Eötvös Loránd Tudományegyetem

Informatikai kar

Algoritmusok és Alkalmazásaik Tanszék

TDK dolgozat

Nemlineáris optimalizálási problémák párhuzamos megoldása grafikus processzorok felhasználásával

Témavezető:
Eichhardt Iván
Doktorandusz

ELTE-IK

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

Budapest, 2016

Tartalomjegyzék

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. Irodalmi áttekintés

A nemlineáris optimalizálás [1] egy alkalmazások szempontjából fontos téma. Számítógépes implemantációkban 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 találkoztam. 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 Numerikus eljárások

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

2.1. Matematikai alapfogalmak

Az alábbiakban megemlítek néhány szükséges analízisbeli fogalmat, tételt.

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

2.1.2. 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]

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

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

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

2.2. 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

2.2.1. 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 szerint[1]:

$$(a,\ b) + (c,\ d) = (a + c,\ b + d) \tag{2.2.1.1}$$

$$(a,\ b) \cdot (c,\ d) = (a \cdot c,\ a \cdot d + b \cdot c) \tag{2.2.1.2}$$

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

2.2.1.3. 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 $f : \mathbb{R} \to \mathbb{R}$ 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 $\hat{f} : \mathbb{D} \to \mathbb{D}$ függvényt, ami az f-ével megegyező hatványsor által definiált, a duális számokon értelmezett analitikus függvény.

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

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

A tétel az 2.1.2 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 $\hat{f}((a,1)) = (f(a), f'(a))$.

2.2.2. Automatikus differenciálás többváltozós esetben

Legyen most $f : \mathbb{R}^n \to \mathbb{R}$. 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 $\hat{f}_i : \mathbb{D}^n \to \mathbb{D}, (i \in \{1, ..., n\})$ 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:

$$\mathbf{x} = ((a_0, 0), ..., (a_i, 1), ..., (a_n, 0)) \tag{2.2.2.1}$$

2.3. 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:

$$f(\mathbf{p},\mathbf{x}) = \sum_{i=0}^{n} f_i(\mathbf{p}, \mathbf{x_i}) \tag{2.3.0.1}$$

, ahol $\boldsymbol{p}$ 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 ismételjük.

2.1.1 alapján a 2.3.0.1 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].

2.3.1. Gradiens módszer

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

$$\mathbf{x_{i+1}} = \mathbf{x_i} - \alpha_i \cdot f'(\mathbf{x_i}), \quad (\alpha_i \in \mathbb{R},\ \alpha_i > 0) \tag{2.3.1.1}$$

- Az α_i valós paraméter megválasztására a módszer nem ad egyértelmű választ, különböző stratégiák léteznek. Erősen feladatfüggő, hogy milyen módszert érdemes választani.
  - Lehet konstans, esetleg néhány lépésenként csökkentett konstans
  - Felhasználhatjuk a gradiens normálására, hiszen a gradiens nagysága valójában semmit sem mond a szélsőértékhelyek távolságáról

2.3.2. Gauss-Newton módszer

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

A 2.1.3 tételből levezethető (a harmadrendű maradéktag elhanyagolásával), hogy:

$$\delta\mathbf{x} = -(f''(\mathbf{x}))^{-1} \cdot f'(\mathbf{x}) \tag{2.3.2.1}$$

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 $F = [f_1, f_2, ..., f_m]$ 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 2.3.2.1 ezen közelítésével az alábbi iterációhoz jutunk[2]:

$$\mathbf{x}_{i+1} = \mathbf{x_i} - \left((F'(\mathbf{x_i}))^T \cdot (F'(\mathbf{x_i})) \right)^{-1} \cdot (F'(\mathbf{x_i}))^T \cdot F(\mathbf{x_i}) \tag{2.3.2.2}$$

Az implementáció során természetesen az itt található számításokra (mátrixszorzás, lineáris egyenletrendszer megoldás[3]) figyelemmel kell lennünk.

2.3.3. 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, méghozzá úgy, hogy egy az iteráció során változó $\lambda > 0$ valós paraméter használatával a LER mátrixához hozzáad $\lambda \cdot E$-t[4]. Tehát az iteráció:

$$\mathbf{x}_{i+1} = \mathbf{x_i} - \left((F'(\mathbf{x_i}))^T \cdot (F'(\mathbf{x_i})) + \lambda \cdot E \right)^{-1} \cdot (F'(\mathbf{x_i}))^T \cdot F(\mathbf{x_i}) \tag{2.3.3.1}$$

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.

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

[3]Numerikus pontossággal kapcsolatos meggondolásokból az invertálást LER megoldással váltjuk ki.

[4]E a megfelelő méretű négyzetes egységmátrix

3 Implementáció

3.1. 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 hardverfüggő, 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).

3.1.1. 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.

3.2. Kódgenerálás

A célom az volt, hogy egy C++-ban megadott függvényhez (itt függvény alatt matematikai függvényt értek, lásd 2.3.0.1) 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 2.2.1.1, 2.2.1.2 és 2.2.1.4 szerint. (Kihasználom, hogy ezek elemi függvények, a deriváltjaik ismertek.)

  3.2.1. 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.

  3.2.2. 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 kódgenerálókból álló duális számon* (a korábban említettek alapján ($"x"$, "1") bemenő értékkel), akkor a kimenő érték

a ($\langle$*függvényérték kódja*$\rangle$, $\langle$*derivált kódja*$\rangle$) pár lesz. Tehát megkapjuk azt a két kifejezést, amit kiértékelve a fv.érték, 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.

