Diplomamunka

CT berendezés képrekonstrukciós algoritmusának implementálása grafikus kártyán

Kiss István

Témavezető: Dr. Légrády Dávid
egyetemi docens
BME Nukleáris Technika Intézet

BME
2013

Önállósági nyilatkozat

Kijelentem, mint a Budapesti Műszaki és Gazdaságtudományi Egyetem Fizikus mesterszak (MSc) hallgatója, hogy ezt a diplomamunkát meg nem engedett segítség igénybevétele nélkül, saját magam készítettem. Minden olyan szövegrészt, adatot, diagramot, ábrát, vagy bármely más elemet, amelyet vagy szó szerint, vagy azonos értelemben, de átfogalmazva másoktól vettem át, a forrás megadásával egyértelműen megjelöltem.

.....

Kiss István

Tartalomjegyzék

1. Bevezetés.....5
2. Grafikuskártya-specifikus megfontolások 7
2.1 A grafikus kártya tulajdonságai és a hozzá tartozó fejlesztői környezet.....7
2.2 Véletlenszám-generátorok.....12
2.2.1 Lineáris kongruenciás generátor (lcg).....13
2.2.2 Additív lineáris kongruenciás generátor (alcg).....14
2.2.3 Xorshift generátor.....15
2.2.4 CUDA generátorok.....15
2.2.5. Véletlenszám-generátorok összehasonlítása.....16
3. Rekonstrukciós algoritmus.....22
3.1 CT berendezések képrekonstrukciós algoritmusai.....22
3.2 Vizsgált elrendezés.....25
3.3 Teszt geometria 28
3.4 Az előrevetítő kernel.....30
3.4.1 Az előrevetítő megvalósítása.....30
3.4.2 Abszorpciós értékek verifikálása.....34
3.4.3 Az energiaspektrum figyelembevételének hatásai.....38
3.5. Visszavetítés 42
3.5.1. Adott voxel csúcsainak a pontforrásból történő vetítése a detektor felületre.....43
3.5.2. A vetítésből kapott pontokat körbehatároló legkisebb konvex poligon pontjainak meghatározása.....45
3.5.3. A konvex poligon darabolása a detektorpixeleket határoló vonalak segítségével.....46
3.5.4. A kapott poligonok területének kiszámítása.....47
4. Rekonstruált képek vizsgálata.....49
4.1 Értékhelyesség ellenőrzése.....53
4.2 Konvergencia vizsgálata.....58
4.3 Spektrum felkeményedés eliminálása a rekonstruált képen fizika vagy szoftveres szűrés nélkül.....60
5. Összefoglalás.....62
6. Irodalomjegyzék.....63
7. Függelék.....65

1. Bevezetés

A komputer tomográfia (CT – computed tomography) az egyik legelterjedtebb orvosi diagnosztikai eszköz (58,8 vizsgálat/1000 fő évente [1]), melyhez mára már számos képrekonstrukciós algoritmus készült, mégis sok pénz és idő ráfordításával fejlesztik az újabbakat. A fejlesztés fő céljai, hogy javuljon a diagnosztikai kép minősége, a leadott dózist minél kisebb legyen, és csökkenjen az előállításához szüksége idő, ami a hatékonyabb kamerakihasználáshoz vezet.

A kamerák által használt leggyakoribb algoritmus a szűrt visszavetítés, mely gyorsaságának és egyszerűségének köszönheti elterjedtségét, mégis meglehetősen sok műterméket ad a rekonstruált képen. A képhibák (műtermék, artefaktum) származhatnak a beteg mozgásából, a kamera geometriájából, vagy valamely, a rekonstrukció során figyelmen kívül hagyott fizikai effektusból. Ezek csak már a megkapott rekonstruált kép szűrésével eliminálhatók, de a kép szűrésének nagy számításigénye lehet, és különböző artefaktumok különböző szűrökkel távolíthatok el. Az előbb leírtaknál esetleg jobb megoldás egy olyan algoritmus implementálása, amely a rekonstrukció során figyelembe veszi az effektust, és a hatás okozta műterméktől mentes képet eredményez.

A munkám célja ezért egy olyan rekonstrukció megalkotása, ami minél több fizikai effektusra van tekintettel. Ezek hátránya, hogy nagy számításigénnyel bírnak, de csoportos részecskekövetés megvalósításával és grafikus kártyán történő futtatással kaphatunk olyan megoldást, ami elfogadható futási idővel rendelkezik, és kevesebb képhibát ad, mint a szűrt visszavetítés.

Ezek a rekonstrukciók nem gyakoriak a mindennapos használatban, mivel költséghatékony eszközökön még megvalósíthatatlanok a humán CT esetében a hatalmas memóriaigény és a sebesség területén támasztott feltételek miatt (minimum 20 szelet/másodperc). Ezért egy kisállatkamera geometriáját vettem alapul a programom megíráshoz és teszteléséhez. A választott kamera méretei kisebbek, de még így sem triviális a grafikus kártyán történő implementálás, mivel grafikus kártyák (GPU -k) memóriakapacitása kisebb a központi számító egységek (CPU -k) által elérhetőknél, és működésükben is eltérő a két egység. Szükséges volt tehát vizsgálnom a grafikus kártyák és az általam használt kártya pontos tulajdonságait, és megoldást kellett találnom a kis memóriaméret okozta gondokra.

Kiválasztottam a fent leírt problémához legmegfelelőbb rekonstrukciós algoritmust, beépítettem a csoportos fotonkövetést, amit Monte Carlo alapú részecsketranszporttal kombináltam, hogy

később könnyen bővíthető legyen a megvalósított kódom. A forrás spektrumából mintavételeztem a foton energiáját. A kapott eljárás részeit vizsgáltam külön-külön, majd kimutattam a spektrum figyelembevételének hatását, ami a spektrum felkeményedését okozta.

A spektrumfelkeményedés lényegében a kis energiás fotonok nagyobb arányú elnyelődése, ami a hatáskeresztmetszet energiafüggéséből ered. Az effektus szűrt visszavetítés használatával homogén anyag esetében inhomogén képet eredményez, az inhomogenitás mértéke a forráson elhelyezett szűrővel(Al) csökkenthető, de csak a megkapott kép szoftveres szűrésével eliminálható.

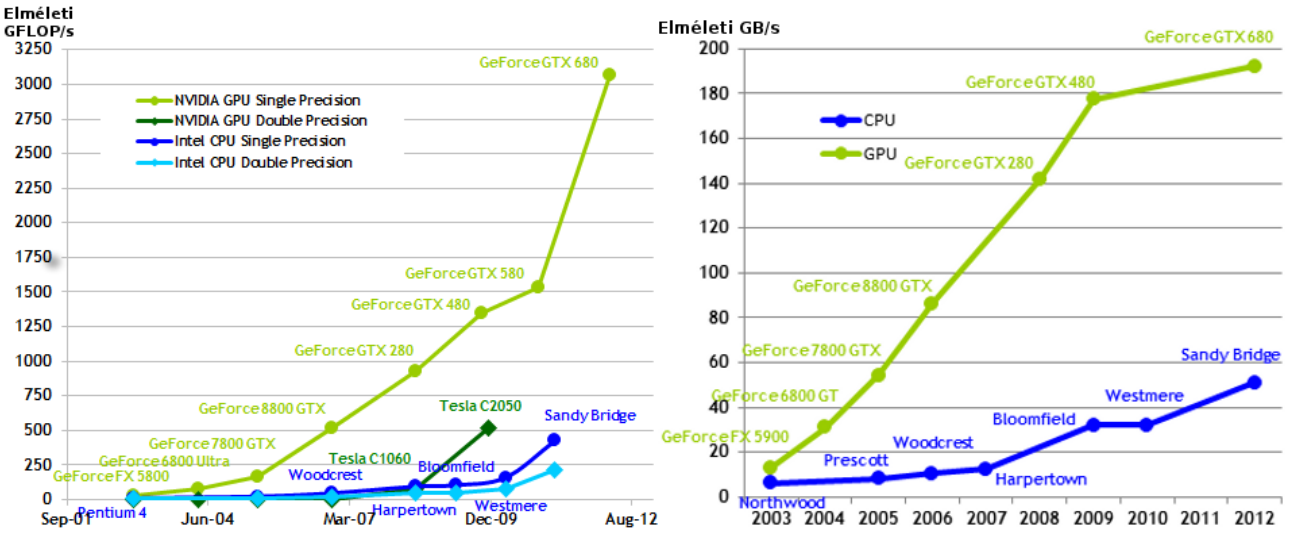
Elvárásaink szerint az általam implementált program szűrés nélkül eliminálja a jelenséget, amit a dolgozatom végén ismertetek és alátámasztok eredményekkel.

2. Grafikuskártya-specifikus megfontolások

2.1 A grafikus kártya tulajdonságai és a hozzá tartozó fejlesztői környezet

Az elmúlt 10 évben a grafikus kártyák (graphical processing unit – GPU) rohamos fejlődése és a programozást segítő új architektúrák megjelenése volt a hajtóereje annak, hogy a nagy számításigényű, könnyen párhuzamosítható algoritmusokat GPU -ra implementálják.

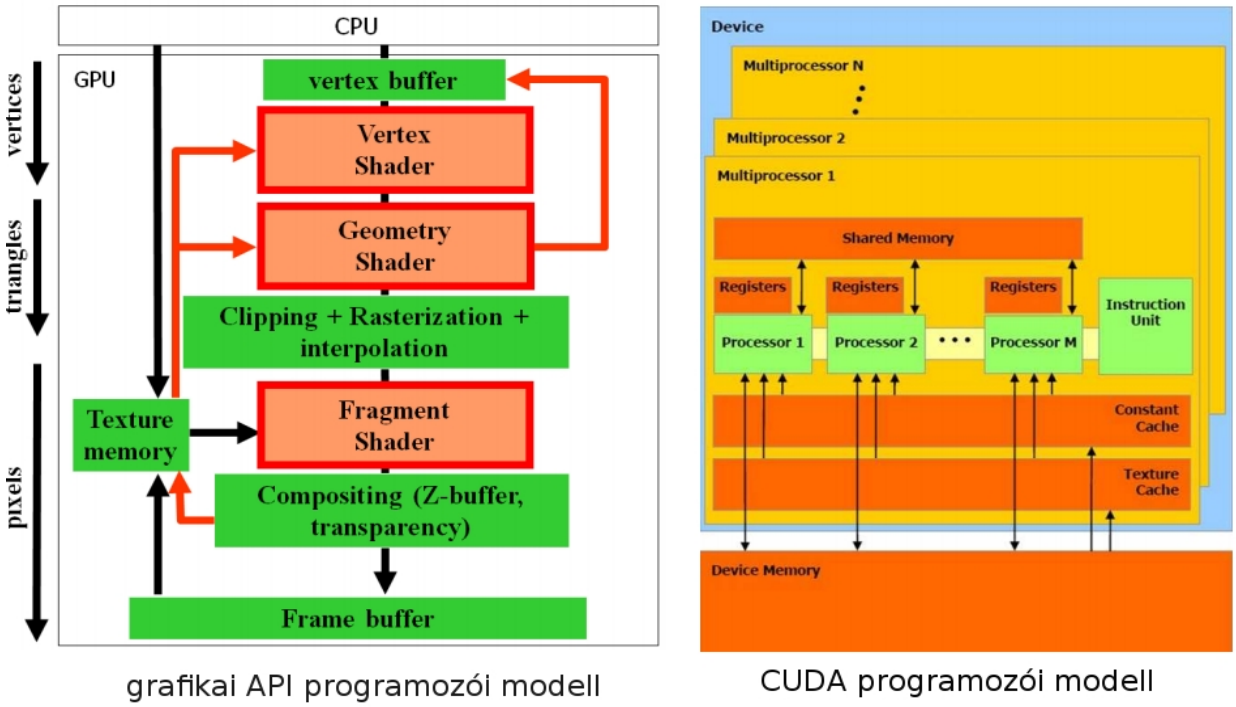
A GPU -k sebességnövekedése nem lenne elegendő az átálláshoz, ha a már eddig használt központi számító egységek (central processing unit – CPU) teljesítménynövekedéséhez képest nem lenne nagyobb. E folyamatot szemlélteti az 1. ábra baloldali (a másodpercenkénti egyszeres és kétszeres pontosságú lebegőpontos műveletek száma), illetve jobboldali (az elméleti memória sávszélessége) grafikonok, amelyek az adott időszak legfejlettebb GPU-nak, illetve CPU –nak egy-egy teljesítményparaméterét ábrázolják. A kétféle számítóegység egyszeres pontosságú műveleti sebességének, illetve a memória sávszélességének különbsége az idő függvényében monoton növekvő, amiből arra lehet következtetni, hogy legalábbis a közeljövőben gyorsabb algoritmusok készíthetők a GPU-k segítségével.



1. ábra Az adott időszak legfejlettebb grafikus kártya és CPU egyszeres és kétszeres pontosságú műveletek száma másodpercenként (bal) illetve a memóriáik elméleti sávszélessége (jobb). [2]

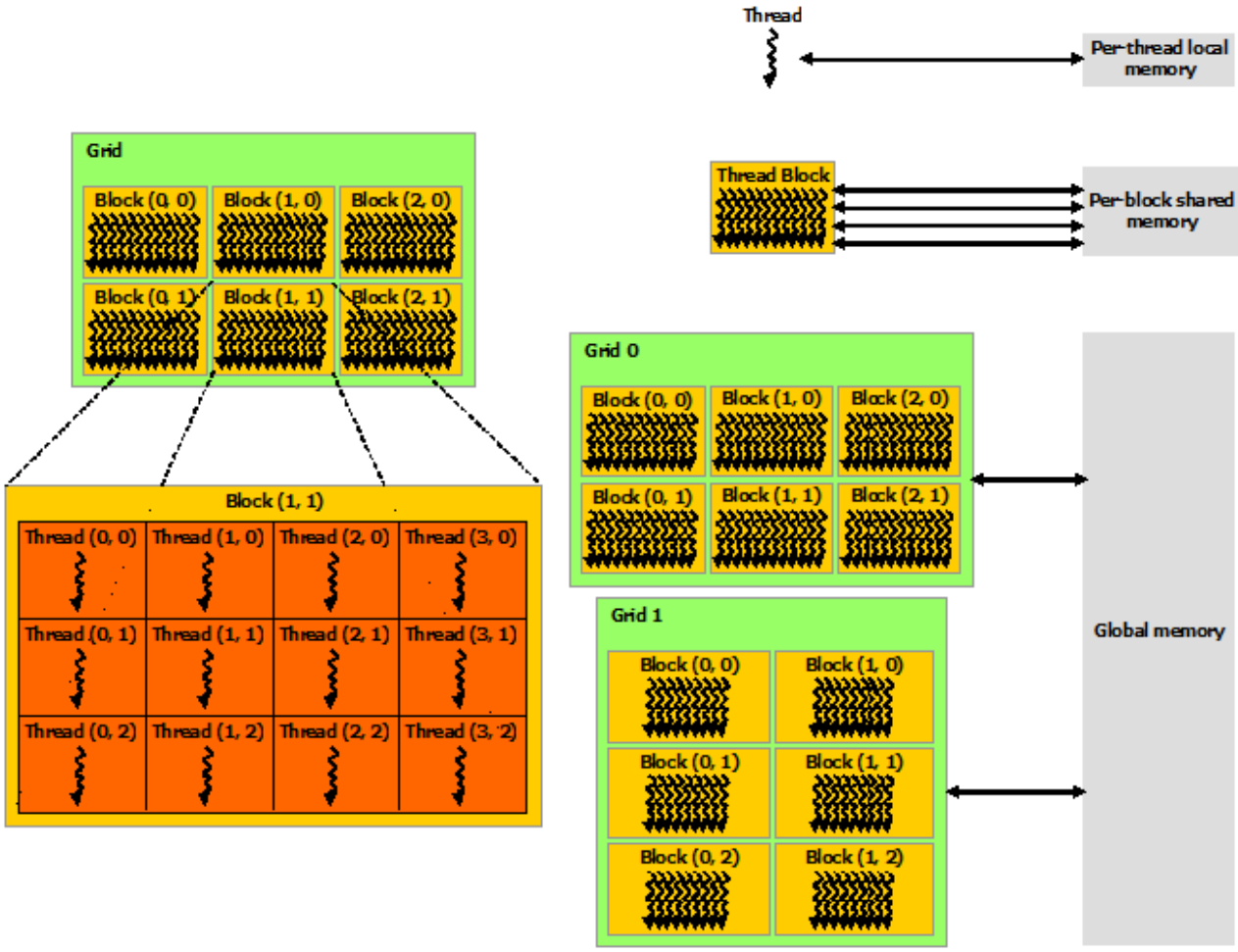
Az új architektúrák megjelenése előtt a grafikus kártyán csak a csővezeték modell (2. ábra bal oldal) pontos ismeretével lehetett programozni. A modell a grafikai problémák kiszámítására és a

grafikai elemek képi megjelenítésére lett kidolgozva, így ezen feladatok megoldására a legalkalmasabb. Az általánosabb problémák megoldására fejlesztették ki a CUDA -t (2. ábra jobb) és az OpenCL -t. Az előbb említett új architektúrák elrejtik a csővezeték modellt és segítségükkel a GPU -t multiprocesszorok gyűjteményeként kezelhetjük. Igaz, hogy az OpenCL több grafikus kártyát támogat, és CPU-ra is használható, de minden grafikus kártyára szánt kódot futás közben kell fordítani. Így a diplomamunkám megvalósításához az NVIDIA által kiadott CUDA programozói modellt választottam.



2. ábra Programozói modellek [3]

A CUDA (teljes nevén Computer Unifed Device Arhitecture) lényegében egy virtuális architektúra, mely segítségével a GPU - n több egymással párhuzamosan futó processzoron lehet egyidejűleg azonos algoritmust futtatni, különböző bemeneti paraméterekkel. Az egyidejűleg futó processzorok száma függ az adott kártya típusától. Az általam használt GTX 280-as 30 multiprocesszort tartalmaz (8 cuda-mag helyezkedik el egy multiprocesszorban), és multiprocesszoronként (MP) 1024 szál kezelhető egy időben, azaz párhuzamosan egyszerre legfeljebb 30720 szál futhat. Az architektúra ezt a fizikai paramétert elrejti a programozó elől, mivel ennél sokkal több szálat engedélyez, melyeket blokkokba (block) szedve, és e blokkokat hálózatba (grid) rendezve érhet el a fejlesztő (3. ábra bal). A GTX 280 -as maximálisan 1024 szálat engedélyez multiprocesszoronként, de ennél kevesebb szál indulhat egyszerre a kevés rendelkezésre álló regiszter vagy memória miatt, vagy ha jóval több szál indul, mint az egy időben futtatható maximális szálak száma. Ebben az esetben a MP -ok ismétlődő futásaival éri el a fejlesztői környezet, annak érdekében, hogy minden szál lefusson.



3. ábra Szálak elrendezése és memória típusok elérése

Az architektúra egy adott blokkján belüli szálakat csak egy adott MP dolgozza fel, így a szálakhoz rendelt lokális memóriát az MP -n belüli regiszterekből osztja ki a vezérlő egység. Az egy szálhoz rendelt regisztereket a többi szál nem éri el, amíg a MP -on belüli megosztott memóriát a blokkon belüli összes szál kezelheti. Különböző blokkok szálainak csak a GPU -n belüli leglassabb memóriatípushoz van egyidejű hozzáférésük: ez a globális memória.

Minden kernel meghíváskor a fejlesztő feladata, hogy meghatározza a blokk és a hálózat méretét az alábbi megkötések és szabályok figyelembevételével:

- A blokkok rendelkeznek egy maximális szálmennyiséggel (GTX 280 -nál 512).
- A blokkok rendelkeznek dimenziókorláttal (GTX 280 -nál 512x512x64).
- A hálózat rendelkezik dimenziókorláttal (GTX 280 -nál 65535 x 65535 x 1).
- A regiszterek száma blokkonként állandó, így az egy szál által felhasznált regiszterek

száma meghatározza az egy multiprocesszorra jutó aktív szálak számát.

- A megosztott memória nagysága állandó, így a blokkonként felhasznált megosztott memória meghatározza a MP -ként aktív blokkok számát.

A különböző szálak azonos időben történő, azonos globális memóriaterületre írásakor 'atomic' függvények használata nélkül a GPU hibás számítási eredményt ad vissza. Ezen függvények a programot jóval lassabbá teszik, és csak egész értékek esetében használhatók. Ebből kifolyólag a diplomamunkám elkészítése során kerültem használatukat, és kerestem az egyszerre történő azonos memóriacímre írást kerülő megoldásokat.

Az NVCC a CUDA fordítója, a számítógép C, C++ vagy Fortran fordítójának kiegészítéseként használható, azaz a kártyára írt kódból az NVCC és a CPU-ra szánt kódból a a számítógép alapfordítója generál gépi kódot. A diplomamunkámat egy 4GB memóriával rendelkező, ASROCK H55 LM alaplapot és Intel i3-560 processzort tartalmazó asztali gépen futtattam. A rekonstrukció kidolgozására használt operációs rendszer egy 64 bites Fedora 17, melyre telepítettem a gcc-4.4.6 fordítót és CUDA 5.0 -át. Az általam használt grafikus kártya GTX 280 -as, amit a Fedora az NVIDA 304.64 verziójú driver -rel vezérel.

2.2 Véletlenszám-generátorok

A rekonstrukciós algoritmus által használt véletlenszám-generátor minősége és gyorsasága kiemelt fontosságú, mivel Monte Carlo (MC) alapú előrevetítést implementáltam, és az MC algoritmus gyorsaságát és statisztikáját nagyban befolyásolják.

Feladataim közé tartozott a megfelelő generátor kiválasztása, amelyhez figyelembe kell venni a GPU tulajdonságait. A GPU -k csak a 32 bites számábrázolást támogatják optimálisan, így nagyobb számábrázolást használó generátorok sokkal rosszabb sebességet érhetnek el, mint amit a CPU-n várnánk. Fontos még, hogy a szálanként használt regiszterek számát ne növelje túlzottan a generátor, mivel (ahogy már korábban említettem) az egy szálra jutó regiszterek száma befolyásolja az egyidejűleg futó szálak számát.

Számos véletlenszám-generátor létezik, ezért egy új kifejlesztése helyett a meglévők közül választottam, esetleg átalakítottam őket a grafikus kártyán való optimálisabb futáshoz. Az új CUDA 5.0 rendelkezik cuRand[7] könyvtárral, amely már jó minőségű generátorokat tartalmaz, így ezeket is megvizsgáltam.

A vizsgált véletlenszám-generátorok:

- lineáris kongruenciás generátor
- additív lineáris kongruenciás generátor
- xorshift generátor
- CUDA generátorok:
  - pszeudo XORWOW generátor
  - pszeudo MRG32k3a generátor
  - pszeudo MTGP32 generátor
  - kvázi SOBOL32 generátor
  - kvázi SOBOL64 generátor
  - kvázi SCRAMBLED SOBOL32 generátor
  - kvázi SCRAMBLED SOBOL64 generátor

2.2.1 Lineáris kongruenciás generátor (lcg)

Az egyik leggyorsabb és legegyszerűbb generátor, mely rendelkezik periódussal. Az *i* -ik véletlen számot az előző véletlen számból számolja ki az alábbi képlet alapján:

$$x_i = (x_{i-1} \cdot a + c) \bmod m \quad [4], (1.)$$

ahol x_i az új szám, x_{i-1} az előző véletlen szám, az *a*, *m* és *c* paraméter generátoronként különböző konstans. *m* általában az adott számábrázolás maximális értéke.

A lcg gyorsaságát annak köszönheti, hogy 3 egész számmal való algebrai művelet elvégzésével ad egy új számot. Hátránya, hogy a periódus hossza egyenlő az *m* paraméterrel, ami a GPU-n történő optimális futáshoz csak a 32 bites maximális egész szám lehet. A diplomamunkám e szakaszához érve még nem rendelkeztem támponttal arra nézve, hogy hány véletlen számra lesz szükségem, így azt vettem alapul, hogy a kártyák csak véletlen számokat generálva, mennyi idő alatt érnének a periódus végére. Tehát a generátorsebesség mérésekor ellenőriznem kellett, hogy a kapott sebesség mellett milyen gyorsan érne a generátor a periódus végére.

Az általam vizsgált lcg-k paraméterei az 1. táblázatban találhatóak. A 32 bites számábrázoláson nem megjeleníthető paramétereket implementáltam 64 bites számábrázolással és 32 bites számok kombinációjával is.

	lcg 32 bites számokkal	lcg 64 bites	lcg 128 bites számokkal
a	1664525	9219741426499971445	250962815189121053421918519178387186 29
m	2^{32}	2^{63}	2^{128}
c	1013904223	1	0
x_0	1	2131323	1

1. táblázat lcg-k paraméterei [5]

A periódusátlapolódás elkerülése végett fel kell tudnunk osztani a generátor teljes periódusát. A szimpla lcg algoritmuson alapuló generátorok periódusát a 'leap ahead' módszerrel daraboltam több részre. Az algoritmus az alábbi képlet alapján lép *n*-et az lcg perióduson

$$x_i = \left(\left(x_{i-1} \cdot a^n \right) \bmod m + c \cdot \left(a^n + a^{(n-1)} + ... + a + 1 \right) \right) \bmod m \quad . (2.)$$

A fenti képlet nem lenne gyorsabb az eredeti lcg algoritmus *n*-szeres lefuttatásánál, ha nem használnánk a moduláris multiplikatív módszert, amely gyorsan számol hatványműveleteket. A megvalósított algoritmus a függelékben található. Az additív lcg periódushosszát az előbb leírt módszerrel sajnos nem tudjuk felosztani, a rendelkezésemre álló időn belül erre nem találtam

megoldást, az additív lcg-t nem tudtam később hasznosítani.

2.2.2 Additív lineáris kongruenciás generátor (alcg)

Az lcg periódushossz meghosszabbításának *m* paraméter változtatásán kívül másik módja, hogy a generált számhoz hozzáadunk egy régebbi véletlen számot. Ez az additív lcg, amely az *i*-ik véletlen számot az alábbi képlettel számolja

$$x_i = (x_{i-r} \cdot a + x_{i-1} \cdot a + c) \bmod m \quad [4].(3.)$$

ahol *r* a két összegzendő véletlen szám közötti távolság, más néven a szóhossz. Az algoritmus alapján az új szám előállításához rendelkeznünk kell az előző számmal és az *r* -el előtte kapott számmal. Ahhoz, hogy ez a feltétel teljesüljön a további számok generálásánál, meg kell őriznünk az *(i-1)* -ik és *r* -ik közötti elemeket is. Ezzel a módszerrel a periódus kap egy 2^{r-1} szorzót, azaz minél nagyobb a szóhossz, annál nagyobb a periódus, de annál több regiszterre van szükségünk szálanként.

A vizsgált alcg paraméterei a 2. táblázatban láthatók és a periódushossza $2^{((r-1)+63)} = 2^{79}$. [6]

additív lcg			
a	9219741426499971445	x[-7]	1853551276924066032
m	2^31	x[-8]	5009278006881457585
c	1	x[-9]	5620239895114384102
x_0	8938803317467288985	x[-10]	6238090374769420575
x[-1]	6504692979264364526	x[-11]	1004296704575865388
x[-2]	6560596919803295175	x[-12]	6011048099338914333
x[-3]	3473783674809037556	x[-13]	9160818560819397698
x[-4]	4855435565765123461	x[-14]	3364998211876029483
x[-5]	3934057098709682890	x[-15]	5813820809681046184
x[-6]	1939052829077506131	x[-16]	1542576533590230729

2. táblázat additív lcg paraméterei

2.2.3 Xorshift generátor

A Xorshift generátor[8] az egyik leggyorsabb nagy periódushosszú véletlenszám-generátor, amely 5 egész számból keveri ki az új véletlen számot az alábbi metódus alapján:

x=123456789,y=362436069,z=521288629,w=88675123,v=886756453 – kezdeti paraméterek

1. t=(x^(x>>7));
2. x=y; y=z; z=w; w=v;
3. v=(v^(v<<6))^(t^(t<<13));
4. new_random_number = (y+y+1)*v;

A generátor periódushossza 2^{160} és 5 regisztert használ szálanként.

2.2.4 CUDA generátorok

A cuRand[7] könyvtárban lévő CUDA generátorok a véletlen számokat a CPU -n (host -on) és a kártyán is generálhatják.

A CPU -n generált számokat globális memóriába kell tölteni, majd innen érik el a generátor eredményeit felhasználó kernel szálai. A megoldás hátrányai, hogy az egyébként is kevés globális memóriából kell elkülönítenünk területet a véletlen számoknak, valamint a globális memóriából olvasás a leglassabb memóriaelérés, tehát maga a számkiolvasás is lassú lesz. Az előrevetítésem minimum 2026 fotont indít és átlagban 5 véletlen számra van szükség fotononként a geometrián való áthaladáshoz. 64 szálat használok blokkonként és 450 blokkot indítok egy kernel futáshoz, így kb. 288 000 000 véletlen szám kell előrevetítésenként. Ez 274 MB -ot jelent, ami az aktuális kártyám globális memóriájának egynegyede. Ekkora hely nem biztosítható a véletlen számok tárolására teljesítménycsökkenés nélkül, emiatt a kódomban nem használtam a host -on történő generálást.

A kártyán működő generátorokkal elkerülhetjük a globális memóriába történő írást, így azonban használhatóságuk felméréséhez ki kell vizsgálnom a generátorok gyorsaságát és minőségét.

A különböző CUDA által implementált generátortípusok kifejtését nem tartalmazza a diplomamunkám, csak a leírásban megtalálható fontosabb paramétereit (3. táblázat). A pontosabb és részletesebb megértés érdekében a cuRand[7] leírását és általa megjelölt irodalmat ajánlom az

érdeklődők figyelmébe. Feladataim korlátozódtak a generátorok minőségének és sebességének vizsgálatára, kivéve az MTGP32-t, amit nem tanulmányozok a továbbiakban, mivel nem rendelkezik leap-ahead függvényhívással.

	Xorwow	MRG32K3A	MTGP32	Sobol32	Sobol Scrambled 32	Sobol64	Sobol Scrambled 64
periódushossz	$> 2^{190}$	$> 2^{190}$	2^{11213}	szál $* 2^{32}$	szál $* 2^{32}$	szál $* 2^{64}$	szál $* 2^{64}$
max szál szám	-	-	-	20000	20000	20000	20000
max periódushossz	$> 2^{190}$	$> 2^{190}$	2^{11213}	2^{47}	2^{47}	2^{79}	2^{79}

3. táblázat A cuRand könyvtár generátoraira jellemző tulajdonságok

2.2.5. Véletlenszám-generátorok összehasonlítása

A „véletlenszám-generátorok” fejezetben ismertetett generátorok periódushosszát, gyorsaságát és az általuk előállított számok véletlenszerűségét vizsgáltam, azért hogy megtaláljam a lehető legmegfelelőbb megoldást a véletlen számok előállítására a rekonstrukciós algoritmusomban.

A GPU -n futó generátorok 64 bites számábrázolás használatát a CUDA nem támogatja optimálisan, a 128 bites számábrázolást pedig egyáltalán nem támogatja, ezért a 64 bites *m* paramétert használó lcg-t implementáltam 32 bites számok használatával és a 128 bites *m* paraméterrel rendelkező lcg-t 64 bites számokkal.

A 128 bites *m* paraméterű számokat az alábbi képlet segítségével bontottam 64 bites számokra

$$x_i = g \cdot 2^{64} + h, a = i \cdot 2^{64} + j, x_i = g\, cdot 2^{64} + h, a = i \cdot 2^{64} + j \text{ , (4.)}$$

ahol *g*, *h*, *i* és *j* 64 bites számok, így érve el, hogy az *a* és az *x* paraméter 128 bites legyen. A generátor az alábbi módon alakul

$$x_{i-1} = (x_i \cdot a) \bmod m = ((g \cdot 2^{64} + h) \cdot (i \cdot 2^{64} + j)) \bmod m = (g \cdot h \cdot 2^{128} + (g \cdot j + h \cdot i) \cdot 2^{64} + h \cdot j) \bmod m \text{ .(5.)}$$

Tudjuk , hogy az általunk használt lcg *m* paramétere 2^{128}, ezért $g \cdot h \cdot 2^{128} \bmod 2^{128} = 0$

$$x_{i-1} = (k \cdot 2^{64} + l) \rightarrow k = (g \cdot j + h \cdot i) \bmod 2^{64} + (h \cdot j - ((h \cdot j) \bmod 2^{64}))/2^{64} \text{ és (6.)}$$

$$l = (h \cdot j) \bmod 2^{64} \text{ . (7.)}$$

Látható, hogy a fenti képletek alapján csak a $h \cdot j$ túlcsordulását kellett kezelnem, amit úgy oldottam meg, hogy a *h* és *j* paramétert felbontottam további két 32 bites alsó és felső számra. A fenti logika alapján valósítottam meg a 128 bites és a 64 bites lcg-t, melyek forráskódja a

függelékben található. A megoldásnak köszönhetően a vizsgálandó lcg -k száma 4-re nőtt, mivel a 64 bites lcg-t implementáltam 32 bites számokat használó megoldással is.

