Università degli Studi di Modena e Reggio Emilia

DIPARTIMENTO DI SCIENZE FISICHE, INFORMATICHE E MATEMATICHE
Corso di Laurea in Fisica

Transizione di fase: modello di Ising in 2D

Angelica Tommasi
gruppo Braidi-Tommasi

Anno Accademico 2022-2023

1 Sommario

Attraverso l'utilizzo di una simulazione numerica (Python) con il metodo Metropolis Monte Carlo, si studia la dinamica di un sistema a spin con reticolo: il `modello di Ising`.

2 Introduzione

2.1 Modello di Ising

2.1.1 Hamiltoniana e medie d'ensemble

Vediamo nel dettaglio la modellizzazione di Ising che presenta una transizione di fase. Consideriamo una hamiltoniana H di N particelle interagenti su un reticolo quadrato. Queste particelle possono avere due stati denotati da s_i $+1$ o -1 che corrispondono ai due possibili stati di spin: up o down. Consideriamo inoltre che la minimizzazione dell'energia si ha quando gli atomi vicini si trovano nello stesso stato.

Per descrivere questo sistema studiare la funzione di partizione è molto complesso pertanto si utilizza una approssimazione detta `primi vicini a campo medio` cioè lo spin al centro interagisce con un campo medio dovuto ai suoi vicini più prossimi. Nel caso di un reticolo 2D questi sono 4.

La funzione di partizione è:

$$Z_N = \sum_{s_i} exp(-\beta[-\sum_{i<j} J_{ij} s_i s_j - \mu_0 B \sum_i s_i]) \tag{1}$$

dove J_{ij} indica la costante di accoppiamento degli spin mentre B è un eventuale campo magnetico esterno.

Se $J_{ij} > 0$ si considera un materiale ferromagnetico che favorisce spin paralleli[1].

Quindi:

$$H = -\sum_{i<j} J_{ij} s_i s_j - \mu_0 B \sum_i s_i \tag{2}$$

Esplicitando l'approssimazione a primi vicini l'hamiltoniana diventa nella sua forma più generale:

$$H = -J[\sum_{nn} s_i m - \mu_0 B \sum s_i] \tag{3}$$

La somma è sui primi vicini indicati con `n n` e m è una costante valida per ogni primo vicino. Definendo quindi `q` il numero dei primi vicini detto numero di coordinazione, l'hamiltoniana si semplifica ancora:

$$H = -Jmq[\sum_i s_i - \mu_0 B \sum_i s_i] = [-Jmq - \mu_0 B] \sum_i s_i \tag{4}$$

Definendo:

$$B' = Jmq/\mu_0 \tag{5}$$

$$H = (B + B')\mu_0 \sum_i s_i \tag{6}$$

con B' campo medio. [1]

Se il campo esterno è nullo avremo quindi che lo stato sarà definito dall'ennupla $\nu = (s_1, s_2, ..., s_N)$ che avrà 4 primi vicini `j` e pertanto possiamo scrivere:

$$E_\nu = H(s_1, s_2, ..., s_N) = -\sum_i {\sum_j}' s_i s_j \tag{7}$$

$$M_\nu = \sum_i s_i \tag{8}$$

dove ${\sum_j}'$ è la somma suoi primi vicini.

Per fare una simulazione è necessario utilizzare delle condizioni periodiche al bordo che appunto ci dicono che se si esce da un lato, si rientra dall'altro.

Partizionando l'energia di un microstato ν come somma dei termini a singola particella avremo che:

$$E_\nu = 1/2 \sum_i^N \epsilon_{\nu,i} \tag{9}$$

[1] In particolare si scelgono le unità $J = 1, \mu_0 = 1$ e $k_B = 1$ per la simulazione.

dove

$$\epsilon_{\nu,i} = -Js_i \sum_j{}' s_j \tag{10}$$

Se si rovescia ad esempio lo spin s_2 lo stato da $\nu = (s_1, s_2, ..., s_N)$ diventa $\nu' = (s_1, -s_2, ..., s_N)$, pertanto la differenza di energia tra due stati è:

$$\Delta_{\nu,\nu'} = (\epsilon_{\nu',i} - \epsilon_{\nu,i})/2 = 1/2[(-J(-s_i)\sum_j{}' s_j) - (-Js_i \sum_j{}' s_j)] = Js_i \sum_j{}' s_j \tag{11}$$

Per quanto riguarda infine le proprietà termodinamiche, fissato il numero di spin N e la temperatura T, le proprietà macroscopiche sono determinate dalle medie M sulla catena di Markov dalle quali è possibile calcolare l'energia interna e la magnetizzazione.

$$< E_\nu >= 1/\mathrm{M} \sum_k^{\mathrm{M}} E_{\nu,k} = U(T, N) \tag{12}$$

$$< M_\nu >= 1/\mathrm{M} \sum_k^{\mathrm{M}} M_{\nu,k} = M(T, N) \tag{13}$$

$$C_B(T, N) = 1/(k_B T^2)[(< E_\nu^2 >) - (< E_\nu >)^2] \tag{14}$$

$$\chi_B = 1/(k_B T)[(< M_\nu^2 >) - (< M_\nu >)^2] \tag{15}$$

con

$$< E_\nu^2 >= 1/\mathrm{M} \sum_k^{\mathrm{M}} E_{\nu,k}^2 \tag{16}$$

$$< M_\nu >= 1/\mathrm{M} \sum_k^{\mathrm{M}} M_{\nu,k}^2 \tag{17}$$

2.1.2 Equazione di autoconsistenza della magnetizzazione e transizione di fase

Considerando campo esterno nullo ($B = 0$) e partendo dall'equazione (2) si può ricavare la cosiddetta `equazione di autoconsistenza` che è così chiamata perché dall'ipotesi dell'esistenza della magnetizzazione M, la si calcola.

$$M = tanh(\frac{JqM}{k_B T}) \tag{18}$$

Si definisce `temperatura critica` quella temperatura in corrispondenza della quale il sistema subisce la transizione di fase.

$$T_c = Jq/k_B \tag{19}$$

In questo caso si tratta di una transizione continua da materiale ferromagnetico a materiale paramagnetico. Subito dopo questo punto, la magnetizzazione diventa nulla.

Prima di T_c (`punto di Curie`), espandendo la tangente iperbolica[2], si ha che la magnetizzazione è data da:

$$M = (T - T_c)^\alpha \tag{20}$$

dove α è detto `esponenziale critico` e, in questo intorno sinistro di T_c, è pari a 1/2 (sempre con approssimazione di campo medio).