3.2.3. 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))));
}
```

[1] Többváltozós esetben persze a 2.2.2.1 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.

3.3. Párhuzamosítás

Korábban említettem (2.3. szakasz), 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 memóriafoglalások 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, objektumorientált kezelését. Például egy *vex::vector* az *std::vector*-hoz hasonló interfaceszel rendelkezik, de megvalósítását tekintve a tárolt elemeket a felhasznált hardverek memóriájában tárolja.

3.3.1. Gradiens módszer párhuzamos implementációja

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

azonban hardveresen gyorsítható a szummás kifejezés tagonkénti kiértékelésével, majd az összegzés GPU oldali elvégzésével[2].

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 *vex::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 *vex::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 3.1 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 lépésközről.

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.

3.3.2. Gauss-Newton módszer párhuzamos implementációja

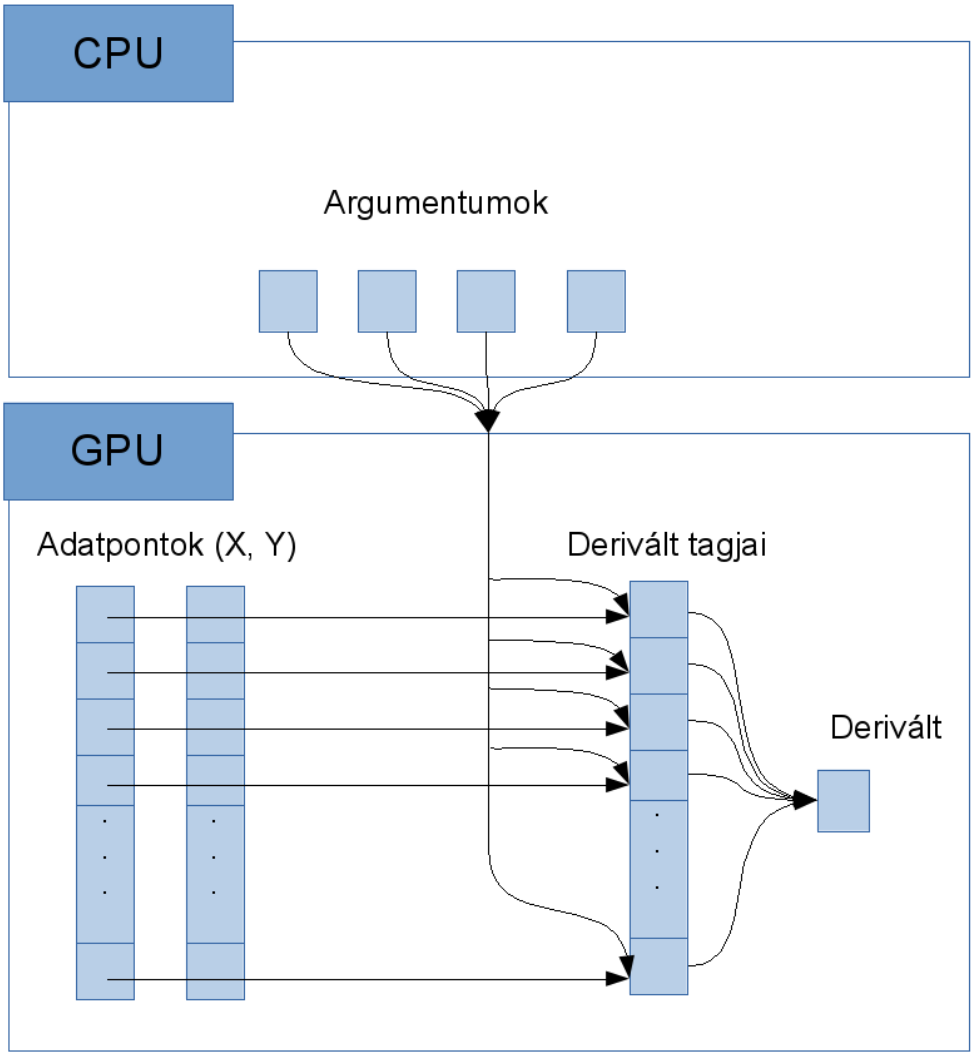
Emlékeztető: a végzendő iteráció:

$$\mathbf{x}_{i+1} = \mathbf{x_i} - \left((F'(\mathbf{x}))^T \cdot (F'(\mathbf{x})) \right)^{-1} \cdot (F'(\mathbf{x}))^T \cdot F(\mathbf{x}) \tag{3.3.2.1}$$