A generátorok sebességének méréséhez a grafikus kártyán indított kernelekkel egy vagy több szálon, a különböző generátorokkal szálanként 1 000 000 véletlen számot sorsoltam 0 és 1 között. Generátorokra külön-külön lebontva mértem az időt a kernel indulásától számítva a szálanként utolsó véletlen szám GPU-memóriából host-memóriába töltéséig. Három különböző szálmennyiség mellet generátoronként három mérést hajtottam végre. Három eset:

1. 450 blokkon 64 szálat futtattam blokkonként, míg a Sobol generátor esetében 225 blokkon 64 szálat futtattam blokkonként
2. 450 blokkon 32 szálat futtattam blokkonként, míg a Sobol generátor 225 blokkon 32 szálat futattam blokkonként,
3. Egy szálon futtattam a generátorokat.

	lcg 32	lcg 64	lcg 64 with 32	lcg 128	additive lcg	xorshift	xorshift eredeti
T [ms] – 1. 288E8 véletlen-szám	8544.16	27546.98	70965.42	128612.81	82392.80	10497.76	10496.82
	8547.61	27546.97	70965.49	128612.86	82450.57	10500.67	10494.57
	8543.97	27547.05	70965.32	128612.60	82364.24	10501.86	10494.51
T átlag [ms]	**8545.25**	**27547.00**	**70965.41**	**128612.76**	**82402.54**	**10500.10**	**10495.30**
v [véletlen-szám / ms]	**3370294.34**	**1045486**	**405831.51**	**223928.02**	**349503.79**	**2742832**	**2744084.99**
T [ms] – 2. 144E8 véletlen-szám	4721.29	14587.29	37528.19	67573.64	48226.27	5880.98	5877.38
	4722.45	14587.32	37528.33	67573.28	48202.47	5880.95	5877.21
	4718.99	14587.39	37528.18	67573.63	48240.23	5880.96	5877.11
T átlag [ms]	**4720.91**	**14587.33**	**37528.23**	**67573.52**	**48222.99**	**5880.96**	**5877.23**
v [véletlen-szám / ms]	**3050260.22**	**987158**	**383711.12**	**213101.24**	**298612.76**	**2448578**	**2450132.34**
T [ms] – 3. 1E6 véletlen-szám	1203.81	2623.61	8179.33	12469.58	13672.04	1419.87	1419.87
	1203.81	2623.60	8179.31	12469.55	13670.80	1419.86	1419.86
	1203.81	2623.60	8179.31	12469.55	13671.51	1419.86	1419.86
T átlag [ms]	**1203.81**	**2623.60**	**8179.31**	**12469.56**	**13671.45**	**1419.86**	**1419.86**
v [véletlen-szám / ms]	**830.70**	**381.15**	**122.26**	**80.20**	**73.15**	**704.29**	**704.29**

4. táblázat Általam implementált generátorok sebessége

	global memory	XORWOW	MRG	Sobol 32	Scramled Sobol 32	Sobol 64	Scrambled Sobol 64
T [ms] – 1. 288E8 véletlen-szám Sobols fele anyit generált	7552.90	17044.62	111919.99	38028.55	43395.97	79181.95	84517.48
	7374.15	17044.53	111848.98	38042.09	43771.21	79650.67	84646.36
	7245.52	17044.69	111814.94	38081.07	50819.80	81990.73	85002.84
T átlag [ms]	**7390.86**	**17044.61**	**111861.30**	**38050.57**	**45995.66**	**80274.45**	**84722.23**
v [véletlen-szám / ms]	**3896706.98**	**1689683.33**	**257461.69**	**756887.47**	**626146.05**	**358769.20**	**339934.41**
T [ms] – 2. 144E8 véletlen-szám Sobols fele anyit generált	6566.49	9073.75	67563.74	21145.83	21503.10	35084.74	34315.73
	6869.99	9073.93	67543.42	21144.82	21501.72	35085.80	34315.79
	6561.83	9073.89	67543.84	21143.86	21503.02	35084.35	34316.81
T átlag [ms]	**6666.10**	**9073.86**	**67550.33**	**21144.84**	**21502.61**	**35084.96**	**34316.11**
v [véletlen-szám / ms]	**2160182.63**	**1586976.72**	**213174.37**	**681017.36**	**669686.06**	**410432.25**	**419627.96**
T [ms] – 3. 1E6 véltetlen-szám	3223.18	5200.81	41035.27	12540.67	12561.02	13410.32	13409.99
	3223.11	5200.80	41035.16	12541.00	12561.09	13410.29	13409.98
	3223.00	5200.80	41035.18	12540.63	12561.11	13410.34	13409.98
T átlag [ms]	**3223.10**	**5200.80**	**41035.20**	**12540.76**	**12561.07**	**13410.32**	**13409.98**
v [véletlen-szám / ms]	**310.26**	**192.28**	**24.37**	**79.74**	**79.61**	**74.57**	**74.57**

5. táblázat CuRand könyvtár generátorainak sebessége

Ahogy a 4. és 5. táblázatban látható, a több szálon való párhuzamos futtatás esetében a globális memóriából olvasás a leggyorsabb metódus, aminek az lehet az oka, hogy a lehető legkevesebb regisztert használja, és sok blokkot futtathatnak egyszerre a MP -ok. Természetesen ez nem kifizetődő, mert minden szálnak előre le kell generálni a véletlen számot, és le kell foglalni az eszköz memóriáján a véletlen szám tárolásához szükséges területet, ami plusz időbe telik, és sok globális memóriát emészt fel.

A 32 bites lcg volt a leggyorsabb az lcg -k közül, és a globális memóriát leszámítva a leggyorsabb generátor. A XORSHIFT a 32 bitesen kívül az összes lcg-t megelőzte sebességben. A XOWOW is egy Xorshift generátor, de lassabban teljesített, mint az általam implementált. Ennek egyik oka lehet, hogy a generátor a státuszparamétert a globális memóriára menti, ami lassíthatja a

működését, mivel minden függvényhívásnál a globális memóriából olvassa be a generátor státuszát. Érdekes eredményt adott a 32 bites számokkal implementált 64 bites lcg az eredetihez képest, mivel jóval lassabban teljesített. Ezért még külön a kétféle generátor 1E10 véletlenszám-generálását és globális memóriába írását mértem CUDA 2.3 SDK -n fordítva, és GTX 285 -n futtatva. Az eredeti 64 bites lcg lefutásához 112902 ms kellett, míg a 32 bites megvalósítással csak 66631 ms. Látható tehát, hogy a 2.3-as CUDA verzióhoz képest jóval optimálisabban használja a 64 bites számábrázolást a CUDA 5.0, hiszen az újabb teszteken a 64 bites számábrázolást használó generátor a gyorsabb.

A generátorok minőségét, azaz az általuk előállított számok véletlenszerűségét a Die Hard[9] tesztelő csomag továbbfejlesztésével, a Die Harder[10] -rel ellenőriztem. A tesztek felteszik azt a nullhipotézist, hogy a véletlenszám-generátor tökéletes, és bármely kezdeti paraméter használatával végtelen véletlen szekvencia előállítására képes. Természetesen ez nem lehet igaz, mivel a vizsgált generátorok rendelkeznek periódussal. Végtelen hosszú szekvenciát nem vizsgálhatunk, így véges hosszú szekvenciát vizsgálunk több tesztel. A tesztek megértéséhez nézzünk egy példát.

A példa tesztünk az lesz, hogy összeadunk t db 0 és 1 közé eső, egyenletesen sorsolt számot. Tudjuk, hogy egy futás esetén a várhatóértéke ennek az összegnek $\mu = 0.5t$, és nagy t esetén a független számok összege közelíthető normális eloszlással, melynek szórása: $\sigma = \sqrt{t/12}$. Több ilyen teszt egymás utáni lefutása tekinthető mérésnek, ahol a szummák közül mintavételezett értéket jelöljük x -el. Annak a valószínűsége, hogy elérjük azt a várhatóértéket, amit egy tökéletes véletlenszám-generátor adott volna az alábbi tesztre, a következő képlettel számolható:

$$p = erfc\left(\frac{|\mu - x|}{\sigma\sqrt{2}}\right) \quad \text{, [8] (8.)}$$

ahol az *erfc* a komplementer-hibafüggvény.

Ez az érték lényegében a *p*-értéke a feltett nullhipotézisnek, melynek kis értéke esetén ($0.05 <$) elutasítjuk azt. Az 1-hez túl közeli érték (< 0.995) ugyancsak hibás eredménynek tekinthető, mivel tudjuk, hogy nem tökéletes a generátor. A csomagokban lévő tesztsorozatok, tehát olyan események vizsgálata, melyeknek eloszlása ismert. A p-értékekből csak annyi határozható meg, hogy a generátor megbukott -e a teszten. Ha a generátor átment a teszten az még nem jelenti azt, hogy a generátor tökéletes.

Nem adna pontos eredményt egy teszt, ha egyetlen p-érték alapján döntenénk el, hogy a generátor jó-e vagy sem. Azonban tudjuk, hogy a tesztek többszöri futásakor kapott p-értékeknek egyenletes eloszlás szerint kell szórnia 0 és 1 között [10]. Tehát tesztek p-értékeinek legenerálása

után vizsgálhatjuk ezen értékek eloszlását, mely a Kolmogorov-Szmirnov (KS) teszt, ami ad egy újabb p-értéket. Ez már sokkal pontosabb képet ad arról, hogy a generátor megbukott a teszten vagy sem. A Die Harder minden tesztet KS teszttel zár, és megbukott eredményt ad $0{,}05 < p$ vagy $0{,}995 < p$ esetén. Gyengén teljesít egy generátor az adott teszten, ha a KS tesztre kapott p-érték, kisebb mint 0,5 vagy nagyobb mint 0,95.

A generált számok véletlenszerűségének vizsgálatához a tesztkészletnek kétféleképpen adtam át a véletlen számokat. Az első megoldás szerint, a grafikus kártyán generált számokat kiírtam a globális memóriába, majd azokat egy bináris fájlba, amelyből a tesztkészlet beolvasta a számokat. Ezzel az a probléma, hogy a tesztenként felhasznált számok mennyisége 10 - 8000 millió, így a program bizonyos tesztek esetében a szükséges mennyiségű véletlen számokat csak a bináris fájl ciklikus olvasásával tudja legenerálni, ami hibás eredményhez vezethet.

E hiba elkerülésére implementáltam a véletlenszám-generátorokat CPU -ra úgy, hogy előjel nélküli egész számokat adjanak, és a Unix csővezeték segítségével a standard bemeneten adtam át azokat (függelékben található a futtatáshoz szükséges szkript).

A generátorokat össze tudjuk hasonlítani az alapján, hogy hány teszten mentek át. A Die Harder minden tesztet KS tesztel zár, amely az adott teszt 100 p-értékét vizsgálja. A tesztek hasonlóak a példateszthez, amelyek pontos listája és leírása megtalálható a tesztkészlet oldalán[10]. A Die Harder-ből kapott eredményeket a függelékben található táblázatok tartalmazzák.

A táblázatok ntuple paraméterei azt jelenti, hogy az adott teszt a bit folyamot hány bitenként osztja fel, amelyekből előjel nélküli egész számokat generál és ezekre futtatja le a tesztet.

A Die Harder tesztkészlet eredményekből az tűnik ki, hogy a Xorshift, XORWOW és MRG32Kta generátorok teljesítettek a legjobban. Ezek a generátorok az összes teszten átmentek és csak 2 vagy 4 helyen mentek át gyengén. A Sobol generátorok több mint a tesztek felén megbuktak, ami az eredmények közül a legrosszabb, azaz a generátorok által szolgáltatott számok a legkevésbé véletlenszerűek. Az lcg -k közel hasonlóan teljesítettek, mivel a generátorok által előállított számokat előjel nélküli egész számmá kellett konvertálnom, így a számábrázolás és *m* modulusbeli különbségek kevésbé jelentek meg az eredményekben.

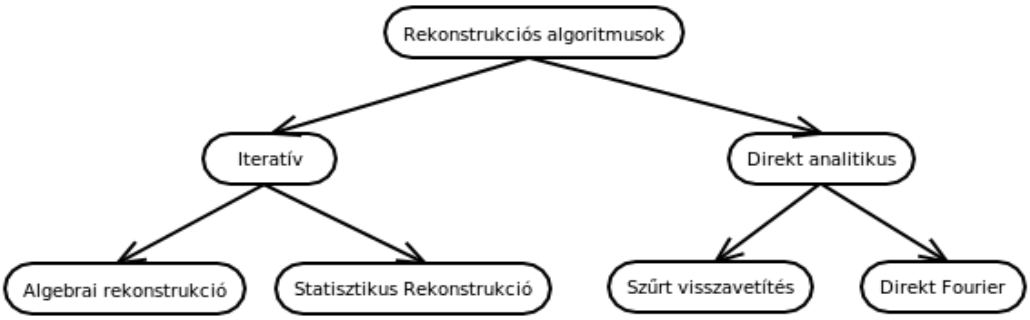
Összességében a vizsgált generátorok közül a XORSHIFT generátorok szerepeltek a legjobban az általam használt teszteken. Sebességük nagyobb a 64 bites lcg generátorénál, és periódushosszuk megfelelően nagy ahhoz, hogy hosszabb futások indítása esetén se érjen körbe a generátor a periódusán. Az általam implementált Xorshift gyorsabb a XORWOW generátornál, hátránya viszont, hogy szálanként 5 regisztert használ. A cuRand könyvtár generátorai nagyon

használhatónak bizonyultak, és sok hasznos függvényt tartalmaznak a generátorokhoz. Képes különböző eloszlások szerint sorsolni a számokat, és könnyen kezelhetőek a generátorok. Sajnos ezeket már nem tudtam implementálni a kódomba, mivel a generátorok megjelenése jóval a rekonstrukcióm lekódolása után történt. A generátorok közül, így a 64 bites lcg -t használtam, mivel ennek kellően nagy a periódushossza, és csak 1 regisztert vesz igénybe, ami az általam használt kártyánál fontos szempont. A jövőben újabb és gyorsabb kártyák használata esetén a Xorshift vagy XORWOW generátor használatát javaslom.

3. Rekonstrukciós algoritmus

3.1 CT berendezések képrekonstrukciós algoritmusai

Az első CT 1974 -es orvosi alkalmazásra való üzembe helyezése óta számos rekonstrukciós algoritmust dolgoztak ki, melyek két fő részre oszthatok: az iteratív vagy a direkt analitikus rekonstrukciók csoportjába. (4. ábra).



4. ábra Rekonstrukciós algoritmusok típusai [11]

Az analitikus megoldások közé tartozik a szűrt visszavetítés, amely a mai napig a leggyakrabban használt rekonstrukciós algoritmus, amit egyszerűségének és gyorsaságának köszönhet. Hátránya, hogy műtermékektől nem mentes, ami javítható további szűrésekkel és utómunkával, de ezek mind időigényes és bonyolult műveletek. Analitikus rekonstrukció még a direkt Fourier-rekonstrukció, amely a központi szelet tételén alapul. E rekonstrukció megvalósításában az a nehézség, hogy az algoritmusban szereplő Fourier-transzformáció során polár koordináta-rendszerben kell mintavételeznünk a Fourier-teret, és ez sok pontatlansághoz vezet.

Az iteratív rekonstrukciós algoritmusok, bár kevesebb műtermékkel rendelkeznek, a nagy számításigény miatt kevésbé elterjedtek, de egyre több fejlesztést indítanak ilyen típusú algoritmusok kidolgozására. E folyamat mögött a GPU -k nyújtotta nagy számítási sebesség megjelenése is állhat. Az implementálandó algoritmus kiválasztásakor a fentebb leírt tényeket figyelembe véve, az iteratív algoritmusok között kerestem a jobb képminőség érdekében. A nagy számításigényéből adódó lassúságát a GPU -n való implementálással fogom javítani.

$$\mathbf{y} = \mathbf{Ax} \quad (9.)$$

Az iteratív megoldások egyik ága az algebrai rekonstrukciók, mely a 9-es egyenletrendszer különböző iteratív algoritmusokkal történő megoldása (például Kaczmarz), ahol ***y*** a mérési eredmények oszlopvektora, ***x*** a keresett együttható vektor, ami a vizsgált objektumra jellemző. Ezek a CT esetében a makroszkópikus hatáskeresztmetszetek (***μ***), és ***A*** a rendszer-mátrix, amely az objektum leképzését írja le.

Az algebrai eljárás bővítései a statisztikus rekonstrukciók, melyek a 9-es egyenletet egészítik ki úgy, hogy a valóság jobb megközelítése érdekében a mért adatokra a statisztikai mennyiségek várható értékeként tekint:

$$E(\mathbf{y}) = \mathbf{Ax}\ , \quad (10.)$$

Az egyenlet megoldásához többségében a becsléselmélet alapvető technikáját, a Maximum-Likelihood elvet alkalmazzák, amelynek lényege, hogy keressük azt az **x** paramétervektort, aminél az adott mérési eredmény (**y**) maximális valószínűséggel fordulna elő

$$\hat{x} = argmax\rho(y|x) \quad (11.)$$

A kódomba statisztikus iteratív algoritmust implementáltam, mivel ez használja fel a lehető legtöbb fizikai információt, mely várhatóan a lehető legkevesebb műterméket tartalmazó képet eredményezzi. A Maximum likelehood kiegészítését az ML-EM módszert vizsgáltam meg először, ami a modellezés kiterjesztését jelenti plusz virtuális fizikai változókkal. Lange és Carson[12] megoldása alapján a fizikai paraméterek Poisson-eloszlást követnek. A plusz virtuális változók X_{j+1}, a *j -ik* voxelt elhagyó fotonok száma és az X_j a *j* -ik voxelbbe belépő fotonok száma ($1 \leq j \leq m\text{-}1$). A virtuális X_{j+1} változó csak a voxel makroszkópikus hatáskeresztmetszetétől, a foton által a voxelben befutott úthossztól, és X_j -től, azaz a belépő fotonok számától függ. A detektált fotonok száma $Y = X_m$, a forrásból indított fotonok száma $W = X1$. Az X_m megkötése mellett ($X_m - X_j$) változók Poisson-eloszlást követnek, így várhó értéke

$$E(X_j - X_m \vee X_m) = E(X_j) - E(X_m) \quad . (12.)$$

Másképp felírva

$$E(X_j | X_m) = X_m + E(X_j) - E(X_m) \quad . (13.)$$

A fenti egyenlet paramétereit értelmezve X_m a mérési eredmény és $E(X_m)$ a számolt várható érték. Vezessük be M_{ij} és N_{ij} változókat, amik az *i* -ik projekció esetén a *j* -ik voxelbe belépő és abból kilépő fotonok száma. A log likelehood függvény minden projekcióra

$$\log\left(g\left(Y,\mu\right)\right)=\sum_i\left\{-d_i\exp\left(-\sum_{j\varepsilon I_i}l_{ij}\mu_j\right)-Y_i\sum_{j\varepsilon I_i}l_{ij}\mu_j+Y_i lnd_i-\ln Y_i!\right\} \quad , (14.)$$

ami a virtuális változókkal kiegészítve az alábbi egyenletre vezet

$$\sum_i\sum_{j\varepsilon I_i}\left\{N_{ij}\ln\left(\exp\left(-l_{ij}\mu_j\right)\right)+\left(M_{ij}-N_{ij}\right)\ln\left(1-\exp\left(-l_{ij}\mu_j\right)\right)\right\}+R \quad , (15.)$$

ahol *R* tartalmazza az összes olyan paramétert, ami nem függ μ_j-től. A 15. egyenlet minimumát keressük μ_k szerint minden *k* -ra, azaz a parciális deriváltjának nullának kell lennie, azaz

$$0=\sum_{i\varepsilon I_k}N_{ik}l_{ik}+\sum_{i\varepsilon I_k}\left(M_{ik}-N_{ik}\right)\frac{l_{ik}\exp\left(-l_{ik}\mu_k\right)}{1-\exp\left(-l_{ik}\mu_k\right)}=\sum_{i\varepsilon I_k}N_{ik}l_{ik}+\sum_{i\varepsilon I_k}\left(M_{ik}-N_{ik}\right)\frac{l_{ik}}{\exp\left(l_{ik}\mu_k\right)-1} \quad , (16.)$$

ahol *l* a befutott szabad úthosszak adott voxelben adott projekció esetén, *i* a projekciós index, *j* az adott detektor pixel és *k* a voxel index.

A fenti egyenletben utolsó tag szorzóját közelíthetjük, mivel $s = l_{ik}\mu_k$ kicsi.

$$\frac{1}{e^s-1}=\frac{1}{s}-\frac{1}{2}+\frac{s}{12}+O\left(s^3\right) \quad (17.)$$

Különböző rekonstrukciókat kapunk több tag figyelembe vételével. Ha csak az első tagot (1/s) vesszük figyelembe

$$\mu_j^{n+1}=\frac{\sum_{i\varepsilon I_k}\left(M_{ik}-N_{ik}\right)}{\sum_{i\varepsilon I_k}\left(N_{ik}l_{ik}\right)} \quad .(18.)$$

A 18. egyenlet értelmezése a 13. egyenlet használatával

$$\sum_{i\varepsilon I_k}\left(M_{ik}-N_{ik}\right)=\sum_{i\varepsilon I_k}\left[E\left(X_k\right)-E\left(X_{k+1}\right)\right] \text{ és } \sum_{i\varepsilon I_k}\left(N_{ik}l_{ik}\right)=\sum_{i\varepsilon I_k}\left\{\left[\left(y_i^{mért}-y_i^{számolt}\right)+X_{k+1}^{számolt}\right]l_{ik}\right\} \quad , (19.)$$

ahol $y_i^{mért}$ az *i* -ik mérési projekció során detektált fotonok száma, $y_i^{számolt}$ a *i* -ik projekció során számolt fotonok száma. Ezek alapján az algoritmus előrevetítése során nem csak a detektált fotonok számát kell lementenünk, hanem az adott Lor esetén az adott pixelbe befutó és kilépő fotonok számát is. A Woodcock becslő használata mellett az utóbbi két paraméter nehezen számolható, és a szűkös időkereten belül nem találtunk megoldást. Így ezen algoritmus implementálásához szükségese a Ray Marching módszer használata.

A 17-es közelítéshez használt egyenlet újabb tag bővítésével (1/s-1/2)

$$\mu_j^{n+1} = \frac{\sum_{i \varepsilon I_k} (M_{ik} - N_{ik})}{\frac{1}{2} \sum_{i \varepsilon I_k} (M_{ik} + N_{ik}) l_{ik}} \quad . (20.)$$

A 20. egyenlet számlálóját a 19. egyenlet már értelmezi, így a nevezőre kell találnunk egy fizikai értelmezést

$$\sum_{i \varepsilon I_k} (M_{ik} + N_{ik}) = 2 \cdot \left(y_i^{mért} - y_i^{számolt} \right) \quad . (21.)$$

A WoodCock becslés használatához és kisebb memóriaigény érdekében a normális eloszlást követő ML – TR (Maximum Liklehood Transsmission) algoritmust használtam a rekonstrukcióhoz, amely az alábbi képlet alapján számol

$$\mu_j^{n+1} = \mu_j^n + \frac{\sum_{i=1}^{I} l_{ij} \cdot (\hat{y}_i - y_i)}{\sum_{i=1}^{I} l_{ij} \cdot [\sum_{h=1}^{J} l_{ih}] \cdot \hat{y}_i} \quad .[13](22.)$$

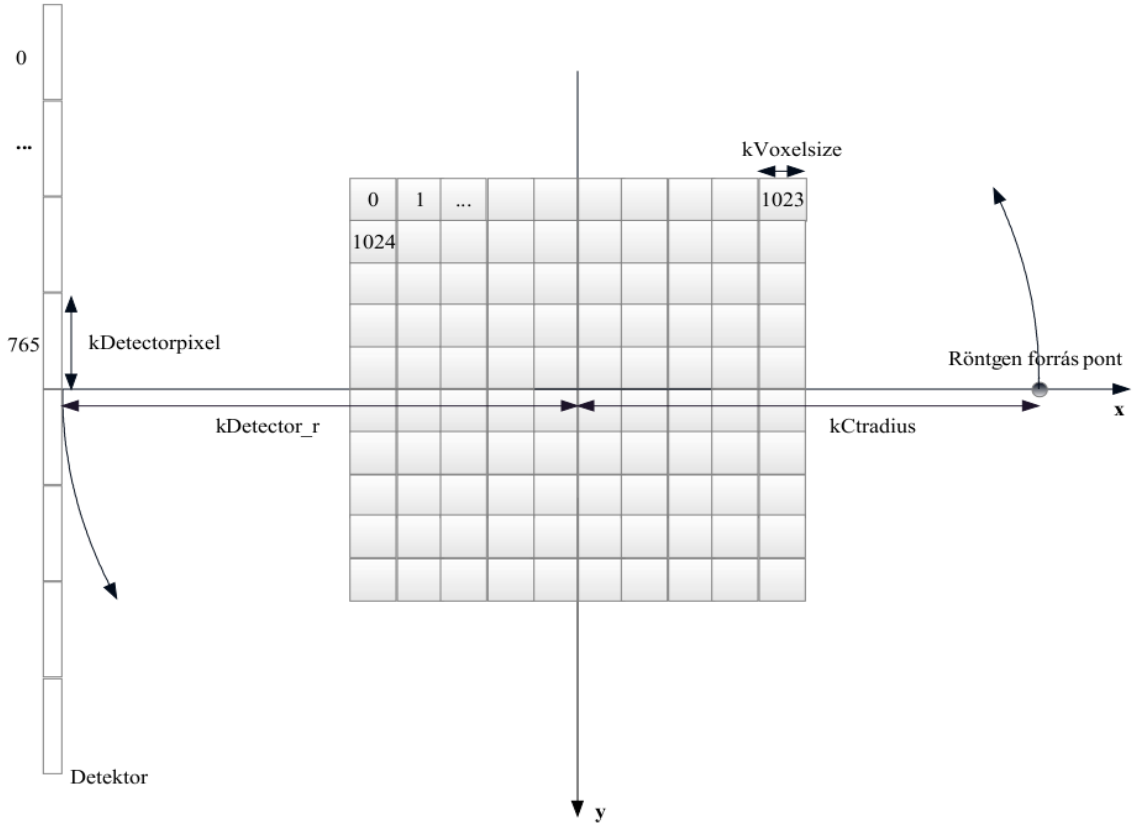
A képletben l_{ij} a j -ik voxelben befutott úthossz az i -ik detektorpixelhez tartozó projekció esetében, $\hat{y}_i$ az i -ik detektorpixelhez tartozó várható beütés, y_i az i -ik detektorpixelhez tartozó mért beütés és μ_j^n a j -ik voxel makroszkópikus hatáskeresztmetszete n. iterációnál.

A képlet alapján szükségünk lesz egy előrevetítés függvényre, amely előállítja a $\hat{y}_i$ értékeket minden iterációnál az aktuális μ_j^n értékek mellett. Ez könnyen párhuzamosítható, tehát a feladatom az, hogy kernel segítségével megvalósítsam az előrevetítést, erre a továbbiakban előrevetítési kernelként fogok hivatkozni. Az l_{ij} paramétereket a visszavetítésből tudjuk meghatározni, amelyre a képlet kiszámítása mellett is sort keríthetünk. Tehát elégséges lesz egy másik kernel megvalósítása, amely tartalmazni fogja a visszavetítést és a fenti képletet. A továbbiakban ezt rekonstrukciós kernelnek nevezem.

3.2 Vizsgált elrendezés

A szimulált geometria alapjának egy cone beam kisállat CT-t választottunk, hogy a vizsgálandó rendszer memóriaigénye elfogadható méretekkel rendelkezzen. A rekonstruálandó térfogat egy voxelekre felosztott téglatest, ahol a voxel egy adott élhosszúságú kocka, amelyben lévő anyagot homogénnek és állandó sűrűségűnek tekintjük. A voxelekből felépített téglatest, nevezzük a

voxelrendszernek, egy 1024x1024x2048 -as mátrix (egyszeres lebegőpontos voxel értékek esetén 8 GB csak a mért anyag). A téglatest középpontja a koordináta rendszer origója, és az e ponton áthaladó z tengely a forgási tengely, ami körül forog a forrás és a detektor (5. ábra). A detektor egy 1536x864-es pixelekből felépített mátrix, amely középpontja az origótól 'kDetector_r' távolságra helyezkedik el, és a pontforrás vele mindig ellentétes oldalon található 'kCtradius' távolságra az origótól.



5. ábra Felépített geometria z metszete

Az előbb említett paraméterek állíthatók a program 'config.h' fájljában, csakúgy mint a 'kVoxelsize', a voxelek oldalainak hossza, és a kDetectrorpixel, a detektorpixelek szélessége.

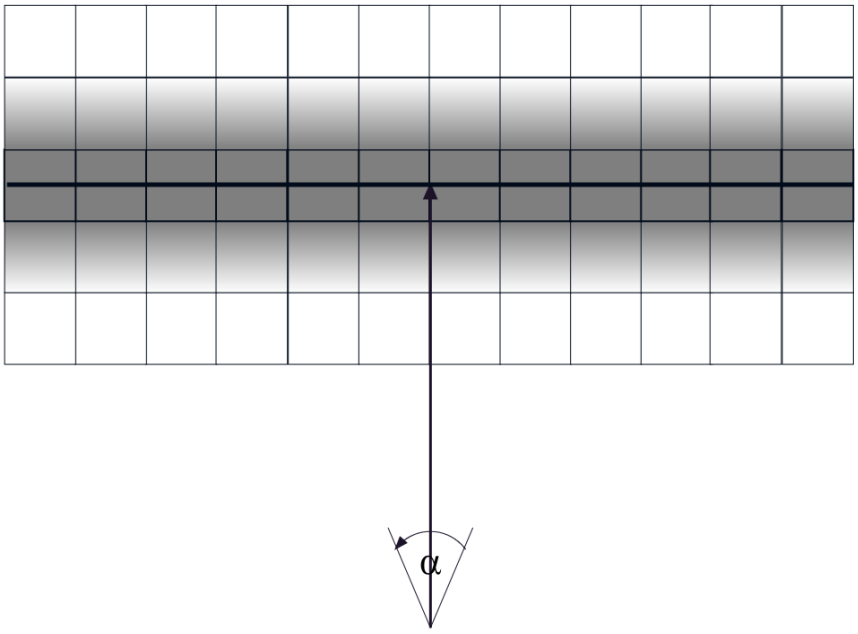
A beállított paramétereknél figyelnünk kell arra, hogy a detektor ne kerüljön a voxelrendszerbe, azaz kVoxelsize és kDetector_r megválasztásakor az alábbi feltételnek kell teljesülnie

$$kDetector\_r < -kVoxelsize \cdot 512 \text{ , (23.)}$$

ahol kDetector_r negatív előjellel rendelkezik, mivel a tengely negatív oldalán található. A pontforrás sem kerülhet a voxelrendszerbe, az erre felírt feltétel

$$kCtradius > kVoxelsize \cdot 512 \text{ . (24.)}$$

Az algoritmus ellenőrzésére csak 2D -s felvételeket vizsgáltam, amit úgy értem el, hogy fan beam CT-t szimuláltam és a voxel- és detektorrendszerek *z* irányú méretét 1-re csökkentettem. A kódom felismeri, ha a voxel- , vagy a detektorrendszer valamely dimenziójának mérete páratlan, és ekkor fél voxellel és detektorpixellel lejjebb tolja a rendszerek origóját, ahogy a 6. ábrán is látszik.



6. ábra A 2D-os képek készítéséhez választott geometria. Látható, hogy a forrásból a detektor középpontjába mutató vektor itt már a detektorpixelek felére mutat, szemben a 3D-os geometriával, ahol a rendszerek mérete a *z* dimenzió mentén páros.

