Simulations in Statistical Physics

Course for MSc physics students

Janos Török

Department of Theoretical Physics

September 16, 2014

Simulations

Experiments	*Simulations*
Principle of measurement	Algorithm
Apparatus	Program + Hardware
Calibration	Calibration + Debugging
Sample	Sample
Measurement	Run

Data collection

Analysis

Simulations

Experiments	*Simulations*
Principle of measurement	Algorithm
Apparatus	Program + Hardware
Calibration	Calibration + Debugging
Sample	Sample
Measurement	Run

Data collection

Analysis

Marked ones: Computer codes!

Programming languages

Simulations codes

- System size must be large
  - Phase transition $\xi \to \infty$
  - Real systems $N \sim 10^{23}$ (memory $< 10^{11}$)
- Simulation time should be long
  - Relaxation time
  - Interesting phenomena take long
  - Separation of time scales

Must be efficient!

It is not bad if program is readable and extensible...

Sample preparation

- Sometimes it is also a simulation

Data analysis

- Anything may happen

Programming languages

Problem to solve:

- Fill an array with product of two random numbers
- Calculate the average of them

python

```
import random
random.seed(12345);
N = 10000
s = []
for i in range(0,N):
  s.append( random.random() * random.random() )

av = 0
for i in range(0,N):
  av += s[i]

print av/N
```

matlab

```
N = 10000;
s = zeros(N,1);
rng( 12345 );
for i = 0:N
  s(i) = rand * rand;
end
% s = rand(N,1);
av = 0;
for i = 0:N
  av = av + s(i);
end
av = av / N;
% av = sum( s ) / N;
disp( av );
```

Programming languages

```
N = 10000;
s = zeros(N,1);
rng( 12345 );
for i = 0:N
  s(i) = rand * rand;
end
% s = rand(N,1);
av = 0;
for i = 0:N
  av = av + s(i);
end
av = av / N;
% av = sum( s ) / N;
disp( av );
```

```
import random
random.seed(12345);
N = 10000
s = []
for i in range(0,N):
  s.append( random.random() * random.random() )

av = 0
for i in range(0,N):
  av += s[i]

print av/N
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main(int argn,char * argv[])
{
  int i,N;
  double *s;
  double av, rm1;

  N=10000000;
  s = (double *)calloc(N, sizeof(double));
  srand(12345);
  rm1 = 1.0 / RAND_MAX;

  for (i=0; i<N; i++) {
/*    s[i] = (double) rand() * rm1 * rand() * rm1;*/
    s[i] = (double) rand() * rand() / RAND_MAX / RAND_MAX;
  }
  av = 0.0;
  for (i=0; i<N; i++) {
    av += s[i];
  }
  printf("%lg\n", av / N);
}
```

Optimization

- Multiplication vs. Division (*not so* old computers)

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main(int argn,char * argv[])
{
  int i,N;
  double *s;
  double av, rm1;

  N=10000000;
  s = (double *)calloc(N, sizeof(double));
  srand(12345);
  rm1 = 1.0 / RAND_MAX;

