Ferromágneses Ising-modell mágnesezettségének hőmérsékletfüggése köbös rácson Swendsen-Wang algoritmusával (H = 0)

Trencséni Márton, ELTE fizikus, VI. évf.
2008. jan. 10.

1. Bevezetés

A programot C++-ban implementáltam Windows alá Visual Studio alatt, így a fordításhoz utóbbi szükséges. A kód struktúrája klasszikus C++, azaz a deklarációk a .hpp fileokban, a definíciók a .cpp fileokban találhatók.

2. Forráskód magyarázat

2.1 Support osztályok

A kód jónéhány segítő osztályt tartalmaz, amelyek elabsztrahálják az alacsony szintű műveleteket a "fizikát" tartalamazó rész elől. Ezek:

Field.hpp és Field.cpp

Field<T> - ez az osztály egy dinamikusan allokált, T tipusú elemeket tartalmazó tömböt absztrahál. Ennek megfelelően a konstruktorban adjuk meg, hogy hány elemre van szükségünk. A Randomize() függvény randomizálja, a Clear() függvény kinulláza az egész tömböt. A Get() és Set() függvényekkel kérdezzük le ill. állítjuk be az elemeket.

BitField – egy bitmezőt absztrahál. Azért nem a fenti template osztályt használom erre, mert nincs beépített 1 bites adattípus a C++-ban (a boolean általában 8 bites), így bit szintű shift műveleteket használok íráshoz és olvasáshoz. Ugyanazokat a műveleteket támogatja, mint a Field<T> osztály.

Lattice.hpp és Lattice.cpp

Lattice<T> - egy 3D-s "köbös" rácsot absztrahál, ahol a rácspontokban T tipusú elemek ülnek. Belül Field<T> osztályt használ tárolásra, és ugyanazokat a függvényeket támogatja. A hozzáadott érték annyi, hogy a Get() és Set() függvények x, y és z koordinátákat vesznek át, amiből kiszámolják a megfelelő 1D indexet.

BitLattice – a BitField osztályt használó 3D-s köbös rács, a fentieknek megfelelően.

Random.hpp és Random.cpp

Random – Sokféle véletlen szám generátor létezik, az általam választott konkrét implementációt absztrahálja a Random osztály. A stat. fiz. szimulációkban kitüntetett szerepe van a véletlen számok generálásának, az eredmények helyessége függ tőle. Viszonylag gyorsan, osztásokkal lehet pszeudo-random számokat generálni; ha erőssebb véletlen számok kellenek, az futási időben drágább. A Win32 platformon a Microsoft a Wincrypt API használatát ajánlja véletlen számok generálásához. Nevének megfelelően ez titkosításhoz is elég erős véletlen számokat generál, így az ajánlásnak megfelelően *először* ezt használtam. A Wincrypt érzésem szerint túl erős ehhez a szimulációhoz, azaz gyengébb de gyorsabb generátor is elég lenne, ezért áttértem az M. Matsumoto & T. Nishimura által javasolt "Mersenne Twister" verzióra (ld. M. Matsumoto & T. Nishimura, *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, 1998, pp. 3-30). A mersenne-féle véletlen szám generátort nem implementáltam, hanem egy standard implementációt használtam, melyet a http://www.agner.org/random/ weblapról lehet letölteni. Méréseim szerint a Wincrypt API-nál a mersenne durván 10-szer gyorsabb, és ezen a problémán nem mutatkozik a véletlen számok gyengesége.

A Random osztály tartalmaz egy GetReal() függvényt, amely visszaad egy 0 és 1 közötti valós számot, a GetBit() egy véletlen bitet, a GetBit(real probability) egy <probability> valószínűséggel 1 bitet, a Randomize() egy buffert tölt ki véletlenen byteokkal.

Timer.hpp és Timer.cpp, TimeSpan.hpp és TimeSpan.cpp

Timer – ezt az osztályt a profiling során használtam időmérésre. A Start() függvény elindítja az időmérést, az End() megállítja. A Start()-ot újra meghívva tovább *folytatja* a mérést. A msec felbontású GetTickCount() Win32 függvényt használom belül.

TimeSpan – belül ezt használja a Timer osztály, hogy az eltelt msec időből kinyomtatható óra + perc + másodperc + msec adatot csináljon.

