Dwuwymiarowy model Isinga - symulacja komputerowa

Dominik Stańczak

25 stycznia 2016

1 Model Isinga

W roku 1924 niemiecki fizyk Ernst Ising zaproponował model, nazwany później od jego nazwiska, mający na celu wytłumaczyć zjawiska zachodzące w ferromagnetykach, a zwłaszcza przejście fazowe w temperaturze Curie. Model Isinga, jak powszechnie wiadomo, opiera się na przedstawieniu spinów[1] w materiale jako dyskretnych cząstek na siatce, obdarzonych spinem mogącym przyjmować wartości $S_i = \pm 1$. Hamiltonian takiego układu w przypadku nieuwzględniającym zewnętrznego pola magnetycznego przedstawia się jako

$$H = -\sum_{i,j \neq j} J_{ij} S_i S_j$$

gdzie J_{ij} jest tak zwaną całką wymiany, wielkością określającą siłę wzajemnego oddziaływania między dowolnymi dwiema cząstkami, zaś sumowanie odbywa się po wszystkich parach cząstek w układzie. Często przyjmuje się, że J_{ij} ma niezerową wartość (często 1) wyłącznie dla najbliższych sąsiadów danej cząstki.

Należy zwrócić uwagę, że dla $J > 0$ korzystna energetycznie jest sytuacja, gdy wszystkie spiny mają identyczny kierunek - materiał jest wtedy ferromagnetyczny. Dla $J < 0$ korzystna energetycznie jest sytuacja, w której wszystkie spiny mają kierunek przeciwny do swoich sąsiadów.

Magnetyzację układu definiuje się w prosty sposób jako sumę orientacji wszystkich spinów w układzie:

$$M = \sum_i S_i$$

[1] Oczywiście można wykorzystać model Isinga do modelowania innych zjawisk, lecz dla skupienia uwagi ograniczmy się do modelowania materiałów magnetycznych.

Ising znalazł w swojej pracy doktorskiej rozwiązanie układu w jednym wymiarze, w którym mowa o tzw. łańcuchu Isinga. Niestety, w jednym wymiarze łańcuch Isinga nie przejawia przejścia fazowego, zaś uporządkowanie układu, którego spodziewamy się w systuacji ferromagnetycznej, maleje wykładniczo w czasie. Spiny są więc zorientowane losowo, a magnetyzacja krąży wokół zera - nie jest to więc dobry model magnesu. Ising błędnie wywnioskował, że jego model będzie się zachowywał podobnie w dowolnej liczbie wymiarów.

2 Dwuwymiarowy model Isinga

W roku 1944 Lars Onsager w swojej własnej pracy doktorskiej rozwiązał analitycznie model Isinga dla dwóch wymiarów, z okresowymi warunkami brzegowymi. Jest to jednoznaczne z założeniem, że materiał ma całkowitą symetrię translacyjną. Jak się okazało, w dwóch wymiarach model faktycznie wykazuje przejście fazowe w temperaturze krytycznej, zakładając izotropię (niezależność całki wymiany od kierunku oddziaływania):

$$T_C = \frac{2}{\ln(1+\sqrt{2})}$$

Poniżej temperatury krytycznej układ ma stabilne minima energetyczne (stany równowagi) dla średniej energii na cząstkę[2]

$$u =< U >= -J \operatorname{ctgh}(2\beta J)\Big(1 + \frac{2}{\pi}(2\operatorname{tgh}^2(2\beta J) - 1)K(x)\Big)$$

gdzie $\beta = (k_B T)^{-1}$, $x = \sinh^2 2\beta J$, zaś $K(x)$ jest całką eliptyczną zupełną pierwszego rodzaju:

$$K(x) = \int_0^{\pi/2} (1 - x\sin^2(t))^{-1/2} dt$$

Należy zwrócić uwagę, że dla $T = T_C$, $K(x) = \infty$.