A detektor és a forrás *z* tengely körül forog, és a felvételek 'step and shoot' módban készülnek. Az így kapott mérési eredményeket egy ***y*** tömbbe töltjük, amelynek méretét a detektorpixelek száma és a szögállások szorzata alapján kapjuk meg.

A kisállat CT paraméterei egy létező CT elnagyolt másolata (6. táblázat), ezt az előrevetítés verifikáláshoz és a rekonstruált kép elkészítéséhez használtam.

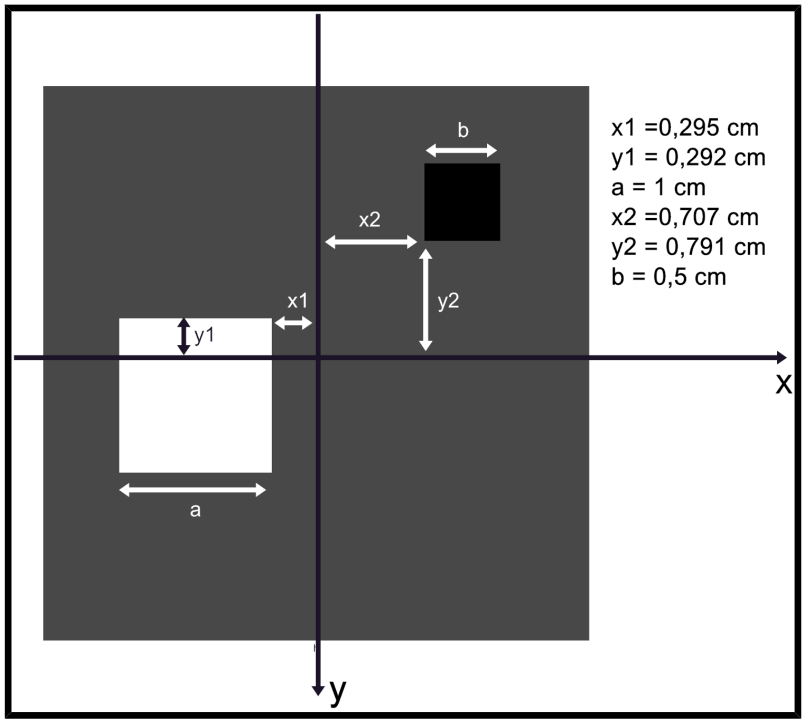
kCtradius [cm]	kDetector_r [cm]
19	-10.2
VoxelpixelnumberX [db]	VoxelpixelnumberY [db]
1024	1024
detectorPixelnumberY [db]	detectorPixelnumberZ [db]
1536	1

6. táblázat Az előrevetítés validálásához használt paraméterek.

A 3D kép készítéséhez a GPU memóriakapacitása miatt szükséges feldarabolni a voxelrendszert, ezért a kódomat már úgy alakítottam ki, hogy a *z* tengely menti feldarabolás esetén a feldarabolt részek egymás utáni futtatása megoldható legyen a jövőben.

3.3 Teszt geometria

Az abszorpciós értékek validálására és a rekonstrukciós algoritmus tesztelésére a 7. ábrán látható szimulált fantomot használtam, amely egy homogén vízzel feltöltött négyzet, ami két eltérő nagyságú és sűrűségű négyzetet foglal magába. A négyzetek különböző sűrűségű vizet tartalmaznak: a fehér négyzet sűrűségének 10 kg/dm^3, míg a kisebb, fekete négyzet sűrűségének 0,001 kg/dm^3 választottam. Az előrevetítés ellenőrzésére a programomban és az MCNP -ben felépítettem a fantomot, és a beütések regisztrálására az MCNP -ben F4 rács 'tally' -t használtam.



7. ábra A validálásra használt fantom.

A két programban spektrum helyett 100 KeV-es fotonokat sorsoltam. Annak érdekében, hogy az MCNP szimuláció se tartalmazzon szórást, az ott kapott beütések energiáját ablakoltam 90 és 100 KeV között. A szimulációk összehasonlítására megvizsgáltam három forrás-detektor pozíciót. A pontforrás és origó között húzott egyenes és az x tengely által közbezárt szögek a három különböző pozícióban: 0°, 30° és 90°. Ezek a pozíciók a CT forgását akarják modellezni, azaz a CT forgása közben készült felvételeket ellenőriztem 2 dimenzióban.

A rekonstrukció ellenőrzésére a fent leírt fantom segítségével szimuláltam mérési eredményeket annyi módosítással, hogy a fehér négyzet sűrűségértékét kisebbre vettem (2,344 kg/dm^3).

3.4 Az előrevetítő kernel

3.4.1 Az előrevetítő megvalósítása

A már korábban említett várható beütések számát Monte Carlo szimulációval becsli a programom, amelyet egy, a GPU -n futó kernellel valósítottam meg. A foton voxelrendszer áthaladás leszimulálására Woodcock módszert választottam. Ennek az a lényege, hogy a vizsgált objektumban előforduló legnagyobb hatáskeresztmetszettel sorsoljuk a szabad úthosszakat, így biztosítva, hogy az elégséges legsűrűbb lépésközt használjuk a foton anyagon való áthaladás

leszimulálására. Foton-kölcsönhatás akkor történik adott pozícióban, ha az ott lévő aktuális hatáskeresztmetszet és a majoráns hatáskeresztmetszet hányadosánál kisebb véletlen számot sorsoltunk. A szabad úthosszakat a 26. egyenlet alapján számoljuk, és hogy végbe ment kölcsönhatás azt a 27 -el.

$$l = \frac{-1 \cdot \log(x)}{\Sigma_m} \quad , (26.)$$

ahol l a szabad úthossz, x egy (0,1] tartományon sorsolt véletlen szám, és Σ_m a legnagyobb hatáskeresztmetszet az objektumban, azaz a majoráns hatáskeresztmetszet.

$$x < \frac{\Sigma_i}{\Sigma_m} \quad (27.)$$

Az *i*-ik voxelbe érkezett foton kölcsönhatásba lép, ha 3.3.2 -es egyenlet teljesül, ahol és Σ_i az *i* -ik voxel makroszkópikus hatáskeresztmetszete. A kutatás jelen fázisában a szórásokat nem modelleztem, így
minden kölcsönhatás esetén fotoeffektus megy végbe. Az előrevetítést végző kernel feladatai a fentiek alapján:

- kiszámolja a foton aktuális indulási pontját, azaz a röntgencső aktuális pozícióját
- irányt sorsol a fotonnak
- bekapcsolt spektrum esetén energiá(ka)t sorsol a fotonnak
- a foton végig halad Woodcock módszerrel a voxelrendszeren
- a vizsgált terület elhagyása után a fotont rávetíti az aktuális sebesség vektorával a detektor felületére, és regisztrálja melyik pixelbe adott beütést, ha a foton nem nyelődött el

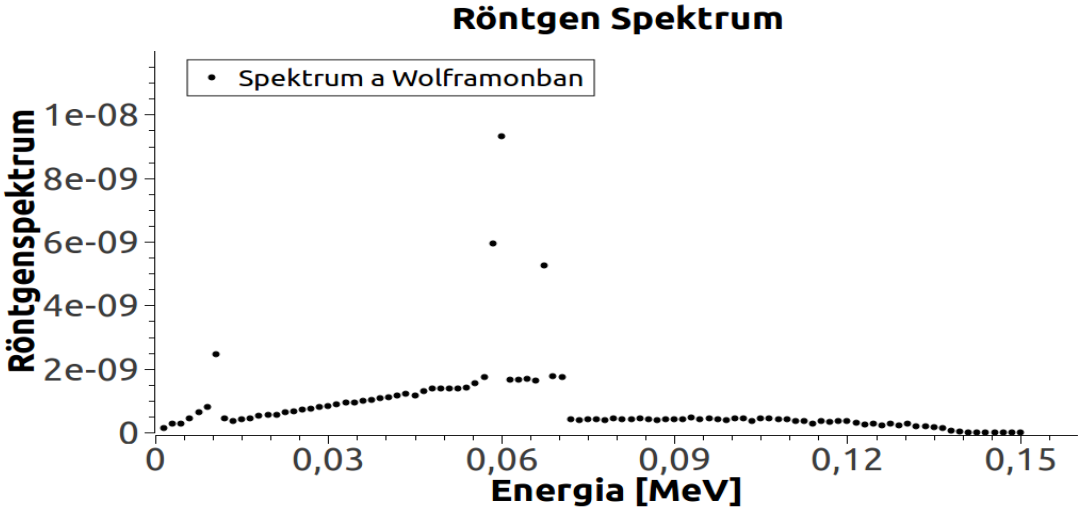
A csoportos fotonkövetés volt a diplomamunkám egyik célkitűzése. Ezt úgy valósítottam meg, hogy egy fotonhoz több energiaértéket sorsoltam, és minden energiaértékhez külön súlyértéket rendeltem, amelyeket a foton indításkor 1-nek választottam. Az effektussorsolást minden energiára külön végrehajtottam, és adott energián bekövetkező fotoeffektusnál az energiához tartozó súlyfaktort 0-ra állítottam.

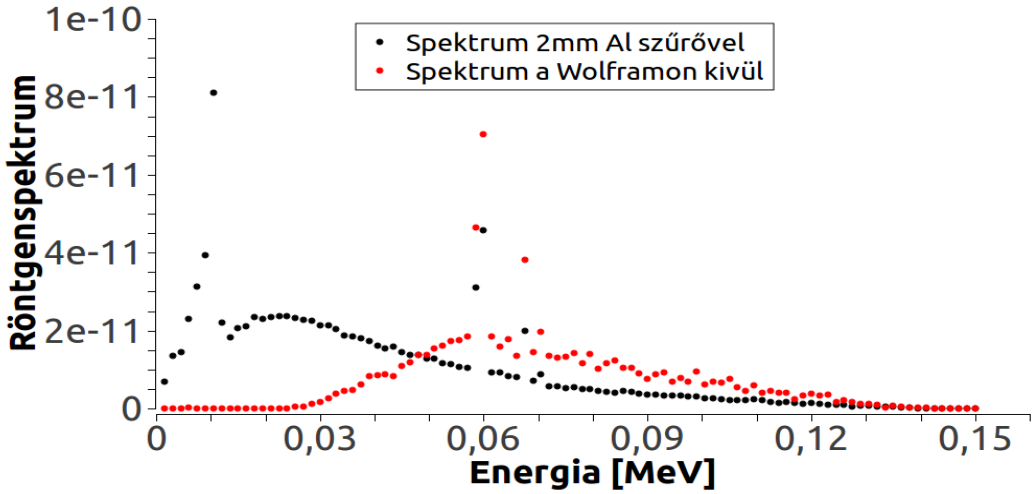
A rekonstrukció futása közben a röntgencső szimulációját, és az ez alapján számolt fotonenergia-számolást nem valósítottam meg, mivel az elektron szimulációk túlzott mértékben lelassítanák a kódom. Helyette a fotonenergiákat a már előre elkészített spektrumból számolt energia eloszlásból sorsolom. A spektrumokat kézhez kaptam, amelyeket MCNP szimulációval generáltak [11,12]. A kapott röntgenspektrumok volfrám elnyelő közegbe irányított elektronok

reakciójából keletkeztek. háromféle különböző spektrumot három különböző tally-zással érték el:

- Volfrámon belül mért fotonok spektruma.
- Volfrámon kívül tapasztalható spektrum, ahol már látható a target anyag elnyelése.
- 2 mm-es alumínium szűrővel ellátott röntgen katódcső spektruma, ahol a target anyag elnyelése mellett a katódcső ablakán elhelyezett szűrő elnyelése is érzékelhető.

A szimulációból kapott spektrumok a 7. ábrán láthatók.





7. ábra Program által használt röntgenspektrumok

A spektrumokon látható csúcsok a volfrám -ra jellemző karakterisztikus röntgencsúcsok, melyek energiája az anyagra jellemző, és pontosan meghatározott. Karakterisztikus röntgensugárzás akkor keletkezik, ha a beeső nagy energiájú elektron az atom egyik belső héj-elektronjával ütközik, és megfelelően nagy energiája van ahhoz, hogy az elektront kiüsse a helyéről. Az ott kialakult vakanciába egy magasabb pályán mozgó elektron kerül, amely eközben fotont bocsát ki. E foton energiája a pályák energiakülönbségével megegyező, tehát az anyagra jellemző. A volfrámra jellemző karakterisztikus röntgenek energiája: 9,6 KeV, 11 KeV, 59 KeV, 67,2 KeV, 69,1 KeV. Ezen csúcsokból az spektrumok energiafelbontása mellett csak három látszódhat. E három csúcs 10, 59 és 68 KeV környékén lehetnek. Ezen csúcsok megtalálhatók a spektrumainkban, így feltételezhetjük, hogy ezek megfelelnek a valóságban a röntgencső által kibocsátott spektrummal. Az alumíniumszűrő használata mellett kapott görbén a kis energiás karakterisztikus csúcs nem jelenik meg, mivel azt kiszűri a szűrő.

A spektrumokból inverz kumulatív eloszlásfüggvény módszerrel lehet energiákat sorsolni, azaz a sűrűségfüggvények kinyeréséhez normáltam a hisztogramokat, majd ezekből kiszámoltam az eloszlásfüggvényt, végül vettem az inverzüket, hogy a [0,1) intervallumon sorsolt véletlen szám energiaértéket adjon vissza.

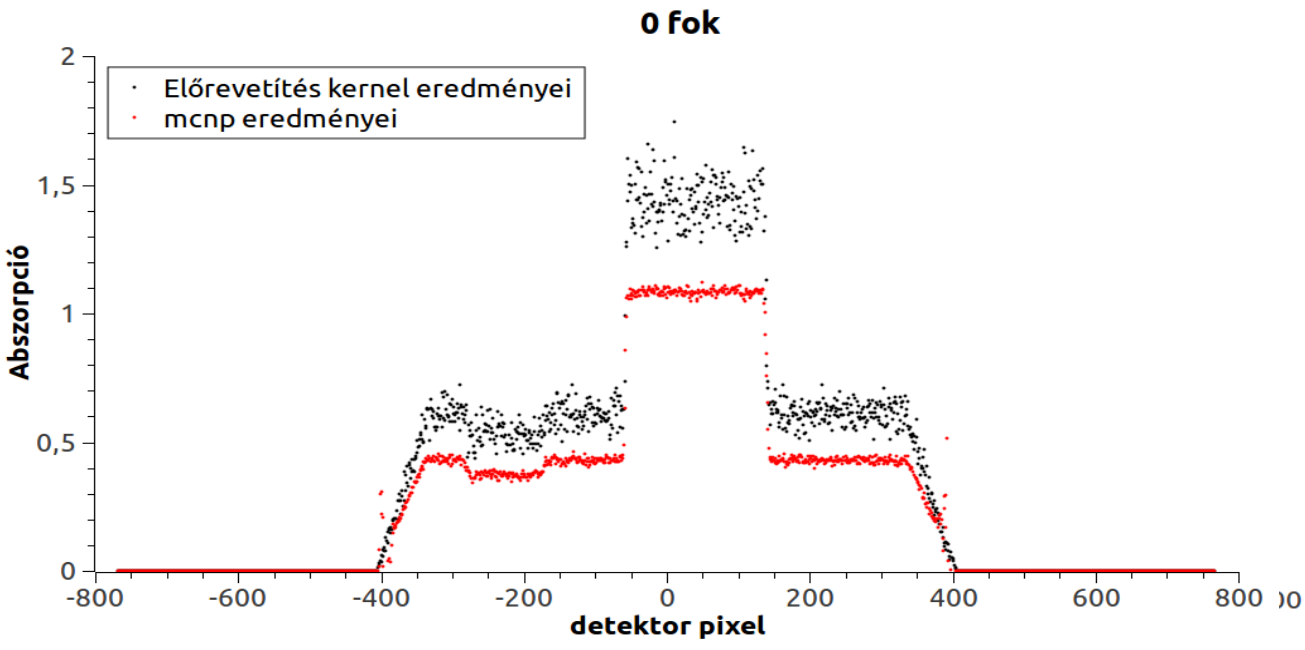
A kernel futása közben valamilyen módon regisztrálnom kell a beütéseket. Ha betöltjük a memóriába a detektorrendszert és oda regisztráljuk a beütéseket, akkor előfordulhat, hogy különböző szálak egyszerre akarnak azonos memóriacímre írni. A probléma elkerülése érdekében a detektorrendszerbe írás helyett regisztrálom a globális memóriába a beütésekhez tartozó

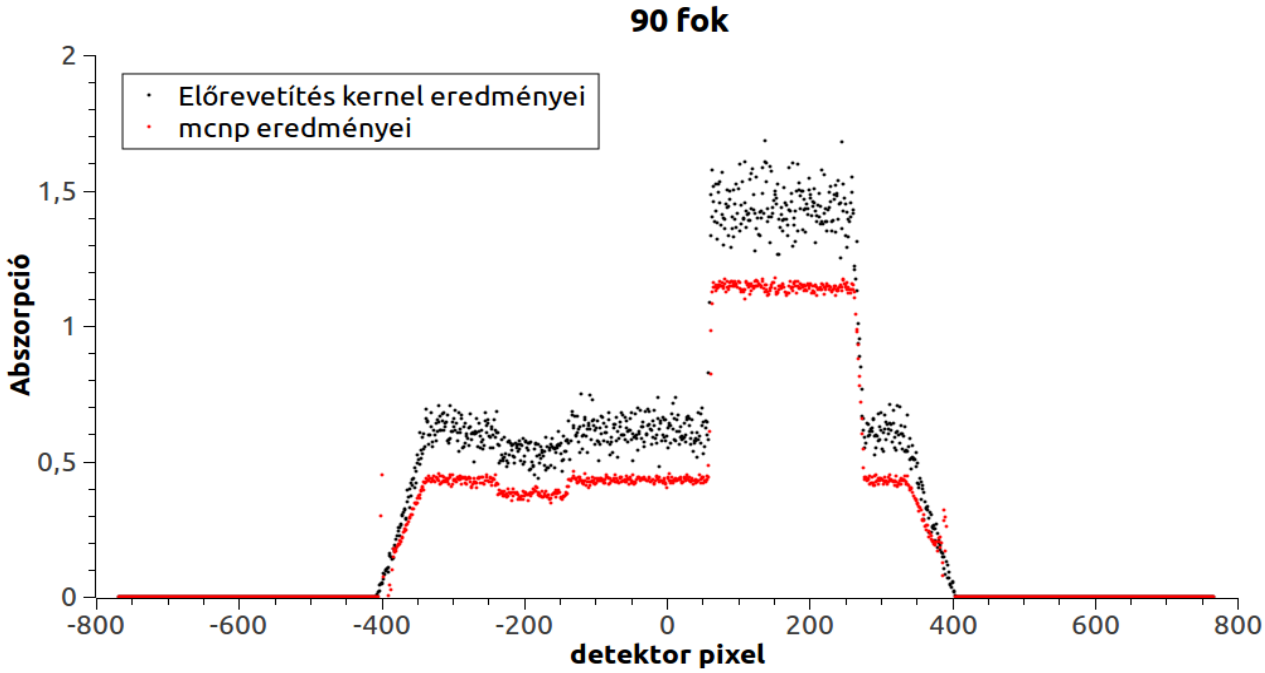
detektorpixel-indexeket, és csak a számítógép memóriájában foglaljuk le a teljes rendszert. A beütésekből abszorpciós értékeket kell számolnom, ezért szükséges elmentenünk az elnyelő anyag nélküli voxelrendszer esetén kapott beütések számát. Az abszorpciót az alábbi képlet alapján számoljuk: $y[i] = -\lg(I[i]/I_0[i])$, ahol $I[i]$ a mért beütések száma az i detektorpixelnél, $I_0[i]$ a beütések száma i detektorpixelnél elnyelő anyag nélküli voxelrendszer esetén.

Az optimális futás érdekében a GPU -n létrehoztunk két egész tömböt, amelyek az I és I_0 beütések indexeit regisztrálja, ezzel elkerülve a kernel kétszer egymás utáni futtatását, azaz a szimuláció alatt egyszerre regisztrálom mindkét típusú beütést. A program a grafikus kártyán fennmaradt maradék memóriát hasznosítja, és abból számolja ki az egy szálra jutó fotonok számát. Az előrevetítő kernelt szükséges verifikálni, amelyet a következő alfejezetben taglalok.

3.4.2 Abszorpciós értékek verifikálása

Az MCNP eredmények abszorpciós értékekre történő átszámításhoz szükséges volt egy fantom nélküli futást is indítanom. A jó statisztikájú MCNP[14-15] eredmények eléréséhez 10^{10} fotont indítottam. A programomból kapott pixelek indexeit 768-al eltoltam negatív irányba, mivel az MCNP eredmények detektorpixel indexei [-768, 767] intervallumon helyezkednek el. A két program által szolgáltatott eredményeket a 9. ábrán láthatóak. A 0º és 90º eredményei lényegében a fantom x és y tengely menti levetítései. A homogén anyagban lévő két eltérő sűrűségű négyzet három plató formájában jelenik meg. A 0º esetében -350 és 350 között egy alapplató látható, amelyből kiemelkedik -40 és 180 pixel között egy hatalmas elnyelés. Ennek oka a nagy sűrűséggel rendelkező négyzet. -330 és -180 között látható az alapplatón egy bemélyedés, amely a kisebb sűrűségű négyzet következménye. A 90º-os elrendezés esetén is három plató látható, csak más detektorpixeleknél, mivel ez az elrendezés a fantom y tengely menti levetítés. A 30º -os detektor-forrás pozíciónál a platók helyet egyre inkább csúcsok láthatók, melynek oka, hogy a fantomon áthaladó fotonok úthossza nem egyenlő.





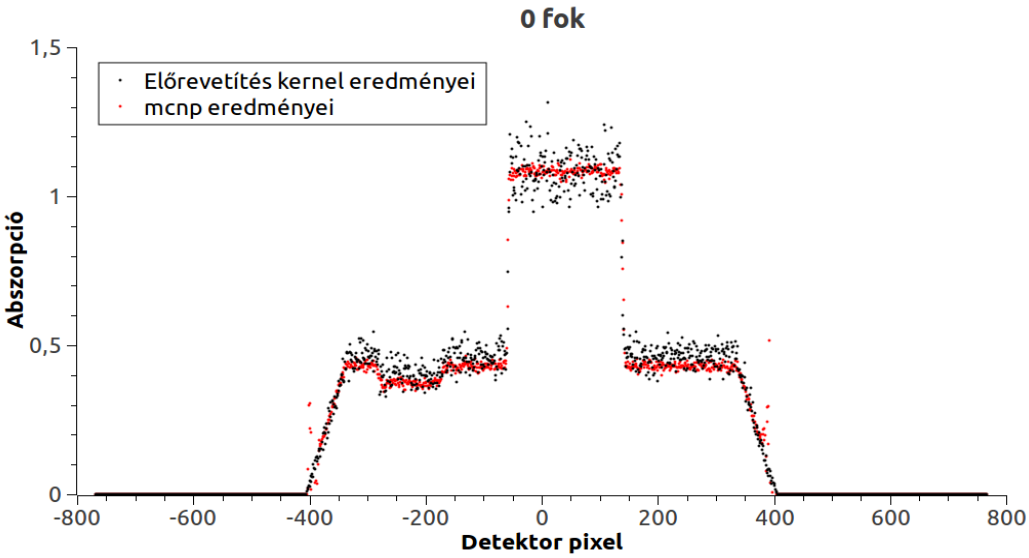
9. ábra A Monte Carlo alapú előrevetítés kernel és az MCNP eredmények összevetése három különböző projekciónál. Az abszorpciós értékek (**y[i] = – lg (I[i]/I_0[i])**) vannak ábrázolva az *i*-ik detektorpixel függvényében.

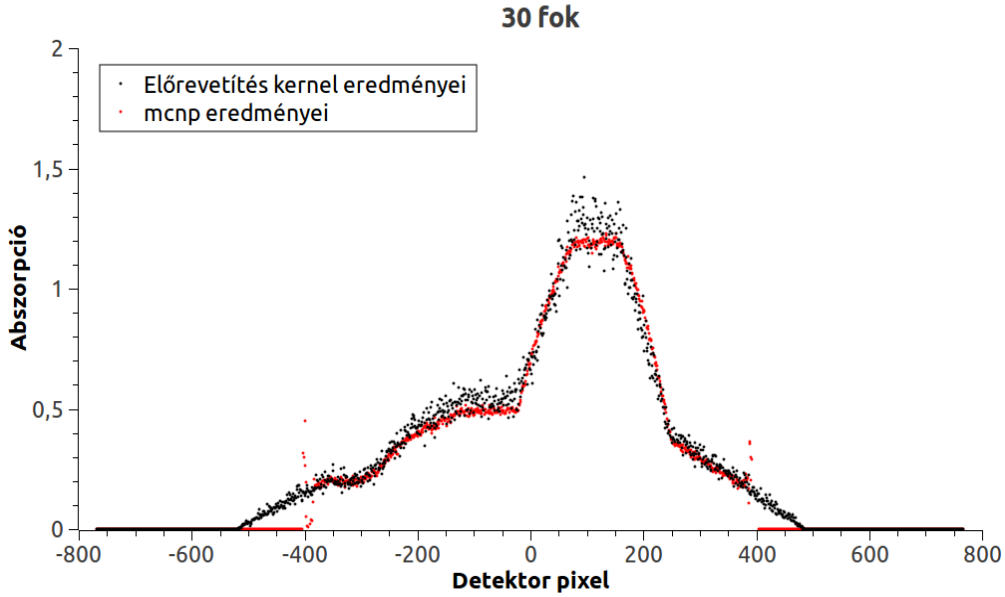
A két különböző programból kapott abszorpciós görbék alakjai hasonlóságot mutatnak, így feltételezhetjük, hogy mindössze egy normálási faktor az eltérés közöttük. Ennek bizonyítására kiszámítottam a faktort, úgy hogy vettem a legnagyobb abszorpciós csúcsokat, azaz a nagy sűrűségből adódó elnyeléseket és átlagoltam a csúcsok platóján lévő pontok értékét. Az így kapott értékek a különböző pozíciók esetében:

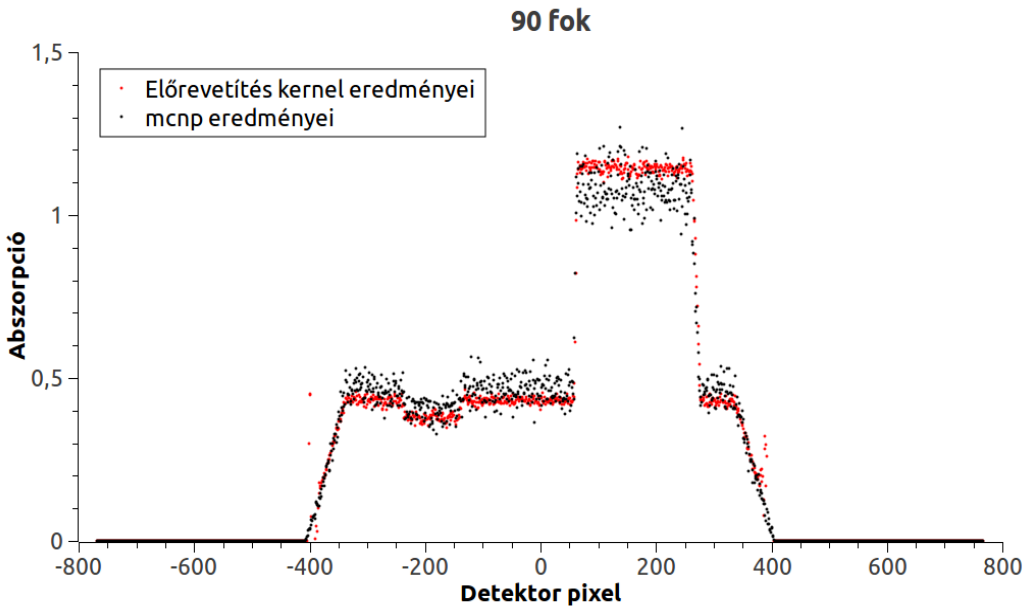
	0 fok	30 fok	90 fok
Maximális plató átlag értéke az mcnp-ből	1.0835	1.1926	1.1380
Maximális plató szórása (mcnp)	0.0125	0.0148	0.0838
Maximális plató értéke az előrevetítés kernelből	1.4367	1.6652	1.4390
Maximális plató értéke szórása (kernel)	0.0865886	0.1008372	0.0485182
normálási faktor (mcnp/kernel)	0.7541	0.7162	0.7908
normálási faktorok hibája	0.0463	0.0443	0.0640

7. táblázat Csúcsértékek összehasonlítása különböző pozícióknál.

A különböző projekciók normálási faktorainak átlag értékével (0,7537 ± 0,0301883) szoroztam a kernelből kapott abszorpciós értékeket, és újra összevetettem az MCNP -ből kapott eredményekkel, amely a 10. ábrán látható.


0 fok
Előrevetítés kernel eredményei
mcnp eredményei
Abszorpció
Detektor pixel
1,5
1
0,5
0
-800
-600
-400
-200
0
200
400
600
800


30 fok
Előrevetítés kernel eredményei
mcnp eredményei
Abszorpció
Detektor pixel
2
1,5
1
0,5
0
-800
-600
-400
-200
0
200
400
600
800



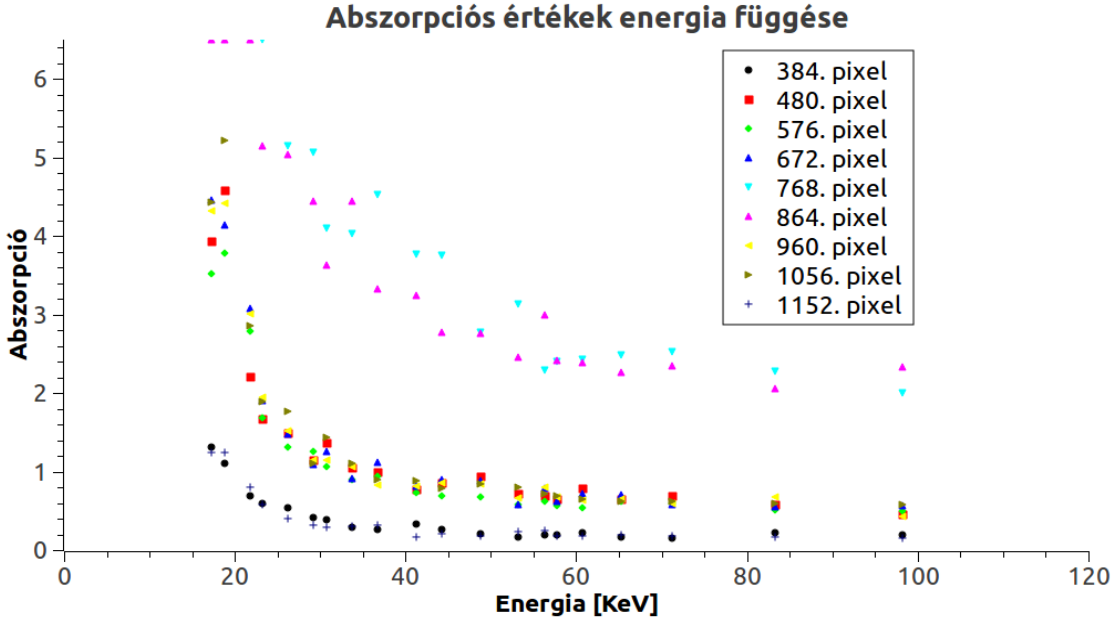
10. ábra Az előrevetítés kernel és MCNP program összevetése normálás után.

Észrevehető, hogy az eredmények eltérése egy normálással eliminálható. Az általam produkált abszorpciós értékek az MCNP eredményektől 25±3%-ban térnek el, melyet jelen kutatási fázisban a kitűzött célhoz mérten megfelelőnek tartottunk. Valós mérési adatok értelmezése esetén ennek a különbségnek a pontos okát fel kell tárni.