2.2 Fizikát tartalmazó osztályok

Az egyetlen fizikát tartalmazó osztály:

Swendsen.hpp és Swendsen.cpp

Swendsen – ez a program legnagyobb osztálya. Kívűlről ugyanakkor ez a legkisebb, csak három publikus metódusa van. A Swendsen() konstruktornak adjuk át, hogy mekkora rácsot szeretnénk szimulálni. Az Init() függvényt kell a szimuláció elején meghívni, és beadni a beta hőmérséklet paramétert és J csatolási állandót. A szimuláció léptetését a Step() függvénnyel végezzük. A Step() után a Magnetization() függvény adja vissza a rács makroszkópikus mágnesezettségét.

A használt member adatstruktúrák:
BitLattice sites – a spinek

BitLattice xbonds, ybonds, zbonds – mivel 3D-ben vagyunk ezért 3x annyi bond van a spin siteok között, mint spin site.
Lattice<int> memberships – melyik spin melyik clusterbe tartozik.
Field<int> clusters – clusterek lánca; itt tudunk két clustert linkelni. Amennyiben az i-edik elem értéke i, a cluster nincs láncolva. Amennyiben az i-edik elem értéke j, az i-edik és j-edik cluster egy clusternek tekintendő.

Az Init() függvény a sites spinrácsot randomizálja. A Step() először meghívja a RandomizeBonds()-ot, utána a Cluster()-t és végül a FlipClusters() függvényt. *Ez maga a Swendsen-Wang algoritmus!*

A RandomizeBonds() minden egyes bond-ra (x, y és z irányú) megnézi, hogy a két spin szomszédja egy irányban áll-e, és ha igen, akkor *1 – exp(-2 * beta * J)* valószínűséggel beállítja a bondot.

A Cluster() függvény implementálja a *Hoshen-Kopelman* cluster algoritmust. Ez a függvény a legrondább az egész kódban, mivel meglátásom szerint a HK algoritmus rengeteg esetkezelést igényel: az eljárás lényege, hogy egy írógép fejét szimulálva bejárjuk a rácsot, és megnézzük, hogy az x, y és z irányban bejárás szerint visszafele lévő bondok be vannak-e állítva. Ha igen, akkor a clustereket linkeljük. A probléma ezzel, hogy a rács széleinél nem szabad visszanézni, így attól függően, hogy hány dimenziós a probléma, rengeteg if-et kell tartalmazzon a kód. Alternatív lehetne n-dimenziós kódot írni, de az is elég bonyolult lenne.

Amikor két cluster-t össze kell linkelni, LinkCluster() függvényt hívjuk meg. Ez rekurzívan végigjárja a clusters láncot, ameddig meg nem találja a lánc végét, és odaragasztja az új clustert.

A Step() utolsó lépése a FlipClusters() függvény. Ez a clusterek linkelését figyelembe véve clusterenként az összes spint együtt 0.5 valószínűséggel megfordítja.

2.3 Driver

Main.c

A program main.c file-ja egy egyszerű driver. Argumentumként átveszi, hogy
- abszolút átlagot számoljon-e
- milyen nevű CSV fileba írja a beta, magnetization adatokat
- mekkora ráccsal dolgozzon (L)
- hány lépést végezzen a szimuláció minden egyes beta értékével
- milyen béta értékkel kezdjen
- milyen béta értékig menjen fel
- mekkorákat ugorjon bétaval
- mekkora legyen J, a csatolási állandó

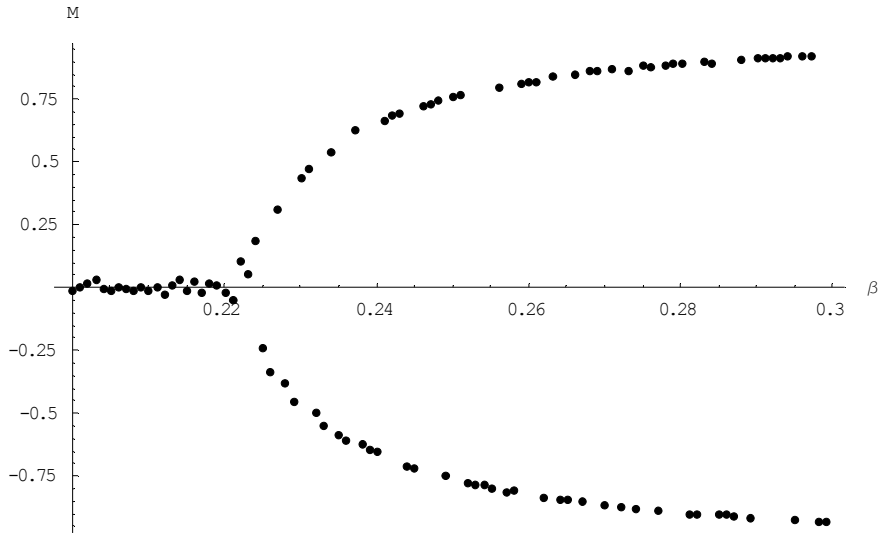
Azaz

```
SwendsenWang.exe <absAvg> <outFile.csv> <length> <nSteps> <betaFrom> <betaTo> <betaIncr> <J>
```