W stanie równowagi teoretyczna magnetyzacja na cząstkę:

$$m =< M >= [1 - \sinh^{-4}(2\beta J)]^{1/8}$$

Układ wykazuje więc spontaniczną magnetyzację - tak, jak spodziewamy się dla ferromagnetyka.

[2] https://en.wikipedia.org/wiki/Square-lattice_Ising_model#Exact_solution

3 Symulacja komputerowa

Ze względu na prostotę swoich podstawowych reguł dwuwymiarowy model Isinga znakomicie nadaje się do symulacji komputerowej. Siatka spinów może być w bardzo łatwy i logiczny sposób modelowana jako dwuwymiarowa tablica liczb całkowitych o rozmiarze NxN. Spiny są wtedy jednoznacznie określone przez indeksy i, j z zakresu $(0, N)$ na siatce. Jako początkowy stan układu przyjmujemy losową tablicę liczb całkowitych $S_i = \pm 1$.

Sąsiedzi danego spinu są łatwi do znalezienia jako spiny o indeksach

$$(i+1, j), (i-1, j), (i, j+1), (i, j-1)$$

W celu uwzględnienia okresowych warunków brzegowych obliczamy indeksy modulo N, na przykład dla spinu w prawym dolnym rogu symulacji o indeksie (N, N) indeksy sąsiadów to:

$$(0, N), (N-1, N), (N, 0), (N, N-1)$$

4 Dynamika układu. Algorytm Metropolis-Hastings

Pozostaje kwestia najważniejsza - implementacja dynamiki układu. Algorytm stosowany w tym celu jest nieskomplikowany:

1. Wybieramy losowy spin jako parę indeksów (i, j).
2. Obliczamy jego energię interakcji tego spinu w przypadku, gdybyśmy zdecydowali się go przerzucić, z jego czterema sąsiadami, potencjalnie uwzględniając okresowe warunki brzegowe, jako sumę $E = -JS_{ij} \sum_{\text{sasiedzi}} S_{\text{sasiad}}$. Należy zwrócić uwagę, że różnica energii między tymi dwoma stanami wynosi $\Delta E = -2E$.
3. Na podstawie zmiany energii interakcji danego spinu decydujemy, czy należy go przerzucić.
4. Opcjonalnie, jeśli nastąpiło przerzucenie, aktualizujemy energię oraz magnetyzację układu.

Oczywiście niewyjaśnioną pozostaje kwestia decyzji, czy należy przerzucić spin. Istnieje jednak proste rozwiązanie tego problemu: algorytm Metropolis-Hastings. Według tego algorytmu na podstawie reguły "równowagi szczególnej"[3] można stwierdzić, że prawdopodobieństwo akceptacji przejścia ze

[3]Tłumaczenie własne, nie znam bowiem polskiej literatury na ten temat.

stanu A (na przykład spin up) do stanu B (spin down) powinno być proporcjonalne do $\frac{P(B)}{P(A)} = \frac{Z}{Z} e^{-\beta(E(B)-E(A))} = e^{-\beta\Delta E}$. Należy zwrócić uwagę, że algorytm ten nie wymaga obliczania sumy statystycznej układu Z - skraca się ona w trakcie dzielenia.

W praktyce wystarczy więc dla proponowanego przerzucenia spinu obliczyć czynnik boltzmannowski $P = e^{-\beta\Delta E}$, zawierający się dla dodatnich temperatur w przedziale $(0, 1)$. Następnie należy wylosować liczbę rzeczywistą z tego przedziału i porównać ją z P. Jeżeli wylosowana liczba jest mniejsza od P, należy zaakceptować przejście (przerzucenie spinu). Warto zauważyć, iż dla $\Delta E < 0$ każde przejście zostaje zaakceptowane: algorytm odwzorowuje przejście do minimum energetycznego jako stanu równowagi.