2.2 Teoria generale del metodo Monte Carlo

Definiamo `traiettoria` una sequenza cronologica di configurazioni di un sistema. Possiamo quindi considerare una configurazione per il modello di Ising come una lista delle variabili di spin $\nu = (s_1, s_2, ..., s_N)$ per un punto nello spazio N-dimensionale [2].

Immaginiamo ora una traiettoria in questo spazio $\nu(t)$. Dato che le proprietà configurazionali cambiano mentre la traiettoria progredisce, una variabile associata al sistema, come ad esempio l'energia, è determinata dal suo valore medio in T passi:

$$< E >_T = \frac{1}{T} \sum_{t=1}^{T} E_{\nu(t)} \tag{21}$$

[2] $tanh(x) \sim x - x^3/3 + 2x^5/15 - 17x^7/315 + ...$

Ciò significa che le traiettorie sono `ergodiche` [3] e vale la `distribuzione di Boltzmann` [4].

Nel caso ideale:

$$< E >= \lim_{T\to\infty} < E >_T \tag{22}$$

In realtà nella pratica il valore medio è calcolato su un tempo finito e ciò porterà ad avere solo una stima di $< E >$.

Illustriamo ora l'algoritmo di Monte Carlo per determinare le traiettorie attraverso l'uso di numeri random, o meglio pseudo-randomici.

Per calcolare il numero totale delle configurazioni, un campione di stati o configurazioni rappresentative è creato da una `random walk` nello spazio delle configurazioni (sempre con consistenza con la distribuzione di equilibrio dell'ensemble). Se consideriamo un modello di Ising in 2D con $20x20 = 400$ spins il numero totale di configurazioni sarebbe 2^{400} ma con il metodo Monte Carlo si può semplificare lo studio del sistema con solo 10^6 configurazioni cioè si trattano solo gli stati che hanno un fattore di Boltzmann significativo (*importance sampling* [2]).

Per descrivere l'algoritmo si parte dalla condizione iniziale. Si sceglie poi uno degli spin tramite il `generatore di numeri pseudo-random` [2] che genera una distribuzione uniforme di numeri compresi tra 0 e 1.

Dalla configurazione ν passiamo quindi alla ν' che si differenzierà dalla precedente perché avrà lo spin scelto random rovesciato (se inizialmente -1, diventerà $+1$).

Il cambio di configurazione implicherà una variazione di energia così definita:

$$\Delta E_{\nu\nu'} = E_{\nu'} - E_{\nu} \tag{23}$$

Questa differenza di energia governa la probabilità relativa delle configurazioni della distribuzione di Boltzmann e pertanto si può utilizzare in un criterio per accettare o rifiutare nuove configurazioni. In particolare si ha che:

- Se $\Delta E_{\nu\nu'} \leq 0$ accettiamo la configurazione;
- Se $\Delta E_{\nu\nu'} > 0$ prendiamo un numero random x, $0 < x < 1$ e, definendo $\beta = 1/k_B T$, accettiamo la configurazione solo se $exp(-\beta\Delta E_{\nu\nu'}) \geq x$. Viceversa la configurazione è rifiutata.

Quindi riassumendo, data $\nu(t) = \nu$,

$$\begin{cases} \nu(t+1) & = \nu', \Delta E_{\nu\nu'} \leq 0 \\ \nu(t+1) & = \nu', \Delta E_{\nu\nu'} > 0 \text{ e } exp(-\beta\Delta E_{\nu\nu'}) \geq x \\ \nu(t+1) & = \nu, \Delta E_{\nu\nu'} > 0 \text{ e } exp(-\beta\Delta E_{\nu\nu'}) < x \end{cases} \tag{24}$$

Questo algoritmo è chiamato `Metodo Metropolis Monte Carlo` (1953).

2.3 Metodo Metropolis Monte Carlo in dettaglio (Python)

Si può utilizzare la libreria `numpy` di Python [5] per generare numeri random uniformemente distribuiti. Per studiare sistemi fisici composti da moltissime particelle si utilizza il metodo Metropolis Monte Carlo ideato nei laboratori di Los Alamos da Metropolis N., Rosenbluth A. W., Rosenbluth M. N., Teller A. H. e Teller E. e pubblicato nel 1953 nell'articolo *"Equation of state calculations by fast computing machines"* [3].

Questo metodo genera una `catena di Markov` i cui elementi sono distribuiti secondo una probabilità limite.

Considerando un sistema classico di N particelle non interagenti confinate in un volume V, quando N è molto grande (circa 10^{23}) non è possibile ricavare le leggi dinamiche di ogni singola particella. E' pertanto necessario determinare il comportamento macroscopico del sistema a temperatura T quando esso giunge all'equilibrio termico attraverso la `Meccanica statistica`. Questa, dalla descrizione microscopica governata dalla `Statistica di Maxwell-Boltzmann`, permette di determinare la termodinamica del sistema.

Se infatti ogni particella ha massa m e velocità (v_x, v_y, v_z) la sua energia cinetica sarà:

$$T = \frac{m}{2}(v_x^2 + v_y^2 + v_z^2) \tag{25}$$

Nel mondo microscopico la conoscenza della velocità è solo un indicatore di `densità di probabilità` data dalla `distribuzione delle velocità di Maxwell` [1].

$$f(v)dv = \frac{4}{\sqrt{\pi}}(m/2k_BT)^{3/2}v^2 exp(-mv^2/2k_BT)dv \tag{26}$$

Utilizzando Phyton possiamo plottare la funzione con questo programma:

[3] Con l'`ipotesi ergodica` si ammette l'equiprobabilità del sistema di visitare tutti i possibili stati.

[4] $P(\text{microstato r}) = exp(-E_r/k_BT)/\sum_i exp(-E_i/k_BT)$, con $Z = \sum_i exp(-E_i/k_BT)$ la somma su tutti i possibili stati [1].

