbiakban összefoglalom a mellékelt DVD-n található állományokat, céljukat és használatukat.

A.1. ábra. A DVD gyökérkönyvtára

	build	2016.05.12. 19:19	Fájlmappa
	dependencies	2016.05.12. 19:19	Fájlmappa
	docs	2016.05.12. 19:19	Fájlmappa
	source	2016.05.12. 19:19	Fájlmappa
	tests	2016.05.12. 19:19	Fájlmappa

A *build* könyvtárban található meg a **lefordított program**, a futtatáshoz szükséges *.dll* kiterjesztésű könyvtárakkal, illetve a futtató *start.bat* állománnyal.

A.2. ábra. A build könyvtár tartalma

	AutoDiff.exe	2016.05.10. 22:41	Alkalmazás	305 KB
	boost_date_time-vc120-mt-1_60.dll	2016.03.16. 17:26	Alkalmazáskiterjes...	40 KB
	boost_filesystem-vc120-mt-1_60.dll	2016.03.16. 17:26	Alkalmazáskiterjes...	106 KB
	boost_system-vc120-mt-1_60.dll	2016.03.16. 17:26	Alkalmazáskiterjes...	17 KB
	freeglut.dll	2015.11.12. 20:21	Alkalmazáskiterjes...	272 KB
	glew32.dll	2015.11.12. 20:21	Alkalmazáskiterjes...	200 KB
	opencv_calib3d2412.dll	2015.08.02. 7:54	Alkalmazáskiterjes...	961 KB
	opencv_core2412.dll	2015.08.02. 7:51	Alkalmazáskiterjes...	2 090 KB
	opencv_features2d2412.dll	2015.08.02. 7:53	Alkalmazáskiterjes...	706 KB
	opencv_flann2412.dll	2015.08.02. 7:51	Alkalmazáskiterjes...	513 KB
	opencv_highgui2412.dll	2015.08.02. 7:52	Alkalmazáskiterjes...	2 097 KB
	opencv_imgproc2412.dll	2015.08.02. 7:52	Alkalmazáskiterjes...	1 848 KB
	opencv_ml2412.dll	2015.08.02. 7:51	Alkalmazáskiterjes...	496 KB
	opencv_nonfree2412.dll	2015.08.02. 7:55	Alkalmazáskiterjes...	544 KB
	opencv_objdetect2412.dll	2015.08.02. 7:52	Alkalmazáskiterjes...	640 KB
	opencv_ocl2412.dll	2015.08.02. 7:54	Alkalmazáskiterjes...	2 053 KB
	start.bat	2016.05.10. 20:12	Windows kötegfájl	1 KB

A *source* könyvtárban található a program forráskódja, Visual Studio 2013 projekt fájl formájában. A program C++ nyelven íródott, a forrásfájlok egy mappával beljebb találhatóak meg.

A.3. ábra. A forráskód szervezése

AutoDiff	2016.05.12. 19:20	VC++ Project	10 KB
AutoDiff.sln	2016.05.11. 14:18	VC++ Project Filte...	2 KB
README.md	2016.05.11. 14:18	Visual Studio Proj...	2 KB
AutoDiff.vcxproj	2016.05.11. 14:18	C/C++ Header	9 KB
AutoDiff.vcxproj.filters	2016.05.11. 14:18	C/C++ Header	2 KB
AutoDiff.vcxproj.user	2016.05.11. 14:18	C/C++ Header	8 KB
Calculation.hpp	2016.05.11. 14:18	C++ Source	16 KB
Circle.hpp	2016.05.11. 14:18	C/C++ Header	3 KB
dualnumber.hpp	2016.05.11. 14:18	C/C++ Header	3 KB
Fitting.cpp	2016.05.12. 0:10	C/C++ Header	2 KB
Fitting.h	2016.05.11. 14:18	C++ Source	10 KB
Functions.hpp	2016.05.11. 14:18	C/C++ Header	7 KB
Homography.hpp	2016.05.11. 14:18	C/C++ Header	8 KB
main.cpp	2016.05.11. 14:18	C/C++ Header	3 KB
MathCodeGen.hpp	2016.05.11. 14:18	C/C++ Header	3 KB
Minimize.hpp	2016.05.11. 14:18	C/C++ Header	3 KB
TestTools.hpp	2016.05.11. 14:18	C/C++ Header	3 KB
utils.h	2016.05.11. 14:18	C/C++ Header	3 KB

A *dependencies* könyvtárban van a program két fő függősége, az OpenCV könyvtár, illetve az OCLPack gyűjtemény, amiben az OpenCL-es könyvtárakat gyűjtöttem össze: OpenCL, VexCL, ViennaCL, illetve a Boost könyvtár ezekhez szükséges részei. A könyvtárak elérési útvonala a projektfájlok által való megtalálásra a `mount_t.bat` parancsfájllal állítható be.