5 Kod symulacji w Pythonie

Program napisany jest w Pythonie 3.5 (dystrybucja Anaconda). Kod dostępny jest również w repozytorium[4].

```
import numpy as np
import numpy.random as random
import matplotlib
import matplotlib.animation
import matplotlib.pyplot as plt
from scipy.special import ellipk
from time import time
import os.path


k=1             #Boltzmann's constant
J=1             #the exchange integral
Theory_TC = J/k*2/np.log(1+np.sqrt(2)) #the theoretical value
    for the critical temperature

def Theory_U_Formula(T):
    """
    Calculates the theoretical energy per site as seen over at
    https://en.wikipedia.org/wiki/Square-lattice_Ising_model#
    Exact_solution
    T: Temperature, in units of Boltzmann's constant.
    """
    beta = 1/k/T
    k_parameter = 1/np.sinh(2*beta*J)**2
    m = 4*k_parameter*(1+k_parameter)**(-2)
    integral = ellipk(m)
```

[4] https://github.com/StanczakDominik/ising

```
25
26      return -J/np.tanh(2*beta*J)*(1+2/np.pi*(2*np.tanh(2*beta*J)
        **2-1)*integral)
27
28  def ising(N, NT, T, plotting=False, show=False, anim=False,
        continue_run=True):
29      """
30      Runs a 2D square lattice Ising model simulation using
        periodic boundary conditions.
31
32      N: number of rows of the square lattice. N=64 gives 64^2
        spins in the system.
33      NT: number of time steps, or single flip trials. Preferably
        as a float:
34          1e6 will run for a million iteration. Please limit this
        to 1e8 at most.
35      T:  Temperature, in units of Boltzmann's constant (this can
        be set to its
36          actual physical value above).
37      plotting: set to True to get a history plot of energy and
        magnetization
38          for the whole simulation.
39      anim: set to True to get an animation of the current run. NT
        ^(1/3) snapshots
40          are taken to conserve memory and speed up the simulation
        .
41      continue_run: set to False to restart the run from its
        starting data.
42          Note that this will overwrite any data already there
        beside starting
43          from the same initial condition.
44      """
45
46      ##================Setup parameters ==========================
47      beta=1/k/T
48      NT = int(NT)
49      saved_parameters=4  #how many parameters do we want to keep
        in history
50      Nsnapshots = int(NT**(1/3)) #how many spin array snapshots
        to take. ^1/3 because it seemed okay.
51      Theory_U = Theory_U_Formula(T)*N**2      #theoretical total
        energy for the simulation area
52      if(T<Theory_TC): #there is no spontaneous magnetization
        otherwise
53          Theory_M = N**2*(1-np.sinh(2*beta*J)**(-4))**(1/8) #see
        wikipedia link above for formula
54
55
```

```
    ##==================File management, history loading, etc
    ==================
    parameter_string = "data/N%d_T%.1f/"%(N,T)
    if not os.path.exists(r"./" + parameter_string):
        continue_run = False
        os.makedirs(r"./" + parameter_string)
        spins = random.randint(0,2, (N,N))*2-1
        np.save(parameter_string+"data_start", spins)
        history = np.zeros((Nsnapshots,saved_parameters))
        starting_iteration = starting_history_iteration = 0
    elif(continue_run):
        spins=np.load(parameter_string+"data_finish.npy")
        history=np.load(parameter_string+"history.npy")
        starting_iteration = int(history[-1,0]+1)
        starting_history_iteration = history.shape[0]
        history=np.append(history,np.zeros((Nsnapshots,
    saved_parameters)), axis=0)
    else:
        spins=np.load(parameter_string+"data_start.npy")
        history = np.zeros((Nsnapshots,saved_parameters))
        starting_iteration = starting_history_iteration = 0

    if anim:
        snapshot_history = np.empty((Nsnapshots, N,N), int)


    ##========== Define useful functions and initial diagnostics
    ==========
    def Energy(spins):
        """
        Takes the array of spins and calculates its energy (with
     PB conditions)
        """
        center = spins
        sides = np.roll(spins, 1, 0) + np.roll(spins, 1, 1) + np