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.

[2]Párhuzamos környezetben az összegzés O(log(n)) műveletigénnyel elvégezhető

3.1. 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.

3.3.3. Levenberg-Marquardt módszer párhuzamos implementációja

A Levenberg-Marquardt módszer iterációja:

$$\mathbf{x}_{i+1} = \mathbf{x_i} - \Big((F'(x))^T \cdot (F'(x)) + \lambda \cdot E\Big)^{-1} \cdot (F'(x))^T \cdot F(x) \tag{3.3.3.1}$$

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 Példák, esettanulmányok

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

4.1. Geometriai illesztések

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

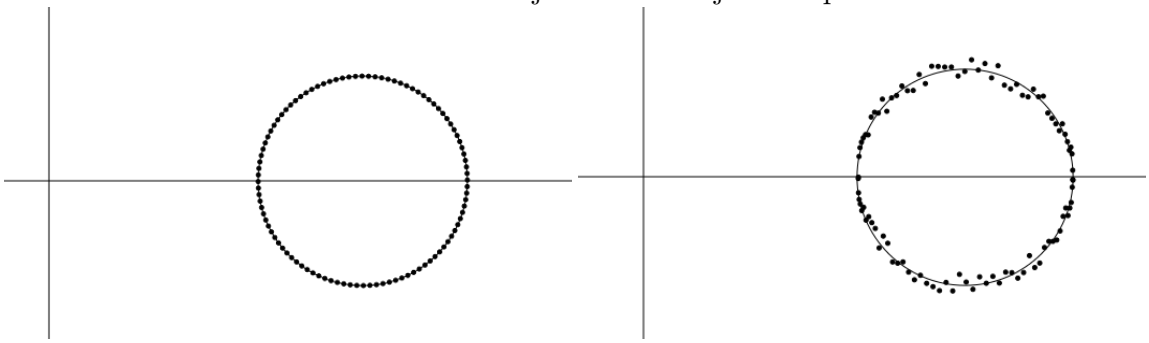
4.1.1. Kör

A körillesztés költségfüggvénye:

$$f(\mathbf{p},\, \mathbf{x}) = \sum_{i=1}^{n} \Big((x_i - c_x)^2 + (y_i - c_y)^2 - r^2 \Big) \tag{4.1.1.1}$$

, 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.).

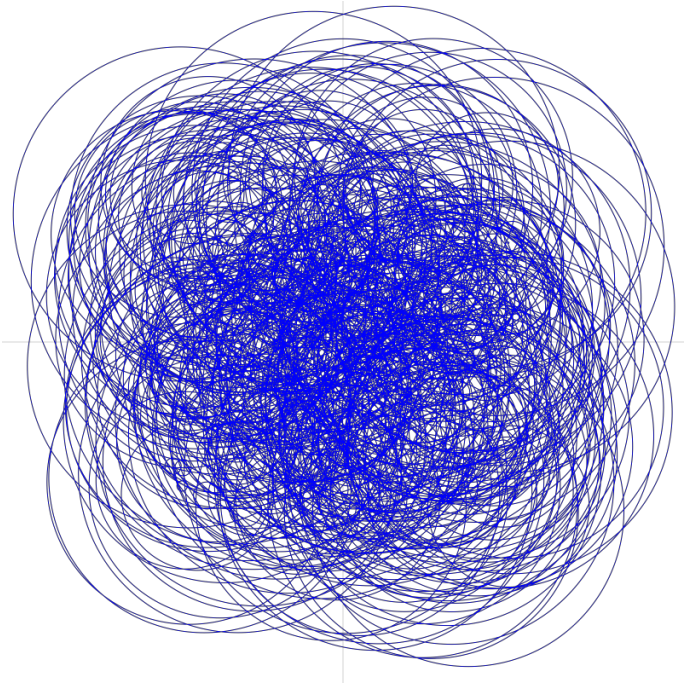
4.1. ábra. Körillesztés zajmentes és zajos adatpontokra



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 4.2 ábrán szemléltetem, aszerint színezve, hogy az iteráció végén mekkora hibával sikerült az illesztés.

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

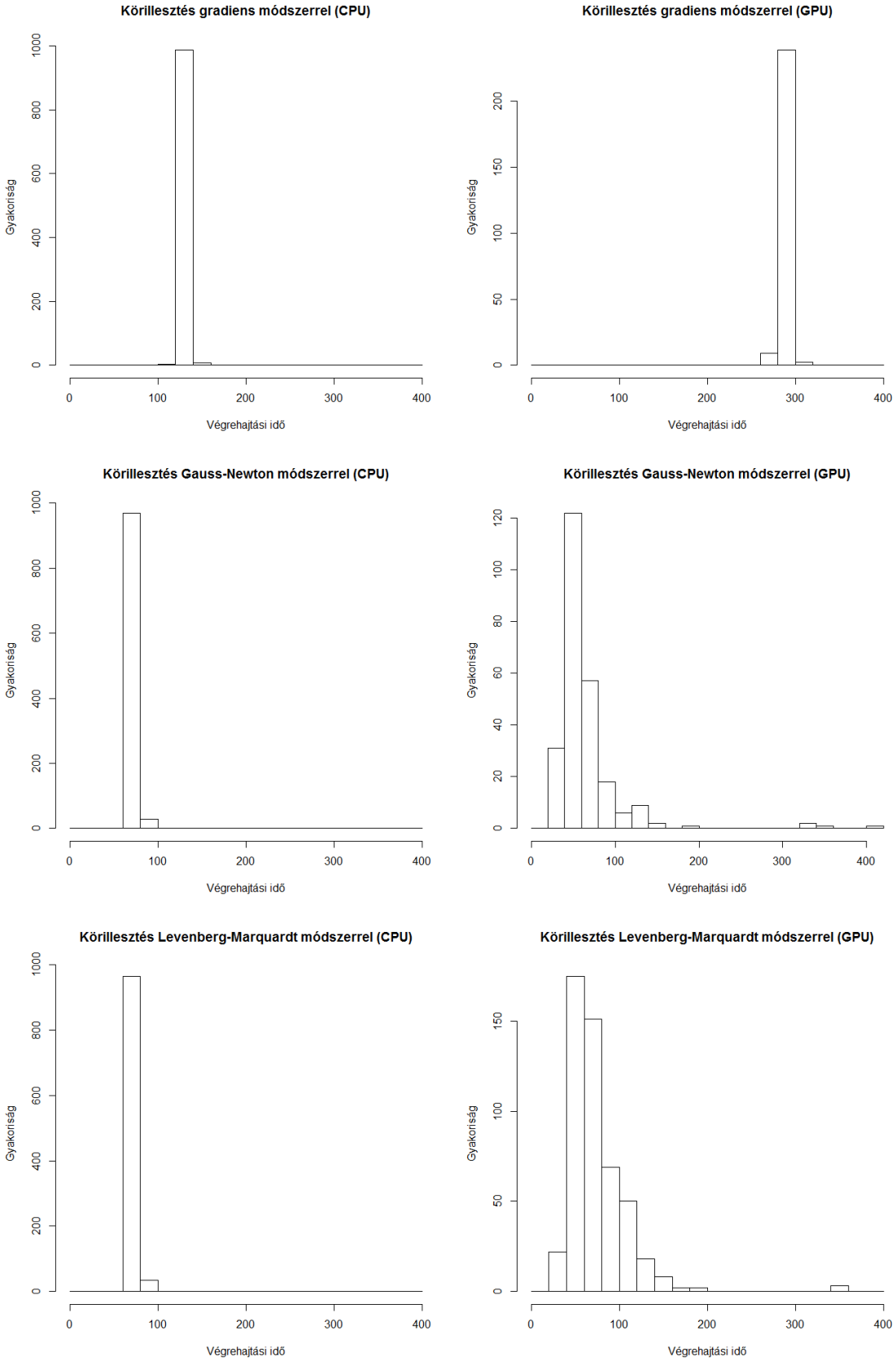