A *tests* könyvtárban néhány tesztfájl található: a *points* mappában illesztésekhez használható adatpontokat tartalmazó fájlok, a *pictures* mappában egy homográfia-illesztéshez használható képpár.

A *docs* mappában ennek a dokumentumnak az elektronikus változata található meg.

Ábrák jegyzéke

2.1. Use-case-ek összefoglalása	6
4.1. A parciális deriváltak számításának modellje.	19
4.2. A program osztályainak kapcsolatai	21
4.3. Szekvenciadiagram körillesztésre	23
5.1. Körillesztés zajmentes és zajos adatpontokra	24
5.2. Körillesztés konvergenciavizsgálata 1000 kiindulópontból	25
5.3. Futásidők összevetése körillesztésre.	26
5.4. Ellipszis-illesztés zajmentes adatpontokra.	27
5.5. Ellipszis-illesztés zajos adatokra.	28
5.6. Zajmentes ellipszisillesztés pontossága - gradiens módszer	28
5.7. Zajmentes ellipszisillesztés pontossága - Gauss-Newton módszer	29
5.8. Zajmentes ellipszisillesztés pontossága - Levenberg-Marquardt módszer	30
5.9. Zajos ellipszisillesztés pontossága - gradiens módszer	30
5.10. Zajos ellipszisillesztés pontossága - Gauss-Newton módszer	31
5.11. Zajos ellipszisillesztés pontossága - Levenberg-Marquardt módszer	31
5.12. Futásidők összevetése ellipszisillesztésre.	32
5.13. Detektált pontpárok két képen szűrés nélkül	34
5.14. Szűrt pontpárok	34
5.15. Kezdeti homográfiabecslés RANSAC algoritmussal	35
5.16. Levenberg-Marquardt finomítás RANSAC-cal adott kezdeti becslésre	36
5.17. Eredeti és transzformált kép összehasonlítása	36
5.18. Egyszerű illesztés	37
A.1. A DVD gyökérkönyvtára	40
A.2. A build könyvtár tartalma	40
A.3. A forráskód szervezése	41

Irodalomjegyzék

- [1] Kaj Madsen, Hans Bruun Nielsen, and Ole Tingleff. Methods for non-linear least squares problems. 2004.
- [2] Jeffrey A. Fike, Juan J. Alonso. The Development of Hyper-Dual Numbers for Exact Second Derivative Calculations. 2011.
- [3] John E. Stone, David Gohara, Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. 2010.
- [4] Grewe, Dominik and Lokhmotov, Anton. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 12. ACM, 2011.
- [5] Sameer Agarwal and Keir Mierle and Others. Ceres Solver. <http://ceres-solver.org>.
- [6] Simon Péter. *Bevezetés az analízisbe I*. 2013.
- [7] Moré, Jorge J. The Levenberg-Marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.
- [8] Demidov, Denis and Ahnert, Karsten and Rupp, Karl and Gottschling, Peter. Programming CUDA and OpenCL: A case study using modern C++ libraries. *SIAM Journal on Scientific Computing*, 35(5):C453–C472, 2013.
- [9] VexCL documentation. <http://vexcl.readthedocs.io/>. Elérés: 2016-04-23.
- [10] Rupp, Karl and Rudolf, Florian and Weinbub, Josef. ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs. In *Intl. Workshop on GPUs and Scientific Applications*, pages 51–56, 2010.
- [11] Gary Bradski, Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.
- [12] OpenCV feature detection tutorial. http://docs.opencv.org/2.4/doc/tutorials/features2d/feature_detection/feature_detection.html. Elérés: 2016-04-23.

- [13] Fischler, Martin A and Bolles, Robert C. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.
- [14] Konstantinos G Derpanis. Overview of the RANSAC Algorithm. *Image Rochester NY*, 4:2–3, 2010.
- [15] Massart, Desire L and Kaufman, Leonard and Rousseeuw, Peter J and Leroy, Annick. Least median of squares: a robust method for outlier and model error detection in regression and calibration. *Analytica Chimica Acta*, 187:171–179, 1986.