    .roll(spins, -1, 0) + np.roll(spins, -1, 1)
        return -J*np.sum(center*sides)/2

    def Mag(spins):
        """
        Takes the array of spins and calculates its total
    magnetization.
        Doesn't get much simpler than this.
        """
        return np.sum(spins)

    def FlipSpin(spins, E, M):
        """
        The core of the Metropolis-Hastings algorithm.
```

```
99          Operates off relative changes in energy and
    magnetization to save time.
100
101         1. pick a spin at random
102         2. calculate the sum of spins of its 4 neighbors
103         3. calculate the change in energy of the system should
    the random spin flip
104         4. calculate the Boltzmann probability factor - how
    likely is the spin to flip?
105             for T>0 this is between 0 to 1.
106         5. pick a float from 0 to 1 at random
107         6. if the random float is lesser than the Boltzmann
    cutoff, flip the spin
108             (in-place) and update the energies.
109         """
110
111         x, y = random.randint(0,N,2)     #1.
112         test = spins[x,y]
113         neighbor_spins = spins[(x+1)%N,(y)%N] + spins[(x-1)%N,y%
    N] + spins[x%N,(y-1)%N] + spins[x%N,(y+1)%N] #2.
114         deltaE=J*2*test*neighbor_spins #3.
115         accepted = 0
116         probability_cutoff = np.exp(-beta*deltaE) #4.
117         uniform_random = random.random() #5.
118         if(uniform_random < probability_cutoff): #6.
119             E += deltaE
120             M -= 2*test
121             accepted = 1
122             spins[x,y] *= -1
123         return E, M, accepted
124
125     def ViewSystem(title):
126         """
127         Print the state of the system to console.
128         """
129         print(title)
130         print("Iteration: %d\tEnergy: %d\tMagnetization: %d\tN:
    %d\tT: %.2f" %(starting_iteration,E,M,N,T))
131         print(spins[1:-1,1:-1])
132
133     ## ========================= Main loop =========================
134     start_time = time()      #start timing the run
135
136     E, M = Energy(spins), Mag(spins)     #first (and only) direct
     calculation
137     ViewSystem("Starting")
138     for i in range(NT):
139         E, M, Accepted = FlipSpin(spins,E,M)
140         if i%(Nsnapshots)==0:     #saves data to history
```

```
141             snapshot_iteration = int((i/NT)*Nsnapshots)
142             parameters = i+starting_iteration, E, M, Accepted
143             history[starting_history_iteration+
    snapshot_iteration]= parameters
144             if anim:
145                 snapshot_history[snapshot_iteration]=spins
146
147     #saves the final spin array
148     np.save(parameter_string+"data_finish", spins)
149     #saves the history array, for the energy and magnetization
    plot
150     np.save(parameter_string+"history", history)
151     ViewSystem("Finished")
152
153
154
155     #diagnostics
156     times = history[:,0]
157     energies = history[:,1]
158     magnetization = history[:,2]
159     acceptance = history[:,3]
160     print(np.mean(acceptance))
161     print("Acceptance ratio: %f" %np.mean(acceptance))
162     print("Runtime: %f" % (time()-start_time))
163
164     def plot():
165         fig, (ax_energy, ax_magnet) = plt.subplots(2, sharex=
    True, sharey=False,figsize=(15,7) )
166         plt.title("Final energy: %d Final magnetization: %d N: %
    d T: %.2f" %(E,M,N,T))
167
168         ax_energy.plot(times,energies, "b-", label="Energy")
169         ax_energy.plot(times, np.ones_like(times)*Theory_U, "r--
    ", label="Theoretical energy")
170         ax_energy.legend()
171         ax_magnet.grid()
172         ax_energy.set_ylabel("Energy")
173
174         ax_magnet.plot(times,magnetization, "g-", label="
    Magnetization")
175         if(T<Theory_TC):
176             ax_magnet.plot(times, np.ones_like(times)*Theory_M,
    "g--", label="Theoretical spontaneous")
177             ax_magnet.plot(times, -np.ones_like(times)*Theory_M,
     "g--")
178         ax_magnet.set_ylabel("Magnetization")
179         ax_magnet.legend()
180         ax_magnet.grid()
181         plt.xlabel("Time")
```