  for (i=0; i<N; i++) {
/*    s[i] = (double) rand() * rm1 * rand() * rm1;*/
    s[i] = (double) rand() * rand() / RAND_MAX / RAND_MAX;
  }
  av = 0.0;
  for (i=0; i<N; i++) {
    av += s[i];
  }
  printf("%lg\n", av / N);
}
```

Optimization

- Programming language
  - In example C is 20 times faster than python
  - On old computers with multiplication is 20% faster
  - Matlab, Maple, Mathematica are expensive
  - Clusters have C, and C++
- Optimization
  - Parallelization
  - Indexing Careful usage of pointers
  - Reformulate operations
  - Does not always worth the pain
  - `gprof`

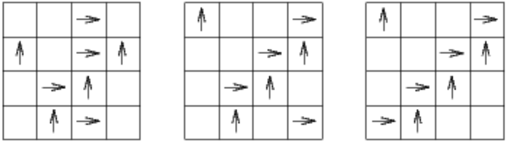
gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 37.66     56.83    56.83 3248064862     0.00     0.00  is_in_community
 25.99     96.05    39.22  1000000     0.04     0.04  e_erode
 11.55    113.47    17.43 21355853     0.00     0.00  weighted_random_link
  6.33    123.03     9.55 11078805     0.00     0.00  weighted_random_link_ban_list
  3.02    127.58     4.55  8406648     0.00     0.01  e_info
  2.77    131.75     4.18                             main
  2.26    135.16     3.40 197988614     0.00     0.00  ct_weight
  2.10    138.33     3.17        4   792.50   792.50  clear_data
  1.85    141.12     2.79 12949626     0.00     0.00  e_single
  1.73    143.74     2.62 164260875     0.00     0.00  ranksz
  1.60    146.16     2.42 12774907     0.00     0.00  strengthen
  0.97    147.62     1.46 19359356     0.00     0.01  communicate
  0.88    148.94     1.33 248428917     0.00     0.00  is_internet
  0.32    149.43     0.48    15380     0.03     0.03  random_agent_with_group_sex
  0.31    149.90     0.47  2042439     0.00     0.00  e_share
  0.24    150.25     0.36                             seed3
```

Optimization

- Programming language
- Optimization
  - Careful with time
  - Too much optimization prevents further development



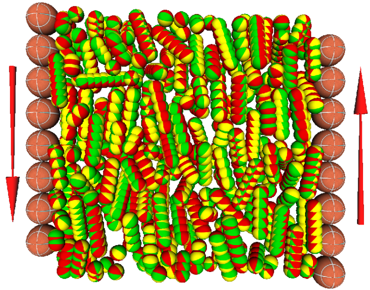
1)	**Jobbra haladó**	: 01101100	↑ → → ␣ → → ↑ ␣
2)	**Felfelé haladó**	: 10000010	
3)	Shift 1)	: 00110110	␣ ␣ → → ␣ → → ␣
4)	2) AND 3)	: 00000010	× (7th position)
5)	3) ADC 4)	: 00111000	× × × (3rd–5th positions)
6)	5) AND NOT 3)	: 00001000	× (5th position)
7)	3) - 4) + 6)	: 00111100	↑ ␣ → → → → ↑ ␣

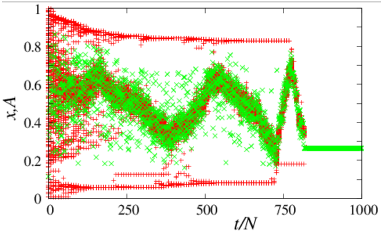
Optimization

- Programming language
- Optimization
  - Careful with time
  - Too much optimization prevents further development
  - Optimize only working code!
- Algorithm
  - The war can be won here

Simulations

- Do what nature does
  - Molecular dynamics
  - Hydrodynamics
- Make use of statistical physics
  - Monte-Carlo dynamics
  - Simulate simplified models
  - Much smaller codes!





Random numbers

- Why?
  - Ensemble average:

$$\langle A \rangle = \sum_i A_i P_i^{\mathrm{eq}}$$

    Random initial configurations
  - Model: e.g. Monte-Carlo
  - Fluctuations
  - Sample
- How?

Generate random numbers

- We need good randomness:
  - Correlations of random numbers appear in the results
  - Must be fast
  - Long cycle
  - Cryptography

Random number generators

- True (Physical phenomena):
  - Shot noise (circuit)
  - Nuclear decay
  - Amplification of noise
    - Atmospheric noise (`random.org`)
    - Thermal noise of resistor
    - Reverse biased transistor
  - Limited speed
  - Needed for cryptography
- Pseudo (algorithm):
  - Deterministic
    - Good for debugging!
  - Fast
  - Can be made reliable

Language provided random numbers

It is good to know what the computer does!