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib widget
def maxwell3d(v,kt=1,m=1):
    denom = -1./(2*kt*m)
    cost = (4./np.sqrt(np.pi))*(np.sqrt(m/2.*kt))**3
    return cost*v**2*np.exp(denom*v**2)
if __name__ == "__main__":
    v = np.linspace(0.,8.,1000)
    #restituisce su specifico intervallo numeri uniformemente distanziati
    P  = maxwell3d(v)
    fig1, ax1 = plt.subplots(1, 1, figsize=(8,4), dpi=100)
    ax1.set_xlabel('v')
    ax1.set_ylabel('f(v)')
    plt.plot(v,P, color = 'mediumorchid')
    plt.show()
     plt.savefig('Maxwell.jpeg')
```

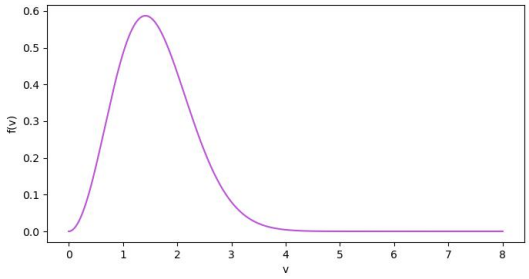


Figure 1: Distribuzione di Maxwell `f(v)`

Illustriamo ora il programma per ottenere un istogramma personalizzato che verrà utilizzato per le prossime considerazioni.

- Istogramma.py

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import matplotlib.path as path
%matplotlib widget

def istogramma(x,M,colore='g',grandezza='velocity'):
    fig1, ax1 = plt.subplots(1, 1, figsize=(8,4), dpi=100)
# histogram our data with numpy
    n, bins = np.histogram(x, 100, density=True) #Compute the histogram of a dataset
# get the corners of the rectangles for the histogram
    left = np.array(bins[:-1])
    right = np.array(bins[1:])
    bottom = np.zeros(len(left))
    top = bottom + n
# we need a (numrects x numsides x 2) numpy array for the path helper
# function to build a compound path
    XY = np.array([[left, left, right, right], [bottom, top, top, bottom]]).T
# get the Path object
    barpath = path.Path.make_compound_path_from_polys(XY)
# make a patch out of it
    patch = patches.PathPatch(barpath, facecolor=colore, alpha=0.4)
    ax1.add_patch(patch)
# update the view limits
```

```
    ax1.set_xlim(left[0], right[-1])
    ax1.set_ylim(bottom.min(), top.max()+.03)
    #ax1.axhline(1.,color='blue', alpha=0.5)
    plt.title(' istogramma %s con %d campioni' % (grandezza,M) )
    """
    % detto estrattore legge dalla tupla ed estrae
    """
    plt.show()
#
```

Si definisce `istogramma` che ha come argomenti i dati stessi dell'istogramma (`x`), il numero dei campioni (`M`), il colore e la grandezza che è un testo printato alla fine.

La funzione `istogramma` restituisce due valori: `n` e `bins`; `n` si riferisce all'altezza di ogni singolo rettangolino e quindi la densità o frequenza relativa delle diverse fasce dell'istogramma, `bins` si riferisce agli intervalli dell'istogramma, cioè agli intervalli entro cui deve essere delimitato. E' infatti utilizzato per definire gli estremi sinistri (`left`) e destri (`right`) che partono rispettivamente o dall'ultimo elemento `-1` o dal primo elemento `+1`. Il `bottom` è definito dalla lunghezza dell'ultimo elemento mentre il `top` dalla base (`bottom`) più la frequenza dei dati all'interno dell'istogramma (`n`).

Si costruisce poi un `array` e lo si traspone per poter ottenere le coordinate `XY` per costruire l'istogramma.

Inoltre il parametro `density = True` fa si che l'integrale nel range sia normalizzato a 1 costante cioè il risultato sia il valore della funzione densità di probabilità al bin. In questo modo si hanno rettangolini tutti uguali in modo da facilitare la comprensione dell'output. [5] [3]

- Istogramma delle velocità a partire da condizione iniziale arbitraria con 10000000 campioni

```
if __name__ == "__main__":
    M = 10000000
    rng = np.random.default_rng()
    vx = rng.normal(0.,1.,M)
    vy = rng.normal(0.,1.,M)
    vz = rng.normal(0.,1.,M)
    v = np.sqrt(vx**2 + vy**2 +vz**2)
    istogramma(v,M,colore='b',grandezza='velocità')
    plt.savefig('istogramma1.jpeg')
```

`numpy.random.default_rng()` è un generatore di numeri random e `numpy.random.normal(loc=0.0, scale=1.0, size=None)`[5], utilizzato per definire le componenti della velocità `vx`, `vy` e `vz`, prende numeri random dalla distribuzione normale Gaussiana ("curva a campana"). Calcolando poi la velocità `v` e inserendola in `istogramma`, con un numero di 10000000 campioni, si otterrà sempre l'andamento della distribuzione di Maxwell, come mostrato in *figura 2.*

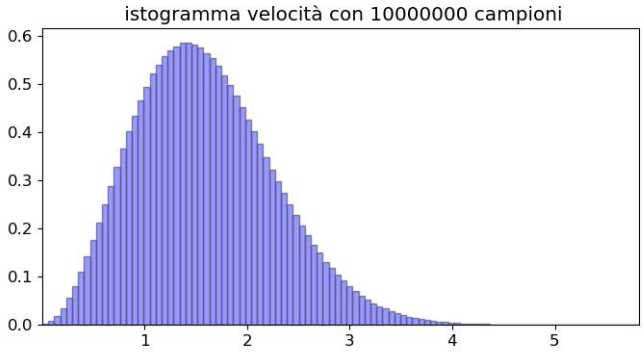


Figure 2: Istogramma delle velocità

Vediamo come con il Metropolis Monte Carlo, a partire da una condizione iniziale scelta arbitrariamente, la catena di Markov si evolve fino a generare configurazioni microscopiche per le velocità di N particelle campionate proporzionalmente alla distribuzione di Maxwell.

Possiamo quindi enunciare nuovamente l'algoritmo di Metropolis.

Esso consiste nell'iterare la procedura seguente, generando una sequenza ordinata di stati, scandita da un indice intero τ a cui ci si riferisce.

Al tempo τ la particella sia nello stato k-esimo descritto dalle componenti k-esime della velocità ed associata una certa probabilità p_k.

Si sceglie a caso una componente estraendo un valore η nel set (x, y, z) con probabilità uniforme pari a $1/3$.

Si genera un valore di prova della componente $v_\eta^{(j)}$ con uno spostamento aleatorio uniforme nell'intervallo $[-d, d)$ cioè si estrae un numero reale pseudoaleatorio ξ distribuito con probabilità uniforme in $[0, 1)$ e si ottiene:

$$v_\eta^{(j)} = v_\eta^{(k)} + (2\xi - 1)d \tag{27}$$

che è la componente dello stato j-esimo in cui le altre due componenti della velocità totale $\vec{v}^{\,(j)}$ non sono state modificate rispetto allo stato k-esimo.

Calcoliamo infine la variazione di energia $\Delta E_{kj} = \frac{1}{2}\left[(v_\eta^{(j)})^2 - (v_\eta^{(k)})^2\right]$ e il rapporto $p_j/pk = exp(-\Delta E_{kj})$.