Az első paraméter némi magyarázatra szorul: ha “absAvg”, akkor a program minden egyes lépés után kiolvassa a makroszkópikus mágnesezettséget, annak az abszolút értékét veszi, és a lépések során átlagol. Így természetesen nem kapjuk vissza a megszokott “kétágú villa” plotot, de az átlagolás miatt jobb eredményt kapunk. Ha “noAbsAvg”-t adunk be, akkor az utolsó lépés után előálló mágnesezettséget veszi a program, és megkapjuk a “kétágú villa” plotot. Egy beta futás után az <outFile.csv> file-ba beírja a „beta, átlagos mágnesezettség” adatot mint egy sort.

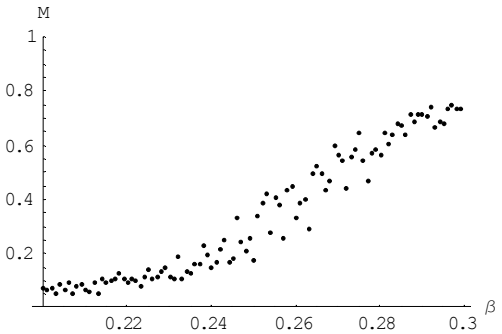
3. Szimulációs eredmények

A szimuláció szépen mutatja a ferromágneses viselkedést. A β paramétert növelve elérkezünk a kritikus β „inverz-hőmérséklethez”, ahol spontán szimmetriasértés következik be.

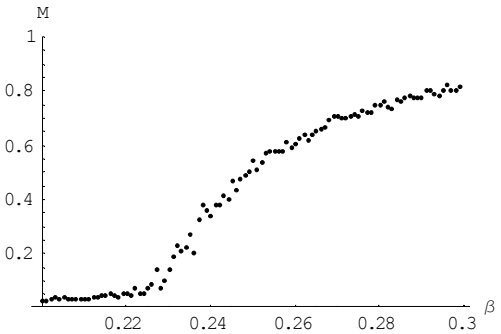


1. ábra: a ferromágneses Ising modell L = 50 rács esetén 50 lépéssel (J = 1)

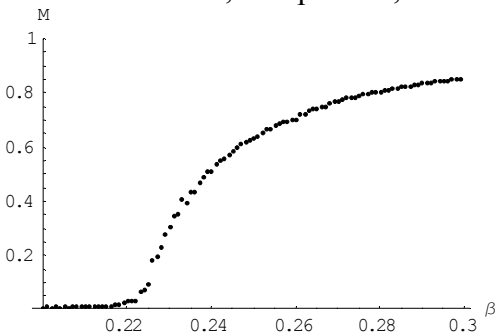
Jól látható, hogy a kritikus β 0.22 körül helyezkedik el. Ahhoz, hogy ertékét pontosan meghatározzam, különböző (L, nSteps) kombinációkkal végeztem méréseket.