3.4.3 Az energiaspektrum figyelembevételének hatásai

A diplomamunkám feladatai közé tartozik a spektrum rekonstrukciós algoritmusomra gyakorolt hatásának ellenőrzése. A spektrum hatásának egyik jele, hogy a kis energiájú beütések abszorpciós értékei jóval magasabbak, mint a nagy energiájú fotonoknak, ami a hatáskeresztmetszet értékek energiafüggéséből következik. Tehát ellenőriznem kellett, hogy az előrevetítés kernel eredményeiben megjelenik-e ez a jelenség, és ha igen, jól látható-e a kapcsolat a hatáskeresztmetszettel. Erre a feladatra átalakítottam az előrevetítés kernelemet úgy, hogy a fotonokat diszkrét energiaértékek mentén sorsoltam, és a már korábban leírt fantomon való áthaladásuk után regisztráltam adott energián történt beütések számát. Adott, 0° -os detektor-forrás pozicionál kilenc különböző detektorpixel energiaspektrumát regisztráltam az elnyelő anyag nélkül, illetve a fantom szimulációja mellett, hogy kiszámolhassam az adott energiához tartozó abszorpciós értékeket. Az energiaspektrumom 20 diszkrét energiapontból áll, amelyet a volfrámon kívüli alumínium szűrő nélküli spektrumból sorsoltam. A pixeleken kapott abszorpciós értékek az energia

függvényében a 11. ábrán láthatók. A nagy abszorpciós értékek nem tekinthetőek pontosnak, mivel a beérkezett fotonok száma ebben az esetben közelít a nullához, így pontos becslést nem ad az előrevetítés kernel nagyon kis energiájú fotonok esetében. A vizsgált intervallum a 20 KeV fölötti energia tartományra korlátozódott.



11. ábra Abszorpciós értékek az energia függvényében.

Az adott detektorpixelhez tartozó abszorpciós értékek víz hatáskeresztmetszetétől való függését kell bizonyítanom. A kapcsolat meghatározásához elméleti megfontolásokat vettem alapul. Tudjuk, hogy a kapott beütés értékek, a Beer-Lambert törvény alapján, a következőképp számolhatók

$$y(E)=y_0(E)\cdot\exp\left(-\int \mu(E,x)\,dx\right) \quad . (28.)$$

Homogén anyagokból felépített fantommal dolgoztam, így az előbbi képlet integrálja felírható szummák segítségével

$$y(E)=y_0(E)\exp-\left(\sum \mu_i(E,x)\,x_i\right) \quad . (29.)$$

Abszorbancia értékekre átírva a képlet az alábbi formára módosul

$$Abszorpció(E)=-\ln\left(\frac{y(E)}{y_0(E)}\right)=\sum \mu_i(E)\,x_i \quad , (30.)$$

ahol a baloldal az energiafüggő abszorpciós érték. A makroszkópikus hatáskeresztmetszete kiszámítható a következő módon

$$\mu(E)=\sigma(E)\cdot\rho \quad , (31.)$$

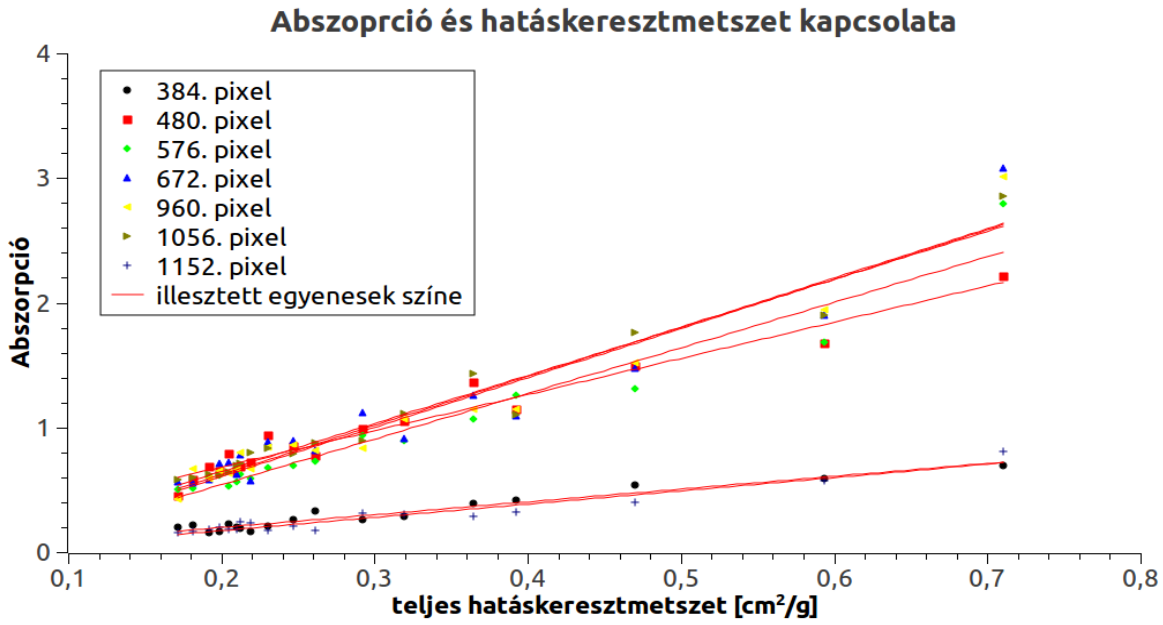
ahol $\sigma(E)$ a teljes hatáskeresztmetszet és ρ az anyag sűrűsége.

Adott detektorpixel esetén a fotonok által az anyagban befutott úthosszak egyenlőnek tekinthetők, így az energiafüggő abszorbancia és a teljes hatáskeresztmetszet között felírható kapcsolat három különböző sűrűségű, de azonos minőségű anyag esetén

$$Abszorpció(E)=\rho_a\cdot\sigma(E)\cdot x_a+\rho_b\cdot\sigma(E)\cdot x_b+\rho_c\cdot\sigma(E)\cdot x_c=\sigma(E)\cdot(\rho_a\cdot x_a+\rho_b\cdot x_b+\rho_c\cdot x_c) \quad , (32.)$$

ahol x valamely anyagban befutott úthossz, $\sigma(E)$ a víz hatáskeresztmetszete és ρ ezen anyag sűrűsége. Azaz a vizsgált fantomom esetében, az adott detektorpixelhez tartozó abszorpciós értékek energiafüggése csak a hatáskeresztmetszetek energiafüggését tartalmazza, és köztük lineáris a kapcsolat, amit be kell bizonyítani a kapott eredményekkel.

E linearitás bizonyítására, adott detektorpixelek mellett ábrázolom az abszorbancia értékeket az adott energiákhoz tartozó hatáskeresztmetszet-értékek függvényében, ahol a pontok kapcsolatát a közös energiaértékek adják. A kapott pontok egy egyenesre illeszkednek, melynek meredeksége információt tartalmaz az áthaladt anyag átlagsűrűségéről, és hogy milyen hosszú utat futott be az anyagban. A kapott pontok a 12. ábrán láthatóak az ezekre a pontokra illesztett egyenesekkel együtt ($y=a+b\cdot x$), amelyek paraméterei a 8. táblázatban találhatóak. Az illesztett egyenesek jósági tényezőjével ellenőrizhetjük a lineáris kapcsolatot, ami verifikálja az előrevetítés során a helyes spektrum használatát.



12. ábra Abszorpció értékek a hatáskeresztmetszet függvényében ábrázolva és az ezekre illesztett egyenesek.

Az illesztett egyenesek jósága mind 0,9 feletti értékek, így lineárisnak tekinthetjük a hatáskeresztmetszet és az abszorpciós értékek kapcsolatát az energia függvényében, tehát az előrevetítés kernelem az elméletnek megfelelően kezeli a különböző energiájú fotonokat.

Y = a+b*x	384. pixel	480. pixel	576. pixel	672. pixel	960. pixel
a	-0.0078	0.1015	-0.1900	-0.1671	-0.1864
a hiba	0.0202	0.0530	0.0771	0.0983	0.0817
b	1.0259	2.9031	3.6552	3.9417	3.9572
b hiba	0.0599	0.1570	0.2283	0.2910	0.2420
R^2	0.9482	0.9553	0.9412	0.9198	0.9435

8. táblázat Illesztett egyenesek paraméterei.

3.5. Visszavetítés

A visszavetítés során a fotonok által az adott voxelben befutott úthosszak átlagát számoltam, melyhez kétféle megoldást is implementáltam. Egyik megoldás szerint az *i* -ik voxel *j* -ik projekcióhoz tartozó átlag befutott úthossz kiszámításához véletlenszerűen sorsoltam a rajta áthaladó fotonutakat, és átlagoltam a voxelekben befutott úthosszakat. A fotonutak mintavételezését úgy oldottam meg, hogy a voxelen belül egyenletes eloszlás szerint sorsoltam egy pontot, melyen áthalad az aktuálisan mintavételezett fotonút. Ez egy gyors, egyszerű megoldása az átlag úthosszak becslésére.

A másik megoldás az lehet, hogy az úthosszak helyett kiszámolhatjuk annak a súlyát, hogy az adott deketorpixel abszorpciójához egy voxel mekkora valószínűséggel járul hozzá, miközben a voxelben megtett úthosszakat egyenlőnek tekintjük.. Ezt a valószínűséget megkaphatjuk területek hányadosaként

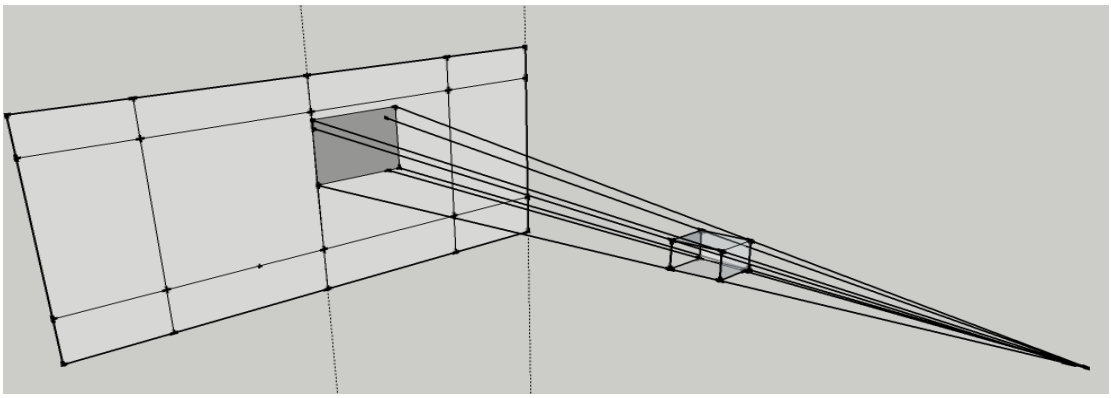
$$p = \frac{i\,.\,voxelárnyéka\ j\,.\ pixelen}{j\,.\ pixelen\ regisztrált\ összeárnyék\ adott\ detektor - forrás\ pozició\ mellett} \quad . \text{(33.)}$$

A 33. egyenletben leírt valószínűség használata okozhat hibákat, mivel a voxelben megtett úthosszakat egyenlőnek tekintjük, és az irodalomban sem találtam ilyesfajta megoldást, így a továbbiakban részletesen kifejtem a módszer működését és az ezzel a visszavetítéssel előállított eredményeket.

A voxelárnyékok területeinek kiszámításához a következő lépéseket kellet implementálnom (ezt a 13. ábra szemlélteti) :

1. adott voxel csúcsainak a pontforrásból történő vetítése a detektor felületre
2. a vetítésből kapott pontokat körbehatároló legkisebb konvex poligon pontjainak meghatározása
3. a konvex poligon darabolása a detektropixeleket határoló vonalak segítségével
4. az így kapott poligonok területének kiszámítása

Az árnyékok alapján történő visszavetítés használata igen ritka, ezért a fenti pontok kifejtésére fókuszál ez a fejezet. Az eredmények taglalását a következő fejezet tartalmazza, mivel a visszavetítést már a rekonstrukciós algoritmussal együtt vizsgáltam.



13. ábra Voxel csúcsok vetítése a detektor felületére.

3.5.1. Adott voxel csúcsainak a pontforrásból történő vetítése a detektor felületre

A függvény az inputként kapott indexből kiszámolja a hozzá tartozó voxel csúcsainak koordinátáit. A hatékonyabb memória kihasználás érdekében csak a két legszélsőbb értékkel rendelkező csúcsok koordinátáit mentem el és a további számításokhoz ezekből keveri ki a program a csúcsokat. A két csúcs kiszámításának módja C -ben:

```
//i a függvény bemeneteként kapott voxel indexe
//Átállás az 1D tömb indexelésről a 3D tömb indexelésére
PositionCord.x = i%pixelnumberX;
PositionCord.y = (int)(i/pixelnumbefelület rX)%pixelnumberY;
PositionCord.z = i/(pixelnumberX*pixelnumberY);
//origótól legtávolabbi pont
VoxelUpPoint.x = ((float)PositionCord.x+1) * kVoxelsize - kXwidth;
VoxelUpPoint.y = ((float)PositionCord.y+1) * kVoxelsize - kYwidth;
VoxelUpPoint.z = ((float)PositionCord.z+1) * kVoxelsize – kZwidth;
//origóhoz legközelebbi pont
VoxelDownPoint.x = (float)PositionCord.x * kVoxelsize - kXwidth;
VoxelDownPoint.y = (float)PositionCord.y * kVoxelsize - kYwidth;
VoxelDownPoint.z =((float)PositionCord.z) * kVoxelsize - kZwidth;
```

A fenti két pontból kikevert csúcsokat összekötve a pontforrással, kiszámíthatók az innen a csúcsokba mutató vektorok:

```
//angle a pontforrás és origó között húzott egyenes és az x tengely által
közbezárt szögek
sindelta = sin(M_PIs2*(angle/360.0f));
cosdelta = cos(M_PIs2*(angle/360.0f));
//A pontforrás koordinátái
StartPoint.x = kCtradius * cosdelta;
StartPoint.y = kCtradius * sindelta;
StartPoint.z = 0.0f;
//A pontforrás koordinátái
v.x = ((Voxel csúcs).x - StartPoint.x);
v.y = ((Voxel csúcs).y - StartPoint.y);
v.z = ((Voxel csúcs).z - StartPoint.z);
```

A fenti kódrészletben a „(Voxel csúcs)” koordinátái a következők: (VoxelUpPoint.x, VoxelUpPoint.y, VoxelUpPoint.z) , (VoxelDownPoint.x, VoxelUpPoint.y, VoxelUpPoint.z) , (VoxelUpPoint.x, VoxelDownPoint.y, VoxelUpPoint.z) ,(VoxelUpPoint.x, VoxelUpPoint.y, VoxelDownPoint.z) , (VoxelDownPoint.x, VoxelDownPoint.y, VoxelDownPoint.z) , (VoxelUpPoint.x, VoxelDownPoint.y, VoxelDownPoint.z) , (VoxelDownPoint.x, VoxelUpPoint.y, VoxelDownPoint.z) , (VoxelDownPoint.x, VoxelDownPoint.y, VoxelUpPoint.z).

A kapott vektorok a továbblépéshez szükséges egyenesek irány vektorai, melyek a forráspont és a voxelcsúcsokat kötik össze. Az egyenesek és a detektorsík döféspontjait kell kiszámolni, amihez szükséges meghatározni a detektorsík egyenletét. Az egyenlethez szükséges a sík normálvektora (cosdelta, sindelta, 0) és a sík egy pontja. A detektor *A* csúcsát használja a program az egyenlet megalkotásához, mivel később szükségem lesz rá. Az *A* pontot kiszámoló kódrészlet:

```
      // A téglalap alakú detektor középpontjába mutató vektor
      n1 = kDetector_r * cosdelta;
      n2 = kDetector_r * sindelta;
      n3 = 0.0f;
      // kDetector_width_y a detektor szélességének a fele
      // A detektorunk egy téglalap és az egyik csúcsa kiszámolható a detektor
középpontjába mutató vektor és az arra merőleges detektor széleséggű pont
összegeként
      Ax = n1 - kDetector_width_y * sindelta;
      Ay = n2 + kDetector_width_y * cosdelta;
      Az = kDetector_width_z;
```

Most már számolható az egyenes és sík döféspontja. A sík egyenlete

$$n1 \cdot x + n2 \cdot y + n3 \cdot z = n1 \cdot Ax + n2 \cdot Ay + n3 \cdot Az \quad , (34.)$$

az egyenes egyenletrendszere

$$x = x_0 + a \cdot t \, , y = y_0 + b \cdot t \, , z = z_0 + c \cdot t \quad . (35.)$$

A döféspont kiszámításához behelyettesíthető az egyenes egyenleteit a sík egyenletébe és az algoritmus kiszámoljuk a *t* paramétert. A *t*-re kapott egyenlet a fentiek alapján

$$t = \frac{Ax \cdot cosdelta + Ay \cdot sindelta - cosdelta \cdot StartPoint\,.\,x - sindelta \cdot StartPoint\,.\,y}{cosdelta \cdot a + sindelta \cdot b} \quad . (36.)$$

A programom a detektor felületére vetített pont koordinátáit a következőképp számolja:

```
      tparam = (Ax* cosdelta + Ay* sindelta - cosdelta*StartPoint.x -
sindelta*StartPoint.y ) / (cosdelta * v.x  + sindelta * v.y);
      (Új pont).x = StartPoint.x + tparam*v.x;
      (Új pont).y = StartPoint.y + tparam*v.y;
      (Új pont).z = StartPoint.z + tparam*v.z;
```

A kapott pont mindhárom koordinátáját ismerjük, de nekünk szükséges áttérni a detektorsík felületére, ami viszont kétdimenziós. A feladat lényegében átállni a 3D-os koordináta-rendszerről a

detektor felületével párhuzamos 2D-os koordináta-rendszerre, amelynek origója a detektor középpontja. Ehhez el kell tolni a 3D-os koordináta-rendszert a detektor középpontjába mutató vektorral, és kiszámolni a pontok új koordinátáit az új rendszerben. Az új rendszer tengelyei y' és x'. A z tengely egybeesik az y' tengellyel, így a pontok y' koordinátái a régi koordináta rendszer z koordinátái. Az új rendszerben a pontok x' koordinátái a következőképp alakulnak

$$x' = \pm\sqrt{(x)^2 + (y)^2} \quad , (37.)$$

ahol x és y a régi koordináta-rendszerben vett első és második koordináta, míg x' az újban vett első koordináta. Az x' koordináta előjelét az határozza meg, hogy a régi koordináta-rendszerbeli pontok z metszetbeli A ponttól számolt távolsága nagyobb vagy kisebb, mint a detektor szélességének fele. Az új rendszerre való áttérésre szolgáló kódrészlet:

```
if( sqrtf((Új pont).x - Ax)*((Új pont).x - Ax) -((Új pont).y - Ay)*((Új
pont).y - Ay)) > kDetector_width_y )  sign = -1.0f;
else sign = 1.0f;
(Új pont).x = (Új pont).x – n1; // eltolás
(Új pont).y = (Új pont).y – n2; // eltolás
// Új koordináta érték kiszámítása:
(Új pont).x = sign*sqrt( (Új pont).x*(Új pont).x + (Új pont).y*(Új pont).y);
```

A program az új 2D-os koordináta-rendszerbeli pontokkal átléphet a következő feladatra.

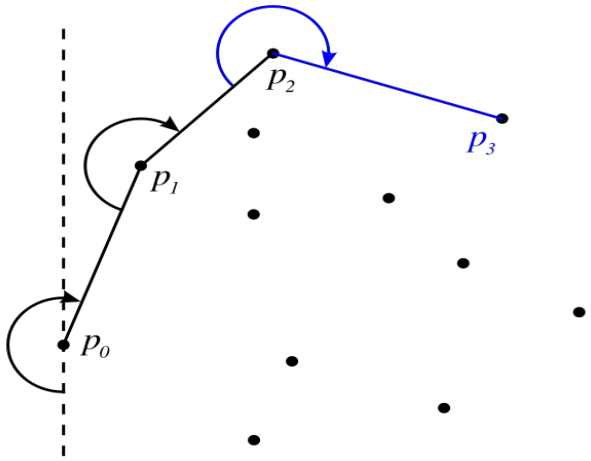
3.5.2. A vetítésből kapott pontokat körbehatároló legkisebb konvex poligon pontjainak meghatározása

Az árnyékot határoló poligon a detektor felületre vetített voxelcsúcsokat körbehatároló legkisebb konvex poligon. Ennek megkeresésére a „Gift wrapping” algoritmust használja a program (az algoritmus szemléltetése látható a 14. ábrán). A szakirodalomban sok más megoldás is megtalálható, melyek lehet, hogy gyorsabbak, de kevés pontról lévén szó (8 pont) a legbiztonságosabb módot választottam. A választott algoritmus működése pontokba szedve a következő:

1. a poligon első pontjának kiválasztja a legkisebb x koordináta értékekkel rendelkező pontot, és beállítja aktuális végpontnak
2. A poligon következő pontja az lesz, amely teljesíti azt a feltételt, hogy az aktuális végpontból a pontba mutató vektortól balra nem található más pont.
3. A kiválasztott következő pontot hozzáadja a poligon pontjaihoz és beállítja aktuális végpontnak.

4. Az előző 2. és 3. pontot addig folytatja, amíg az aktuális végpont nem egyezik meg a kezdőponttal.

5. A végfeltétel teljesülésekor az algoritmus kimenetként biztosítja a konvex poligon pontjainak egymás utáni halmazát és a kapott poligon pontjainak számát.



14. ábra Gift wrapping algoritmust szemléltető ábra[16]

Az algoritmust és a hozzá szükséges függvényt, amely vizsgálja, hogy egy pont a vektortól balra helyezkedik-e el, a függelékben található. A fenti algoritmust többször használom egy adott voxel területeinek számításánál, így a későbbiekben csak „LegkisebbKonvexPoli()” néven fogok rá hivatkozni.

3.5.3. A konvex poligon darabolása a detektorpixeleket határoló vonalak segítségével

Előfordulhat, hogy a kapott árnyék több detektorpixelen helyezkedik el. A pontos eredményhez szükséges kiszámítani, hogy adott pixelen az árnyék mekkora területet takar ki, azaz fel kell darabolnunk a poligonunkat. Az optimális futáshoz feltettünk egy újabb feltételt: egy voxel árnyéka maximum négy pixelen helyezkedik el. Ezt a feltételt teljesíti az előrevetítés verifikálásához használt geometria. Tehát a területek kiszámításához a program az alábbi feladatokat végzi el:

1. Vizsgálom, hogy az árnyék poligonját metszi-e valamely függőleges és/vagy vízszintes detektorpixel-határ:

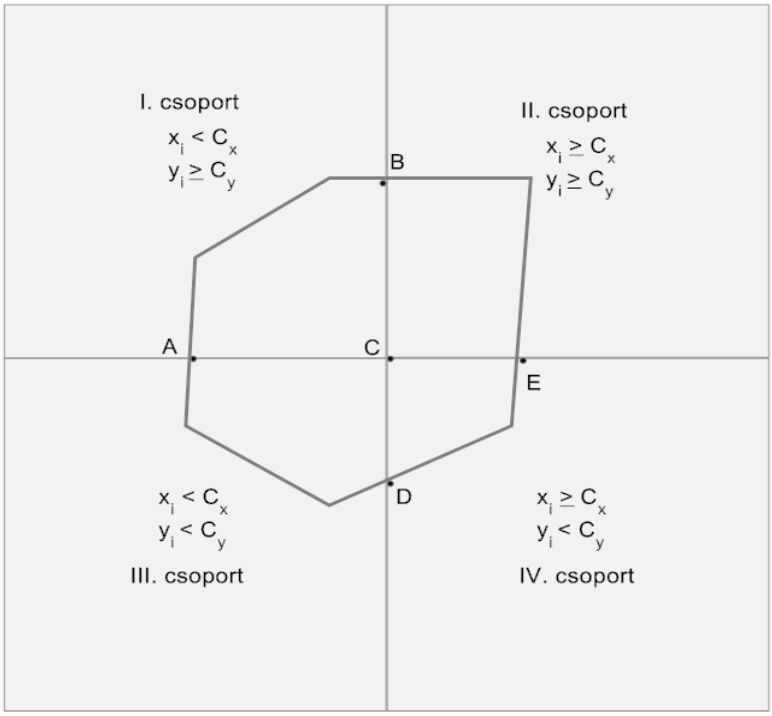
Ha a poligont egy detektorpixel-határ sem metszi, akkor tovább lépünk a poligon terület

kiszámítására

a) Ha a poligont csak egy függőleges határ metszi, akkor kiszámítjuk a B és D pontokat (15. ábra) és a poligon pontjait két csoportra választjuk az alapján, hogy a pontok első koordinátája kisebb, vagy nagyobb mint a B pont első koordinátája. Mindkét ponthalmazhoz hozzáadjuk a B és D pontokat, majd a halmazokon lefuttatjuk a LegkisebbKonvexPoli() függvényt.

b) Ha a poligont csak egy vízszintes határ metszi, akkor kiszámítjuk az A és E pontokat (15. ábra) és a poligon pontjait két csoportra bontjuk az alapján, hogy a pontok második koordinátája kisebb, vagy nagyobb mint az A pont második koordinátája. Mindkét ponthalmazhoz hozzáadjuk az A és E pontokat, majd a halmazokon lefuttatjuk a LegkisebbKonvexPoli() függvényt.

c) Ha a poligont függőleges és vízszintes határ is metszi, akkor kiszámítjuk az A - E pontokat és a poligon pontjait négy csoportra választjuk a 15. ábrán látható módon. Mindegyik csoporthoz hozzáadjuk a csoport határán lévő metszetpontokat és lefuttatjuk a halmazokon a LegkisebbKonvexPoli() függvényt.



15. ábra Poligon felosztassa a detektorpixel határokkal és a poligon metszéspontjai

3.5.4. A kapott poligonok területének kiszámítása

A poligonok, amelyek területét ki kell számolnunk, pontjai az óramutató járásával megegyező

irányban vannak rendezve a pontokat tartalmazó tömbben. Ez a tulajdonság a LegkisebbKonvexPoli() függvénynek köszönhető, amely eredményeképpen a poligonok területe számolható az alábbi képlet segítségével:

$$area = \frac{1}{2} \sum_{i=0}^{N-1} \left(x_i y_{i+1} - x_{i+1} y_i \right) \quad , (38.)$$

ahol N a pontok száma, x_i az i -ik pont első koordinátája, y_i az i -ik pont második koordinátája.

A p valószínűség kiszámításához szükséges tudnunk az adott pixelen regisztrált összes árnyékot adott detektor-forrás pozíciók mellett. Ehhez a visszavetítés algoritmust le kell futtatni a rekonstrukciós kernel előtt, és így a kernel futása közben újra futó visszavetítés algoritmus már valószínűségeket ad vissza.

A visszavetítés eredményeit külön nem taglalom, mivel csak a rekonstrukciós algoritmussal együtt értelmezhető, és ezt a következő részben fejtem ki.

4. Rekonstruált képek vizsgálata

A rekonstrukció helyességének bizonyítására ellenőrizni kell a kapott kép minőségét és értékhelyességét, majd meg kell állapítanom, hogy a rekonstrukció képes csökkenteni, vagy akár teljesen eliminálni a spektrumfelkeményedés okozta artefaktumokat.

Az értékhelyesség ellenőrzésére a már korábban említett fantomot használva legeneráltam a mérési eredményeket az előrevetítő segítségével, majd ezeket rekonstruáltam. Kétféle forrás típust használtam: monokromatikus forrást, vagy spektrummal rendelkező forrást, de egy adott teszt esetnél az előrevetítő és rekonstrukció azonos forrástípust használt. Mindkét visszavetítést lefuttattam, azaz négy különböző teszt esetet generáltam le, és vizsgáltam a kapott kép homogén térrészeinek átlag és szórás értékeit. Az átlag értékekből és a szórásból a kapott képek értékhelyességét tudjuk ellenőrizni.

Az algoritmus minőségét jellemzi az L_2 norma, melyet iterációs számok függvényében vizsgáltam az átlaghoz és a szóráshoz hasonlóan a homogén térrészekre. Az L_2 norma az alábbi képlettel számítható:

$$L_2 norma = \frac{\sum_{i=0}^{N} \left(\mu_i - \mu_i^{id}\right)^2}{\sum_{i=0}^{N} \left(\mu_i^{id}\right)^2} \quad , (39.)$$

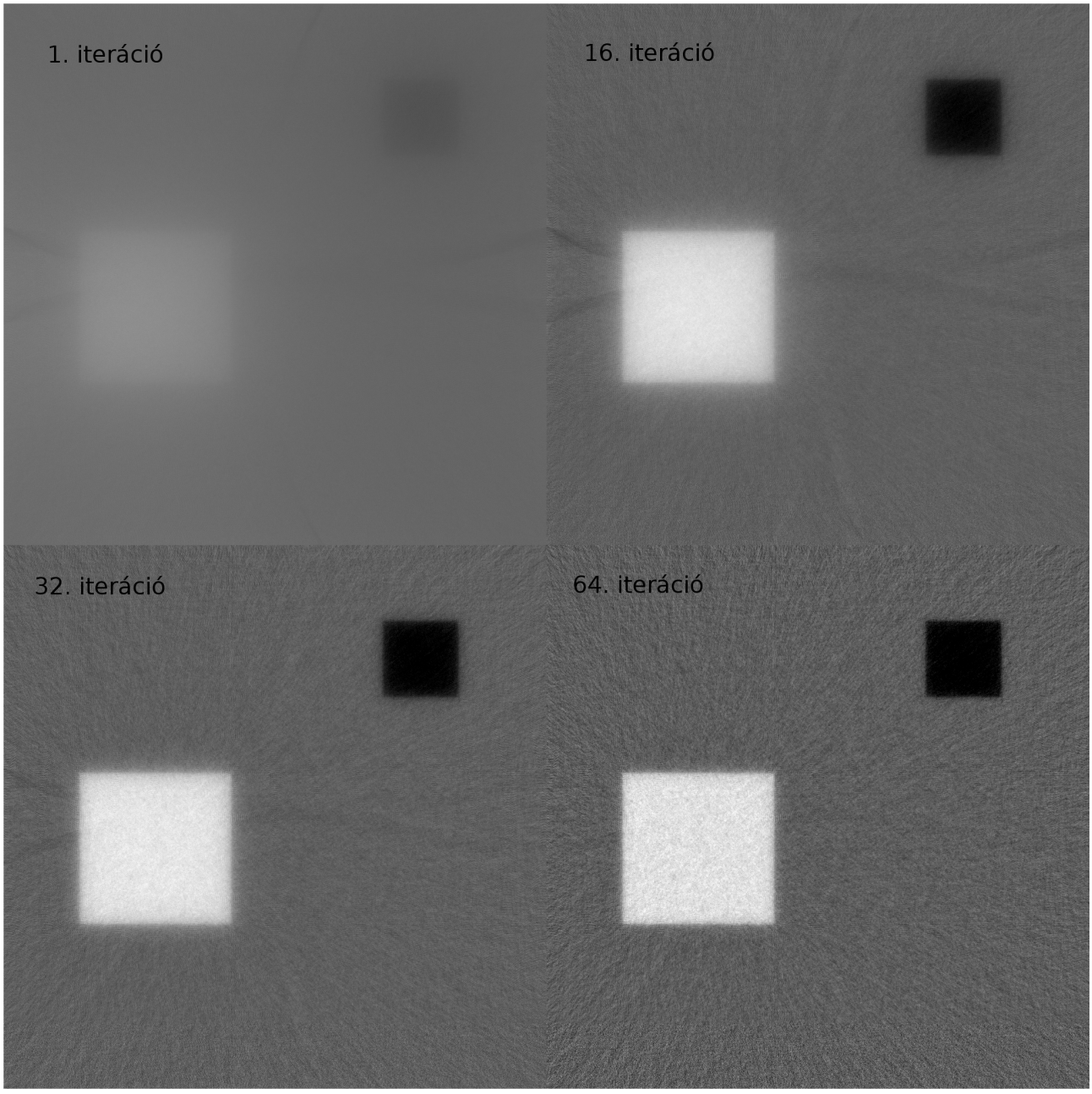
ahol N a voxelek száma, μ_i^{id} az i -ik voxel ideális gyengítési együtthatója és μ_i az i -ik voxel gyengítési együtthatója. Az L_2 norma érték fizikai jelentése, hogy adott képek közül az hasonlít jobban az eredetire, amely kisebb L_2 norma értékkel rendelkezik.