Da qui si decide come scegliere la configurazione successiva nel seguente:

- se $\Delta E_{kj} \leq 0$ cioè $E_j \leq E_k$ si ha che $p_j/pk \geq 1$ e pertanto al tempo $\tau + 1$ si passa allo stato j;
- se $\Delta E_{kj} > 0$ cioè $E_j > E_k$ si ha che è stato estratto un numero pseudoaleatorio χ in $[0, 1)$ e se $\chi \geq p_j/pk$ allora si passa allo stato j al tempo $\tau + 1$; viceversa si rimane nello stato k.

3 Codice

3.1 `Class Ising`

Il programma è strutturato attorno alla `classe Ising`.

Le `classi` sono utilizzate per tenere insieme funzioni e attributi e impiegate per modellizzare un problema tramite la creazione di `oggetti` che appartengono alla `classe` e che hanno associati i parametri della stessa che possono essere facilmente chiamati all'interno di un programma.

In particolare la `classe Ising` (file `Isinga.py`) contiene tutte le funzioni che serviranno per descrivere l'evoluzione del modello di Ising.

```
import math
import random
import numpy as np
class Ising:

    def __init__ (self,temp, conf, N=100):

        self.N = N
        self.temp = temp
        self.J = 1
        self.conf = conf
        if (conf == 1):
            start = np.random.random((N,N))
            self.table = np.zeros((N,N))
            self.table[start>=0.25] = 1
            self.table[start<0.25] =-1
        elif (conf == 2):
            start = np.random.random((N,N))
            self.table = np.zeros((N,N))
            self.table[start>=0.75] = 1
            self.table[start<0.75] =-1
        else:
            self.table = np.random.choice([-1,1], size = (N,N))

        self.Et = 0
        self.Mt = 0
        self.Ct = 0

    def evoluzionemc(self):
        """
        Fa un passo di evoluzione (1 flip)
        """
        for i in range (self.N**2):
            riga=np.random.randint(0,self.N)
            colonna=np.random.randint(0,self.N)
            s = self.table[riga,colonna]
            primi = self.table[(riga-1)%self.N,colonna]+self.table[(riga+1)%self.N,colonna]+\
                          self.table[riga,(colonna-1)%self.N]+self.table[riga,(colonna+1)%self.N]
            delta_en = 2*s*primi

            if(self.flip_accettato(delta_en)):
                self.table[riga,colonna]*=-1
        return self.table

    def flip_accettato(self,delta_en):
        """
        Si passa la delta_energia per calcolare la
        probabilità di accettare la mossa
        """
        accettato = False
        exp = math.exp(-(delta_en/self.temp))
```

```
    if(delta_en<0):
        accettato = True
    else:
        x = random.random()
        if(exp>x):
            accettato = True
        else:
            accettato = False
    return accettato


def energia(self):
    """
    Calcola l'energia totale del sistema in una data configurazione
    """
    en = 0
    for i in range(len(self.table)):
        for j in range(len(self.table)):
            primi_vicini = self.table[(i-1)%self.N,j]+self.table[(i+1)%self.N,j]+\
                      self.table[i,(j-1)%self.N]+self.table[i,(j+1)%self.N]
            en -= self.table[i,j]*primi_vicini
    return en/4.

def magnetizzazione(self):
    """
    Calcola la magnetizzazione del sistema in una data configurazione
    """
    return np.sum(self.table)
```

Importiamo all'inizio `math` per fare calcoli, `random` perché servono numeri casuali e `numpy`.
La classe ha le seguenti:

- Funzione 1: `_init_(self, temp, conf, N = 100)` detto `costruttore`. `Self` è il riferimento a sé stesso, all'istanza e serve per prendere dall'interno della classe il nome dell'oggetto. `Temp` è una temperatura, `conf` fa riferimento alla configurazione iniziale del sistema e `N` è la dimensione della griglia del modello di Ising in 2D (quadrata). Impostiamo `N = 100` in modo tale che se non viene dato in input, il programma utilizzerà questo come default.

  La funzione crea una variabile con `self.N = N` che si chiama `N` e avrà un valore associato in base all'oggetto (ex. oggetto1.N o oggetto2.N) e appartiene alla classe. Si fa lo stesso con `temp`.

  Si da inoltre il valore a `J`, costante di accoppiamento degli spin e si crea una tabella che è la lattice del modello di Ising. Questa è un quadrato NxN ed è generata da un generatore di numeri casuali con in output valori -1 e $+1$. Questa è una scelta dovuta al fatto che il sistema è basato sugli `spin`. In particolare nella configurazione 1 (`conf == 1`) si costruisce una lattice di 75% di spin up e il 25% down viceversa con la configurazione 2 (`conf == 2`). Nella terza scelta si ha una lattice generata con numeri casuali nell'intervallo $[-1, 1]$.

  Crea inoltre due variabili vuote `Et`, `Mt` e `Ct` che sono rispettivamente energia, magnetizzazione e capacità.

- Funzione 2: `evoluzionemc(self)`. E' la funzione principale che quando chiamata fa avviare un ciclo in un range da 0 a N^2. Si definiscono le variabili riga e colonna generate da `numpy.random.randint`, quindi a valori interi. In questo modo si scelgono una riga e una colonna a caso e quindi un punto a caso nella matrice.

  Si calcola quindi la configurazione degli spin primi vicini (`primi`), la differenza di energia se scambiamo lo spin in quella posizione (flip).

- Funzione 3: `flip-accettato(self, delta_en)`. Questa riceve il valore della ΔE e ha come default `accettato = False`. Calcoliamo la probabilità secondo la formula esponenziale e attraverso `if` indichiamo le possibilità, come indicato dalla teoria. In return abbiamo `accettato` che sarà `True` o `False`, in base a quanto accaduto. Questo valore è quindi utilizzato dalla funzione `evoluzionemc(self)`.

  Si avrà che se il filp è accettato il punto della tabella che abbiamo considerato viene moltiplicato per -1 (`self.table[riga,colonna]*=-1`).

- Funzione 4: `energia(self)` calcola l'energia totale del sistema in una data configurazione secondo la formula della teoria (con i due cicli `for` che simulano la sommatoria) che prevede che ogni spin abbia 4 primi vicini. In

uscita si dovrà quindi dividere l'energia per 4 perché ogni punto è stato calcolato 4 volte. Nel calcolo dei primi vicini è utilizzata la condizione al bordo periodica espressa con `%self.N` dove il percentuale significa "modulo self.N" in modo che se usciamo da un lato rientriamo dall'altro.

