Számítógépes Szimulációs Labor

3. Monte Carlo módszerek

Készítette:
Sztreha Balázs
Informatikus fizikus

November 5, 2007

1 Bevezetés

Monte Carlo módszereket gyakran nevezik olyan metódusoknak, amelyek véletlen számot generálnak. A nevet Monacoban lévő *Casino Monte Carlo* helyről kapta, ahol a híres szerencsejátékok folynak. Ez a módszer hasznos lehet:
- atomfizikai problémák, neutron mozgásának szimulációjára
- természetben előforduló problémáknál, amikor nincs lehetőségünk az összes problémát megvizsgálni, így véletlenszerűen választunk ki állapotokat, így feltérképezve a rendszert.
- bonyolult numerikus analízisnél
- véletlen bemeneti adat generálása programok tesztelésére

2 Feladatok

2.1 Integrál számítás

$$I = \int_0^\infty x \cdot e^{-x} dx$$

2.1.1 Exponenciális eloszlás egyenletes eloszlásból (gauss.cpp)

Importance sampling esetben a súlyfüggvény bevezetésével azt érjük el, hogy a mintavételi pontok nagyobb valószínűséggel esnek olyan helyre, ahol a függvény nagyobb értékeket vesz fel. A konkrét esetben a súlyfüggvény w(x) = x, és a véletlen számokat exponenciális eloszlással kell előállítani. Az exponenciális eloszlású véletlen számsorozatokat a következő függvénnyel állítottam elő:

```
double exponential(int seed){
  double dum;
  while(dum==0)
    dum=ran2(seed);
  return -log(dum);
}
```

Várhatóan a pontok számának a nagysága (N) fogja pontosítani az integrál értékét.

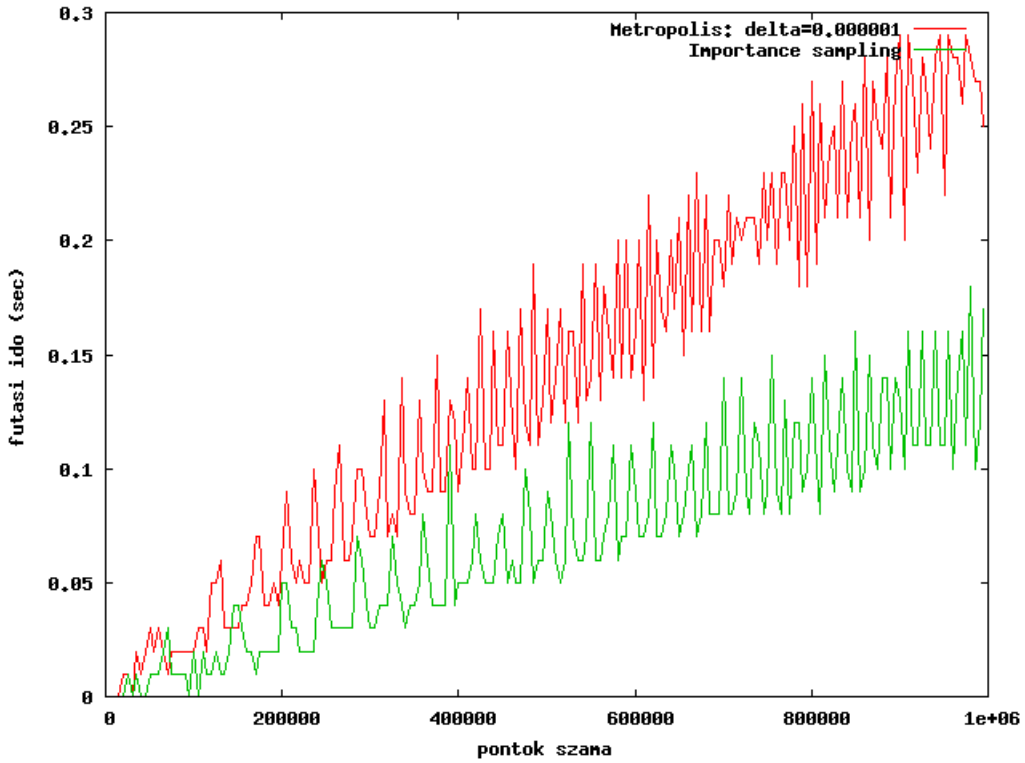

Figure 1: Futási idő összehasonlítása

2.1.2 Metropolis Monte Carlo módszerrel (metropolis.cpp)

Ennél a módszernél a mintavételi pontokat máshogy állítjuk elő. Egy adott x_0 pontból $0 \prec$ dx $\prec$ delta méretű lépést tehetünk. Az algoritmus azt részesíti előnyben, ha a nagyobb P(x) értékek felé lépünk, de kisebb mértékben az ellentétes irányba történő lépést is megengedi. A metropolis.cpp kódot módosítani kellett, hogy a kívánt integrál értékét számolja ki:

```
double P(double x) {
  return exp(-x);
}
double f_over_w(double x) {
 return x;
}
```