- Algorithm
  - Performance
  - Precision
  - Limit cycle
  - Historically a catastrophe
- Seed
  - From true random source
  - Time
  - Manual
    - Allows debugging
    - Ensures difference

First only uniform random numbers

Multiplicative congruential algorithm

- Let r_j be an integer number, the next is generated by

$$r_{j+1} = (ar_j + c)\text{mod}(m),$$

- Sometimes only k bits are used
- Values between 0 and $m - 1$ or $2^k - 1$
- Three parameters (a, c, m).
- If $m = 2^X$ is fast. Use AND (&) instead of modulo (%).
- Good:
  - Historical choice:
    $a = 7^5 = 16807$, $m = 2^{31} - 1 = 2147483647$, $c = 0$
  - gcc built-in $(k = 31)$:
    $a = 1103515245$, $m = 2^{31} = 2147483648$, $c = 12345$
- Bad:
  - RANDU: $a = 65539$, $m = 2^{31} = 2147483648$, $c = 0$

Tausworth, Kirkpatrick-Stoll generator

- Fill an array of 256 integers with random numbers

$$J[k] = J[(k-250)\&255] \hat{} J[(k-103)\&255]$$

- Return $J[k]$, increase k by one
- Can be 64 bit number
- Extremely fast, but short cycles for certain seeds

XOR function

^	1	0
1	0	1
0	1	0

Tausworth, Kirkpatrick-Stoll generator corrected by Zipf

The one the lecturer uses

- Fill an array of 256 integers with random numbers

$$J[k] = J[(k-250)\&255]\hat{}J[(k-103)\&255]$$

Increase k by one

$$J[k] = J[(k-30)\&255]\hat{}J[(k-127)\&255]$$

- Return $J[k]$, increase k by one
- Extremely fast, reliable also on bit level

General transformation $x \in [0:1[$

$$x = r/(RAND\_MAX + 1)$$

Tests

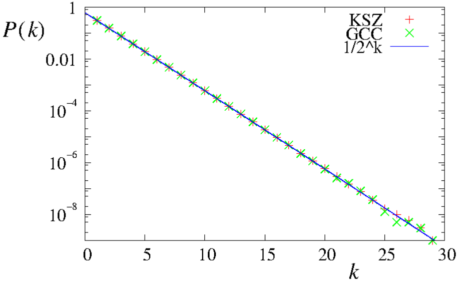
- General: e.g. TESTU01
- Diehard tests:
  - Birthday spacings (spacing is exponential)
  - Monkey tests (random typewriter problem)
  - Parking lot test
  - Moments: $m = \int_0^1 \frac{1}{n+1}$
  - Correlation
    $$C_{q,q'}(t) = \int_0^1 \int_0^1 x^q x'^{q'} P[x, x'(t)] dx dx' = \frac{1}{(q+1)(q'+1)}$$
  - Fourier-spectra
  - Fill of d dimensional lattice
  - Random walks

Red ones are not always fulfilled!

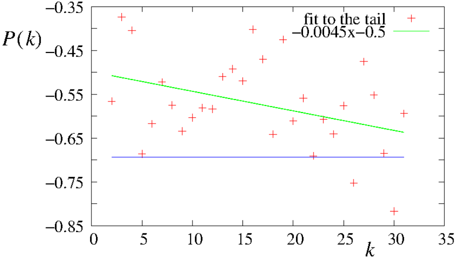
- Certain Multiplicative congruential generators are bad on bit series distribution, not completely position independent.

Bit series distribution

Probability of having k times the same bit

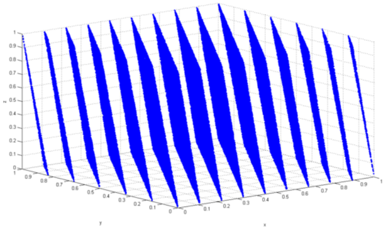


Fit to the tail for different bit positions show



Fill of d dimensional lattice

- Generate d random numbers $c_i \in [0, L]$
- Set $x[c_1, c_2, \ldots, c_d] = 1$
- The Marsaglia effect is that for all congruential multiplicative generators there will be unavailable points (on hyperplanes) if d is large enough.
- For RANDU $d = 3$



Solution for Marsaglia effect

- Instead of d random numbers only 1 (x)
- Divide it int d parts
  ```
  c_1=x%d, x/=d
  c_2=x%d, x/=d
  ...
  ```
- Better to have $L = 2^k$.
- In this case much faster!