- Funzione 5: `magnetizzazione(self)` calcola la magnetizzazione della configurazione che è la somma su tutta la tabella dei valori.

3.2 Codice `main`

Passiamo ora al `main`, il programma principale. La prima parte del programma, di seguito riportata, serve a studiare l'andamento della configurazione degli spin del sistema.

```
import numpy as np
import matplotlib.pyplot as plt
from Isinga import Ising
from time import process_time,time
%matplotlib widget

temp = 0.5
N = 50
obj = Ising(temp=temp,N=N, conf = 0)

f = plt.figure(figsize=(10, 10), dpi=80);
tic = process_time()
print_times = [1,10,100,200,500,999]

for i in range (1000):
    table = obj.evoluzionemc()
    if(i == print_times[0]):
        fig0 = f.add_subplot(231)
        fig0.imshow(table)
    if(i == print_times[1]):
        fig1 = f.add_subplot(232)
        fig1.imshow(table)
        f.show()
    if(i == print_times[2]):
        fig2 = f.add_subplot(233)
        fig2.imshow(table)
    if(i == print_times[3]):
        fig3 = f.add_subplot(234)
        fig3.imshow(table)
    if(i == print_times[4]):
        fig4 = f.add_subplot(235)
        fig4.imshow(table)
    if(i == print_times[5]):
        fig5 = f.add_subplot(236)
        fig5.imshow(table)
    f.show()
print(" cpu time for cycle t=", process_time()-tic)
```

Importiamo innanzitutto `numpy` [5], `matplotlib.pyplot` [4], libreria per creare qualsiasi tipo di visualizzazione in Python, `class Ising` e `process_time,time` che ci permette di calcolare il tempo impiegato dalla cpu per eseguire il programma.

Impostiamo `temp`, la temperatura e `N`, dimensione della griglia. Creiamo poi l'oggetto `obj` specificando il valore di temperatura e di `N` secondo la definizione del `costruttore` della `classe Ising`. In questo caso con `conf = 0` scegliamo che la griglia iniziale sia costruita a caso dal generatore di numeri random.

Creiamo una figura `f` utilizzando `plt.figure()` definendo la sua grandezza con `figsize` e la risoluzione in dots-per-inch (`dpi`) e si fa partire la misura del tempo.

`print_times` sono i valori di tempo di step ai quali si vuole una "foto" in output del sistema realizzata con `matplotlib.pyplot.imshow` [4].

Successivamente vi è un ciclo `for` che nel range 1000 definisce `table = obj.evoluzione()` secondo la chiamata alla classe Ising e ritorna la tabella degli spin.

Vi sono poi una serie di `if`. Questi ci dicono che se `i` coincide con un valore di `print_times`, allora tale situazione va stampata.

Infine calcoliamo energia, magnetizzazione e capacità.

```
import numpy as np
import matplotlib.pyplot as plt
from Isinga import Ising
from time import process_time,time
%matplotlib widget

N = 15
Npoints = 30
step_stabilizzazione = 1001
step_media = 1001
temp = np.linspace(1,5,Npoints,endpoint=False)
E = np.zeros(len(temp))
M = np.zeros(len(temp))
C = np.zeros(len(temp))
Xi = np.zeros(len(temp))
tic = process_time()

for i in range(len(temp)):
    Et = 0
    Mt = 0
    Et2 = 0
    Mt2 = 0
    obj = Ising(temp=temp[i],N=N, conf = 1)

    for j in range(step_stabilizzazione):
        obj.evoluzionemc()
    for j in range(step_media):
        obj.evoluzionemc()
        Et += obj.energia()
        Mt += obj.magnetizzazione()
        #valori medi quadri
        Et2 += obj.energia()**2
        Mt2 += obj.magnetizzazione()**2

    norm1 = ((N**2)*step_media)
    norm2 = ((N**2)*(step_media**2))
    E[i] = Et/norm1
    M[i] = Mt/norm1
    C[i] = ( Et2/norm1 - Et*Et/norm2 ) / ( temp [i] **2 )
    Xi[i] = (Mt2/norm1 - Mt*Mt/norm2 ) / ( temp [i] )

print(" cpu time for cycle t=", process_time()-tic)
```

Con `Npoints` indichiamo il numero di punti utilizzati per descrivere la curva; `step_stabilizzazione` indica quanti step di evoluzione deve compiere il sistema per stabilizzare gli spin, `step_media` su quanti step di evoluzione fare la media.

Si crea allora un vettore `temp` che avrà `Npoints` valori equidistanziati da 1 a 5 utilizzando `np.linespace(start, stop, num=...)` [5] (`endpoint = False` significa che il 5 è in realtà ignorato).

Inizialmente `E, M, C e Xi` sono vettori composti da zeri di dimensione `len(temp)` che poi conterranno i vari valori a differenti temperature.

Vi è un ciclo `for` di `i` nel range di `len(temp)` e si immettono i valori `Et, Mt, Et2, Mt2` inizialmente a zero. Si crea poi un oggetto `obj` con la temperatura corrente (`temp=temp[i]`) e viene ogni volta aggiornato durante il ciclo.

All'interno di questo ciclo ve ne è un altro che è un ciclo di stabilizzazione, `for j in range(step_stabilizzazione)`, durante il quale si fa evolvere il sistema fino alla stabilizzazione. Si procede con il ciclo della media, `for j in range(step_media)`, chiamiamo `obj.evoluzione()` cioè evolviamo di uno step alla volta e tutte le volte otteniamo l'energia e la aggiungiamo all'energia con `obj.energia()` e così via si calcolano le altre grandezza. I valori per ogni

sito sono normalizzati cioè dividendo la somma dei contributi di tutti i passi per il numero di mosse Monte Carlo totali.

4 Risultati

4.1 Configurazioni del modello di Ising

Si è variata la temperatura mantenendo costante N, dimensione della griglia. Per $T = 0.5$, sotto la temperatura critica T_c, dopo 1000 iterazioni gli spin sono tutti allineati nella configurazione di minima energia.

Quando invece si è in prossimità della temperatura critica, sono presenti ampie zone a spin up e a spin down che si mescolano tra loro.