4.2. ábra. Körillesztés konvergenciavizsgálata 1000 kiindulópontból



Az iteráció mindenhonnan tökéletesen konvergált, ezért az összes kör egyszínű. (A zajmentes esetben zöld, míg a zajos esetben az 4.2 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.

4.3. 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 el[1]. A gradiens módszer esetén egyenesen romlik a futásidő a kontextusváltások miatt. Érdekes továbbá, hogy GPU-n futtatva megnő az adatsorok szórása, ez még magyarázatra vár.

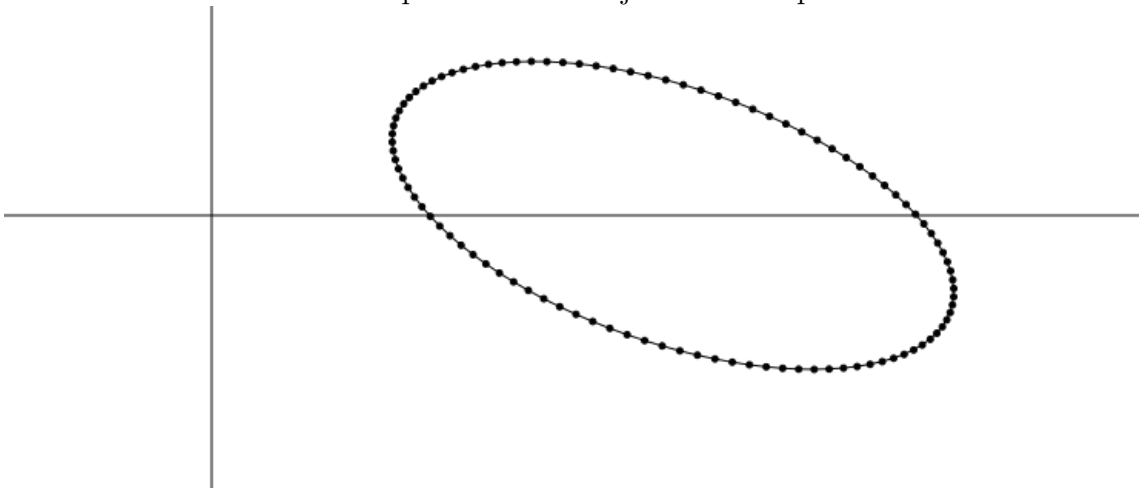
4.1.2. Ellipszis

Az ellipszisillesztésre felírható költségfüggvény[2]:

$$\sum_{i=1}^{n}\left(((x_i-c_x)\cos(A)+(y_i-c_y)\sin(A))^2\frac{b}{a}+((x_i-c_x)\sin(A)-(y_i-c_y)\cos(A))^2\frac{a}{b}-ab\right), \tag{4.1.2.1}$$

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.

4.4. ábra. Ellipszis-illesztés zajmentes adatpontokra.



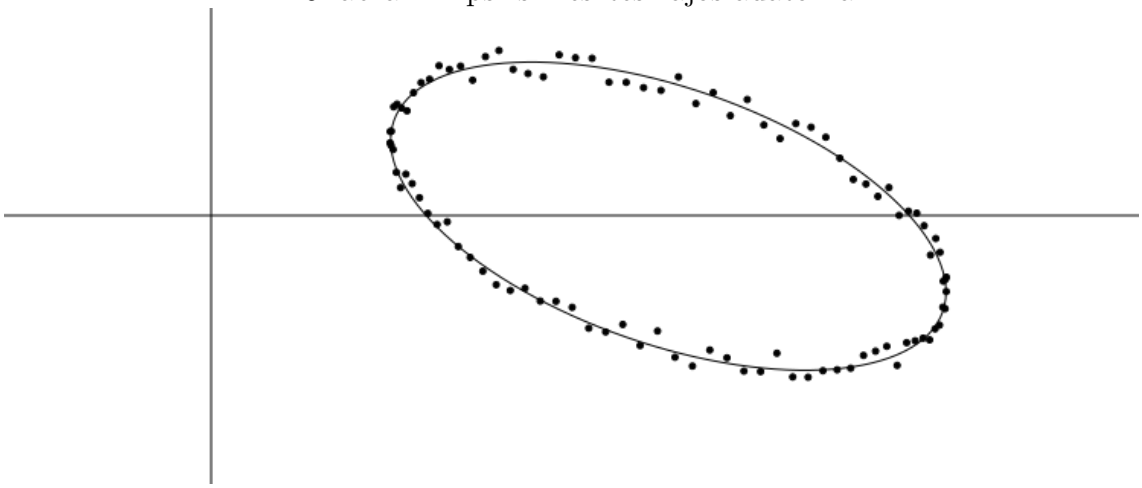
Az illesztések többnyire pontosak, a konvergenciát a 4.6 - 4.11 ábrákon elemzem. Általában a zajmentes esetben a hibafüggvény értéke numerikusan elenyésző, míg a zajos esetben jól simítja a pontokat.

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

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

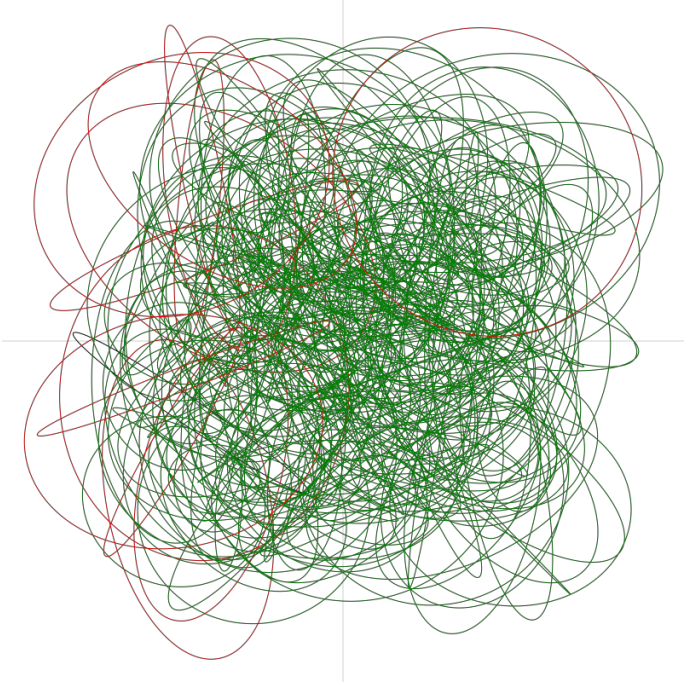