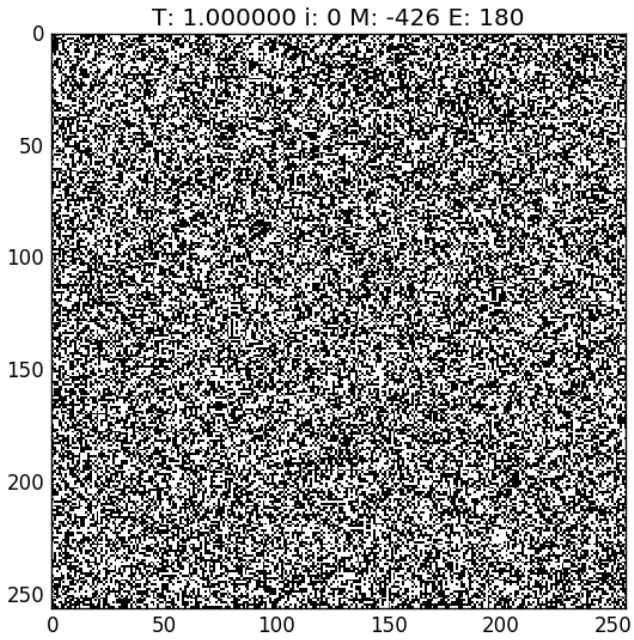
```
182
183         plt.savefig(parameter_string+"plot.png")
184         if(show):
185             plt.show()
186         else:
187             plt.clf()
188     if(plotting):
189         plot()
190
191     def animate():
192         fig=plt.figure()
193         ax = fig.add_subplot(111)
194
195         ims = [[ax.imshow(snapshot_history[i], interpolation='
    nearest', animated=True, cmap='Greys_r'),\
196          plt.title("T: %f i: %d M: %d E: %d"%(T,times[i],
    magnetization[i], energies[i]))] for i in np.arange(0,
    Nsnapshots)]
197         ani = matplotlib.animation.ArtistAnimation(fig, ims,
    interval=30, blit=True, repeat_delay=1000)
198         ani.save(parameter_string+"video%d.mp4"%
    starting_iteration, fps=30, extra_args=['-vcodec', 'libx264'
    ])
199         if(show):
200             plt.show()
201         else:
202             plt.clf()
203     if(anim):
204         animate()
```

6 Wyniki

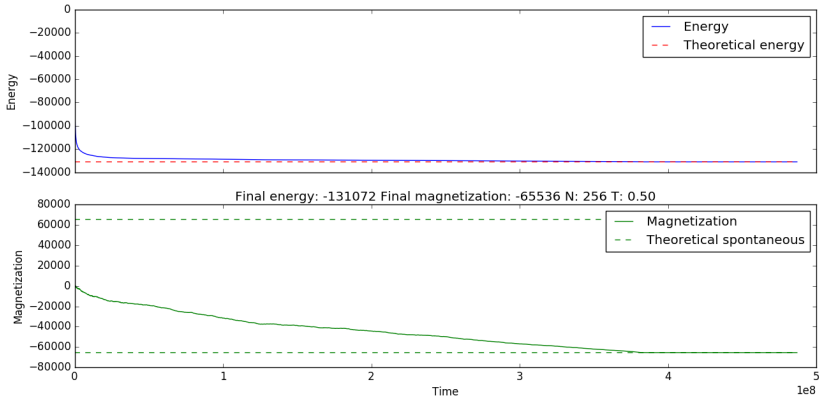
Wykonałem trzy oddzielne symulacje dla układu $256^2 = 65536$ cząstek w temperaturach

$$0.5, T_C = \frac{2}{\ln(1+\sqrt{2})}, 3.5$$

w układzie jednostek znormalizowanym do stałej Boltzmanna $k_B = 1$. Jako przykładową demonstrację warunków początkowych, wygenerowany początkowo losowy stan: (dla $T = 1$, lecz de facto niezależnie od temperatury):