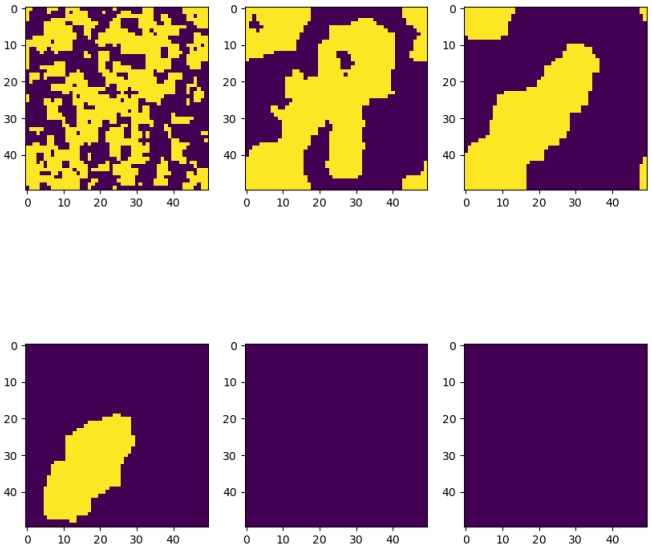


Figure 3: Similazione per $T = 0.5$, molto sotto la temperatura critica.

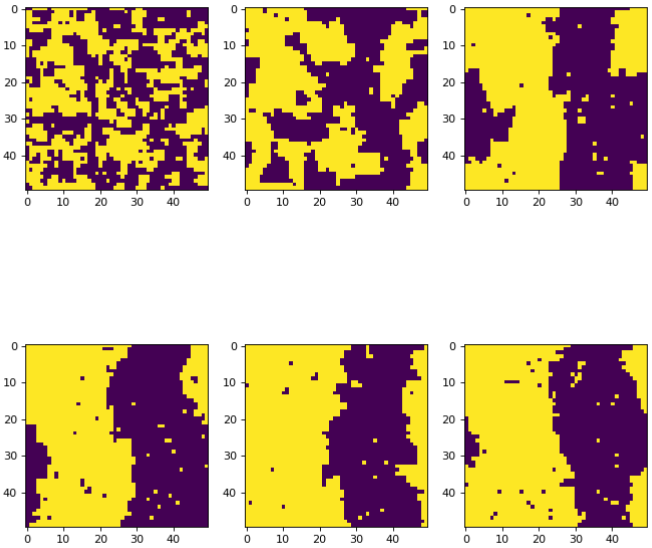


Figure 4: Similazione per $T = 1.7$, sotto ma vicino la temperatura critica.

Infine quando siamo a temperatura molto maggiore di quella critica si ha una randomizzazione degli spin dove statisticamente possiamo dire che metà spin sono up e l'altra metà down e lo stato di ogni spin individuale perde la correlazione con i suoi vicini. [1]

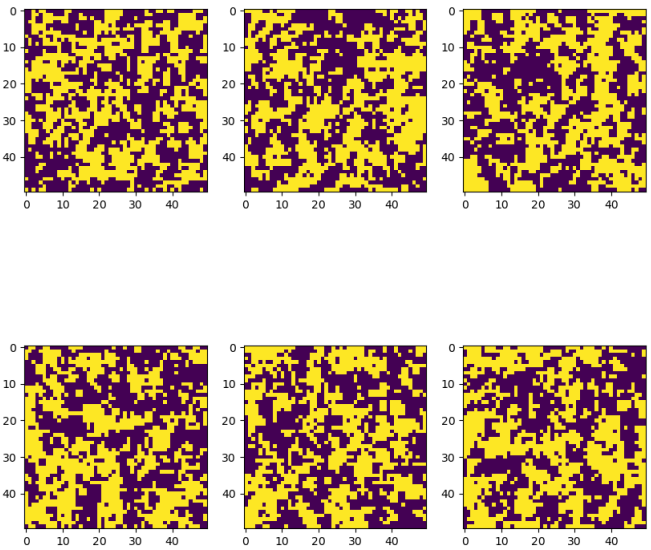


Figure 5: Similazione per $T = 3$, molto sotto la temperatura critica.

4.2 Medie d'ensemble

4.2.1 Caso configurazione 1

Possiamo plottare i risultati di energia, magnetizzazione, suscettività e capacità ottenuti per $N = 15$ e $Npoints = 30$ e configurazione 1 cioé scegliamo come stato iniziale che il 75% di spin sia up e il 25% down. In questo modo si ha una magnetizzazione positiva.

- Energia

```
#grafico energia
fig, ax = plt.subplots()
ax.plot(temp, E, linewidth=1.0, marker='.', color='g')
ax.set_xlabel('Temperatura')
ax.set_ylabel('Energia')
plt.show()
```

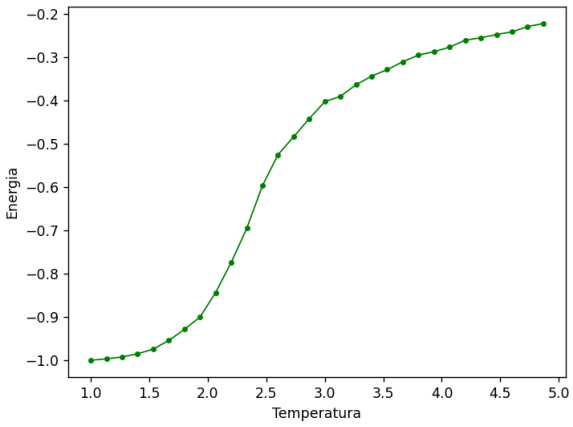


Figure 6: Grafico energia in funzione della temperatura.

Come si vede dal grafico, l'energia è minima alle basse temperature cioè il sistema si trova in una configurazione di spin allineati mentre cresce a maggiori temperature dove la configurazione è di disordine.

- Magnetizzazione

```
#grafico magnetizzazione
fig, ax = plt.subplots()
ax.plot(temp, M, linewidth=1.0, marker='.', color = 'hotpink')
ax.set_xlabel('Temperatura')
ax.set_ylabel('Magnetizzazione')
plt.show()
```

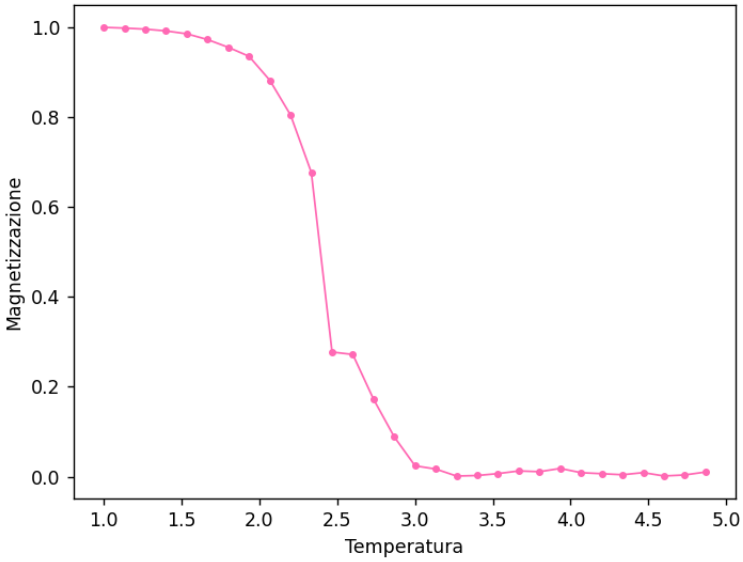


Figure 7: Grafico magnetizzazione in funzione della temperatura.