[2] Korábban a költségfüggvény ab-vel leosztott változatát használtam, de ez túl kis függvényértékeket, ezáltal lassú konvergenciát és pontatlanságokat eredményezett.

4.5. ábra. Ellipszis-illesztés zajos adatokra.



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

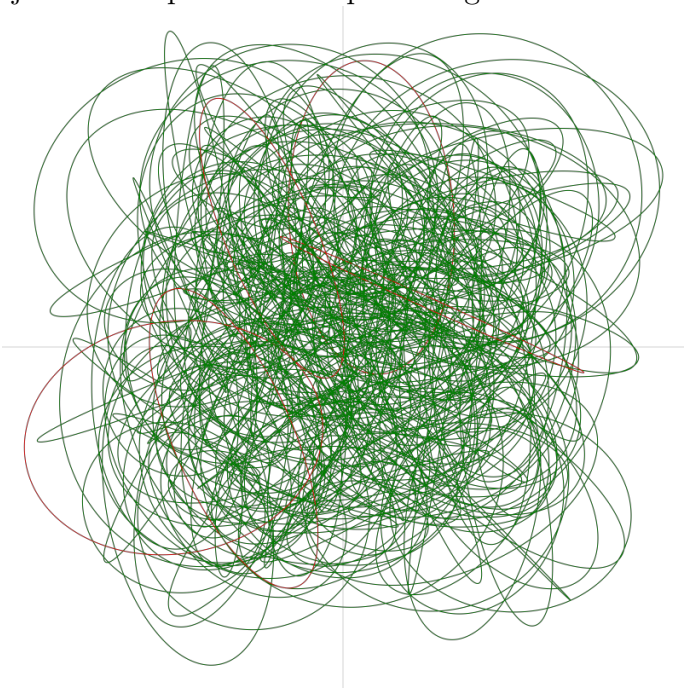
4.6. ábra. Zajmentes ellipszisillesztés pontossága - gradiens módszer



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.

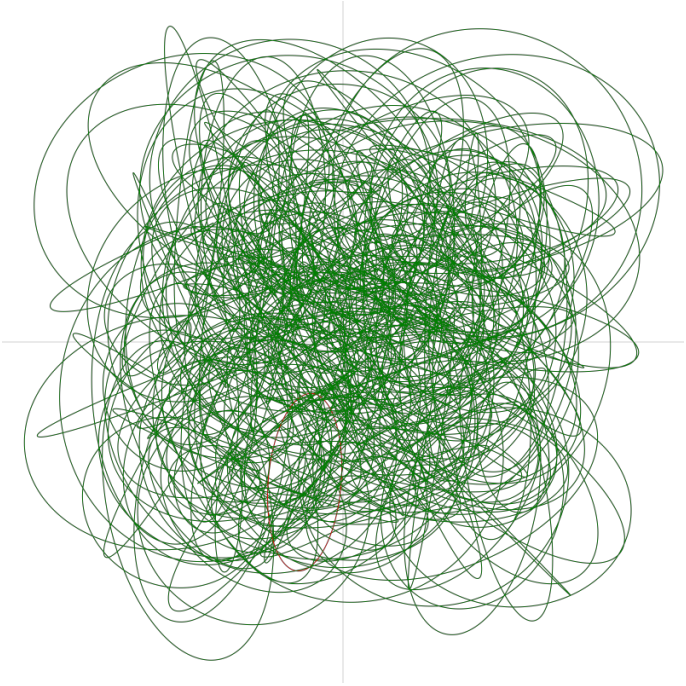
A zajos esetben annyival romlik a helyzet, hogy a hiba már nem tud 0.1 alá csökkenni (ezért a fekete szín), de ez az adatpontok elhelyezkedésének következménye. Némiképp több olyan kezdőpont van, ahonnan nem konvergál az iteráció, de a számuk legalábbis a Levenberg-módszernél még mindig nem jelentős.

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

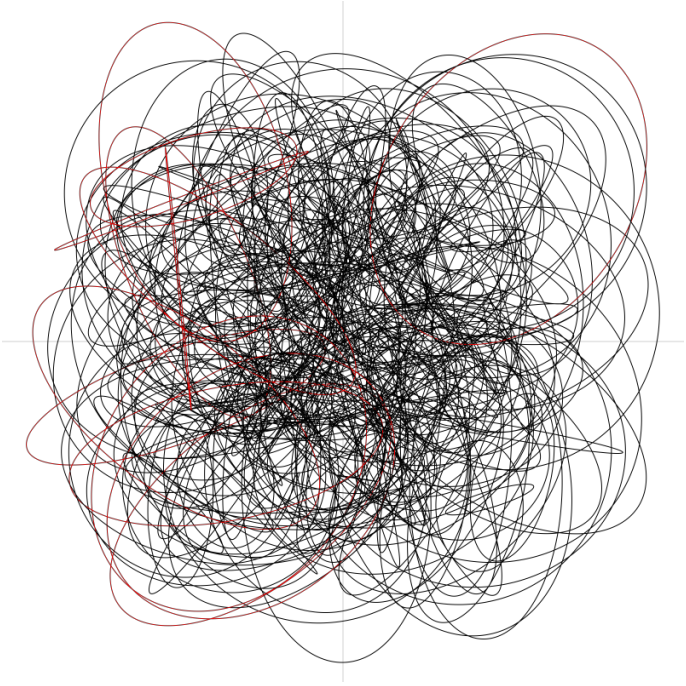


A futásidőket a körillesztéshez hasonlóan mértem. Mindhárom módszerre összevetettem a CPU-n és GPU-n futtatott változatokat.

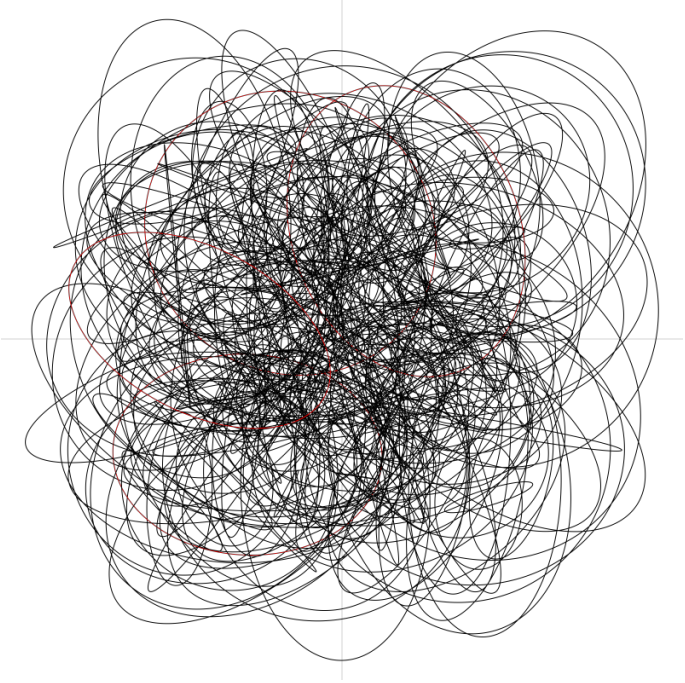
4.8. ábra. Zajmentes ellipszisillesztés pontossága - Levenberg-Marquardt módszer



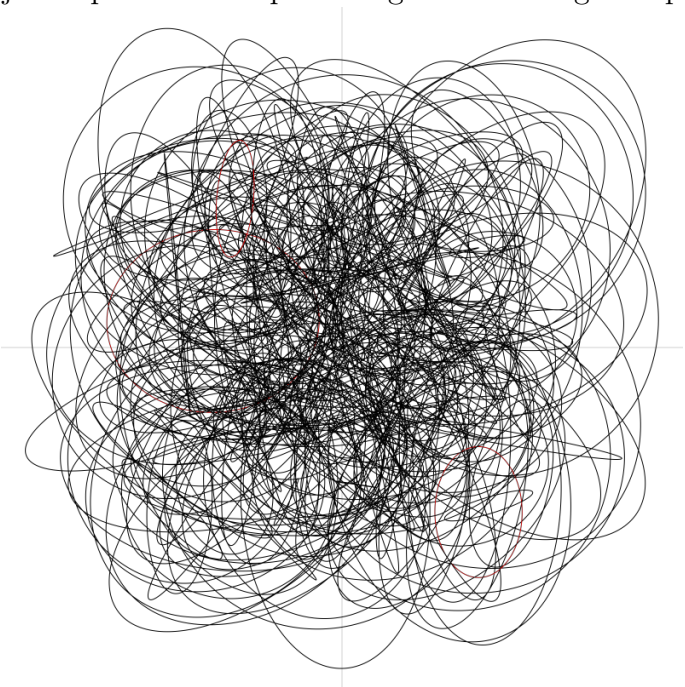
4.9. ábra. Zajos ellipszisillesztés pontossága - gradiens módszer



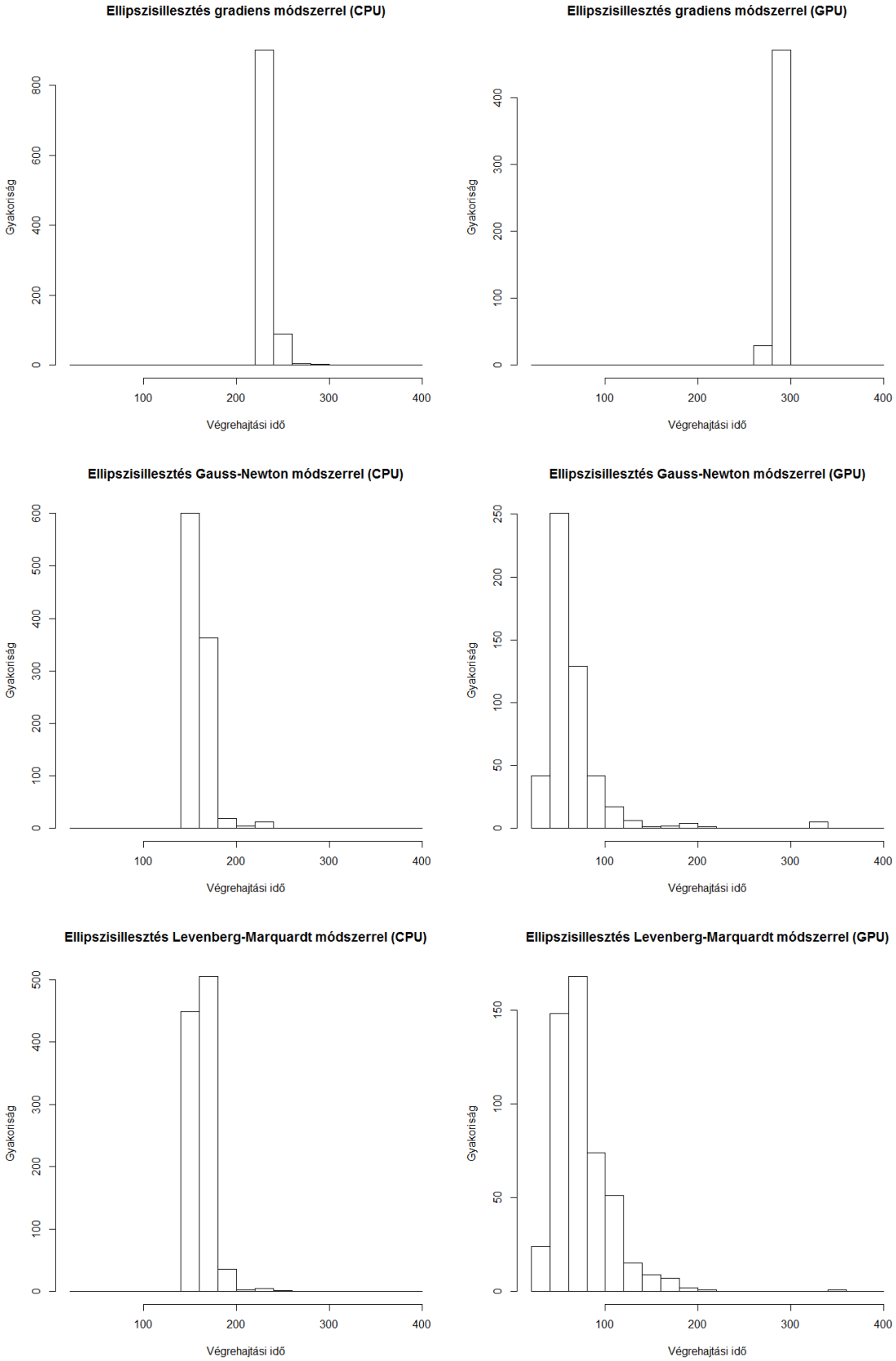
4.10. ábra. Zajos ellipszisillesztés pontossága - Gauss-Newton módszer



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