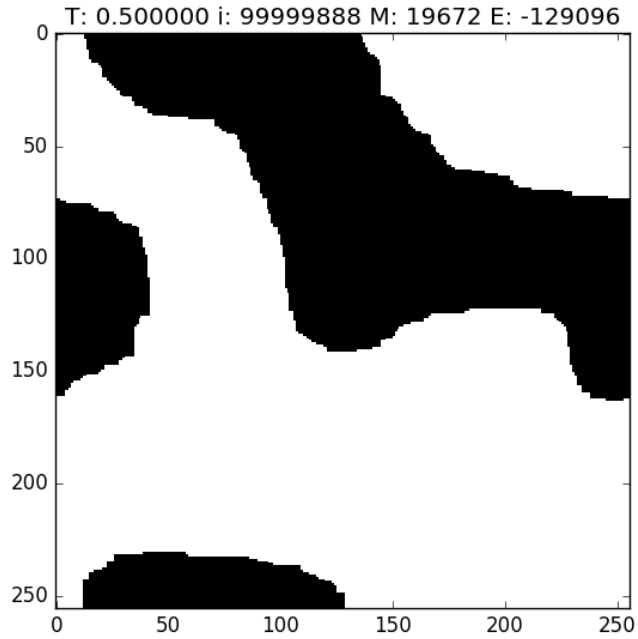


6.1 $T = 0.5$

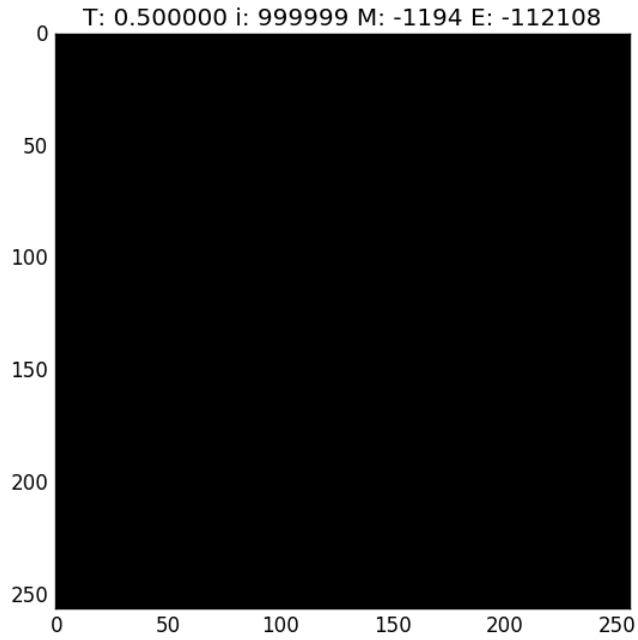


Dla temperatury poniżej T_C następuje bardzo szybka zbieżność energii do wartości bliskiej przewidywanej teoretycznie. Magnetyzacja również zbiega do wartości przewidywanej teoretycznie, lecz tym razem dużo, dużo wolniej. Ma to swoje uzasadnienie: układ przez długi czas składa się nie z