A tesztesetek legenerálása során a mérési eredmények előállításához $8 \cdot 10^8$ fotont indítottam, míg a rekonstrukció során futó előrevetítés $16 \cdot 10^7$ fotont generált iterációnként és összesen 64 ciklus futott le. A fantom makroszkópikus hatáskeresztmetszetei 100 KeV -en: 0,0001711 1/cm (kis sűrűségű víz), 0,4011 1/cm (nagy sűrűségű víz) és 0,1711 1/cm (1 g/cm^3 sűrűségű víz) és a rekonstrukcióhoz 360 projekciót vettünk fel, azaz fokonként egyet. A forrás monokromatikus használata esetén 100 KeV -es fotonokat generált és spektrumként a már korábban említett alumíniumszűrő nélküli volfrámból kijutó spektrumot használtam. A négyféle esetben kapott rekonstruált képeket a 16, 17, 18, és 19. ábra tartalmazza.

A kapott képeken jól felismerhető a három különböző anyag, a határok sem mosódtak el. A

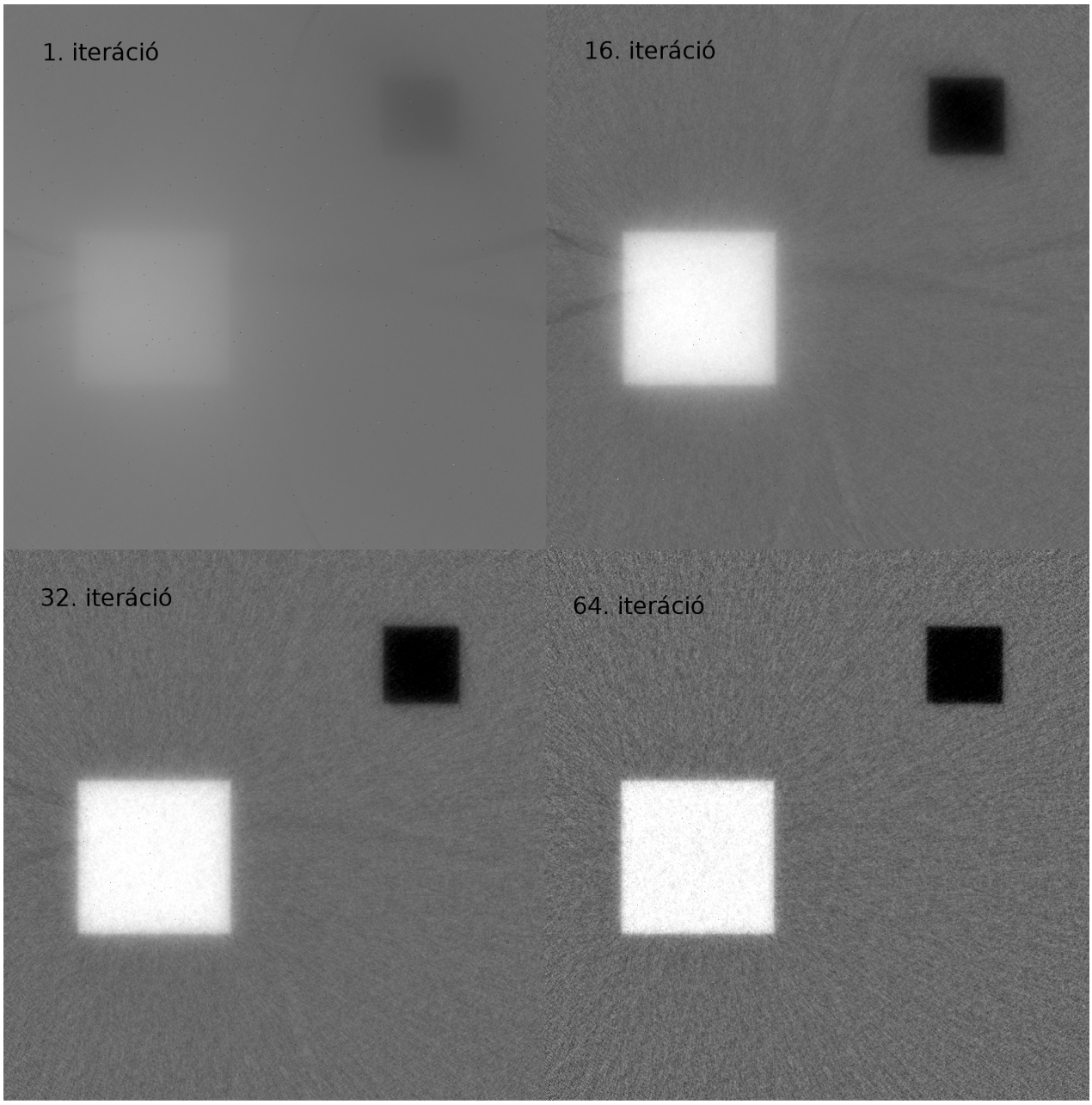
nagyobb sűrűségű négyzet körüli artefaktum nagyobb iterációs számoknál eltűnik. Az iteráció előre haladtával a normál sűrűségű víz becsíkozódik, mely nagyobb mértékben megjelent, amíg nem lokalizáltuk az előrevetítőben egy programhibát. E hiba miatt az előrevetítő nem volt képes megfelelő számú fotont indítani, amit végül Molnár Balázs[17] segítségével javítottunk.

A kétféle vetítő használatából adódó képek között (16. és 17. ábra között) lényegi különbség nem látható, azaz a két fajta visszavetítő első ránézésre hasonló eredményeket adott.



16. ábra Az előrevetítés teszt geometriájáról szimulált mérési adatokból kapott rekonstrukciós képek különböző iterációs lépéseknél a Monte Carlo alapú visszavetítéssel monokromatikus forrás

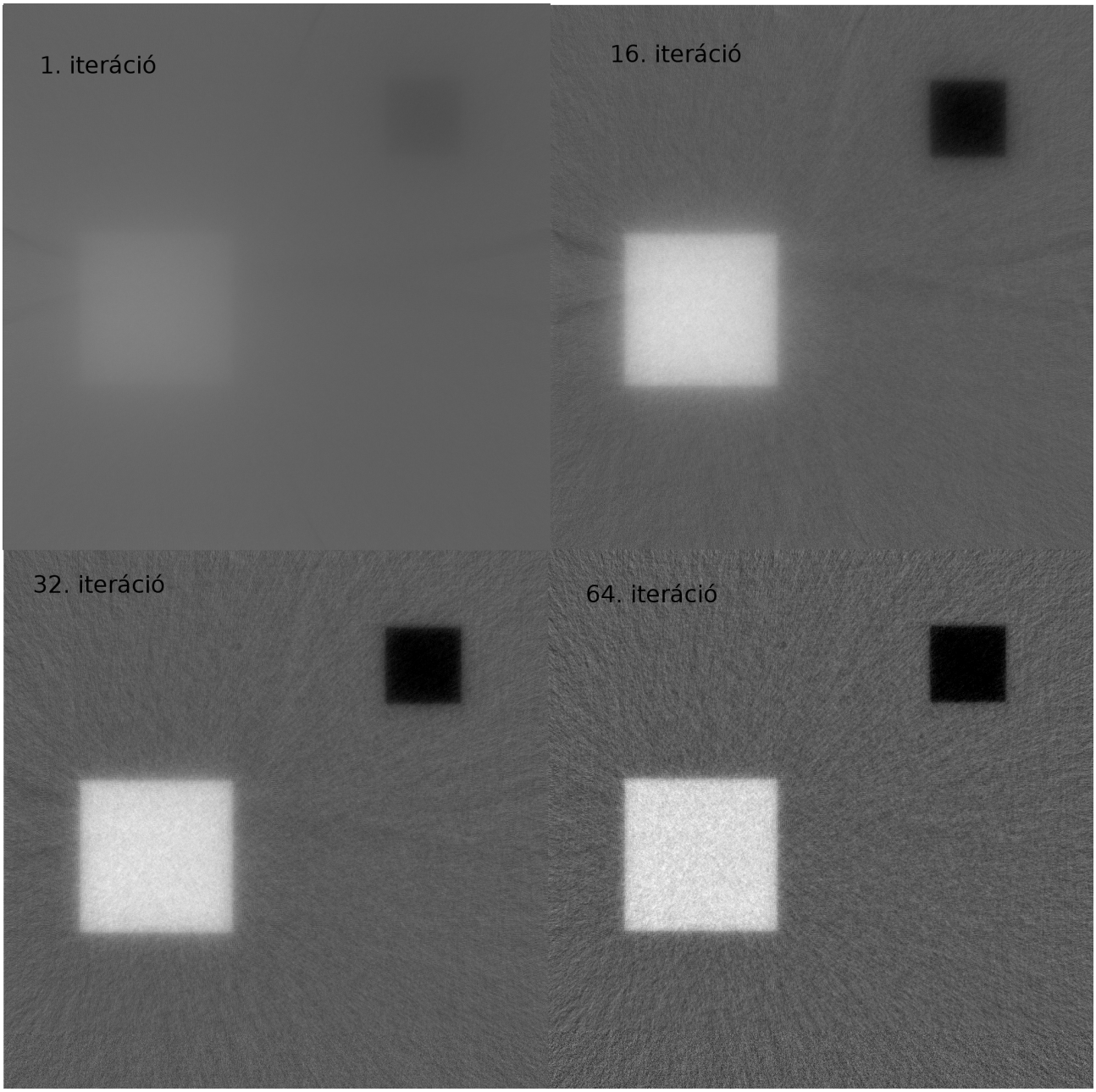
esetén.



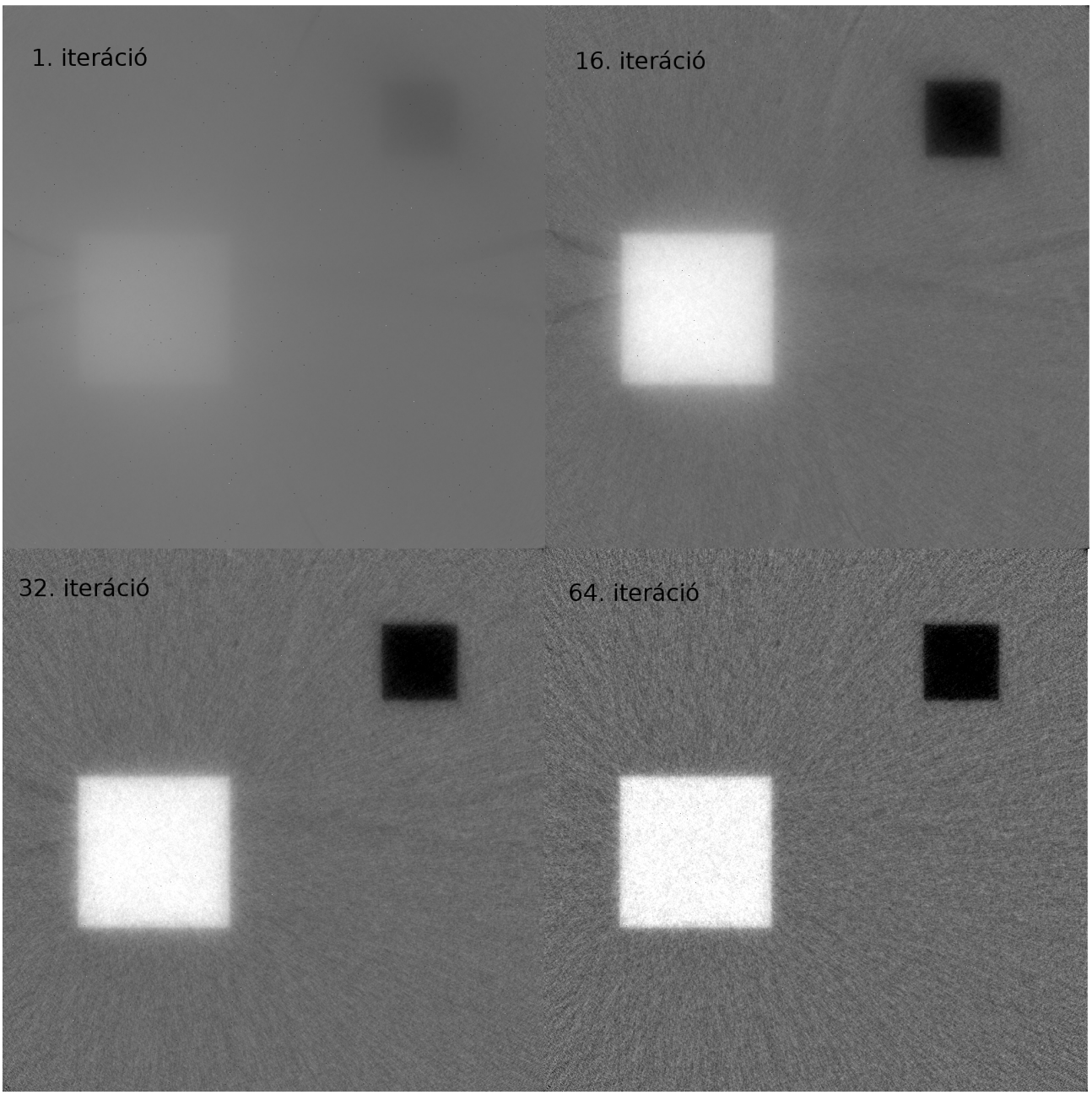
17.ábra Az előrevetítés teszt geometriájáról szimulált mérési adatokból kapott rekonstrukciós képek különböző iterációs lépéseknél az árnyék alapú visszavetítéssel monokromatikus forrás esetén.

A spektrum bekapcsolása egyik esetben sem hoz szemmel látható eltérést, azaz a spektrum használata nem ront a képminőségen, habár fenn áll a lehetőség, hogy alulmintavételezett képet

kapunk, hiszen a kisebb energiás fotonok nagyobb arányban abszorbeálódnak az anyagban. Tehát a tesztesetek foton számának a jó megválasztása eredményezte, hogy nincs látható eltérés a monokromatikus esetekhez képest.(18 19. ábra)



18.ábra Az előrevetítés teszt geometriájáról szimulált mérési adatokból kapott rekonstrukciós képek különböző iterációs lépéseknél spektrummal rendelkező forrás esetén a Monte Carlo alapú visszavetítéssel

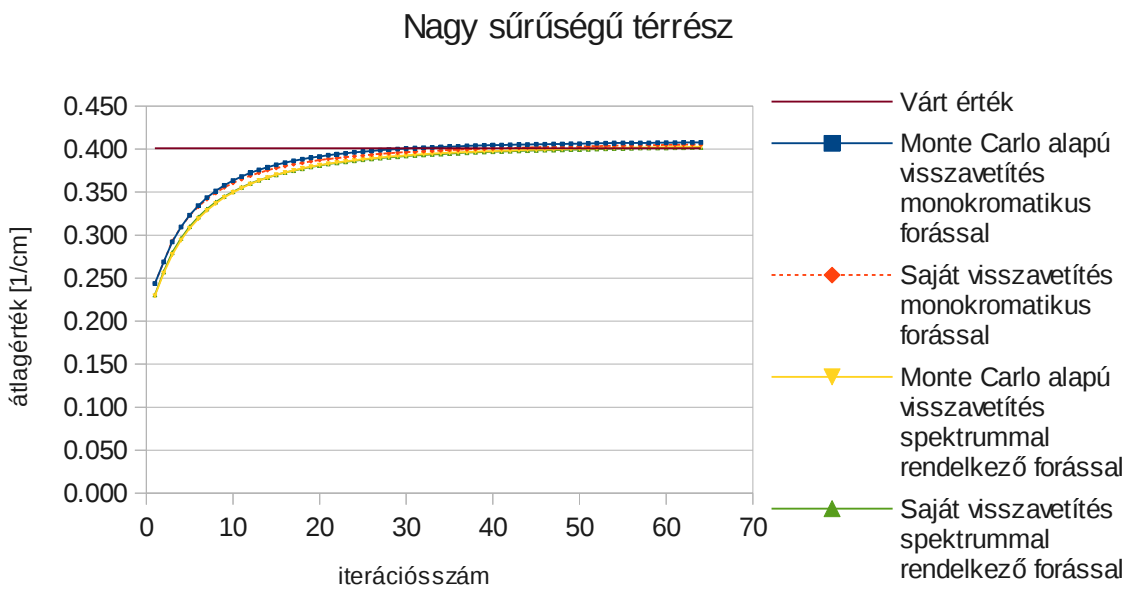


19.ábra Az előrevetítés teszt geometriájáról szimulált mérési adatokból kapott rekonstrukciós képek különböző iterációs lépéseknél spektrummal rendelkező esetén a Monte Carlo alapú visszavetítéssel

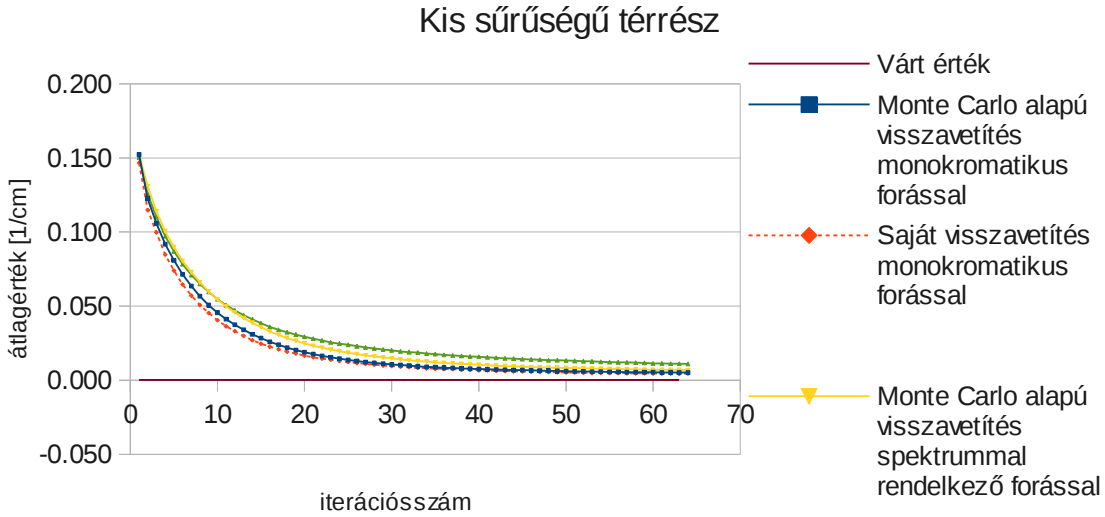
4.1 Értékhelyesség ellenőrzése

A négyféle eset során kapott képet a már korábban leírt mérőszámokkal vizsgáltam (átlag érték és egyszeres szórás) az iterációs szám függvényében. Egy jó algoritmustól azt várhatjuk el, hogy a

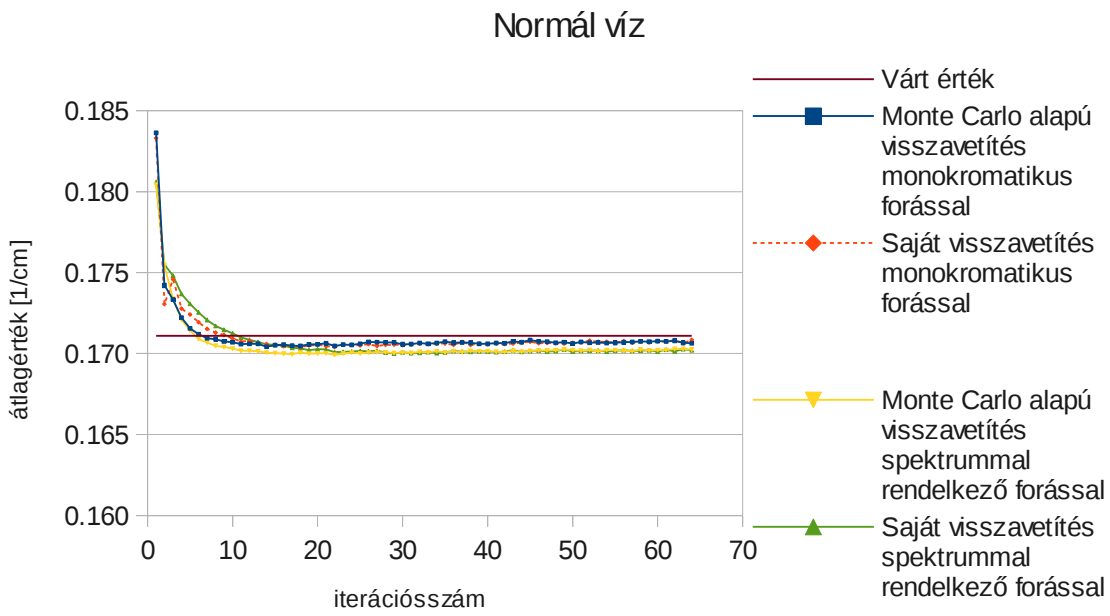
homogén térrészekre számolt átlag értékek egybeesnek a fantom eredeti értékeivel, míg a szórás értékek minél kisebbek. A kapott átlag értékek a különböző térrészekre a 20, 21 és 22. ábrán láthatók.



20. ábra Nagy sűrűségű négyzet várható értéke és a különböző módszerek nagy sűrűségű térrészen kapott átlag értéke az iterációs szám függvényében.



21. ábra Kis sűrűségű négyzet várható értéke és a különböző módszerek kis sűrűségű térrészen kapott átlag értéke az iterációs szám függvényében.



22. ábra Normál víz várható értéke és a különböző módszerek normál víz térrészén kapott átlag értéke az iterációs szám függvényében.

Észrevehető, hogy az átlag értékek mindig a várható értékhez konvergálnak, és a konvergálás sebessége főképp a kiindulási értéktől függ, a használt futási módtól nem. Az átlag értékeket 0,02 pontossággal megközelíti minden esetben és a kis sűrűségű anyagot leszámítva ezred pontosságú a várt érték megközelítése. Ezek alapján kimondhatjuk, hogy mindkét visszavetítés a forrástípusoktól függetlenül értékhelyes.

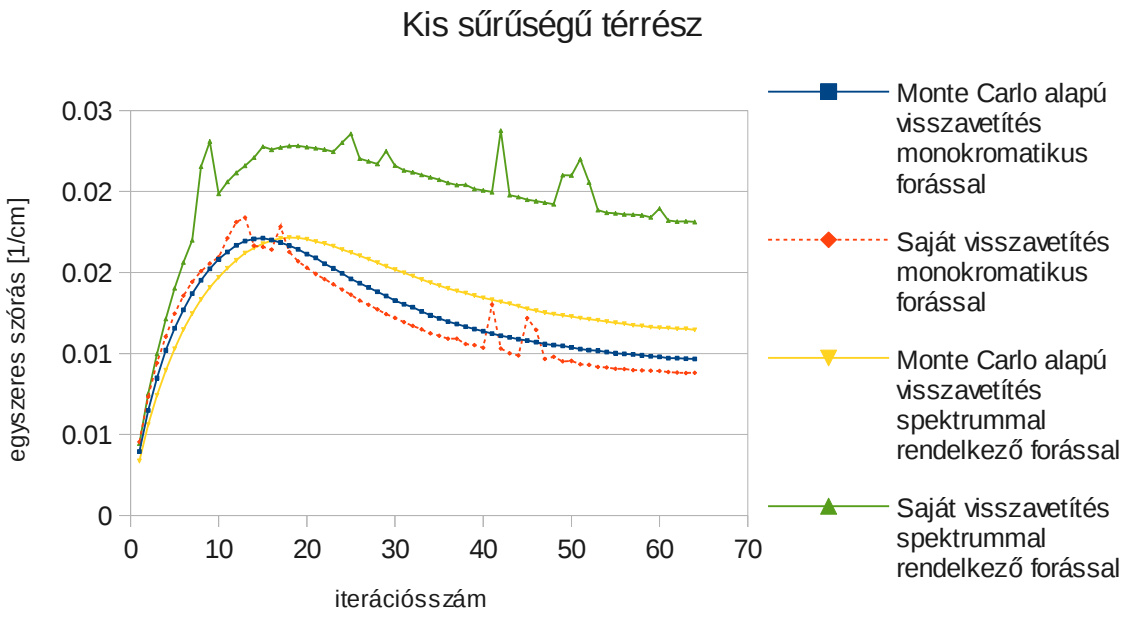
A szórás segítségével meg tudjuk határozni a rekonstrukció hibáját és képet kapunk a homogenitás megtartásának mértékéről (23, 24 és 25. ábra). Minél kisebb a szórás adott terülten, annál homogénebb az adott rész, ami kevésbé szemcsés képre utal. A szemmel összehasonlított képeken nem látunk különbséget, de a szórási értékek eltérők a különböző futási módoknál. A kapott értékekből arra következtethetünk, hogy a sűrűbb anyagok esetében közel azonosan viselkedik a rekonstrukció. Először monoton nő, amíg a 10 iteráció környékén el nem ér egy lokális maximumot, majd monoton csökken a 30. iterációig, végül újra el kezd növekedni. Ennek a görbének a jellege abból származhat, hogy a kezdeti érték távol van a várt értéktől. Továbbá a rekonstrukció során nem homogén módon növeli a voxelrendszer makroszkópikus hatáskeresztmetszet értékeit, majd közelebb a várt értékhez átlagban kisebb változásokat eredményez a rekonstrukció a voxelrendszeren, így az inhomogenitás el kezd csökkenni (10 -> 30

iteráció). Az újbóli növekedés oka, az iteratív rekonstrukció jellegéből adódik. Magas iterációs számoknál a kép elkezd ”becsomósodni”, amely hibát visz a képbe.

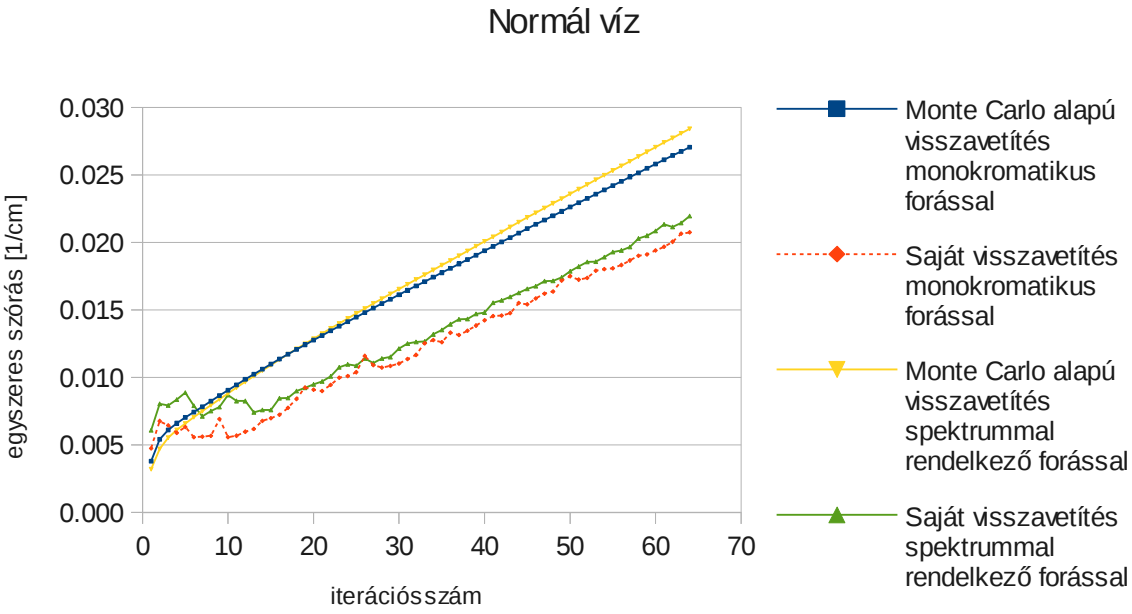
A kis sűrűségű négyzet esetén az árnyék alapú visszavetítéssel generált képek szórása átlagban nagyobb. Ennek az lehet az oka, hogy a bonyolultabb számításból álló árnyék alapú visszavetítő pontatlanabb eredményt ad néhány esetben és több kiszóró pontot eredményez a képen. A spektrum használata ebben az esetben nagyobb szórási értékeket eredményez, míg a Monte Carlo alapú visszavetítőnél kisebbet. Erre pontos magyarázatot nem találtam.

A normál sűrűségű vízre számolt szórás értékek mind a négy esetben monoton növekvő függvényt eredményeztek, amely abból származik, hogy a kezdeti érték közel helyezkedik el a várt értékhez, és az iteratív rekonstrukciókra jellemző módon, a homogenitás romlik a csomósodásnak köszönhetően.

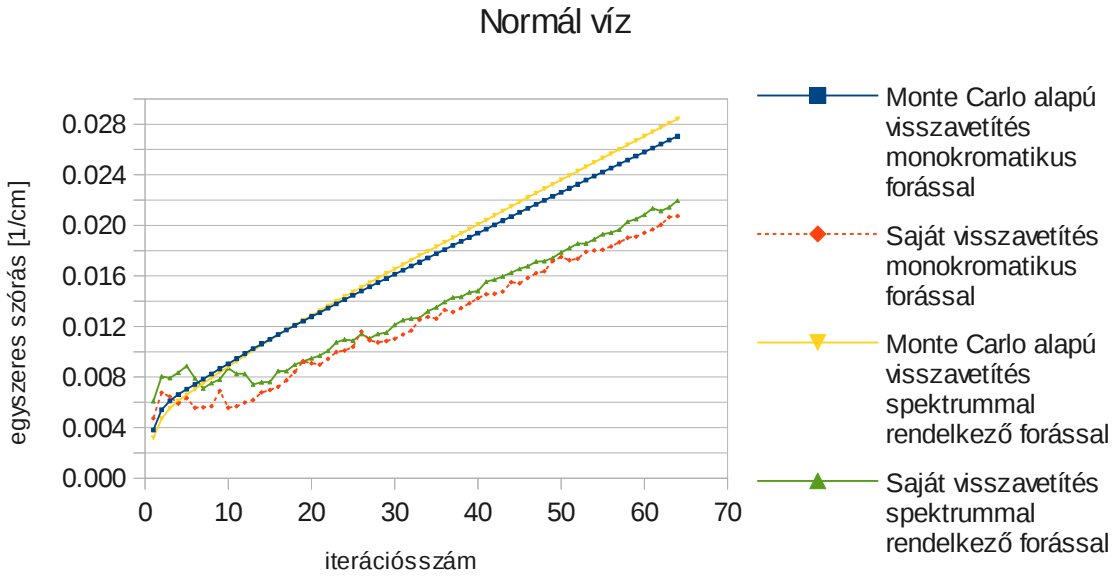
A szórási értékek 0,03 1/cm körül alakulnak, azt tekinthetjük az egyszeres lebegőpontos számításokból adódó hibának. A kutatás jelen fázisán ez a hiba elfogadható és az átlag értékeket is e tartományon belül közelítik meg a várt értékeket. Ez a későbbiekben javítható a szűréssel, vagy a paraméter tér pontos vizsgálatával (több foton, több projekció, ...).



22. ábra Nagy sűrűségű négyzetnél mért egyszeres szórás a különböző módszereknél az iterációs szám függvényében.



23. ábra Kis sűrűségű négyzetnél mért egyszeres szórás a különböző módszereknél az iterációs szám függvényében.