Ügyelni kell arra, hogy a mintavételi pontok csak pozitívak legyenek, ezért a *MetropolisStep()* függvényt át kellett írnom:

```
void MetropolisStep(double delta) {
```

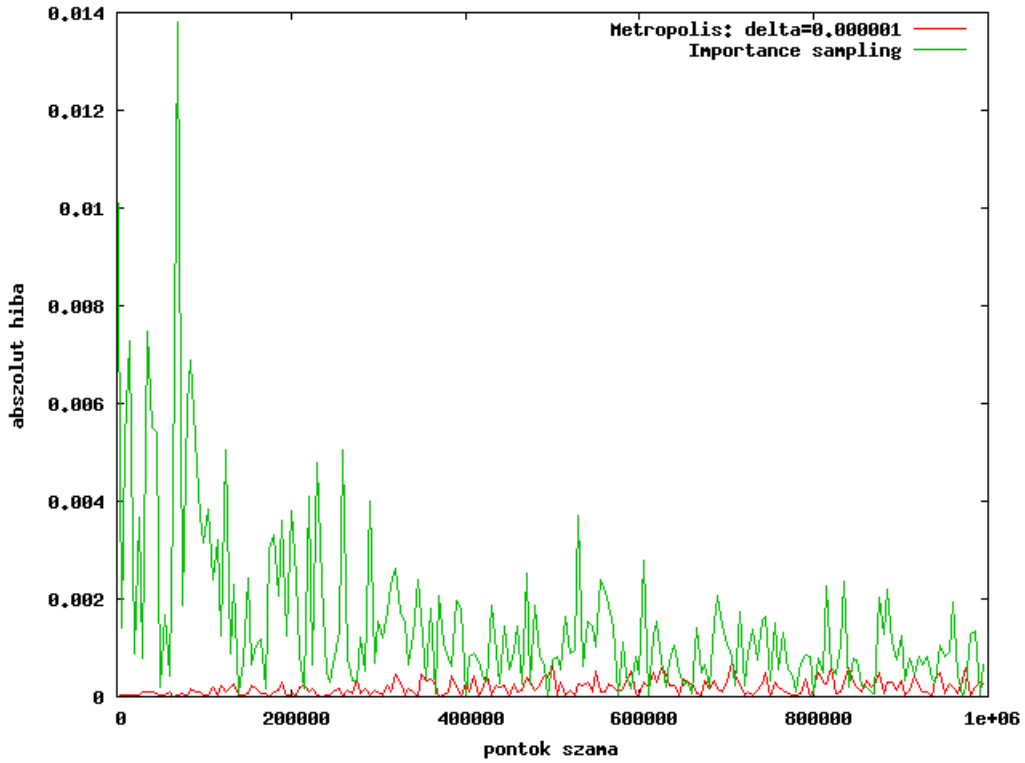

Figure 2: Abszolút hiba összehasonlítása

```
double xTrial;
do
   xTrial = x + (2 * ran2(seed) - 1) * delta;
while(xTrial<0);
...
```

Itt arra számítunk, hogy a delta csökkentésével kapunk pontos eredményt.

2.1.3 Összehasonlítás

Különböző N-ekre és *delta*-kra futtatva az algoritmusokat, arra jutottam, hogy a *gauss.cpp* a pontok számának növelésével lesz pontosabb, míg a *metropolis.cpp* a delta csökkentésével ad hamarabb pontos eredményt már kis N-re is.
Futási időben a Metropolis a lassabb, de mind a kettő kb O(N) futási idejű.