4.12. 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 kontextusinicializálásnak 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.

4.2. Síkhomográfia

A síkhomográfia egy projektív térbeli leképezés két sík között. Természetes megközelítésből úgy adódik, hogy két különböző orientációjú kamerával lefényképezünk egy jól felismerhető mintázattal rendelkező síklapot[3], majd a feladatunk meghatározni azt a perspektív transzformációt, amely az egyik síklapot a másikba viszi át.

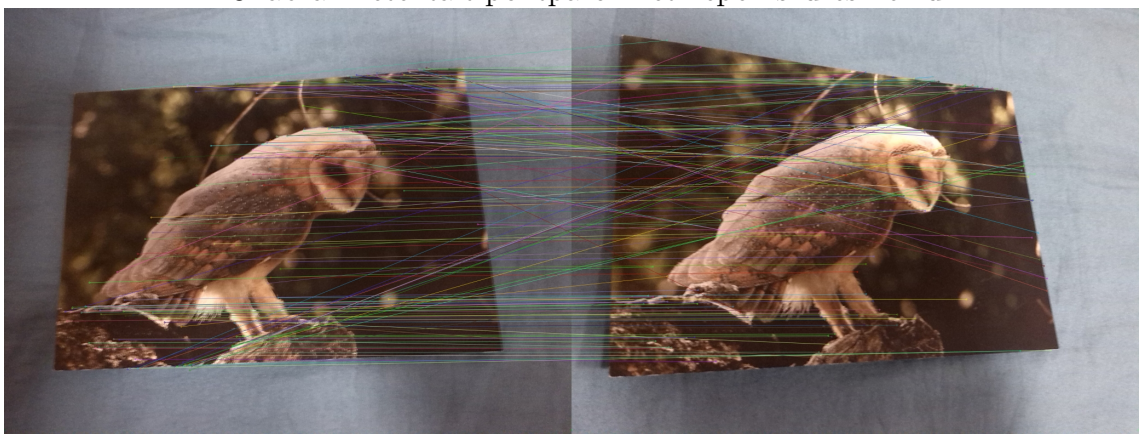
A perspektív transzformációkat leghatékonyabban a homogén koordinátarendszerek segítségével írhatjuk le. Egy kétdimenziós, (u, v) pont homogén koordinátarendszerbeli alakja a $(\lambda u, \lambda v, \lambda)$, $\lambda \in \mathbb{R}$ vektorcsalád tetszőleges eleme. A pontok ilyen felírásával a perspektív 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.

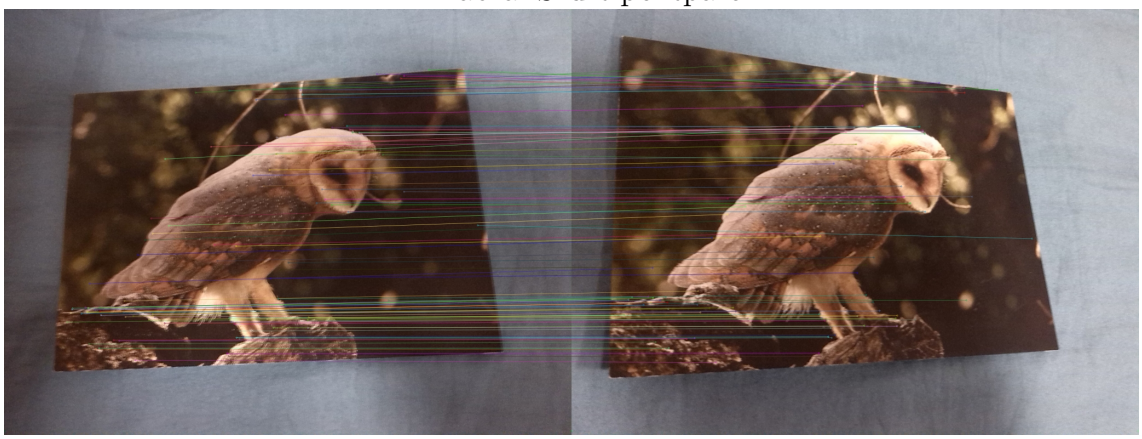
[3]Például sakktábla, plakát

4.13. ábra. Detektált pontpárok két képen szűrés nélkül



Kétféle lehetőséget biztosítok a rossz párosítások eldobására: az OpenCV becslő algoritmusa által adottat, 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.

4.14. 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 $H \in R^{3\times3}$ az a homográfia, amely $\mathbf{x}_i$-t $\mathbf{y}_i$-be viszi (homogén koordinátás alakban)