La magnetizzazione è pari a 1 a basse temperature dove appunto gli spin sono allineati mentre si annulla ad alte temperature.

- Capacità

```
#grafico capacità
fig, ax = plt.subplots()
ax.plot(temp, C, linewidth=1.0, marker='.', color = 'darkorange')
ax.set_xlabel('Temperatura')
ax.set_ylabel('Capacità')
plt.show()
```

La capacità è nulla a basse e alte temperature e presenta un picco attorno alla temperatura critica che è quella temperatura per la quale il sistema inizia ad avere delle fluttuazioni e quindi un aumento di energia.

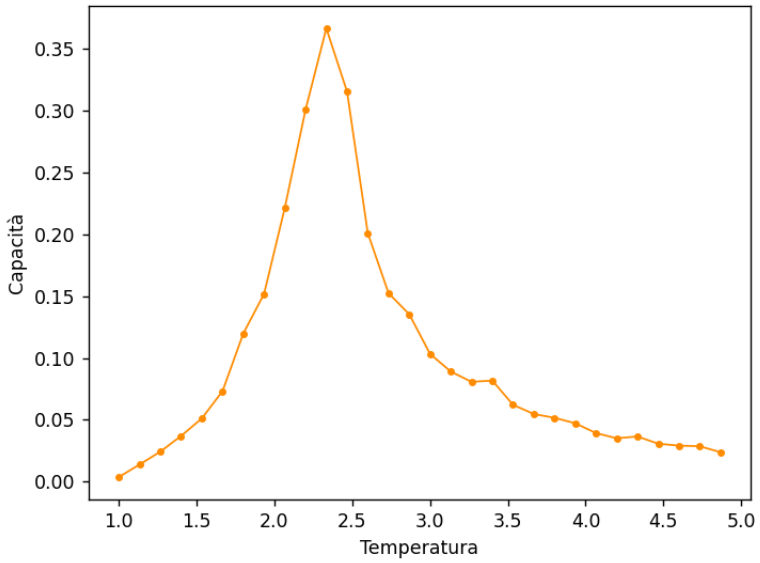


Figure 8: Grafico capacità in funzione della temperatura.

- Suscettività

```
#grafico suscettività
fig, ax = plt.subplots()
ax.plot(temp, Xi, linewidth=1.0, marker='.', color = 'steelblue')
ax.set_xlabel('Temperatura')
ax.set_ylabel('Suscettività')
plt.show()
```

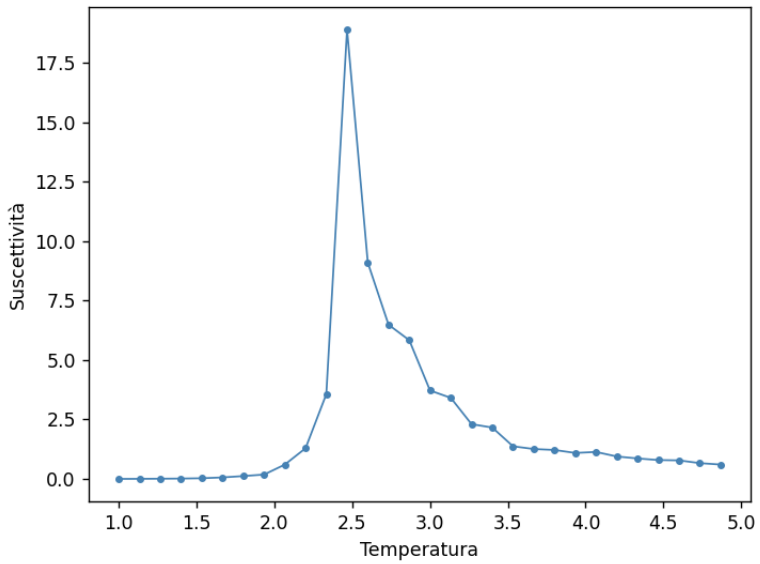


Figure 9: Grafico suscettività in funzione della temperatura.

Anche la suscettività presenta un picco quando secondo la teoria dovrebbe presentare un punto di discontinuità nella temperatura critica.

Di seguito si riportano i grafici per diversi valori di N per un confronto sulle grandezze calcolate.

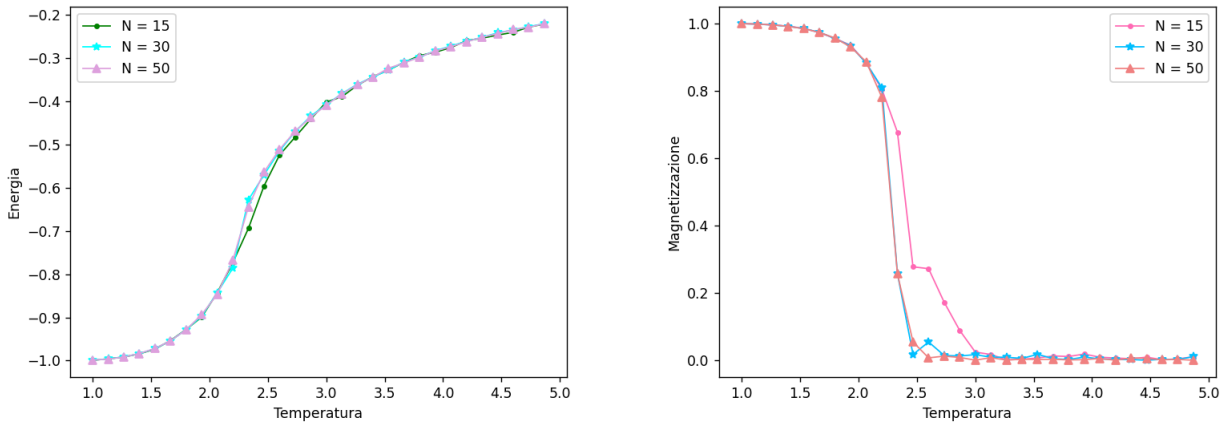


Figure 10: Andamenti a varie valori di N di energia e magnetizzazione.