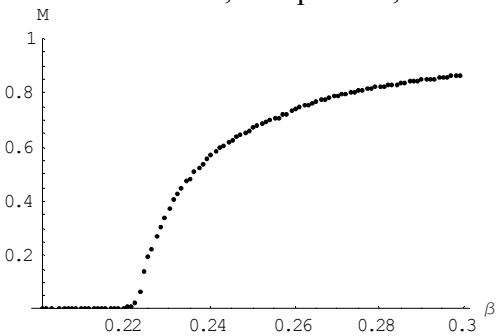
2. ábra: L = 10, nSteps = 20, J = 1



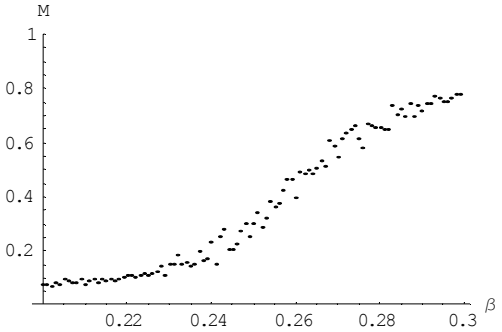
3. ábra: L = 20, nSteps = 20, J = 1



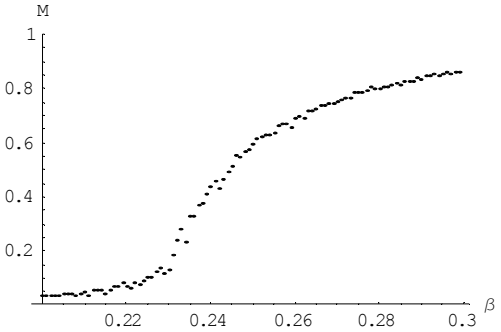
4. ábra: L = 50, nSteps = 20, J = 1



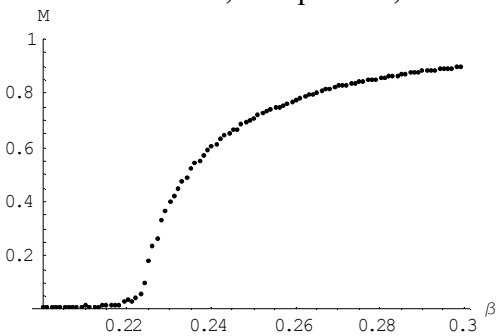
5. ábra: L = 100, nSteps = 20, J = 1



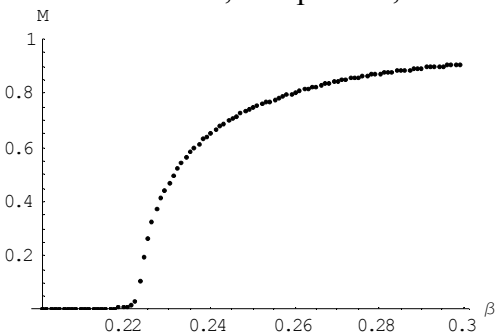
6. ábra: L = 10, nSteps = 50, J = 1



7. ábra: L = 20, nSteps = 50, J = 1

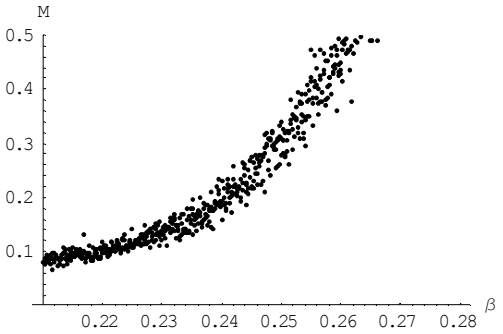


8. ábra: L = 50, nSteps = 100, J = 1

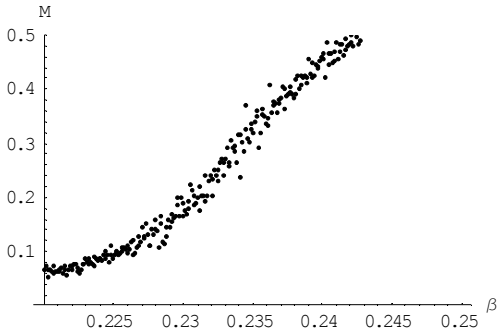


9. ábra: L = 100, nSteps = 50, J = 1

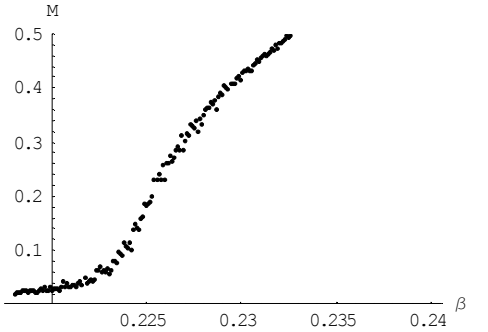
Hogy jobban lehessen látni a mágnesezettség viselkedését a kritikus β körül, nagyfelbontású szimulációkat is végeztem:



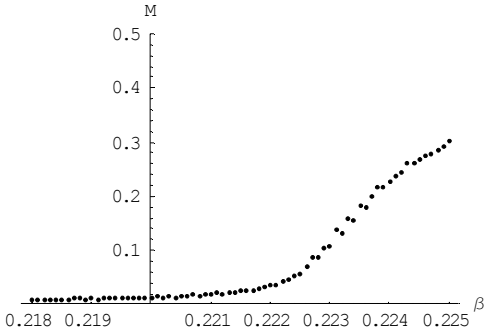
10. ábra: L = 10, nSteps = 100, J = 1



11. ábra: L = 20, nSteps = 100, J = 1



12. ábra: L = 50, nSteps = 100, J = 1

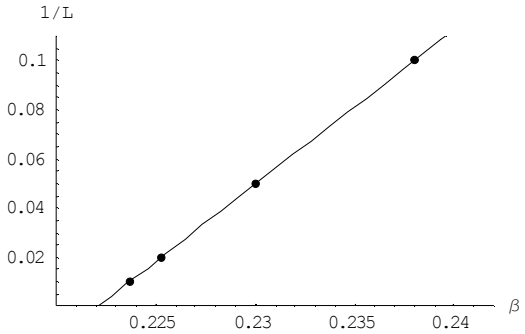


13. ábra: L = 100, nSteps = 100, J = 1

Az ábrákon szépen látszik a *finite size scaling* effektus, azaz, hogy β_{crit} értéke függ L-től. Ahhoz, hogy a végtelen limeszt meghatározzuk, először rögzíteni kell, hogy hogyan határozzuk meg β_{crit} –et. Erre a konvenció itt: ahol M = 0.2. A fenti ábrák alapján:

L	**β_{crit}**
10	0.238
20	0.230
50	0.2253
100	0.2237

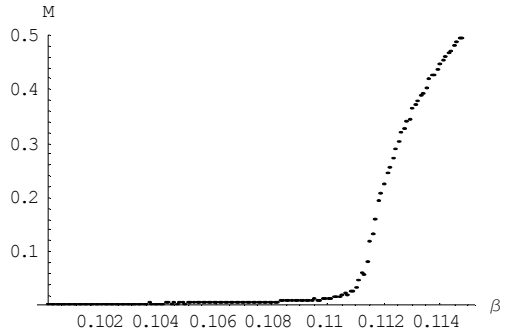
A β_{crit} – 1/L összefüggést ábrázolva:



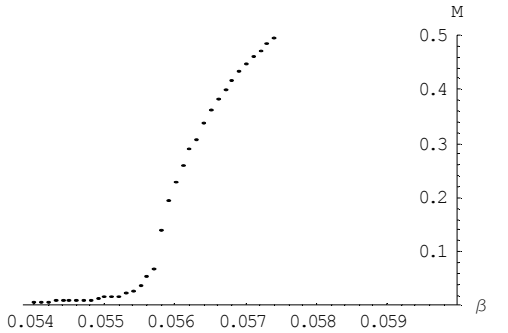
14. ábra: finite size scaling

Az egyenes egyenlete 1/L = -1.3988 + 6.2980 β_{crit} . Az x-tengelyt a β_{crit} = 0.2221 pontban metszi, így ezt fogadjuk el kritikus értéknek J = 1 esetén.

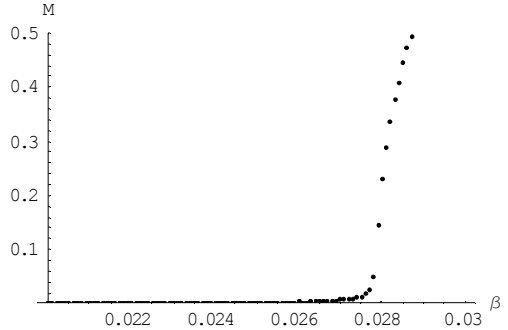
Kérdés még, hogy hogyan függ β_{crit} J-től? Az összefüggést L = 100 méreten vesszük fel. Ehhez az alábbi szimulációkat végeztem:



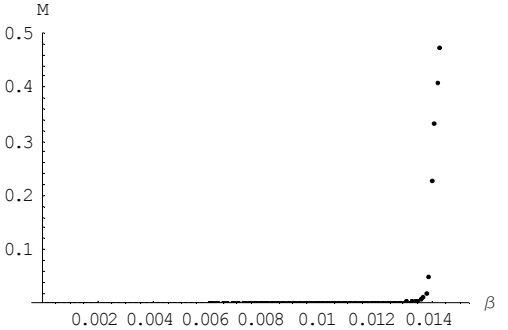
15. ábra: L = 100, nSteps = 100, J = 2



16. ábra: L = 100, nSteps = 100, J = 4



17. ábra: L = 100, nSteps = 100, J = 8



18. ábra: L = 100, nSteps = 100, J = 16

Ezek alapján:

J	**β_{crit}**
1.0	0.222
2.0	0.111
4.0	0.055
8.0	0.027
16.0	0.012

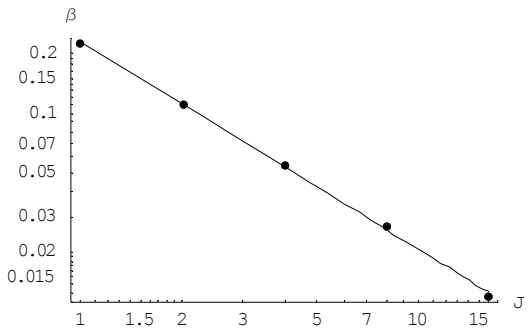
Az összefüggésről látszik, hogy hatványfüggvény. A legjobb illesztés:

$$\mathrm{Log}[\beta_{crit}] = -1.477 - 1.045\ \mathrm{Log}[J]$$

azaz

$$\beta_{crit} = \mathrm{Exp}[-1.477] \times J^{-1.045}$$

Log-Log ploton ábrázolva:



19. ábra: β_{crit} függése J-től

Az összefüggés jól láthatóan 1/J-s, azaz kétszer akkora J esetén fele akkora lesz β_{crit}. Mivel $\beta \sim 1/T$ (Kelvin), ez azt jelenti, hogy J egyenes arányos T_{crit}-tel a ferromágneses Ising modell esetén (H = 0).

4. Ábrák előállítása

1. `SwendsenWang.exe noAbsAvg SW_V_50_50.csv 50 50 0.2 0.3 0.001 1.0`
2. `SwendsenWang.exe absAvg SW_F_10_20.csv 10 20 0.2 0.3 0.001 1.0`
3. `SwendsenWang.exe absAvg SW_F_20_20.csv 20 20 0.2 0.3 0.001 1.0`
4. `SwendsenWang.exe absAvg SW_F_50_20.csv 50 20 0.2 0.3 0.001 1.0`
5. `SwendsenWang.exe absAvg SW_F_100_20.csv 100 20 0.2 0.3 0.001 1.0`
6. `SwendsenWang.exe absAvg SW_F_10_50.csv 10 50 0.2 0.3 0.001 1.0`
7. `SwendsenWang.exe absAvg SW_F_20_50.csv 20 50 0.2 0.3 0.001 1.0`
8. `SwendsenWang.exe absAvg SW_F_50_50.csv 50 50 0.2 0.3 0.001 1.0`
9. `SwendsenWang.exe absAvg SW_F_100_50.csv 100 50 0.2 0.3 0.001 1.0`
10. `SwendsenWang.exe absAvg SW_HR_10_100.csv 10 100 0.218 0.225 0.0001 1.0`
11. `SwendsenWang.exe absAvg SW_HR_20_100.csv 20 100 0.21 0.28 0.0001 1.0`
12. `SwendsenWang.exe absAvg SW_HR_50_100.csv 50 100 0.22 0.25 0.0001 1.0`
13. `SwendsenWang.exe absAvg SW_HR_100_100.csv 100 100 0.218 0.225 0.0001 1.0`
15. `SwendsenWang.exe absAvg SW_HR_100_100_2.csv 100 100 0.10 0.115 0.0001 2.0`
16. `SwendsenWang.exe absAvg SW_HR_100_100_4.csv 100 100 0.054 0.06 0.0001 4.0`
17. `SwendsenWang.exe absAvg SW_HR_100_100_8.csv 100 100 0.02 0.03 0.0001 8.0`
18. `SwendsenWang.exe absAvg SW_HR_100_100_16.csv 100 100 0.00 0.015 0.0001 16.0`