$$f(\mathbf{p}, \mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} \left\| trf(H, \mathbf{x}_i) - \mathbf{y}_i \right\|_2^2 \tag{4.2.0.1}$$

Itt a $trf(H, \mathbf{x}_i)$ 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.

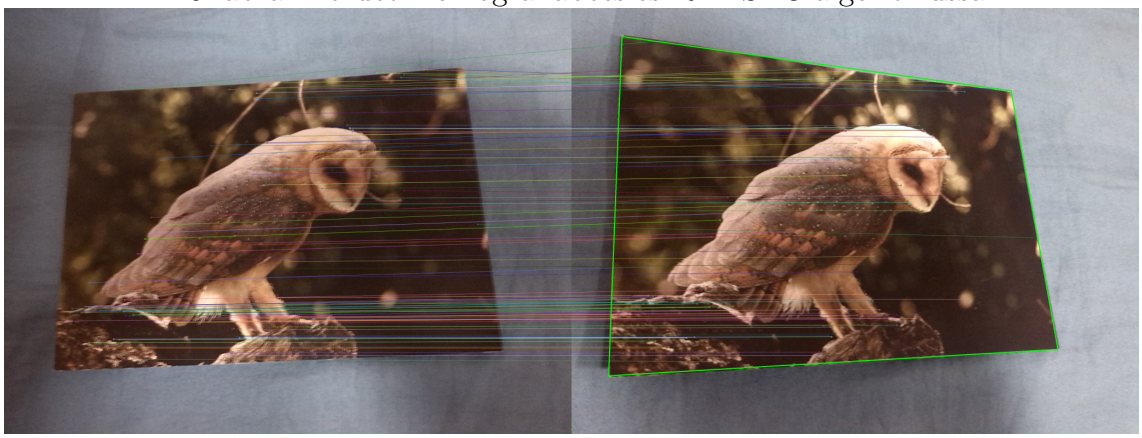
A becslések minőségét a jobb oldali képre illesztett zöld kerettel szemléltetem, ami a bal oldali kép sarokpontjainak a számított homográfiával való transzformálásából adódik.

A másik szemléltetés a 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 pondeldobó módszeremmel mutatom be az eredményeket.

A RANSAC algoritmus egy nagyon jó kezdeti becslést adott (4.15 ábra). A hibafüggvény kezdeti értéke RMSE felírásban 13.7122 volt (ez lényegében az átlagos, pixelben mért távolság a transzformált és a valódi pontok között).

4.15. ábra. Kezdeti homográfiabecslés RANSAC algoritmussal



A kezdeti hibát a Levenberg-módszerrel 13.6625-re sikerült csökkenteni. Az eredmény a 4.16 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 kevébé hangsúlyos.[4]

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

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

Az eredeti és a transzformált kép összehasonlítása a 4.17 ábrán látható. A két kép meglehetősen jól illeszkedik egymásra (a környezetükre pedig ezt nem is garantálja a homográfia).