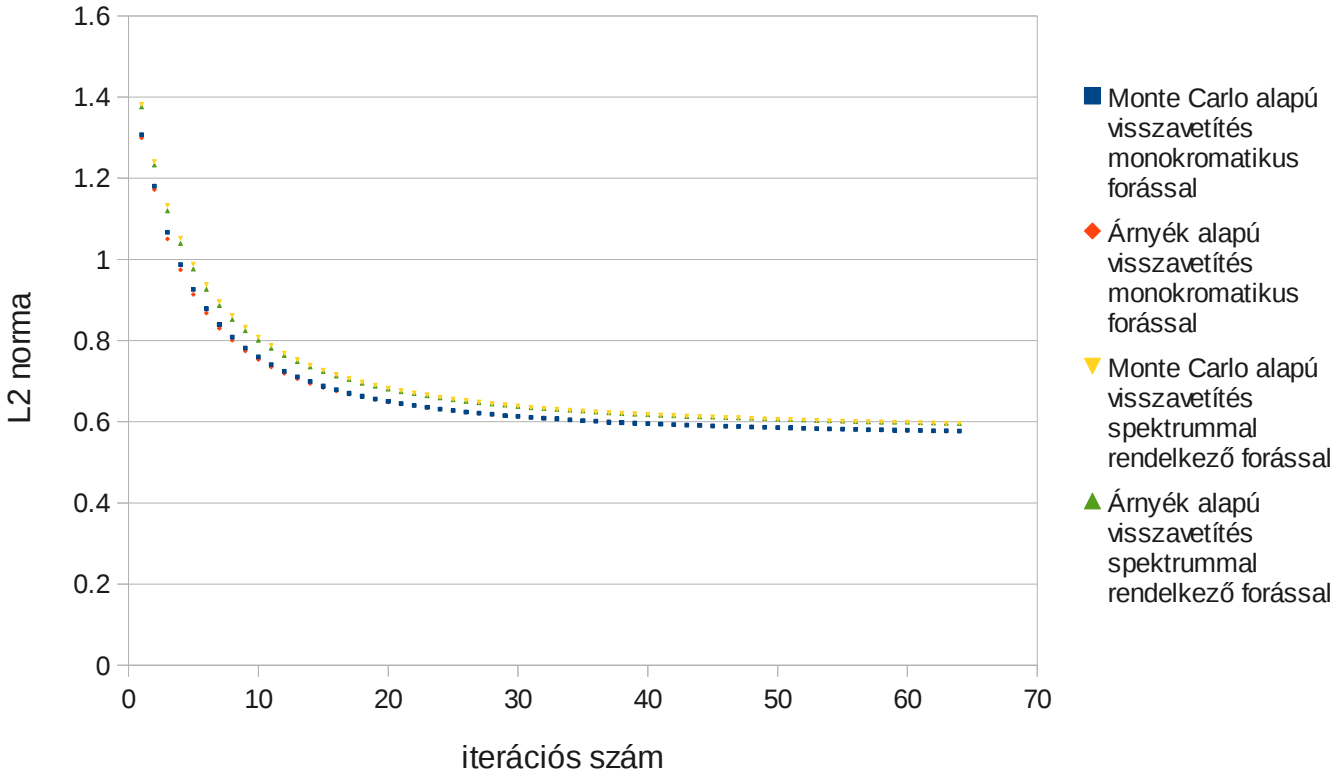


24. ábra Normál víz területén lévő egyszeres szórás a különböző módszereknél az iterációs szám függvényében.

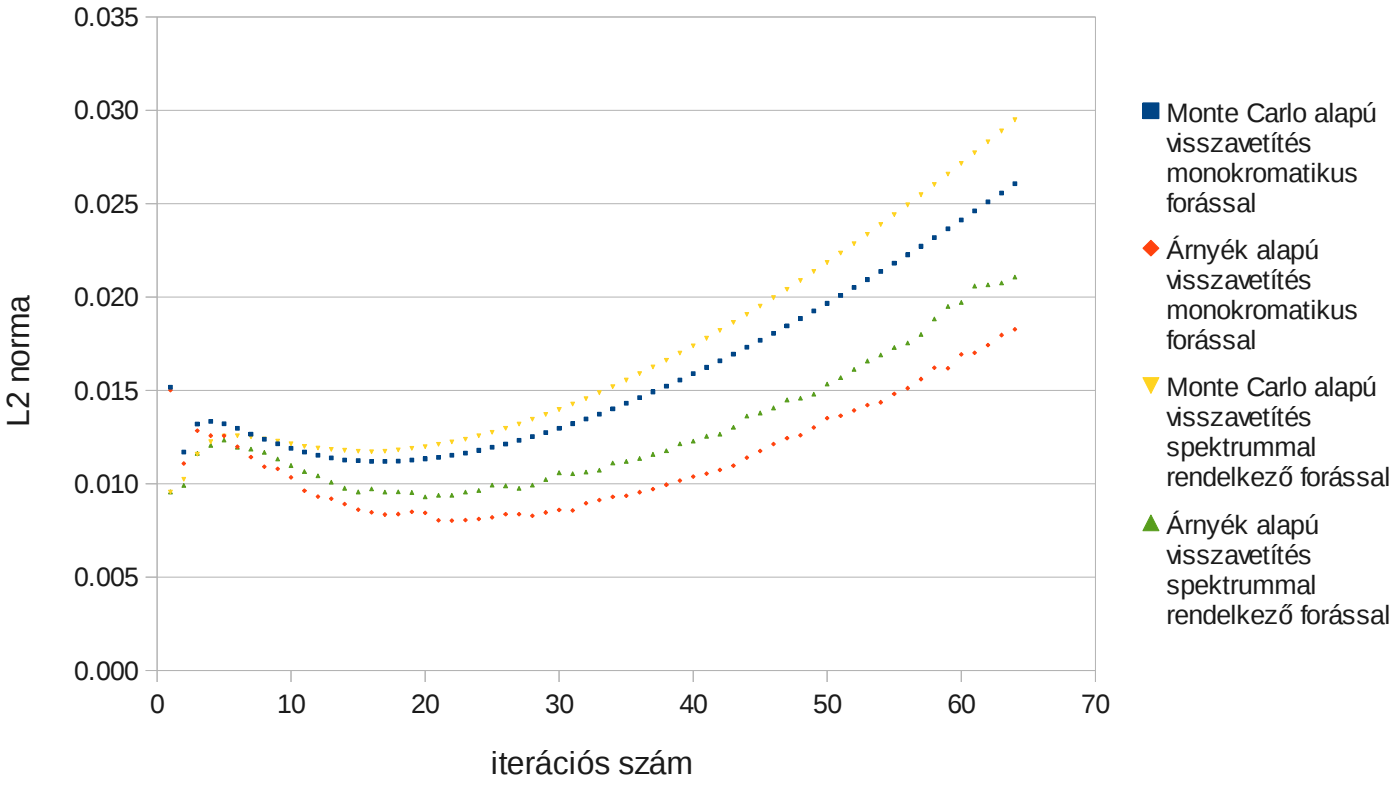
4.2 Konvergencia vizsgálata

A rekonstrukciók konvergencia menetének vizsgálatára használhatjuk az L2 norma értéket, melyek a 25, 26 és 27.ábrán az iterációs szám függvényében láthatók a különböző anyagoknál. Az L_2 norma szigorúan monoton csökken és láthatóan nem ér el minimumot a 64 iterációs szám alatt a kis sűrűségű és normál sűrűségű víz esetében, míg a nagy sűrűségű térrésznél látható lokális minimum.

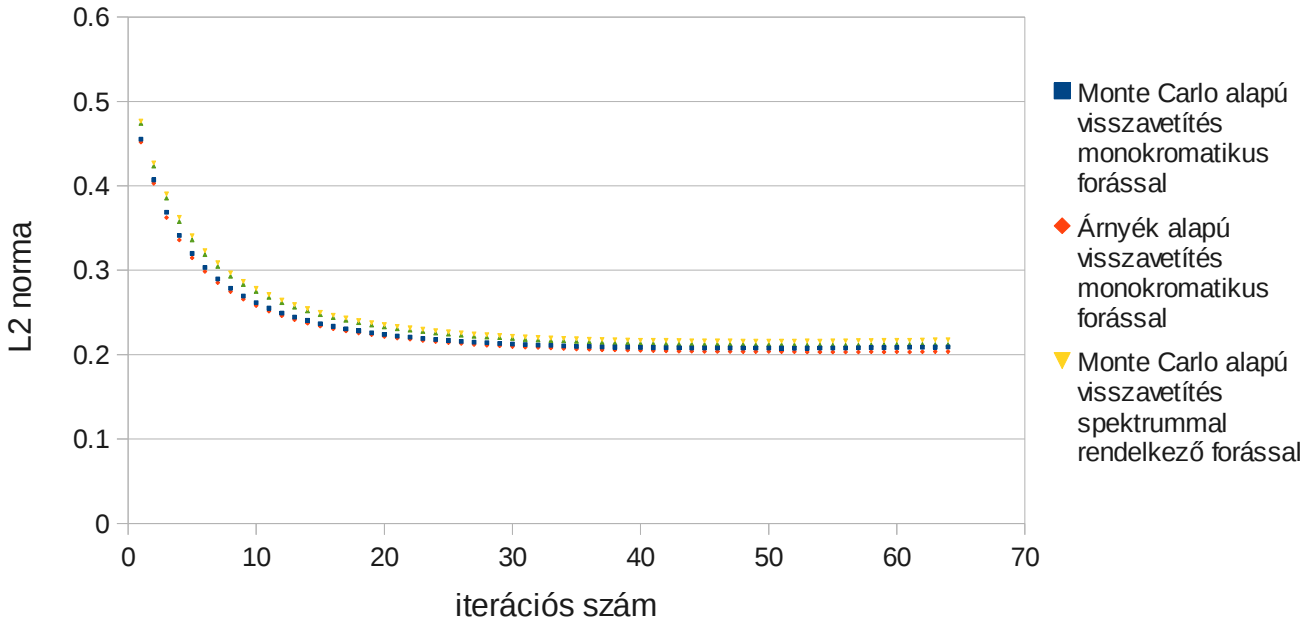
Eltérés még a két esett között, hogy a nagy sűrűségű négyzetnél az árnyék alapú algoritmus kisebb minimumot ér el és gyorsabban konvergál e minimum felé, míg a másik két térrésznél a Monte Carlo alapú vetítő konvergál gyorsabban. A kutatás e szakaszában az alábbi megállapításokra nem találtam magyarázatot.



25.ábra L2 norma az iterációs szám függvényében a kis sűrűségű négyzet helyén.



26.ábra L2 norma az iterációs szám függvényében a nagy sűrűségű területén.

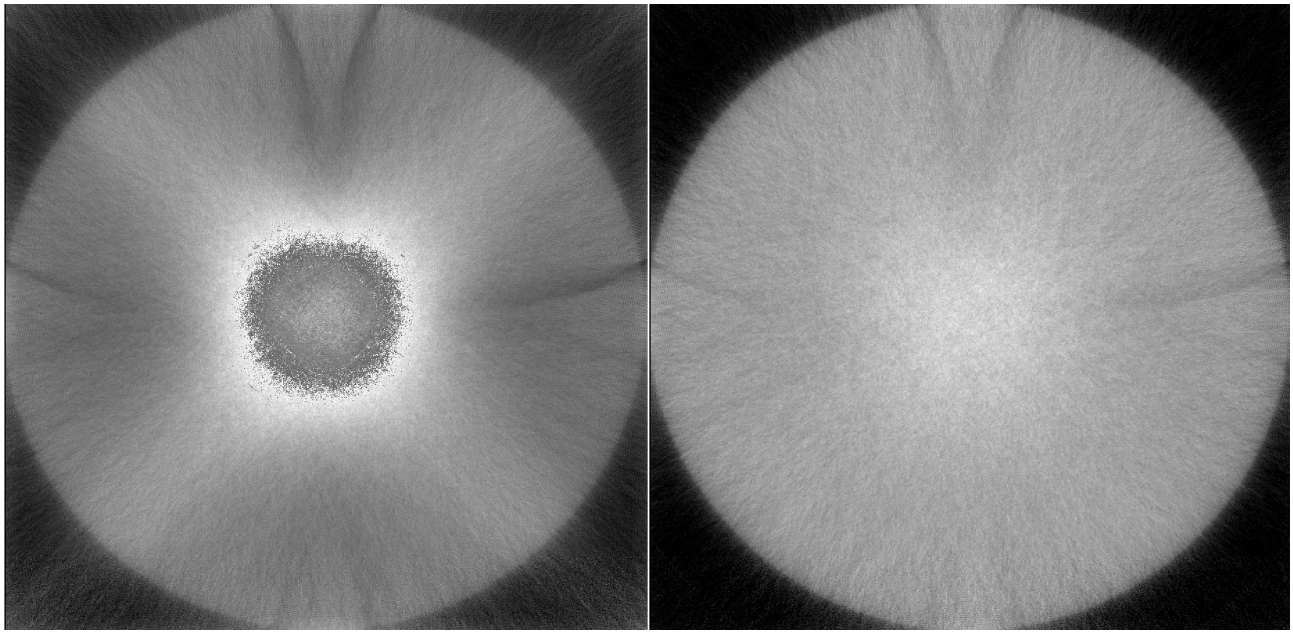


27.ábra L2 norma az iterációs szám függvényében a normál víz területén.

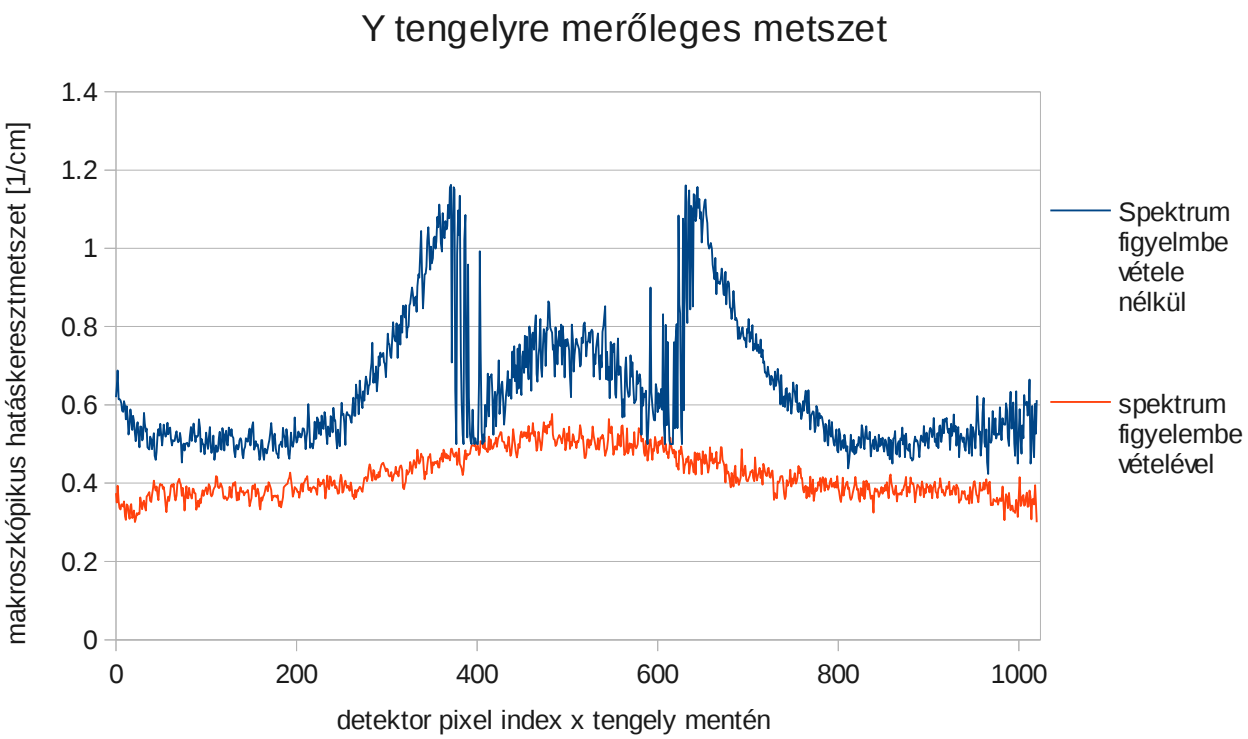
4.3 Spektrum felkeményedés eliminálása a rekonstruált képen fizika vagy szoftveres szűrés nélkül

Molnár Balázs az általam fejlesztett kód tovább verifikációját szolgáló szakdolgozati munkájában generált olyan fantomot és input paramétereket, mely jól szemlélteti a rekonstrukció során figyelembe vett spektrum, spektrum felkeményedést elimináló hatását. Kétféle esetben futtatta a rekonstrukciós algoritmust:

- A mérési eredmények generálása során bekapcsolt spektrum, de a rekonstrukció során nem. Ekkor látható a jelenség.(28. ábra bal oldala)
- A mérési eredmények generálása során bekapcsolt spektrum, és a rekonstrukció során is. Ekkor megszűnik az effektus. (28. ábra jobb oldala)



28.ábra A spektrum felkeményedés megjelenése (baloldal) és eliminálása a rekonstrukció előrevetítése során figyelembe vett spektrummal (jobb oldal).[17]



29.ábra A 28. ábrán látható kép y tengely menti metszete az origónál.

A spektrum használata nélkül kapott képen jól felismerhető a spektrum felkeményedésből eredő hiba (28. ábra), és a spektrum használat nélküli metszeti ábrán megfigyelhető a spektrum felkeményedésre jellemző inhomogenitás (29. ábra). Az anyag közepén megjelenő bemélyedés oka, hogy a kis energiás fotonok már kis mélységben elnyelődtek a mérési szimulálása során.

A programom a spektrum figyelembe vételével jól láthatóan eliminálta a spektrum felkeményedésből adódó képhibát.

5. Összefoglalás

Sikerült létre hozni egy CT rekonstrukcióra képes kódot, amely Monte Carlo alapú részecske transzportot használ az előrevetítő során. A megalkotott algoritmus értékhelyes, képes spektrum kezelésére és a kód egy változata alkalmas csoportos foton követésre is. A beütés regisztrálás index alapú megvalósításának köszönhetően, az algoritmus képes nagyobb méretű geometria kezelésére memóriahiány megjelenése nélkül, azaz a grafikus kártyák kisebb memória kapacitása okozta problémákat kisállat CT esetében hatékonyan kezeli a kódom.

Az algoritmus gyors futásához sikerült implementálnom grafikus kártyára a ML-TR algoritmust, melyet régi kártyán vizsgáltam. Habár a diplomamunkámban a rekonstrukció futási idejét nem taglaltam, a megvalósított rekonstrukció így is nagyságrendekkel lassabb, mint egy normál szűrt visszavetítés. Előnye a MC becslő alapú előrevetítés kernel, amely figyelembe veszi a röntgenforrás spektrumát, és könnyen bővíthető más effektusokkal. Ez pontosabb és artefaktumoktól mentes képet eredményez.

Kimutattam, hogy a spektrum felkeményedést sikeresen korrigálja az algoritmus, így az effektusra jellemző műtermékek nem jelennek meg a rekonstruált képen.

A kód további folytatásához javaslom, az MCNP segítségével szimulált, és a programom által generált előrevettés eredmények eltérésének elemzését. Érdemes lenne beilleszteni és megvizsgálni más iterációs séma használatát, mint például az ML-EM -et. A kód könnyen kibővíthető 3D-ra, így hasznos lenne vizsgálni a 3D képek generálásában nyújtott teljesítményét és a kapott képek minőségét. A pontosabb fizikai modell használata érdekében előrelépés lenne egy koherens és inkoherens szórást is támogató kód.

6. Irodalomjegyzék

1. OECD Imdicators „Health at a Glance” (2009)
http://www.oecd-ilibrary.org/
2. NVIDIA CUDA™, „NVIDIA CUDA C Programming Guide ” (2012)
3. Szirmay-Kalos, Szécsi, „GPGPU: General Pupose on Graphics Processing Units”**,** Algorithms of Informatics (2011) 1451-1495
http://sirkan.iit.bme.hu/~szirmay/gpgpu.pdf
4. **Karl Entacher, „A collection of classical pseudorandom number generators with linear structures -advanced version” (2000)**
http://random.mat.sbg.ac.at/results/karl/server/server.html
5. P. L'Ecuyer. , „Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure.”, Mathematics of Computation, 68(225):249-260, 1999.
6. A.D. Barnard et al., Guaranteeing the period of linear recurring sequences (mod 2**e), IEE Proceedings E140, 243 (1993)
7. NVIDIA CUDA™**, „CUDA Toolkit 5.0 CURAND Guide ” (2012)**
http://docs.nvidia.com/cuda/curand/index.html#topic_1 **.**
8. **George Marsaglia. Xorshift RNGs.** *Journal of Statistical Software* **8(14), 2003.**
http://www.jstatsoft.org/v08/i14/paper **.**
9. Marsaglia, G., „DIEHARD: A battery of Tests of Randomness” (1996)
http://stat.fsu.edu/~geo
10. Robert G. Brow, „DieHarder: A Gnu Public Licensed Random Number Tester” (2008)
http://www.phy.duke.edu/~rgb/General/dieharder.php
11. G. Jakab, T. Huszár, B. Csébfalvi „Iterative CT Recpnstruction on the GPU” Sixth Hungarian Conference on Computer Graphics and Geometry, 124-131 (2012)
12. K. Lange, R. Carson „EM Reconstruction Algorithms for Emission and Transmission Tomography” Journal of Computed Assisted Tomography 8, 306- 316 (1984)
13. B. De Man „Iterative Reconstruction for Reduction of Metal Artifacts in Computed Tomography ” (2001)
14. X-5 Monte Carlo Team , MCNP „A General Monte Carlo N-Particle Transport Code, Version 5 ” (2003)

15. J. K. Shultis , R. E. Faw , „AN MCNP PRIMER ” (2004)
16. http://en.wikipedia.org/wiki/Gift_wrapping_algorithm
17. B. Molnár „Iteratív képrekonstrukciós eljárás GPU-n gamma tomográfiához: tesztelés és verifikáció” (2013)

7. Függelék

A ' leap ahed ' algoritmus modulált multiplikatív módszerrel:

```
//result = base^exp (mod M)
#ifdef __CUDACC__
        __device__
#endif
unsigned long long mod_mul(unsigned long long base, unsigned long long exp,
unsigned long long mod){
      unsigned long long result;
      if(exp == 0) return 1;
      result = 1ULL;
      while(exp){
            if(exp&1ULL){
            //multiply
            result = result * base;
            result %= mod;
            exp = exp - 1;
            }

      base = base * base;
      base %= mod;
      exp = exp/2ULL;
      }
return result;
}
#ifdef __CUDACC__
        __device__
#endif
unsigned long long mod_mul_C(unsigned long long base, unsigned long long exp,
unsigned long long mod, unsigned long long C){
      unsigned long long result;
      unsigned long long f;
      if(exp == 0) return C;
      result = 0ULL;
      f = C;
      while(exp){
            if(exp&1ULL){
            //multiply
            result = result * base  + f;
            result %= mod;
            exp = exp - 1;
            }
      //square
      // Ha már páros a kitevő, négyzetre emelem az alapot, és felezem a
      // kitevő értékét
      f = f * (base +1ULL);
      f %= mod;
      base = base * base;
      base %= mod;
      exp = exp/2ULL;
      }
```

```
return result;
}


// leap ahead method:
#ifdef __CUDACC__
        __device__
#endif
unsigned long long lcg_leap(unsigned long long x0, unsigned long long a,
unsigned long long c, unsigned long long m, unsigned long long n){
      unsigned long long temp;
      unsigned long long p, prod;

      if(a <= 1){
            return 0;
      }
      if(x0 == 1){
            if(c == 0){
            return mod_mul(a, n, m);
            }
            if(c > 0){
            p = mod_mul(a,n,m);
            temp = mod_mul_C(a,n,m, c);
            return ((p + temp)%m);
            }
      }
      if(x0 == 0){
            if(c == 0)
            return 0;
            if(c > 0){
            temp = mod_mul_C(a,n,m, c);
            return (temp % m);
            }
      }
      if(x0 > 1){
            if(c == 0){
            p = mod_mul(a,n-1,m);
            return ((p*x0)%m);
            }
            if(c > 0 ){
            p = mod_mul(a,n,m);
            prod = (p * x0)%m;
            temp = mod_mul_C(a,n,m, c);
            return (prod + temp)%m;
            }
      }
return 0;
}
```

128 bites lcg:

```
// x[i] = a*x[i-1] + c
float m_rand_128 (unsigned long long* seed, unsigned long long* seed2) {
      unsigned long long aup, adown; // a paraméter két 64 bites számként
      unsigned long long temp1, temp2; // az alsó seed érték további két számra
```

```
        bontásához szükséges tárolók
        unsigned long long a1, a2; // az alsó 'a' érték további két számra
        bontásához szükséges tárolók
        // 'a' paraméter betöltése a két tárolóba
        adown = 334654252753519171 7ULL;
        aup = 136047214720561598 2ULL;
        // túlcsordulást tároló paraméter
        int carry = 0;
        *seed = (*seed)*adown + (*seed2)*aup; // (g*j + h*i) mod 64 a képlet
        alapján

        // (*seed2) * adown alsó 64 bitje a seed2 új értéke, míg a felső 64 bit
        hozzáadódik a seed értékéhez
        temp1 = (*seed2)&4294967295ULL; // seed2 alsó 32 bitje
        temp2 = ((*seed2)&18446744069414584320ULL)>>32; // seed2 felső 32 bitje
        a1 = adown&4294967295ULL; // 'adown' alsó 32 bitje
        a2 = (adown&18446744069414584320ULL)>>32; // 'adown' alsó 32 bitje
        *seed2 = temp1*a1; // kiszámolom a két legalsó 32 bites szám szorzatát
        // vizsgálom, hogy van e túlcsordulás, mert lehetséges, hogy 64 bit is
        kevés
        if( (*seed2) + ((temp2*a1)*4294967296ULL) < (*seed2) ) carry++;
        *seed2 += (temp2*a1)*4294967296ULL;
        // vizsgálom, hogy van e túlcsordulás, mert lehetséges, hogy 64 bit is
        kevés
        if( (*seed2) + ((temp1*a2)*4294967296ULL) < (*seed2) ) carry++;
        *seed2 += (temp1*a2)*4294967296ULL;
        // az alsó bitek számításnál kapott túlcsordult érték hozzácsapása a
        felső bitekhez
        *seed += (unsigned long long)carry + temp2*a2  + ( ((temp2*a1) &
        18446744069414584320ULL)/4294967296ULL ) + ( ((temp1*a2) &
        18446744069414584320ULL ) /4294967296ULL );
        // visszatér egy float értékkel,amit úgy kapunk meg, hogy csak a felső 64
        bitetet nézzük azaz seed *1/2^64
        return (float)5.42101086E-20f*(*seed);
}
```

64 bites lcg:

```
float m_rand_withint(unsigned int* seed, unsigned int* seed2)
{
        unsigned int aup, adown; // the 'a' parameter in two 32 bit number
        unsigned int temp1, temp2;
        unsigned int a1, a2, m1, m2;

        // load the 'a' and 'm' parameter
        adown = 2809915765;
        aup = 2146638330;
        m1 = 2147483648;
        m2 = 0;

        // overload handler variable
        unsigned int carry = 0;

        // a*x
```

```
        *seed = (*seed)*adown + (*seed2)*aup; // (g*j + h*i) mod 64
        temp1 = (*seed2)&65535;
        temp2 = ((*seed2)&4294901760)>>16;
        a1 = adown&65535;
        a2 = (adown&4294901760)>>16;
        *seed2 = temp1*a1;
        if( (*seed2) + ((temp2*a1)*65536) < (*seed2) ) carry++;
        *seed2 += (temp2*a1)*65536;
        if( (*seed2) + ((temp1*a2)*65536) < (*seed2) ) carry++;
        *seed2 += (temp1*a2)*65536;
        *seed += carry + temp2*a2  + ( ((temp2*a1) &    4294901760)/65536 ) + (
((temp1*a2)&4294901760) /65536 );

    // +C
        temp2 = *seed2;
        *seed2 +=1;
    if (temp2>*seed2)
        {
        *seed+=1;
    }
        //%m
        if(m1<=*seed)
        {
        *seed -=m1;
    }
    // * 1/2^63
    return 1.08420217E-19f*(float)(*seed2) + 4.65661287E-10f*(float)(*seed);
}
```

Dieharder szkript:

```
#!/bin/sh  mcanp
echo -n 'filename: '
read filename
i="1"
while [ $i -le 19 ]
do
        echo './a.out | dieharder -g 200 -d '$i '>>' $filename
        ./a.out | dieharder -g 200 -d $i >> $filename
        i=`expr $i + 1`
done
./a.out | dieharder -g 200 -a >> $filename
```

64 bites deiharder-nél elégséges a” ./a.out | dieharder -g 200 -a >> $filename „ parancs használata

Dieharder teszt eredmények:

dieharder version 3.31.1 Copyright 2003 Robert G. Brown						rgb_bitdist	5	100000	100	0.15	PASSED
XORWOW						rgb_bitdist	6	100000	100	0.54	PASSED
test_name	ntup	tsamples	psamples	p-value	Assessment	rgb_bitdist	7	100000	100	0.72	PASSED
diehard_birthdays	0	100	100	0.81	PASSED	rgb_bitdist	8	100000	100	0.99	PASSED
diehard_operm5	0	1000000	100	0.20	PASSED	rgb_bitdist	9	100000	100	0.99	PASSED
diehard_rank_32x32	0	40000	100	0.16	PASSED	rgb_bitdist	10	100000	100	0.17	PASSED
diehard_rank_6x8	0	100000	100	0.97	PASSED	rgb_bitdist	11	100000	100	0.47	PASSED
diehard_bitstream	0	2097152	100	0.76	PASSED	rgb_bitdist	12	100000	100	0.46	PASSED
diehard_opso	0	2097152	100	0.70	PASSED	rgb_minimum_distance	2	10000	###	0.96	PASSED
diehard_oqso	0	2097152	100	0.96	PASSED	rgb_minimum_distance	3	10000	###	0.02	PASSED
diehard_dna	0	2097152	100	0.13	PASSED	rgb_minimum_distance	4	10000	###	0.89	PASSED
diehard_count_1s_str	0	256000	100	0.70	PASSED	rgb_minimum_distance	5	10000	###	0.45	PASSED
diehard_count_1s_byt	0	256000	100	0.77	PASSED	rgb_permutations	2	100000	100	0.75	PASSED
diehard_parking_lot	0	12000	100	0.41	PASSED	rgb_permutations	3	100000	100	0.11	PASSED
diehard_2dsphere	2	8000	100	0.37	PASSED	rgb_permutations	4	100000	100	0.99	PASSED
diehard_3dsphere	3	4000	100	0.18	PASSED	rgb_permutations	5	100000	100	0.46	PASSED
diehard_squeeze	0	100000	100	0.98	PASSED	rgb_lagged_sum	0	1000000	100	0.50	PASSED
diehard_sums	0	100	100	0.21	PASSED	rgb_lagged_sum	1	1000000	100	0.45	PASSED
diehard_runs	0	100000	100	0.68	PASSED	rgb_lagged_sum	2	1000000	100	0.37	PASSED
diehard_runs	0	100000	100	0.29	PASSED	rgb_lagged_sum	3	1000000	100	0.11	PASSED
diehard_craps	0	200000	100	0.81	PASSED	rgb_lagged_sum	4	1000000	100	0.62	PASSED
diehard_craps	0	200000	100	0.04	PASSED	rgb_lagged_sum	5	1000000	100	0.25	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.66	PASSED	rgb_lagged_sum	6	1000000	100	0.03	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.01	PASSED	rgb_lagged_sum	7	1000000	100	0.52	PASSED
sts_monobit	1	100000	100	0.98	PASSED	rgb_lagged_sum	8	1000000	100	0.87	PASSED
sts_runs	2	100000	100	0.62	PASSED	rgb_lagged_sum	9	1000000	100	0.54	PASSED
sts_serial	1	100000	100	0.83	PASSED	rgb_lagged_sum	10	1000000	100	0.91	PASSED
sts_serial	2	100000	100	0.35	PASSED	rgb_lagged_sum	11	1000000	100	0.64	PASSED
sts_serial	3	100000	100	0.29	PASSED	rgb_lagged_sum	12	1000000	100	0.50	PASSED
sts_serial	3	100000	100	0.52	PASSED	rgb_lagged_sum	13	1000000	100	0.89	PASSED
sts_serial	4	100000	100	0.18	PASSED	rgb_lagged_sum	14	1000000	100	0.82	PASSED
sts_serial	4	100000	100	0.07	PASSED	rgb_lagged_sum	15	1000000	100	0.44	PASSED
sts_serial	5	100000	100	0.46	PASSED	rgb_lagged_sum	16	1000000	100	0.11	PASSED
sts_serial	5	100000	100	0.00	WEAK	rgb_lagged_sum	17	1000000	100	0.03	PASSED
sts_serial	6	100000	100	0.48	PASSED	rgb_lagged_sum	18	1000000	100	0.87	PASSED
sts_serial	6	100000	100	0.73	PASSED	rgb_lagged_sum	19	1000000	100	0.42	PASSED
sts_serial	7	100000	100	0.06	PASSED	rgb_lagged_sum	20	1000000	100	0.91	PASSED
sts_serial	7	100000	100	0.01	PASSED	rgb_lagged_sum	21	1000000	100	0.29	PASSED
sts_serial	8	100000	100	0.74	PASSED	rgb_lagged_sum	22	1000000	100	0.56	PASSED
sts_serial	8	100000	100	0.56	PASSED	rgb_lagged_sum	23	1000000	100	1.00	WEAK
sts_serial	9	100000	100	0.34	PASSED	rgb_lagged_sum	24	1000000	100	0.20	PASSED
sts_serial	9	100000	100	0.65	PASSED	rgb_lagged_sum	25	1000000	100	0.76	PASSED
sts_serial	10	100000	100	0.50	PASSED	rgb_lagged_sum	26	1000000	100	0.15	PASSED
sts_serial	10	100000	100	0.66	PASSED	rgb_lagged_sum	27	1000000	100	0.96	PASSED
sts_serial	11	100000	100	0.92	PASSED	rgb_lagged_sum	28	1000000	100	0.18	PASSED
sts_serial	11	100000	100	0.98	PASSED	rgb_lagged_sum	29	1000000	100	0.69	PASSED
sts_serial	12	100000	100	0.90	PASSED	rgb_lagged_sum	30	1000000	100	0.71	PASSED
sts_serial	12	100000	100	0.81	PASSED	rgb_lagged_sum	31	1000000	100	0.04	PASSED
sts_serial	13	100000	100	0.82	PASSED	rgb_lagged_sum	32	1000000	100	0.58	PASSED
sts_serial	13	100000	100	0.63	PASSED	rgb_kstest_test	0	10000	###	0.63	PASSED
sts_serial	14	100000	100	0.68	PASSED	dab_bytedistrib	0	51200000	1	0.04	PASSED
sts_serial	14	100000	100	0.26	PASSED	dab_dct	256	50000	1	0.35	PASSED
sts_serial	15	100000	100	0.24	PASSED	Preparing to run test 207. ntuple = 0					
sts_serial	15	100000	100	0.07	PASSED	dab_filltree	32	15000000	1	0.26	PASSED
sts_serial	16	100000	100	0.20	PASSED	dab_filltree	32	15000000	1	0.82	PASSED
sts_serial	16	100000	100	0.53	PASSED	Preparing to run test 208. ntuple = 0					
rgb_bitdist	1	100000	100	0.44	PASSED	dab_filltree2	0	5000000	1	0.12	PASSED
rgb_bitdist	2	100000	100	0.97	PASSED	dab_filltree2	1	5000000	1	0.56	PASSED
rgb_bitdist	3	100000	100	0.83	PASSED	Preparing to run test 209. ntuple = 0					
rgb_bitdist	4	100000	100	0.34	PASSED	dab_monobit2	12	65000000	1	0.49	PASSED