General advice: Save time by generating less random numbers

Random numbers with different distributions

- Let us have a good random number $r \in [0, 1]$.
- The probability density function is $P(x)$
- The cumulative distribution is

$$D(x) = \int_{-\infty}^{x} P(x')dx'$$

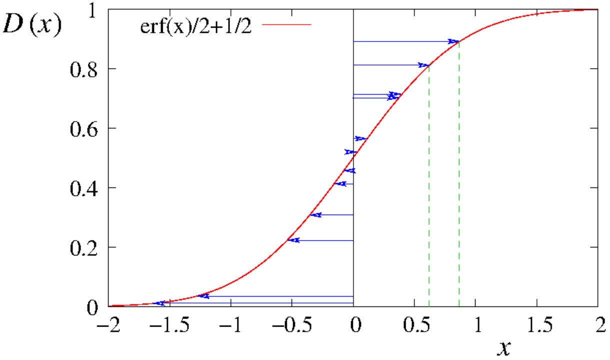
- Obviously:

$$P(x) = D'(x)$$

- The numbers $D^{-1}(x)$ will be distributed according to $P(x)$
- $D^{-1}(x)$ is the inverse function of $D(x)$ not always easy to get!

Random numbers with different distributions

Graphical representation



Box-Müller method

Normally distributed random numbers

$$P(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

- Generate independent uniform $r_1, r_2 \in (0, 1)$
- r_1, r_2 cannot be zero!
- Two independent normally distributed random numbers:

$$x_1 = \sqrt{-2 \log r_1} \cos 2\pi r_2$$

$$x_2 = \sqrt{-2 \log r_1} \sin 2\pi r_2$$

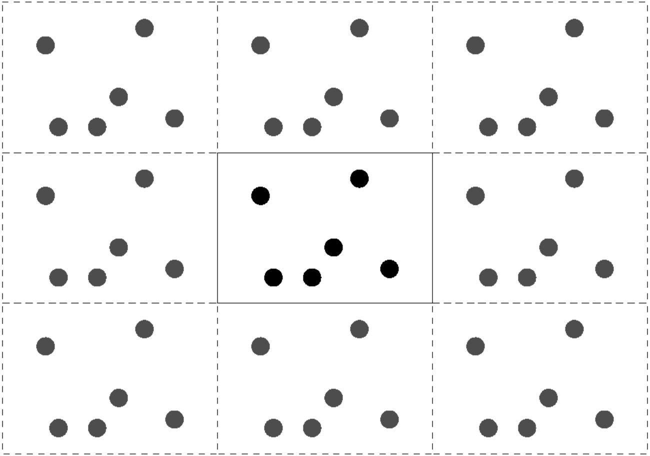
- It uses radial symmetry:

$$P(x, y) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \frac{1}{\sqrt{2\pi}} e^{-y^2/2} = \frac{1}{\sqrt{2\pi}} e^{-(x^2+y^2)/2}$$