4.17. 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

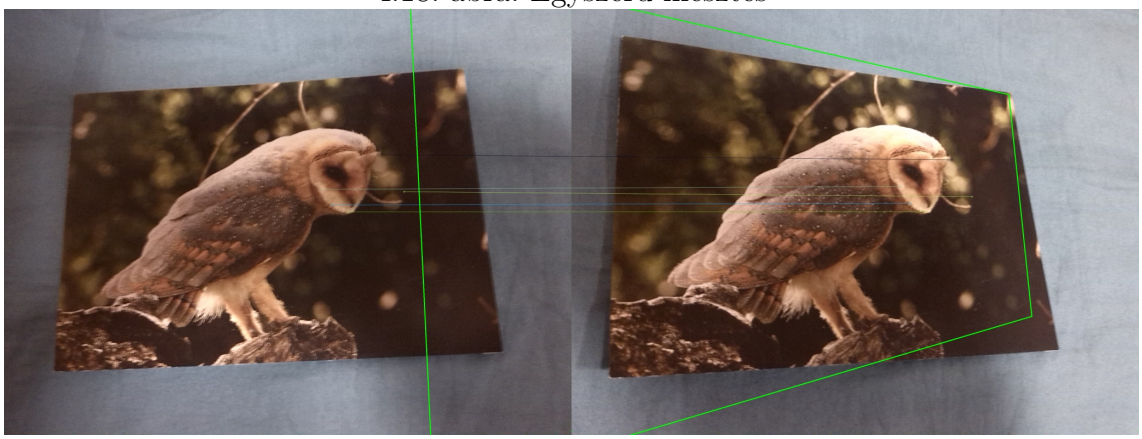
Látható tehát, hogy a futásidő túlnyomó részét az OpenCV különböző algoritmusai tették ki, a finomítás ehhez képest (különösen párhuzamos futtatás esetén) csekély többletet jelent.

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 dobnia 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 4.18 ábra.
- A Least Median módszer egy kevéssel adott rosszabb becslést, mint a RANSAC, és ezt ahhoz hasonlóan is sikerült „javítani”, azaz a több pontot tartalmazó részeken fejleszteni, a nem szignifikáns részeket nem figyelembe véve.

4.18. ábra. Egyszerű illesztés



Összegzésként elmondható, hogy a homográfiaillesztésben sikerült eredményeket elérni, és csökkenteni a költségfüggvényt. 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.

5 Konklúzió

5.1. Összegzés

Elkészítettem egy OpenCL backendet falhasználó 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 könyváraiban adott matematikai függvények véges számú alkalmazásával megadható, egy függvénysorral, vagy más iterációval megadott nem elemi függvény megadása problémákat jelent.

5.2. 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.

Ábrák jegyzéke

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.