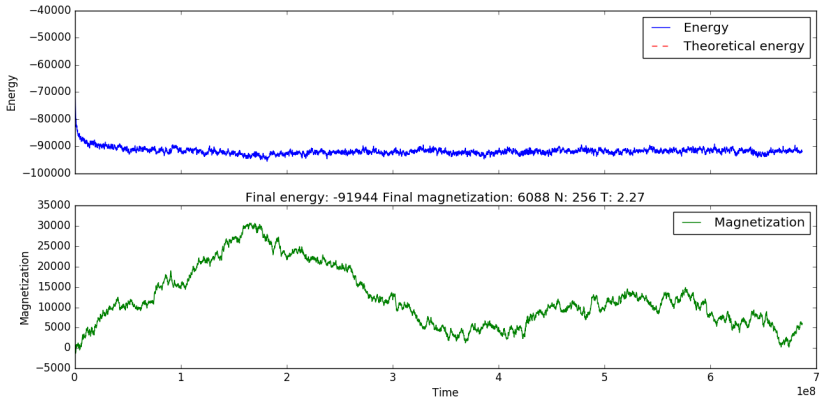
jednego układu o wszystkich spinach wskazujących w tę samą stronę, lecz z dyskretnych domen magnetycznych. Te zaś są stabilne na długich skalach czasowych. Jest to zachowanie jako żywo przypominające znane z rzeczywistych magnesów. Obrazuje to poniższe zdjęcie z animacji oraz sama animacja, pokazująca pierwsze 100 miliardów iteracji. znadująca się pod linkiem: `https://youtu.be/KWMnoyFeenU`



Czuję się również w obowiązku zademonstrować jedno z końcowych zdjęć w symulacji, demonstrujące końcowy stan równowagi:



6.2 $T = T_C$



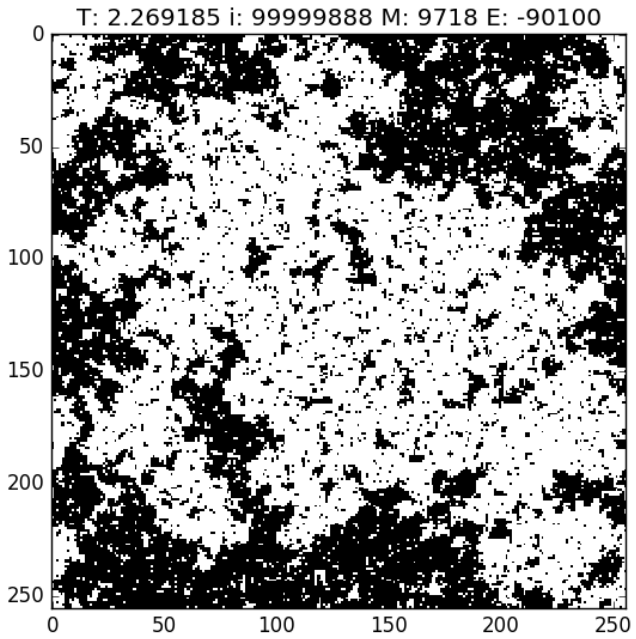
Dla temperatury równej T_C, choć energia szybko zdaje się osiągać minimum, jest to jednak minimum lokalne (na poziomie $-90k$ zamiast $-120k$, jak dla przypadku $T = 0.5$). Zachodzą intensywne oscylacje tak energii jak

i magnetyzacji. Magnetyzacja zdaje się oscylować, osiągając co najwyżej połowę wartości maksymalnej dla przypadku ferromagnetycznego.

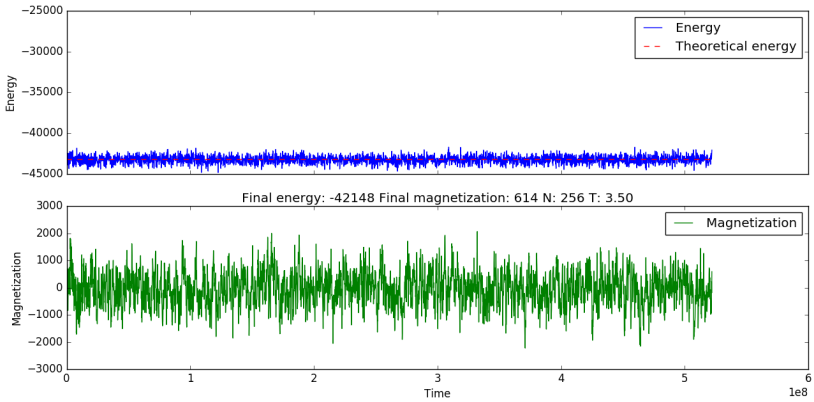
Należy zwrócić uwagę, że algorytm Metropolis-Hastings jest znany ze spowalniania blisko przejścia fazowego, ponieważ duża część przerzuceń spinów zostaje odrzucona. Symulacja tym naiwnym algorytmem w tym reżimie jest więc utrudniona.