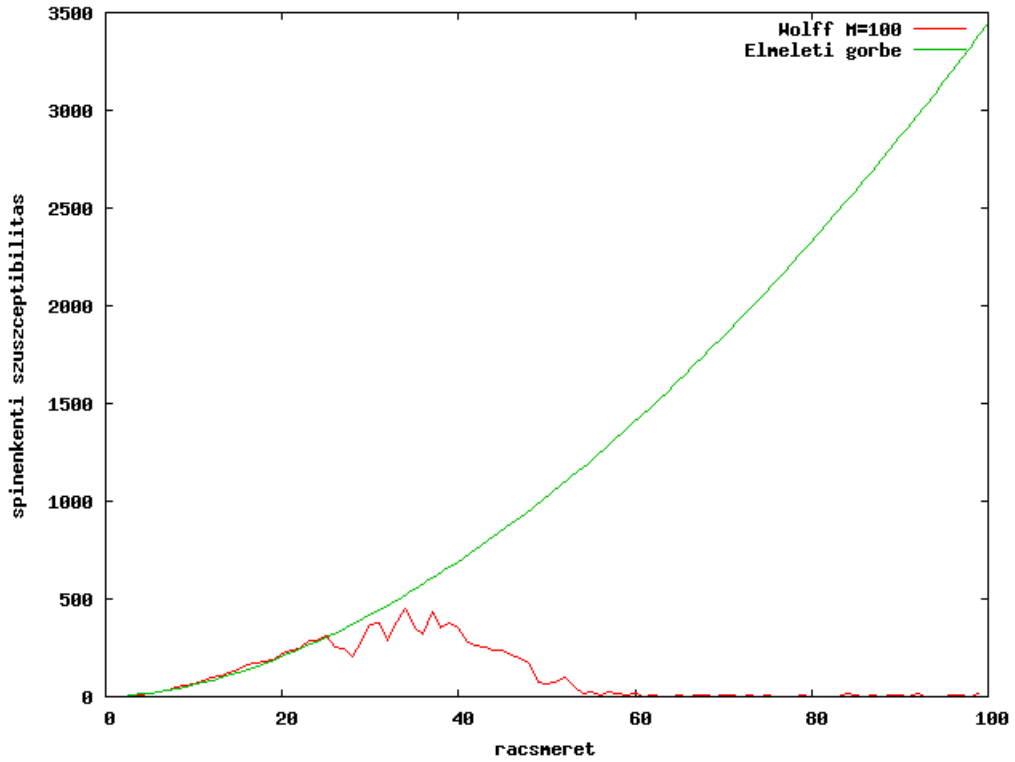

Figure 3: Wolff klaszter algoritmus M = 100 esetén, T = 2.2699 kritikus hőmérsékleten

2.2 Wolff klaszter algoritmus

2.2.1 Spinenkénti szuszceptibilitás

A *wolff.cpp* programot úgy módosítottam, hogy csak a rács vízszintes irányú méretét várja a bemenetről, és a hőmérsékletet a T = 2.2699 kritikus értékre állítottam. A programot a következő shell scripttel futtattam, majd a kimeneti adatokat ábrázoltam.

```
i=0;
while test $((i=i+1)) -lt 100; do
   echo "Lx=" $i". Szamolok..."
   ./wolff $i >> wolff.out
done;
```

Illesztett elméleti görbém a cikk alapján:

$$f(x) = a \cdot x^{2-h},$$

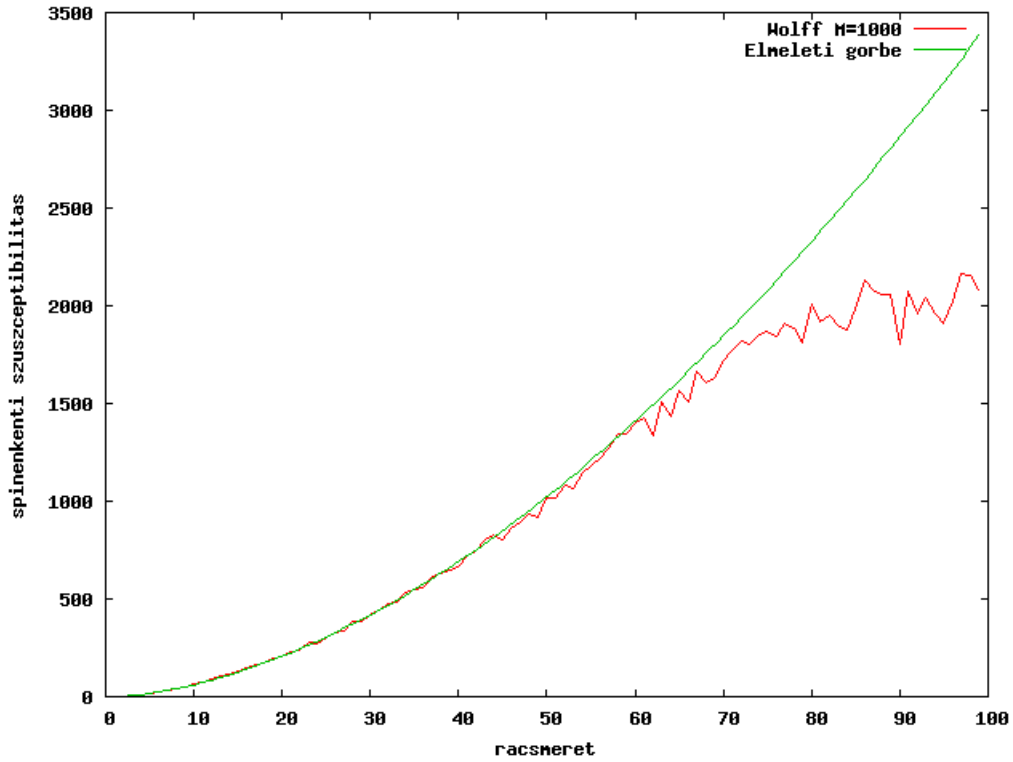

Figure 4: Wolff klaszter algoritmus M = 1000 esetén, T = 2.2699 kritikus hőmérsékleten

ahol h=1/4 és x paraméter a bemeneti rácsméret.
Az ábrák alapján látható, hogy a Monte Carlo lépések számának növelésével egyre nagyobb rendszerek szimulálhatók a programmal, mert később térnek el az elméleti görbétől.