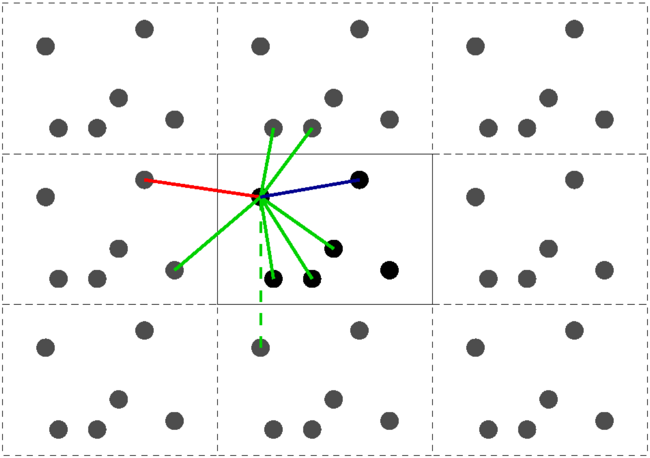
Boundary conditions

- Real boundary conditions
  - Closed (nothing)
  - Walls (with temperature)
  - Substrate (often too expensive)
- Computer based boundary conditions
  - **Periodic boundary conditions**
  - Absorbing (whatever leaves is gone)
  - Reflecting (everything is reflected back)

Periodic boundary conditions

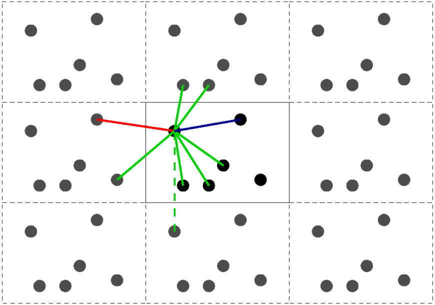


Periodic boundary conditions → contacts



Periodic boundary conditions

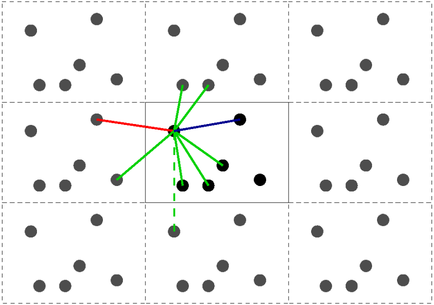
- Infinitely many neighboring cells if long range interactions
- Possibility of self interaction (must be charge neutral)
  - General solution: long range interactions are handled in k-space
- Linear momentum is conserved
- Angular momentum is not conserved



Periodic boundary conditions

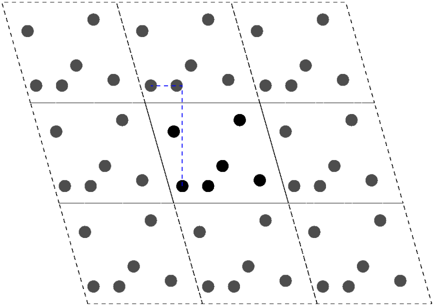
Distance

```
dx = x[i] - x[j]
if (dx < -Lx/2) dx+=Lx;
if (dx >  Lx/2) dx-=Lx;
```



Periodic boundary conditions deformed box

- Box is tilted, positions of particles artificially moved
- Homogeneous shear

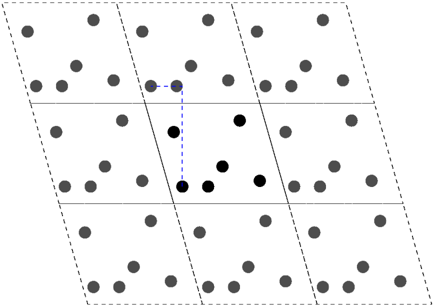


Periodic boundary conditions deformed box

Distance

- Order matters
- Tilted: by D_{xy}, D_{xz}, D_{yz}

```
dx = x[i] - x[j]
dy = y[i] - y[j]
dz = z[i] - z[j]
if (dz < -Lz/2) { dz+=Lz; dx+=Dxz; dy+=Dyz; }
if (dz >  Lz/2) { dz-=Lz; dx-=Dxz; dy-=Dyz; }
if (dy < -Ly/2) { dy+=Ly; dx+=Dxy; }
if (dy >  Ly/2) { dy-=Ly; dx-=Dxy; }
if (dx < -Lx/2) dx+=Lx;
if (dx >  Lx/2) dx-=Lx;
```



Periodic boundary conditions Lees-Edwards boundary conditions → shear

- Images are shifted
- Different from shear by walls
- Different from box tilt
- Stress propagation is important