Chciałbym też zauważyć, że teoretyczna energia przewidywana równaniami Onsagera wynosi $+\infty$ i, o ile mogę to stwierdzić, nie ma sensownego fizycznego znaczenia poza sygnalizowaniem przejścia fazowego - coś w układzie idzie "nie tak".

W animacji układ wygląda jak superpozycja uporządkowanej sytuacji w magnetyku - występują w nim wyraźne domeny ferromagnetyczne - oraz silnych szumów w paramagnetyku. Animacja pierwszych stu miliardów iteracji znajduje się pod tym linkiem: `https://www.youtube.com/watch?v=kHgOxCc_jPI`

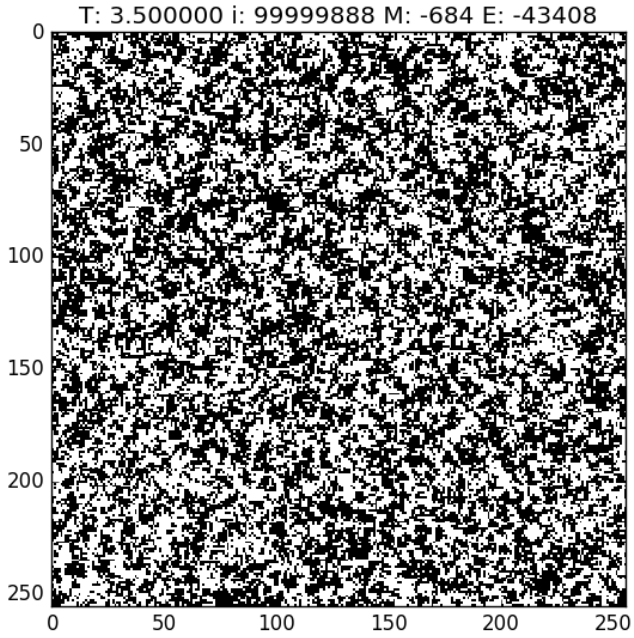


6.3 $T = 3.5$



Dla temperatury powyżej temperatury krytycznej układ bardzo szybko (praktycznie natychmiastowo) zbiega do stabilnego minimum swojej energii, przewidywanej zresztą równaniami Onsagera. Występują jednak silne oscylacje wokół tego minimum, zaś samo minimum jest względnie wysokoenergetyczne ($-45k$ wobec $-120k$ dla przypadku ferromagnetycznego). Jedyna magnetyzacja jaka występuje w układzie to intensywna, choć o rząd wielkości mniejsza niż w poprzednich przypadkach, oscylacja wokół zera. Układ w tej temperaturze jest paramagnetykiem. Model Isinga działa!

Animacja wyniku tej symulacji znajduje się pod następującym linkiem: `https://www.youtube.com/watch?v=rCm1EI82TXk`. Wyraźnie widać, że układ zaczynający z losowego ustawienia pozostaje losowy - oddziaływanie między spinami nie jest na tyle silne, aby utrzymać jakikolwiek porządek w układzie.



7 Bibliografia

- `https://www.coursera.org/course/smac` - "Statistical Mechanics - Algorithms and Computations", wyśmienity internetowy kurs wprowadzający w tajniki symulacji układów statystycznych, w tym modelu Isinga.
- `https://en.wikipedia.org/wiki/Square-lattice_Ising_model` - szybki podgląd wzorów analitycznych dla modelu Isinga 2D
- `https://en.wikipedia.org/wiki/Ising_model#The_Metropolis_algorithm` - przypomnienie formalnej wersji algorytmu Metropolis-Hastings
- `https://github.com/StanczakDominik/ising` - repozytorium programu