2.2.2 Klaszterek átlagos mérete

Az s mátrixban van tárolva az összes spin iránya. Ebből ki lehet olvasni a klaszterek méretét a Hoshen-Kopelman algoritmussal:

```
int klaszter(){
   for(int x=1;x<Lx;x++){
    for(int y=1;y<Ly;y++){
      if(s[x][y]==1){
        bal = s[x-1][y];
        fent = s[x][y-1];
        if((bal == -1)&&(fent == -1)){
```

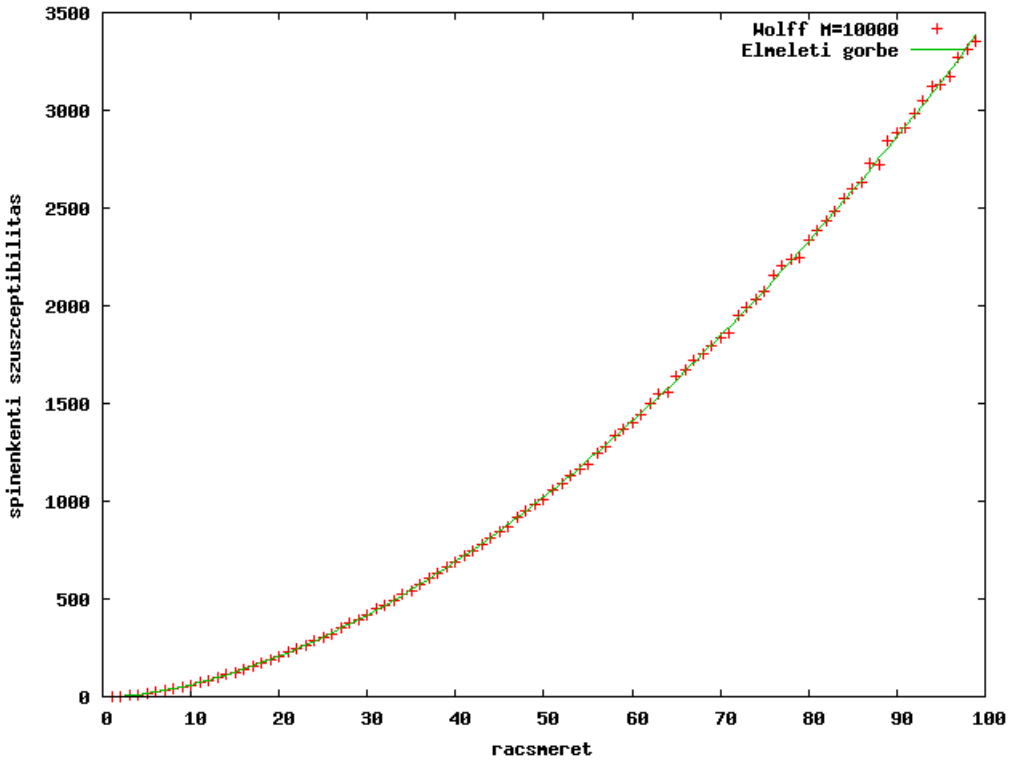

Figure 5: Wolff klaszter algoritmus M = 10000 esetén, T = 2.2699 kritikus hőmérsékleten

```
            max_cimke++;
            cimke[x][y] = max_cimke;
          } else {
            if(bal != -1){
              if(jobb != -1){
                egyesit(bal,fent);
                cimke[x][y] = keres(fent);
              }
            } else {
              cimke[x][y] = keres(jobb);
            }
          }
        }
      }
    }
}
```

```
//kikeres egy reprezentatív tagot az x-szel jelzett klaszterből
int keres(int x){
  while(cimkek[x] != x)
    x = cimkek[x];
  return x;
}

void egyesit(int x, int y) {
  cimkek[keres(x)] = keres(y);
}
```

A fenti eljárás a többszörös cimkézés módszerével a klasztereket megszámozza. A cimke tömbben össze kell számolni, hogy hány azonos címke van (ez az egyes klaszterek mérete), és ennek az átlaga volt a kérdés.