dieharder version 3.31.1 Copyright 2003 Robert G. Brown					
MRG32Ka					
test_name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.81	PASSED
diehard_operm5	0	1000000	100	0.30	PASSED
diehard_rank_32x32	0	40000	100	0.82	PASSED
diehard_rank_6x8	0	100000	100	0.54	PASSED
diehard_bitstream	0	2097152	100	0.39	PASSED
diehard_opso	0	2097152	100	0.56	PASSED
diehard_oqso	0	2097152	100	0.16	PASSED
diehard_dna	0	2097152	100	0.68	PASSED
diehard_count_1s_str	0	256000	100	0.94	PASSED
diehard_count_1s_byt	0	256000	100	0.06	PASSED
diehard_parking_lot	0	12000	100	0.46	PASSED
diehard_2dsphere	2	8000	100	0.82	PASSED
diehard_3dsphere	3	4000	100	0.50	PASSED
diehard_squeeze	0	100000	100	0.70	PASSED
diehard_sums	0	100	100	0.01	PASSED
diehard_runs	0	100000	100	0.84	PASSED
diehard_runs	0	100000	100	0.65	PASSED
diehard_craps	0	200000	100	0.17	PASSED
diehard_craps	0	200000	100	0.92	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.82	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.57	PASSED
sts_monobit	1	100000	100	0.98	PASSED
sts_runs	2	100000	100	0.90	PASSED
sts_serial	1	100000	100	0.35	PASSED
sts_serial	2	100000	100	0.93	PASSED
sts_serial	3	100000	100	0.96	PASSED
sts_serial	3	100000	100	0.47	PASSED
sts_serial	4	100000	100	0.68	PASSED
sts_serial	4	100000	100	0.43	PASSED
sts_serial	5	100000	100	0.49	PASSED
sts_serial	5	100000	100	0.83	PASSED
sts_serial	6	100000	100	0.88	PASSED
sts_serial	6	100000	100	0.76	PASSED
sts_serial	7	100000	100	0.74	PASSED
sts_serial	7	100000	100	0.37	PASSED
sts_serial	8	100000	100	0.34	PASSED
sts_serial	8	100000	100	0.76	PASSED
sts_serial	9	100000	100	0.12	PASSED
sts_serial	9	100000	100	0.54	PASSED
sts_serial	10	100000	100	0.76	PASSED
sts_serial	10	100000	100	0.58	PASSED
sts_serial	11	100000	100	0.94	PASSED
sts_serial	11	100000	100	0.84	PASSED
sts_serial	12	100000	100	0.18	PASSED
sts_serial	12	100000	100	0.06	PASSED
sts_serial	13	100000	100	0.31	PASSED
sts_serial	13	100000	100	0.62	PASSED
sts_serial	14	100000	100	0.54	PASSED
sts_serial	14	100000	100	0.45	PASSED
sts_serial	15	100000	100	0.98	PASSED
sts_serial	15	100000	100	0.16	PASSED
sts_serial	16	100000	100	0.42	PASSED
sts_serial	16	100000	100	0.49	PASSED
rgb_bitdist	1	100000	100	0.25	PASSED
rgb_bitdist	2	100000	100	0.71	PASSED
rgb_bitdist	3	100000	100	0.44	PASSED
rgb_bitdist	4	100000	100	0.31	PASSED
rgb_bitdist	5	100000	100	0.45	PASSED
rgb_bitdist	6	100000	100	0.78	PASSED
rgb_bitdist	7	100000	100	0.02	PASSED
rgb_bitdist	8	100000	100	0.18	PASSED
rgb_bitdist	9	100000	100	0.77	PASSED
rgb_bitdist	10	100000	100	0.45	PASSED
rgb_bitdist	11	100000	100	0.76	PASSED
rgb_bitdist	12	100000	100	0.28	PASSED
rgb_minimum_distance	2	10000	1000	0.06	PASSED
rgb_minimum_distance	3	10000	1000	0.76	PASSED
rgb_minimum_distance	4	10000	1000	0.20	PASSED
rgb_minimum_distance	5	10000	1000	0.77	PASSED
rgb_permutations	2	100000	100	0.46	PASSED
rgb_permutations	3	100000	100	0.95	PASSED
rgb_permutations	4	100000	100	0.00	WEAK
rgb_permutations	5	100000	100	0.03	PASSED
rgb_lagged_sum	0	1000000	100	0.33	PASSED
rgb_lagged_sum	1	1000000	100	0.53	PASSED
rgb_lagged_sum	2	1000000	100	0.87	PASSED
rgb_lagged_sum	3	1000000	100	0.34	PASSED
rgb_lagged_sum	4	1000000	100	0.18	PASSED
rgb_lagged_sum	5	1000000	100	0.86	PASSED
rgb_lagged_sum	6	1000000	100	0.94	PASSED
rgb_lagged_sum	7	1000000	100	0.62	PASSED
rgb_lagged_sum	8	1000000	100	0.18	PASSED
rgb_lagged_sum	9	1000000	100	0.86	PASSED
rgb_lagged_sum	10	1000000	100	0.94	PASSED
rgb_lagged_sum	11	1000000	100	0.48	PASSED
rgb_lagged_sum	12	1000000	100	0.95	PASSED
rgb_lagged_sum	13	1000000	100	0.08	PASSED
rgb_lagged_sum	14	1000000	100	0.34	PASSED
rgb_lagged_sum	15	1000000	100	0.38	PASSED
rgb_lagged_sum	16	1000000	100	0.72	PASSED
rgb_lagged_sum	17	1000000	100	1.00	WEAK
rgb_lagged_sum	18	1000000	100	0.95	PASSED
rgb_lagged_sum	19	1000000	100	0.25	PASSED
rgb_lagged_sum	20	1000000	100	0.98	PASSED
rgb_lagged_sum	21	1000000	100	0.54	PASSED
rgb_lagged_sum	22	1000000	100	0.92	PASSED
rgb_lagged_sum	23	1000000	100	0.94	PASSED
rgb_lagged_sum	24	1000000	100	0.31	PASSED
rgb_lagged_sum	25	1000000	100	0.31	PASSED
rgb_lagged_sum	26	1000000	100	0.21	PASSED
rgb_lagged_sum	27	1000000	100	0.46	PASSED
rgb_lagged_sum	28	1000000	100	0.68	PASSED
rgb_lagged_sum	29	1000000	100	0.73	PASSED
rgb_lagged_sum	30	1000000	100	0.53	PASSED
rgb_lagged_sum	31	1000000	100	0.76	PASSED
rgb_lagged_sum	32	1000000	100	0.17	PASSED
rgb_kstest_test	0	10000	1000	0.17	PASSED
dab_bytedistrib	0	5E+007	1	0.79	PASSED
dab_dct	256	50000	1	0.82	PASSED
Preparing to run test 207. ntuple = 0					
dab_filltree	32	2E+007	1	0.16	PASSED
dab_filltree	32	2E+007	1	0.75	PASSED
Preparing to run test 208. ntuple = 0					
dab_filltree2	0	5000000	1	0.97	PASSED
dab_filltree2	1	5000000	1	0.08	PASSED
Preparing to run test 209. ntuple = 0					
dab_monobit2	12	7E+007	1	0.95	PASSED

dieharder version 3.31.1 Copyright 2003 Robert G. Brown						rgb_bitdist	5	100000	100	0.00	FAILED
Sobol32						rgb_bitdist	6	100000	100	0.00	FAILED
test_name	ntup	tsamples	psamples	p-value	Assessment	rgb_bitdist	7	100000	100	0.00	FAILED
diehard_birthdays	0	100	100	0.00	FAILED	rgb_bitdist	8	100000	100	0.00	FAILED
diehard_operm5	0	1000000	100	0.00	FAILED	rgb_bitdist	9	100000	100	0.00	FAILED
diehard_rank_32x32	0	40000	100	0.00	FAILED	rgb_bitdist	10	100000	100	0.00	FAILED
diehard_rank_6x8	0	100000	100	0.00	FAILED	rgb_bitdist	11	100000	100	0.00	FAILED
diehard_bitstream	0	2097152	100	0.00	FAILED	rgb_bitdist	12	100000	100	0.00	FAILED
diehard_opso	0	2097152	100	0.00	FAILED	rgb_minimum_distance	2	10000	1000	0.00	FAILED
diehard_oqso	0	2097152	100	0.00	FAILED	rgb_minimum_distance	3	10000	1000	0.00	FAILED
diehard_dna	0	2097152	100	0.00	FAILED	rgb_minimum_distance	4	10000	1000	0.00	FAILED
diehard_count_1s_str	0	256000	100	0.00	FAILED	rgb_minimum_distance	5	10000	1000	0.00	FAILED
diehard_count_1s_byt	0	256000	100	0.00	FAILED	rgb_permutations	2	100000	100	0.00	FAILED
diehard_parking_lot	0	12000	100	0.00	FAILED	rgb_permutations	3	100000	100	0.00	FAILED
diehard_2dsphere	2	8000	100	0.00	FAILED	rgb_permutations	4	100000	100	0.00	FAILED
diehard_3dsphere	3	4000	100	0.00	FAILED	rgb_permutations	5	100000	100	0.00	FAILED
diehard_squeeze	0	100000	100	0.00	FAILED	rgb_lagged_sum	0	1000000	100	0.00	FAILED
diehard_sums	0	100	100	0.00	FAILED	rgb_lagged_sum	1	1000000	100	0.00	FAILED
diehard_runs	0	100000	100	0.00	FAILED	rgb_lagged_sum	2	1000000	100	0.11	PASSED
diehard_runs	0	100000	100	0.00	FAILED	rgb_lagged_sum	3	1000000	100	0.00	WEAK
diehard_craps	0	200000	100	0.00	FAILED	rgb_lagged_sum	4	1000000	100	0.51	PASSED
diehard_craps	0	200000	100	0.00	FAILED	rgb_lagged_sum	5	1000000	100	0.91	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED	rgb_lagged_sum	6	1000000	100	0.19	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED	rgb_lagged_sum	7	1000000	100	0.25	PASSED
sts_monobit	1	100000	100	0.00	FAILED	rgb_lagged_sum	8	1000000	100	0.23	PASSED
sts_runs	2	100000	100	0.00	FAILED	rgb_lagged_sum	9	1000000	100	0.48	PASSED
sts_serial	1	100000	100	0.00	FAILED	rgb_lagged_sum	10	1000000	100	0.11	PASSED
sts_serial	2	100000	100	0.00	FAILED	rgb_lagged_sum	11	1000000	100	0.05	PASSED
sts_serial	3	100000	100	0.00	FAILED	rgb_lagged_sum	12	1000000	100	0.95	PASSED
sts_serial	3	100000	100	0.00	FAILED	rgb_lagged_sum	13	1000000	100	1.00	WEAK
sts_serial	4	100000	100	0.00	FAILED	rgb_lagged_sum	14	1000000	100	0.56	PASSED
sts_serial	4	100000	100	0.00	FAILED	rgb_lagged_sum	15	1000000	100	0.60	PASSED
sts_serial	5	100000	100	0.00	FAILED	rgb_lagged_sum	16	1000000	100	0.51	PASSED
sts_serial	5	100000	100	0.00	FAILED	rgb_lagged_sum	17	1000000	100	0.60	PASSED
sts_serial	6	100000	100	0.00	FAILED	rgb_lagged_sum	18	1000000	100	0.85	PASSED
sts_serial	6	100000	100	0.00	FAILED	rgb_lagged_sum	19	1000000	100	0.63	PASSED
sts_serial	7	100000	100	0.00	FAILED	rgb_lagged_sum	20	1000000	100	0.80	PASSED
sts_serial	7	100000	100	0.00	FAILED	rgb_lagged_sum	21	1000000	100	0.90	PASSED
sts_serial	8	100000	100	0.00	FAILED	rgb_lagged_sum	22	1000000	100	0.06	PASSED
sts_serial	8	100000	100	0.00	FAILED	rgb_lagged_sum	23	1000000	100	0.81	PASSED
sts_serial	9	100000	100	0.00	FAILED	rgb_lagged_sum	24	1000000	100	0.35	PASSED
sts_serial	9	100000	100	0.00	FAILED	rgb_lagged_sum	25	1000000	100	0.25	PASSED
sts_serial	10	100000	100	0.00	FAILED	rgb_lagged_sum	26	1000000	100	0.19	PASSED
sts_serial	10	100000	100	0.00	FAILED	rgb_lagged_sum	27	1000000	100	0.24	PASSED
sts_serial	11	100000	100	0.00	FAILED	rgb_lagged_sum	28	1000000	100	0.42	PASSED
sts_serial	11	100000	100	0.00	FAILED	rgb_lagged_sum	29	1000000	100	0.97	PASSED
sts_serial	12	100000	100	0.00	FAILED	rgb_lagged_sum	30	1000000	100	0.52	PASSED
sts_serial	12	100000	100	0.00	FAILED	rgb_lagged_sum	31	1000000	100	0.65	PASSED
sts_serial	13	100000	100	0.00	FAILED	rgb_lagged_sum	32	1000000	100	0.59	PASSED
sts_serial	13	100000	100	0.00	FAILED	rgb_kstest_test	0	10000	1000	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED	dab_bytedistrib	0	51200000	1	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED	dab_dct	256	50000	1	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED	Preparing to run test 207. ntuple = 0					
sts_serial	15	100000	100	0.00	FAILED	dab_filltree	32	15000000	1	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED	dab_filltree	32	15000000	1	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED	Preparing to run test 208. ntuple = 0					
rgb_bitdist	1	100000	100	0.00	FAILED	dab_filltree2	0	5000000	1	0.00	FAILED
rgb_bitdist	2	100000	100	0.00	FAILED	dab_filltree2	1	5000000	1	0.00	FAILED
rgb_bitdist	3	100000	100	0.00	FAILED	Preparing to run test 209. ntuple = 0					
rgb_bitdist	4	100000	100	0.00	FAILED	dab_monobit2	12	65000000	1	1.00	FAILED

Scrambled Sobol32					
test_name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.00	FAILED
diehard_operm5	0	1000000	100	0.00	FAILED
diehard_rank_32x32	0	40000	100	0.00	FAILED
diehard_rank_6x8	0	100000	100	0.00	FAILED
diehard_bitstream	0	2097152	100	0.00	FAILED
diehard_opso	0	2097152	100	0.00	FAILED
diehard_oqso	0	2097152	100	0.00	FAILED
diehard_dna	0	2097152	100	0.00	FAILED
diehard_count_1s_str	0	256000	100	0.00	FAILED
diehard_count_1s_byt	0	256000	100	0.00	FAILED
diehard_parking_lot	0	12000	100	0.00	FAILED
diehard_2dsphere	2	8000	100	0.00	FAILED
diehard_3dsphere	3	4000	100	0.00	FAILED
diehard_squeeze	0	100000	100	0.00	FAILED
diehard_sums	0	100	100	0.00	FAILED
diehard_runs	0	100000	100	0.00	FAILED
diehard_runs	0	100000	100	0.00	FAILED
diehard_craps	0	200000	100	0.00	FAILED
diehard_craps	0	200000	100	0.00	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED
sts_monobit	1	100000	100	0.00	FAILED
sts_runs	2	100000	100	0.00	FAILED
sts_serial	1	100000	100	0.00	FAILED
sts_serial	2	100000	100	0.00	FAILED
sts_serial	3	100000	100	0.00	FAILED
sts_serial	3	100000	100	0.00	FAILED
sts_serial	4	100000	100	0.00	FAILED
sts_serial	4	100000	100	0.00	FAILED
sts_serial	5	100000	100	0.00	FAILED
sts_serial	5	100000	100	0.00	FAILED
sts_serial	6	100000	100	0.00	FAILED
sts_serial	6	100000	100	0.00	FAILED
sts_serial	7	100000	100	0.00	FAILED
sts_serial	7	100000	100	0.00	FAILED
sts_serial	8	100000	100	0.00	FAILED
sts_serial	8	100000	100	0.00	FAILED
sts_serial	9	100000	100	0.00	FAILED
sts_serial	9	100000	100	0.00	FAILED
sts_serial	10	100000	100	0.00	FAILED
sts_serial	10	100000	100	0.00	FAILED
sts_serial	11	100000	100	0.00	FAILED
sts_serial	11	100000	100	0.00	FAILED
sts_serial	12	100000	100	0.00	FAILED
sts_serial	12	100000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED
sts_serial	14	100000	100	0.00	WEAK
sts_serial	15	100000	100	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED
rgb_bitdist	1	100000	100	0.00	FAILED
rgb_bitdist	2	100000	100	0.00	FAILED
rgb_bitdist	3	100000	100	0.00	FAILED
rgb_bitdist	4	100000	100	0.00	FAILED
rgb_bitdist	5	100000	100	0.00	FAILED
rgb_bitdist	6	100000	100	0.00	WEAK
rgb_bitdist	7	100000	100	0.83	PASSED
rgb_bitdist	8	100000	100	0.00	FAILED
rgb_bitdist	9	100000	100	0.97	PASSED
rgb_bitdist	10	100000	100	0.38	PASSED
rgb_bitdist	11	100000	100	0.13	PASSED
rgb_bitdist	12	100000	100	0.66	PASSED
rgb_minimum_distance	2	10000	1000	0.00	FAILED
rgb_minimum_distance	3	10000	1000	0.00	FAILED
rgb_minimum_distance	4	10000	1000	0.00	FAILED
rgb_minimum_distance	5	10000	1000	0.00	FAILED
rgb_permutations	2	100000	100	0.00	FAILED
rgb_permutations	3	100000	100	0.00	FAILED
rgb_permutations	4	100000	100	0.00	FAILED
rgb_permutations	5	100000	100	0.00	FAILED
rgb_lagged_sum	0	1000000	100	0.00	FAILED
rgb_lagged_sum	1	1000000	100	0.00	FAILED
rgb_lagged_sum	2	1000000	100	0.93	PASSED
rgb_lagged_sum	3	1000000	100	0.00	WEAK
rgb_lagged_sum	4	1000000	100	0.86	PASSED
rgb_lagged_sum	5	1000000	100	0.92	PASSED
rgb_lagged_sum	6	1000000	100	0.55	PASSED
rgb_lagged_sum	7	1000000	100	0.65	PASSED
rgb_lagged_sum	8	1000000	100	0.87	PASSED
rgb_lagged_sum	9	1000000	100	0.57	PASSED
rgb_lagged_sum	10	1000000	100	0.40	PASSED
rgb_lagged_sum	11	1000000	100	0.64	PASSED
rgb_lagged_sum	12	1000000	100	0.98	PASSED
rgb_lagged_sum	13	1000000	100	0.81	PASSED
rgb_lagged_sum	14	1000000	100	0.74	PASSED
rgb_lagged_sum	15	1000000	100	0.56	PASSED
rgb_lagged_sum	16	1000000	100	0.78	PASSED
rgb_lagged_sum	17	1000000	100	0.62	PASSED
rgb_lagged_sum	18	1000000	100	0.59	PASSED
rgb_lagged_sum	19	1000000	100	0.48	PASSED
rgb_lagged_sum	20	1000000	100	0.66	PASSED
rgb_lagged_sum	21	1000000	100	0.55	PASSED
rgb_lagged_sum	22	1000000	100	0.72	PASSED
rgb_lagged_sum	23	1000000	100	0.00	WEAK
rgb_lagged_sum	24	1000000	100	0.95	PASSED
rgb_lagged_sum	25	1000000	100	0.48	PASSED
rgb_lagged_sum	26	1000000	100	0.43	PASSED
rgb_lagged_sum	27	1000000	100	0.50	PASSED
rgb_lagged_sum	28	1000000	100	0.48	PASSED
rgb_lagged_sum	29	1000000	100	0.90	PASSED
rgb_lagged_sum	30	1000000	100	0.21	PASSED
rgb_lagged_sum	31	1000000	100	0.12	PASSED
rgb_lagged_sum	32	1000000	100	0.83	PASSED
rgb_kstest_test	0	10000	1000	0.00	FAILED
dab_bytedistrib	0	51200000	1	1.00	FAILED
dab_dct	256	50000	1	0.00	FAILED
Preparing to run test 207. ntuple = 0					
dab_filltree	32	15000000	1	0.00	FAILED
dab_filltree	32	15000000	1	0.00	FAILED
Preparing to run test 208. ntuple = 0					
dab_filltree2	0	5000000	1	0.00	FAILED
dab_filltree2	1	5000000	1	0.00	FAILED
Preparing to run test 209. ntuple = 0					
dab_monobit2	12	65000000	1	1.00	FAILED

dieharder version 3.31.1 Copyright 2003 Robert G. Brown						rgb_bitdist	5	100000	100	0.00	WEAK
Sobol64						rgb_bitdist	6	100000	100	0.00	WEAK
test_name	ntup	tsamples	psamples	p-value	Assessment	rgb_bitdist	7	100000	100	0.67	PASSED
diehard_birthdays	0	100	100	0.00	FAILED	rgb_bitdist	8	100000	100	0.00	FAILED
diehard_operm5	0	1000000	100	0.00	FAILED	rgb_bitdist	9	100000	100	0.85	PASSED
diehard_rank_32x32	0	40000	100	0.00	FAILED	rgb_bitdist	10	100000	100	0.69	PASSED
diehard_rank_6x8	0	100000	100	0.00	FAILED	rgb_bitdist	11	100000	100	0.83	PASSED
diehard_bitstream	0	2097152	100	0.00	FAILED	rgb_bitdist	12	100000	100	0.25	PASSED
diehard_opso	0	2097152	100	0.00	FAILED	rgb_minimum_distance	2	10000	1000	0.00	FAILED
diehard_oqso	0	2097152	100	0.00	FAILED	rgb_minimum_distance	3	10000	1000	0.00	FAILED
diehard_dna	0	2097152	100	0.00	FAILED	rgb_minimum_distance	4	10000	1000	0.00	FAILED
diehard_count_1s_str	0	256000	100	0.00	FAILED	rgb_minimum_distance	5	10000	1000	0.00	FAILED
diehard_count_1s_byt	0	256000	100	0.00	FAILED	rgb_permutations	2	100000	100	0.00	FAILED
diehard_parking_lot	0	12000	100	0.00	FAILED	rgb_permutations	3	100000	100	0.00	FAILED
diehard_2dsphere	2	8000	100	0.00	FAILED	rgb_permutations	4	100000	100	0.00	FAILED
diehard_3dsphere	3	4000	100	0.00	FAILED	rgb_permutations	5	100000	100	0.00	FAILED
diehard_squeeze	0	100000	100	0.00	FAILED	rgb_lagged_sum	0	1000000	100	0.00	FAILED
diehard_sums	0	100	100	0.00	FAILED	rgb_lagged_sum	1	1000000	100	0.00	FAILED
diehard_runs	0	100000	100	0.00	FAILED	rgb_lagged_sum	2	1000000	100	0.21	PASSED
diehard_runs	0	100000	100	0.00	FAILED	rgb_lagged_sum	3	1000000	100	0.00	WEAK
diehard_craps	0	200000	100	0.00	FAILED	rgb_lagged_sum	4	1000000	100	0.00	WEAK
diehard_craps	0	200000	100	0.00	FAILED	rgb_lagged_sum	5	1000000	100	0.74	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED	rgb_lagged_sum	6	1000000	100	0.53	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED	rgb_lagged_sum	7	1000000	100	0.00	WEAK
sts_monobit	1	100000	100	0.00	FAILED	rgb_lagged_sum	8	1000000	100	0.72	PASSED
sts_runs	2	100000	100	0.00	FAILED	rgb_lagged_sum	9	1000000	100	0.00	WEAK
sts_serial	1	100000	100	0.00	FAILED	rgb_lagged_sum	10	1000000	100	0.37	PASSED
sts_serial	2	100000	100	0.00	FAILED	rgb_lagged_sum	11	1000000	100	0.03	PASSED
sts_serial	3	100000	100	0.00	FAILED	rgb_lagged_sum	12	1000000	100	0.28	PASSED
sts_serial	3	100000	100	0.00	FAILED	rgb_lagged_sum	13	1000000	100	0.77	PASSED
sts_serial	4	100000	100	0.00	FAILED	rgb_lagged_sum	14	1000000	100	0.02	PASSED
sts_serial	4	100000	100	0.00	FAILED	rgb_lagged_sum	15	1000000	100	0.02	PASSED
sts_serial	5	100000	100	0.00	FAILED	rgb_lagged_sum	16	1000000	100	0.63	PASSED
sts_serial	5	100000	100	0.00	FAILED	rgb_lagged_sum	17	1000000	100	0.50	PASSED
sts_serial	6	100000	100	0.00	FAILED	rgb_lagged_sum	18	1000000	100	0.85	PASSED
sts_serial	6	100000	100	0.00	FAILED	rgb_lagged_sum	19	1000000	100	0.08	PASSED
sts_serial	7	100000	100	0.00	FAILED	rgb_lagged_sum	20	1000000	100	0.95	PASSED
sts_serial	7	100000	100	0.00	FAILED	rgb_lagged_sum	21	1000000	100	0.06	PASSED
sts_serial	8	100000	100	0.00	FAILED	rgb_lagged_sum	22	1000000	100	0.98	PASSED
sts_serial	8	100000	100	0.00	FAILED	rgb_lagged_sum	23	1000000	100	0.01	PASSED
sts_serial	9	100000	100	0.00	FAILED	rgb_lagged_sum	24	1000000	100	0.01	PASSED
sts_serial	9	100000	100	0.00	FAILED	rgb_lagged_sum	25	1000000	100	0.22	PASSED
sts_serial	10	100000	100	0.00	FAILED	rgb_lagged_sum	26	1000000	100	0.99	PASSED
sts_serial	10	100000	100	0.00	FAILED	rgb_lagged_sum	27	1000000	100	0.10	PASSED
sts_serial	11	100000	100	0.00	FAILED	rgb_lagged_sum	28	1000000	100	0.99	PASSED
sts_serial	11	100000	100	0.00	FAILED	rgb_lagged_sum	29	1000000	100	0.00	WEAK
sts_serial	12	100000	100	0.00	FAILED	rgb_lagged_sum	30	1000000	100	0.94	PASSED
sts_serial	12	100000	100	0.00	FAILED	rgb_lagged_sum	31	1000000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED	rgb_lagged_sum	32	1000000	100	0.27	PASSED
sts_serial	13	100000	100	0.00	FAILED	rgb_kstest_test	0	10000	1000	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED	dab_bytedistrib	0	51200000	1	1.00	WEAK
sts_serial	14	100000	100	0.00	FAILED	dab_dct	256	50000	1	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED	Preparing to run test 207. ntuple = 0					
sts_serial	15	100000	100	0.00	FAILED	dab_filltree	32	15000000	1	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED	dab_filltree	32	15000000	1	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED	Preparing to run test 208. ntuple = 0					
rgb_bitdist	1	100000	100	0.00	FAILED	dab_filltree2	0	5000000	1	0.00	FAILED
rgb_bitdist	2	100000	100	0.00	FAILED	dab_filltree2	1	5000000	1	0.00	FAILED
rgb_bitdist	3	100000	100	0.00	FAILED	Preparing to run test 209. ntuple = 0					
rgb_bitdist	4	100000	100	0.00	FAILED	dab_monobit2	12	65000000	1	1.00	FAILED

dieharder version 3.31.1 Copyright 2003 Robert G. Brown

Scrambled Sobol64

test_name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.00	FAILED
diehard_operm5	0	1000000	100	0.00	FAILED
diehard_rank_32x32	0	40000	100	0.00	FAILED
diehard_rank_6x8	0	100000	100	0.00	FAILED
diehard_bitstream	0	2097152	100	0.00	FAILED
diehard_opso	0	2097152	100	0.00	FAILED
diehard_oqso	0	2097152	100	0.00	FAILED
diehard_dna	0	2097152	100	0.00	FAILED
diehard_count_1s_str	0	256000	100	0.00	FAILED
diehard_count_1s_byt	0	256000	100	0.00	FAILED
diehard_parking_lot	0	12000	100	0.00	FAILED
diehard_2dsphere	2	8000	100	0.00	FAILED
diehard_3dsphere	3	4000	100	0.00	FAILED
diehard_squeeze	0	100000	100	0.00	FAILED
diehard_sums	0	100	100	0.00	FAILED
diehard_runs	0	100000	100	0.00	FAILED
diehard_runs	0	100000	100	0.00	FAILED
diehard_craps	0	200000	100	0.00	FAILED
diehard_craps	0	200000	100	0.00	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED
sts_monobit	1	100000	100	0.00	FAILED
sts_runs	2	100000	100	0.00	FAILED
sts_serial	1	100000	100	0.00	FAILED
sts_serial	2	100000	100	0.00	FAILED
sts_serial	3	100000	100	0.00	FAILED
sts_serial	3	100000	100	0.00	FAILED
sts_serial	4	100000	100	0.00	FAILED
sts_serial	4	100000	100	0.00	FAILED
sts_serial	5	100000	100	0.00	FAILED
sts_serial	5	100000	100	0.00	FAILED
sts_serial	6	100000	100	0.00	FAILED
sts_serial	6	100000	100	0.00	FAILED
sts_serial	7	100000	100	0.00	FAILED
sts_serial	7	100000	100	0.00	FAILED
sts_serial	8	100000	100	0.00	FAILED
sts_serial	8	100000	100	0.00	FAILED
sts_serial	9	100000	100	0.00	FAILED
sts_serial	9	100000	100	0.00	FAILED
sts_serial	10	100000	100	0.00	FAILED
sts_serial	10	100000	100	0.00	FAILED
sts_serial	11	100000	100	0.00	FAILED
sts_serial	11	100000	100	0.00	FAILED
sts_serial	12	100000	100	0.00	FAILED
sts_serial	12	100000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED
rgb_bitdist	1	100000	100	0.00	FAILED
rgb_bitdist	2	100000	100	0.00	FAILED
rgb_bitdist	3	100000	100	0.00	FAILED
rgb_bitdist	4	100000	100	0.00	FAILED
rgb_bitdist	5	100000	100	0.00	WEAK
rgb_bitdist	6	100000	100	0.00	WEAK
rgb_bitdist	7	100000	100	0.67	PASSED
rgb_bitdist	8	100000	100	0.00	FAILED
rgb_bitdist	9	100000	100	0.85	PASSED
rgb_bitdist	10	100000	100	0.69	PASSED
rgb_bitdist	11	100000	100	0.83	PASSED
rgb_bitdist	12	100000	100	0.25	PASSED
rgb_minimum_distance	2	10000	1000	0.00	FAILED
rgb_minimum_distance	3	10000	1000	0.00	FAILED
rgb_minimum_distance	4	10000	1000	0.00	FAILED
rgb_minimum_distance	5	10000	1000	0.00	FAILED
rgb_permutations	2	100000	100	0.00	FAILED
rgb_permutations	3	100000	100	0.00	FAILED
rgb_permutations	4	100000	100	0.00	FAILED
rgb_permutations	5	100000	100	0.00	FAILED
rgb_lagged_sum	0	1000000	100	0.00	FAILED
rgb_lagged_sum	1	1000000	100	0.00	FAILED
rgb_lagged_sum	2	1000000	100	0.21	PASSED
rgb_lagged_sum	3	1000000	100	0.00	WEAK
rgb_lagged_sum	4	1000000	100	0.00	WEAK
rgb_lagged_sum	5	1000000	100	0.74	PASSED
rgb_lagged_sum	6	1000000	100	0.53	PASSED
rgb_lagged_sum	7	1000000	100	0.00	WEAK
rgb_lagged_sum	8	1000000	100	0.72	PASSED
rgb_lagged_sum	9	1000000	100	0.00	WEAK
rgb_lagged_sum	10	1000000	100	0.37	PASSED
rgb_lagged_sum	11	1000000	100	0.03	PASSED
rgb_lagged_sum	12	1000000	100	0.28	PASSED
rgb_lagged_sum	13	1000000	100	0.77	PASSED
rgb_lagged_sum	14	1000000	100	0.02	PASSED
rgb_lagged_sum	15	1000000	100	0.02	PASSED
rgb_lagged_sum	16	1000000	100	0.63	PASSED
rgb_lagged_sum	17	1000000	100	0.50	PASSED
rgb_lagged_sum	18	1000000	100	0.85	PASSED
rgb_lagged_sum	19	1000000	100	0.08	PASSED
rgb_lagged_sum	20	1000000	100	0.95	PASSED
rgb_lagged_sum	21	1000000	100	0.06	PASSED
rgb_lagged_sum	22	1000000	100	0.98	PASSED
rgb_lagged_sum	23	1000000	100	0.01	PASSED
rgb_lagged_sum	24	1000000	100	0.01	PASSED
rgb_lagged_sum	25	1000000	100	0.22	PASSED
rgb_lagged_sum	26	1000000	100	0.99	PASSED
rgb_lagged_sum	27	1000000	100	0.10	PASSED
rgb_lagged_sum	28	1000000	100	0.99	PASSED
rgb_lagged_sum	29	1000000	100	0.00	WEAK
rgb_lagged_sum	30	1000000	100	0.94	PASSED
rgb_lagged_sum	31	1000000	100	0.00	FAILED
rgb_lagged_sum	32	1000000	100	0.27	PASSED
rgb_kstest_test	0	10000	1000	0.00	FAILED
dab_bytedistrib	0	51200000	1	1.00	WEAK
dab_dct	256	50000	1	0.00	FAILED
Preparing to run test 207. ntuple = 0					
dab_filltree	32	15000000	1	0.00	FAILED
dab_filltree	32	15000000	1	0.00	FAILED
Preparing to run test 208. ntuple = 0					
dab_filltree2	0	5000000	1	0.00	FAILED
dab_filltree2	1	5000000	1	0.00	FAILED
Preparing to run test 209. ntuple = 0					
dab_monobit2	12	65000000	1	1.00	FAILED

dieharder version 3.31.1 Copyright 2003 Robert G. Brown

Lcg 32

test_name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.85	PASSED
diehard_operm5	0	1000000	100	0.78	PASSED
diehard_rank_32x32	0	40000	100	0.35	PASSED
diehard_rank_6x8	0	100000	100	0.00	FAILED
diehard_bitstream	0	2097152	100	0.00	FAILED
diehard_opso	0	2097152	100	0.00	FAILED
diehard_oqso	0	2097152	100	0.00	FAILED
diehard_dna	0	2097152	100	0.00	FAILED
diehard_count_1s_str	0	256000	100	0.00	FAILED
diehard_count_1s_byt	0	256000	100	0.00	FAILED
diehard_parking_lot	0	12000	100	1.00	WEAK
diehard_2dsphere	2	8000	100	0.62	PASSED
diehard_3dsphere	3	4000	100	0.59	PASSED
diehard_squeeze	0	100000	100	0.60	PASSED
diehard_sums	0	100	100	0.41	PASSED
diehard_runs	0	100000	100	0.54	PASSED
diehard_runs	0	100000	100	0.39	PASSED
diehard_craps	0	200000	100	0.27	PASSED
diehard_craps	0	200000	100	0.14	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED
sts_monobit	1	100000	100	0.06	PASSED
sts_runs	2	100000	100	0.25	PASSED
sts_serial	1	100000	100	0.05	PASSED
sts_serial	2	100000	100	0.00	WEAK
sts_serial	3	100000	100	0.00	FAILED
sts_serial	3	100000	100	0.00	WEAK
sts_serial	4	100000	100	0.00	FAILED
sts_serial	4	100000	100	0.00	WEAK
sts_serial	5	100000	100	0.00	FAILED
sts_serial	5	100000	100	0.00	FAILED
sts_serial	6	100000	100	0.00	FAILED
sts_serial	6	100000	100	0.00	FAILED
sts_serial	7	100000	100	0.00	FAILED
sts_serial	7	100000	100	0.00	FAILED
sts_serial	8	100000	100	0.00	FAILED
sts_serial	8	100000	100	0.00	FAILED
sts_serial	9	100000	100	0.00	FAILED
sts_serial	9	100000	100	0.00	FAILED
sts_serial	10	100000	100	0.00	FAILED
sts_serial	10	100000	100	0.00	FAILED
sts_serial	11	100000	100	0.00	FAILED
sts_serial	11	100000	100	0.00	FAILED
sts_serial	12	100000	100	0.00	FAILED
sts_serial	12	100000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED
rgb_bitdist	1	100000	100	0.00	FAILED
rgb_bitdist	2	100000	100	0.00	FAILED
rgb_bitdist	3	100000	100	0.00	FAILED
rgb_bitdist	4	100000	100	0.00	FAILED
rgb_bitdist	5	100000	100	0.00	FAILED
rgb_bitdist	6	100000	100	0.00	FAILED
rgb_bitdist	7	100000	100	0.00	FAILED
rgb_bitdist	8	100000	100	0.00	FAILED
rgb_bitdist	9	100000	100	0.00	FAILED
rgb_bitdist	10	100000	100	0.00	WEAK
rgb_bitdist	11	100000	100	0.26	PASSED
rgb_bitdist	12	100000	100	0.18	PASSED
rgb_minimum_distance	2	10000	1000	0.00	FAILED
rgb_minimum_distance	3	10000	1000	0.00	WEAK
rgb_minimum_distance	4	10000	1000	0.00	FAILED
rgb_minimum_distance	5	10000	1000	0.00	FAILED
rgb_permutations	2	100000	100	0.63	PASSED
rgb_permutations	3	100000	100	0.41	PASSED
rgb_permutations	4	100000	100	0.91	PASSED
rgb_permutations	5	100000	100	0.96	PASSED
rgb_lagged_sum	0	1000000	100	0.71	PASSED
rgb_lagged_sum	1	1000000	100	0.99	PASSED
rgb_lagged_sum	2	1000000	100	0.57	PASSED
rgb_lagged_sum	3	1000000	100	0.41	PASSED
rgb_lagged_sum	4	1000000	100	0.85	PASSED
rgb_lagged_sum	5	1000000	100	0.73	PASSED
rgb_lagged_sum	6	1000000	100	0.96	PASSED
rgb_lagged_sum	7	1000000	100	0.96	PASSED
rgb_lagged_sum	8	1000000	100	0.65	PASSED
rgb_lagged_sum	9	1000000	100	0.54	PASSED
rgb_lagged_sum	10	1000000	100	0.33	PASSED
rgb_lagged_sum	11	1000000	100	1.00	WEAK
rgb_lagged_sum	12	1000000	100	0.70	PASSED
rgb_lagged_sum	13	1000000	100	1.00	WEAK
rgb_lagged_sum	14	1000000	100	0.54	PASSED
rgb_lagged_sum	15	1000000	100	0.95	PASSED
rgb_lagged_sum	16	1000000	100	0.64	PASSED
rgb_lagged_sum	17	1000000	100	0.79	PASSED
rgb_lagged_sum	18	1000000	100	0.45	PASSED
rgb_lagged_sum	19	1000000	100	0.74	PASSED
rgb_lagged_sum	20	1000000	100	0.68	PASSED
rgb_lagged_sum	21	1000000	100	0.45	PASSED
rgb_lagged_sum	22	1000000	100	1.00	WEAK
rgb_lagged_sum	23	1000000	100	0.81	PASSED
rgb_lagged_sum	24	1000000	100	0.97	PASSED
rgb_lagged_sum	25	1000000	100	0.46	PASSED
rgb_lagged_sum	26	1000000	100	0.32	PASSED
rgb_lagged_sum	27	1000000	100	0.87	PASSED
rgb_lagged_sum	28	1000000	100	0.23	PASSED
rgb_lagged_sum	29	1000000	100	0.60	PASSED
rgb_lagged_sum	30	1000000	100	0.61	PASSED
rgb_lagged_sum	31	1000000	100	0.36	PASSED
rgb_lagged_sum	32	1000000	100	0.11	PASSED
rgb_kstest_test	0	10000	1000	0.46	PASSED
dab_bytedistrib	0	51200000	1	1.00	FAILED
dab_dct	256	50000	1	0.00	FAILED
Preparing to run test 207. ntuple = 0					
dab_filltree	32	15000000	1	0.57	PASSED
dab_filltree	32	15000000	1	0.95	PASSED
Preparing to run test 208. ntuple = 0					
dab_filltree2	0	5000000	1	0.00	FAILED
dab_filltree2	1	5000000	1	0.00	FAILED
Preparing to run test 209. ntuple = 0					

dieharder version 3.31.1 Copyright 2003 Robert G. Brown

Lcg 64

test_name	ntup	tsamples	psamples	p-val	Assessment
diehard_birthdays	0	100	100	0.98	PASSED
diehard_operm5	0	1000000	100	0.02	PASSED
diehard_rank_32x32	0	40000	100	0.55	PASSED
diehard_rank_6x8	0	100000	100	0.00	FAILED
diehard_bitstream	0	2097152	100	0.00	FAILED
diehard_opso	0	2097152	100	0.00	FAILED
diehard_oqso	0	2097152	100	0.00	FAILED
diehard_dna	0	2097152	100	0.00	FAILED
diehard_count_1s_str	0	256000	100	0.00	FAILED
diehard_count_1s_byt	0	256000	100	0.00	FAILED
diehard_parking_lot	0	12000	100	0.99	PASSED
diehard_2dsphere	2	8000	100	0.67	PASSED
diehard_3dsphere	3	4000	100	0.35	PASSED
diehard_squeeze	0	100000	100	0.49	PASSED
diehard_sums	0	100	100	0.00	WEAK
diehard_runs	0	100000	100	0.09	PASSED
diehard_runs	0	100000	100	0.70	PASSED
diehard_craps	0	200000	100	0.05	PASSED
diehard_craps	0	200000	100	0.42	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED
sts_monobit	1	100000	100	0.37	PASSED
sts_runs	2	100000	100	0.47	PASSED
sts_serial	1	100000	100	0.00	WEAK
sts_serial	2	100000	100	0.00	FAILED
sts_serial	3	100000	100	0.00	FAILED
sts_serial	3	100000	100	0.00	FAILED
sts_serial	4	100000	100	0.00	FAILED
sts_serial	4	100000	100	0.00	FAILED
sts_serial	5	100000	100	0.00	FAILED
sts_serial	5	100000	100	0.00	FAILED
sts_serial	6	100000	100	0.00	FAILED
sts_serial	6	100000	100	0.00	FAILED
sts_serial	7	100000	100	0.00	FAILED
sts_serial	7	100000	100	0.00	FAILED
sts_serial	8	100000	100	0.00	FAILED
sts_serial	8	100000	100	0.00	FAILED
sts_serial	9	100000	100	0.00	FAILED
sts_serial	9	100000	100	0.00	FAILED
sts_serial	10	100000	100	0.00	FAILED
sts_serial	10	100000	100	0.00	FAILED
sts_serial	11	100000	100	0.00	FAILED
sts_serial	11	100000	100	0.00	FAILED
sts_serial	12	100000	100	0.00	FAILED
sts_serial	12	100000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED
rgb_bitdist	1	100000	100	0.00	FAILED
rgb_bitdist	2	100000	100	0.00	FAILED
rgb_bitdist	3	100000	100	0.00	FAILED
rgb_bitdist	4	100000	100	0.00	FAILED
rgb_bitdist	5	100000	100	0.00	FAILED
rgb_bitdist	6	100000	100	0.00	FAILED
rgb_bitdist	7	100000	100	0.00	FAILED
rgb_bitdist	8	100000	100	0.00	FAILED
rgb_bitdist	9	100000	100	0.00	WEAK
rgb_bitdist	10	100000	100	0.07	PASSED
rgb_bitdist	11	100000	100	0.63	PASSED
rgb_bitdist	12	100000	100	0.20	PASSED
rgb_minimum_distance	2	10000	1000	0.00	WEAK
rgb_minimum_distance	3	10000	1000	0.00	WEAK
rgb_minimum_distance	4	10000	1000	0.00	FAILED
rgb_minimum_distance	5	10000	1000	0.00	FAILED
rgb_permutations	2	100000	100	0.73	PASSED
rgb_permutations	3	100000	100	0.04	PASSED
rgb_permutations	4	100000	100	0.17	PASSED
rgb_permutations	5	100000	100	0.60	PASSED
rgb_lagged_sum	0	1000000	100	0.87	PASSED
rgb_lagged_sum	1	1000000	100	0.21	PASSED
rgb_lagged_sum	2	1000000	100	0.12	PASSED
rgb_lagged_sum	3	1000000	100	0.93	PASSED
rgb_lagged_sum	4	1000000	100	0.82	PASSED
rgb_lagged_sum	5	1000000	100	1.00	WEAK
rgb_lagged_sum	6	1000000	100	0.40	PASSED
rgb_lagged_sum	7	1000000	100	0.32	PASSED
rgb_lagged_sum	8	1000000	100	0.99	PASSED
rgb_lagged_sum	9	1000000	100	0.52	PASSED
rgb_lagged_sum	10	1000000	100	0.20	PASSED
rgb_lagged_sum	11	1000000	100	0.68	PASSED
rgb_lagged_sum	12	1000000	100	0.90	PASSED
rgb_lagged_sum	13	1000000	100	0.87	PASSED
rgb_lagged_sum	14	1000000	100	0.56	PASSED
rgb_lagged_sum	15	1000000	100	0.35	PASSED
rgb_lagged_sum	16	1000000	100	0.43	PASSED
rgb_lagged_sum	17	1000000	100	0.12	PASSED
rgb_lagged_sum	18	1000000	100	0.34	PASSED
rgb_lagged_sum	19	1000000	100	0.62	PASSED
rgb_lagged_sum	20	1000000	100	0.08	PASSED
rgb_lagged_sum	21	1000000	100	0.63	PASSED
rgb_lagged_sum	22	1000000	100	1.00	WEAK
rgb_lagged_sum	23	1000000	100	0.40	PASSED
rgb_lagged_sum	24	1000000	100	0.66	PASSED
rgb_lagged_sum	25	1000000	100	0.06	PASSED
rgb_lagged_sum	26	1000000	100	0.76	PASSED
rgb_lagged_sum	27	1000000	100	0.75	PASSED
rgb_lagged_sum	28	1000000	100	0.56	PASSED
rgb_lagged_sum	29	1000000	100	0.03	PASSED
rgb_lagged_sum	30	1000000	100	0.59	PASSED
rgb_lagged_sum	31	1000000	100	0.50	PASSED
rgb_lagged_sum	32	1000000	100	0.41	PASSED
rgb_kstest_test	0	10000	1000	0.87	PASSED
dab_bytedistrib	0	51200000	1	1.00	FAILED
dab_dct	256	50000	1	0.00	FAILED
Preparing to run test 207. ntuple = 0					
dab_filltree	32	15000000	1	0.37	PASSED
dab_filltree	32	15000000	1	0.12	PASSED
Preparing to run test 208. ntuple = 0					
dab_filltree2	0	5000000	1	0.00	FAILED
dab_filltree2	1	5000000	1	0.00	FAILED
Preparing to run test 209. ntuple = 0					
dab_monobit2	12	65000000	1	1.00	FAILED

dieharder version 3.31.1 Copyright 2003 Robert G. Brown

Lcg 128

test_name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.75	PASSED
diehard_operm5	0	1000000	100	0.01	PASSED
diehard_rank_32x32	0	40000	100	0.00	FAILED
diehard_rank_6x8	0	100000	100	0.00	FAILED
diehard_bitstream	0	2097152	100	0.00	FAILED
diehard_opso	0	2097152	100	0.00	FAILED
diehard_oqso	0	2097152	100	0.00	FAILED
diehard_dna	0	2097152	100	0.00	FAILED
diehard_count_1s_str	0	256000	100	0.00	FAILED
diehard_count_1s_byt	0	256000	100	0.00	FAILED
diehard_parking_lot	0	12000	100	0.52	PASSED
diehard_2dsphere	2	8000	100	0.01	PASSED
diehard_3dsphere	3	4000	100	0.54	PASSED
diehard_squeeze	0	100000	100	0.64	PASSED
diehard_sums	0	100	100	0.11	PASSED
diehard_runs	0	100000	100	0.36	PASSED
diehard_runs	0	100000	100	0.90	PASSED
diehard_craps	0	200000	100	0.93	PASSED
diehard_craps	0	200000	100	0.19	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED
marsaglia_tsang_gcd	0	10000000	100	0.00	FAILED
sts_monobit	1	100000	100	0.08	PASSED
sts_runs	2	100000	100	0.00	FAILED
sts_serial	1	100000	100	0.08	PASSED
sts_serial	2	100000	100	0.00	FAILED
sts_serial	3	100000	100	0.00	FAILED
sts_serial	3	100000	100	0.00	FAILED
sts_serial	4	100000	100	0.00	FAILED
sts_serial	4	100000	100	0.00	FAILED
sts_serial	5	100000	100	0.00	FAILED
sts_serial	5	100000	100	0.00	FAILED
sts_serial	6	100000	100	0.00	FAILED
sts_serial	6	100000	100	0.00	FAILED
sts_serial	7	100000	100	0.00	FAILED
sts_serial	7	100000	100	0.00	FAILED
sts_serial	8	100000	100	0.00	FAILED
sts_serial	8	100000	100	0.00	FAILED
sts_serial	9	100000	100	0.00	FAILED
sts_serial	9	100000	100	0.00	FAILED
sts_serial	10	100000	100	0.00	FAILED
sts_serial	10	100000	100	0.00	FAILED
sts_serial	11	100000	100	0.00	FAILED
sts_serial	11	100000	100	0.00	FAILED
sts_serial	12	100000	100	0.00	FAILED
sts_serial	12	100000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED
sts_serial	13	100000	100	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED
sts_serial	14	100000	100	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED
sts_serial	15	100000	100	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED
sts_serial	16	100000	100	0.00	FAILED
rgb_bitdist	1	100000	100	0.00	FAILED
rgb_bitdist	2	100000	100	0.00	FAILED
rgb_bitdist	3	100000	100	0.00	FAILED
rgb_bitdist	4	100000	100	0.00	FAILED
rgb_bitdist	5	100000	100	0.00	FAILED
rgb_bitdist	6	100000	100	0.00	FAILED
rgb_bitdist	7	100000	100	0.00	FAILED
rgb_bitdist	8	100000	100	0.00	FAILED
rgb_bitdist	9	100000	100	0.00	FAILED
rgb_bitdist	10	100000	100	0.00	FAILED
rgb_bitdist	11	100000	100	0.00	FAILED
rgb_bitdist	12	100000	100	0.00	FAILED
rgb_minimum_distance	2	10000	1000	0.00	FAILED
rgb_minimum_distance	3	10000	1000	0.00	FAILED
rgb_minimum_distance	4	10000	1000	0.00	FAILED
rgb_minimum_distance	5	10000	1000	0.00	FAILED
rgb_permutations	2	100000	100	0.87	PASSED
rgb_permutations	3	100000	100	0.96	PASSED
rgb_permutations	4	100000	100	1.00	WEAK
rgb_permutations	5	100000	100	0.95	PASSED
rgb_lagged_sum	0	1000000	100	0.56	PASSED
rgb_lagged_sum	1	1000000	100	0.55	PASSED
rgb_lagged_sum	2	1000000	100	0.19	PASSED
rgb_lagged_sum	3	1000000	100	0.18	PASSED
rgb_lagged_sum	4	1000000	100	0.44	PASSED
rgb_lagged_sum	5	1000000	100	0.93	PASSED
rgb_lagged_sum	6	1000000	100	0.01	PASSED
rgb_lagged_sum	7	1000000	100	0.99	PASSED
rgb_lagged_sum	8	1000000	100	0.71	PASSED
rgb_lagged_sum	9	1000000	100	0.18	PASSED
rgb_lagged_sum	10	1000000	100	0.09	PASSED
rgb_lagged_sum	11	1000000	100	0.08	PASSED
rgb_lagged_sum	12	1000000	100	0.41	PASSED
rgb_lagged_sum	13	1000000	100	0.24	PASSED
rgb_lagged_sum	14	1000000	100	0.93	PASSED
rgb_lagged_sum	15	1000000	100	0.34	PASSED
rgb_lagged_sum	16	1000000	100	0.98	PASSED
rgb_lagged_sum	17	1000000	100	0.90	PASSED
rgb_lagged_sum	18	1000000	100	0.32	PASSED
rgb_lagged_sum	19	1000000	100	0.64	PASSED
rgb_lagged_sum	20	1000000	100	0.10	PASSED
rgb_lagged_sum	21	1000000	100	0.34	PASSED
rgb_lagged_sum	22	1000000	100	0.85	PASSED
rgb_lagged_sum	23	1000000	100	0.99	PASSED
rgb_lagged_sum	24	1000000	100	0.13	PASSED
rgb_lagged_sum	25	1000000	100	0.33	PASSED
rgb_lagged_sum	26	1000000	100	0.88	PASSED
rgb_lagged_sum	27	1000000	100	0.99	PASSED
rgb_lagged_sum	28	1000000	100	0.09	PASSED
rgb_lagged_sum	29	1000000	100	0.67	PASSED
rgb_lagged_sum	30	1000000	100	0.48	PASSED
rgb_lagged_sum	31	1000000	100	0.00	WEAK
rgb_lagged_sum	32	1000000	100	0.92	PASSED
rgb_kstest_test	0	10000	1000	0.63	PASSED
dab_bytedistrib	0	51200000	1	0.00	FAILED
dab_dct	256	50000	1	0.00	FAILED
Preparing to run test 207. ntuple = 0					
dab_filltree	32	15000000	1	0.45	PASSED
dab_filltree	32	15000000	1	0.21	PASSED
Preparing to run test 208. ntuple = 0					
dab_filltree2	0	5000000	1	0.00	FAILED
dab_filltree2	1	5000000	1	0.00	FAILED
Preparing to run test 209. ntuple = 0					
dab_monobit2	12	65000000	1	1.00	FAILED

dieharder version 3.31.1 Copyright 2003 Robert G. Brown						rgb_bitdist	5	100000	100	0.55	PASSED
XorShift						rgb_bitdist	6	100000	100	0.14	PASSED
test_name	ntup	tsamples	psamples	p-value	Assessment	rgb_bitdist	7	100000	100	0.70	PASSED
diehard_birthdays	0	100	100	0.80	PASSED	rgb_bitdist	8	100000	100	0.58	PASSED
diehard_operm5	0	1000000	100	0.95	PASSED	rgb_bitdist	9	100000	100	0.26	PASSED
diehard_rank_32x32	0	40000	100	0.89	PASSED	rgb_bitdist	10	100000	100	0.26	PASSED
diehard_rank_6x8	0	100000	100	0.50	PASSED	rgb_bitdist	11	100000	100	0.61	PASSED
diehard_bitstream	0	2097152	100	0.06	PASSED	rgb_bitdist	12	100000	100	0.56	PASSED
diehard_opso	0	2097152	100	0.63	PASSED	rgb_minimum_distance	2	10000	1000	0.52	PASSED
diehard_oqso	0	2097152	100	0.96	PASSED	rgb_minimum_distance	3	10000	1000	1.00	WEAK
diehard_dna	0	2097152	100	0.72	PASSED	rgb_minimum_distance	4	10000	1000	0.36	PASSED
diehard_count_1s_str	0	256000	100	0.43	PASSED	rgb_minimum_distance	5	10000	1000	0.35	PASSED
diehard_count_1s_byt	0	256000	100	0.29	PASSED	rgb_permutations	2	100000	100	0.97	PASSED
diehard_parking_lot	0	12000	100	0.15	PASSED	rgb_permutations	3	100000	100	0.51	PASSED
diehard_2dsphere	2	8000	100	0.84	PASSED	rgb_permutations	4	100000	100	0.29	PASSED
diehard_3dsphere	3	4000	100	0.68	PASSED	rgb_permutations	5	100000	100	0.00	WEAK
diehard_squeeze	0	100000	100	0.82	PASSED	rgb_lagged_sum	0	1000000	100	0.46	PASSED
diehard_sums	0	100	100	0.35	PASSED	rgb_lagged_sum	1	1000000	100	0.93	PASSED
diehard_runs	0	100000	100	1.00	WEAK	rgb_lagged_sum	2	1000000	100	0.84	PASSED
diehard_runs	0	100000	100	0.65	PASSED	rgb_lagged_sum	3	1000000	100	0.15	PASSED
diehard_craps	0	200000	100	0.91	PASSED	rgb_lagged_sum	4	1000000	100	0.08	PASSED
diehard_craps	0	200000	100	0.99	PASSED	rgb_lagged_sum	5	1000000	100	0.58	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.22	PASSED	rgb_lagged_sum	6	1000000	100	0.52	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.72	PASSED	rgb_lagged_sum	7	1000000	100	0.97	PASSED
sts_monobit	1	100000	100	0.27	PASSED	rgb_lagged_sum	8	1000000	100	0.86	PASSED
sts_runs	2	100000	100	0.47	PASSED	rgb_lagged_sum	9	1000000	100	0.68	PASSED
sts_serial	1	100000	100	0.64	PASSED	rgb_lagged_sum	10	1000000	100	0.52	PASSED
sts_serial	2	100000	100	0.19	PASSED	rgb_lagged_sum	11	1000000	100	0.66	PASSED
sts_serial	3	100000	100	0.16	PASSED	rgb_lagged_sum	12	1000000	100	0.29	PASSED
sts_serial	3	100000	100	0.58	PASSED	rgb_lagged_sum	13	1000000	100	0.80	PASSED
sts_serial	4	100000	100	0.90	PASSED	rgb_lagged_sum	14	1000000	100	0.18	PASSED
sts_serial	4	100000	100	0.75	PASSED	rgb_lagged_sum	15	1000000	100	1.00	WEAK
sts_serial	5	100000	100	0.97	PASSED	rgb_lagged_sum	16	1000000	100	0.99	PASSED
sts_serial	5	100000	100	0.22	PASSED	rgb_lagged_sum	17	1000000	100	0.50	PASSED
sts_serial	6	100000	100	0.32	PASSED	rgb_lagged_sum	18	1000000	100	0.24	PASSED
sts_serial	6	100000	100	0.47	PASSED	rgb_lagged_sum	19	1000000	100	0.21	PASSED
sts_serial	7	100000	100	0.95	PASSED	rgb_lagged_sum	20	1000000	100	0.32	PASSED
sts_serial	7	100000	100	0.39	PASSED	rgb_lagged_sum	21	1000000	100	0.45	PASSED
sts_serial	8	100000	100	0.82	PASSED	rgb_lagged_sum	22	1000000	100	0.12	PASSED
sts_serial	8	100000	100	0.45	PASSED	rgb_lagged_sum	23	1000000	100	0.14	PASSED
sts_serial	9	100000	100	0.21	PASSED	rgb_lagged_sum	24	1000000	100	0.91	PASSED
sts_serial	9	100000	100	0.68	PASSED	rgb_lagged_sum	25	1000000	100	0.55	PASSED
sts_serial	10	100000	100	0.76	PASSED	rgb_lagged_sum	26	1000000	100	0.26	PASSED
sts_serial	10	100000	100	0.56	PASSED	rgb_lagged_sum	27	1000000	100	0.89	PASSED
sts_serial	11	100000	100	0.18	PASSED	rgb_lagged_sum	28	1000000	100	0.89	PASSED
sts_serial	11	100000	100	0.20	PASSED	rgb_lagged_sum	29	1000000	100	0.24	PASSED
sts_serial	12	100000	100	0.10	PASSED	rgb_lagged_sum	30	1000000	100	0.80	PASSED
sts_serial	12	100000	100	0.42	PASSED	rgb_lagged_sum	31	1000000	100	0.08	PASSED
sts_serial	13	100000	100	0.21	PASSED	rgb_lagged_sum	32	1000000	100	0.59	PASSED
sts_serial	13	100000	100	0.86	PASSED	rgb_kstest_test	0	10000	1000	0.15	PASSED
sts_serial	14	100000	100	0.59	PASSED	dab_bytedistrib	0	51200000	1	0.27	PASSED
sts_serial	14	100000	100	0.56	PASSED	dab_dct	256	50000	1	0.01	PASSED
sts_serial	15	100000	100	0.58	PASSED	Preparing to run test 207. ntuple = 0					
sts_serial	15	100000	100	0.70	PASSED	dab_filltree	32	15000000	1	0.17	PASSED
sts_serial	16	100000	100	0.80	PASSED	dab_filltree	32	15000000	1	0.18	PASSED
sts_serial	16	100000	100	0.38	PASSED	Preparing to run test 208. ntuple = 0					
rgb_bitdist	1	100000	100	0.22	PASSED	dab_filltree2	0	5000000	1	0.09	PASSED
rgb_bitdist	2	100000	100	0.42	PASSED	dab_filltree2	1	5000000	1	0.81	PASSED
rgb_bitdist	3	100000	100	0.65	PASSED	Preparing to run test 209. ntuple = 0					
rgb_bitdist	4	100000	100	0.87	PASSED	dab_monobit2	12	65000000	1	0.09	PASSED

LegkisebbKonvexPoli() C kódja a hozzű tartozó függvénnyel:

```
bool isPointLeftOfRayCPU(float x, float y, float raySx, float raySy, float
rayEx, float rayEy) {

  float theCos, theSin, dist ;
  //  Eltoljuk a koordináta rendszert a vektor kezdőpontjába
  rayEx-=raySx; rayEy-=raySy;
  x    -=raySx; y    -=raySy;
  //  Megállapítjuk a vektor hosszát
  dist=sqrt(rayEx*rayEx+rayEy*rayEy);
  if(dist == 0) return false;
  // Elforgatjuk a koordináta rendszert
  theCos=rayEx/dist;
  theSin=rayEy/dist;
  y=y*theCos-x*theSin;
  // Ha az új rendszerben y tengely fölött van az új y koordináta akkor a
vektortól balra van a pont
  return y>0.0f;
}

int findSmallestPolygonCPU(float3* S, float3 *P, int N) {
   float3 endPoint, pointOnHull;
   pointOnHull = S[0];
   for (int i=0; i<N; i++) if (S[i].x<pointOnHull.x || S[i].x==pointOnHull.x &&
S[i].z<pointOnHull.z) {
    pointOnHull = S[i];
   }
   float x, y, x1, y1;
   int polyCorners = 0;
   do{
      P[polyCorners] = pointOnHull;
      endPoint = S[0];
      for(int j=1; j<N;j++){
         x = S[j].x; y = S[j].z; x1 = endPoint.x; y1 = endPoint.z;
         if ((endPoint.x == pointOnHull.x && endPoint.z == pointOnHull.z) ||
isPointLeftOfRayCPU(x, y, P[polyCorners].x, P[polyCorners].z, x1, y1) ){
            endPoint = S[j];
          }
      }
      polyCorners ++;
      pointOnHull = endPoint;
   }while(( endPoint.x!=P[0].x || endPoint.z!=P[0].z));
   return polyCorners;
}
```