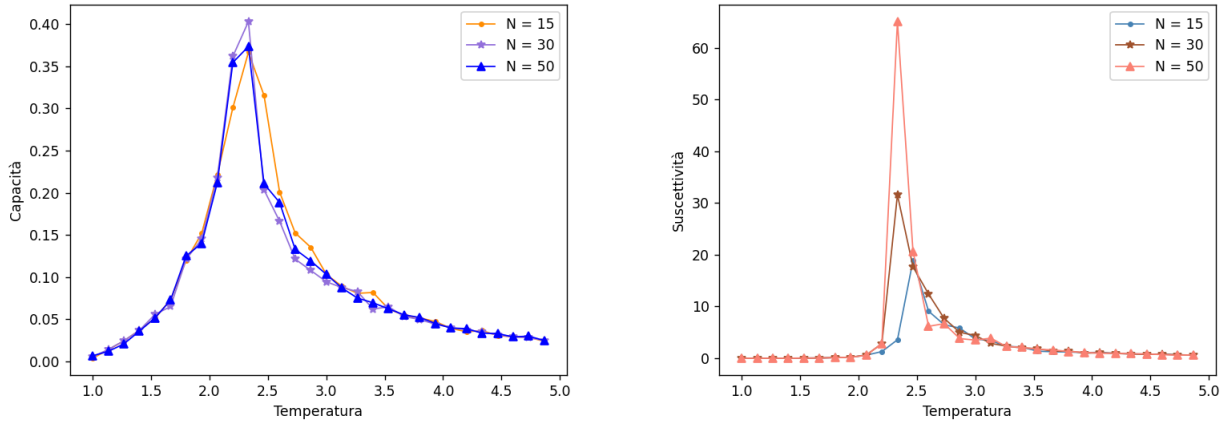


Figure 11: Andamenti a varie valori di N di capacità e suscettività.

Osserviamo che aumentando le dimensioni del lattice la suscettività ha il picco sempre più pronunciato.

4.2.2 Caso configurazione 2

Infine considerando la seconda configurazione che ha il 75% di spin down e il 25% up.

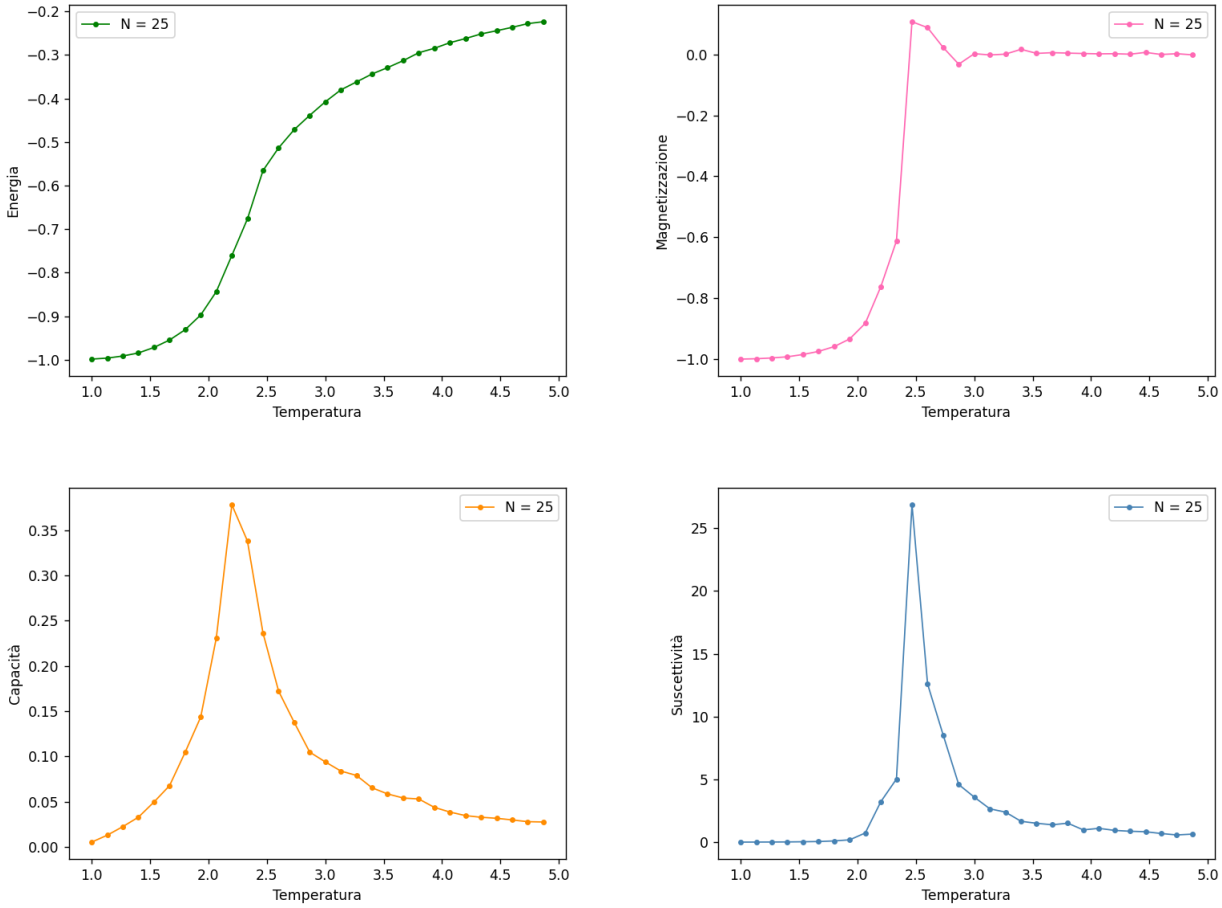


Figure 12: Andamenti di energia e magnetizzazione in alto ; capacità e suscettività in basso.

Si verifica che la magnetizzazione in questo caso è negativa mentre le altre grandezza hanno lo stesso andamento.

4.3 Calcolo temperatura critica

Come detto nel paragrafo 2.1.2, possiamo esprimere la magnetizzazione come una funzione della temperatura critica nel seguente: $M = (T - T_c)^{\alpha}$, $\alpha_{teorico} = 1/2$.

Vogliamo quindi determinare la temperatura critica ma anche il coefficiente α per verificare la validità della formula che prevede che sia $1/2$.

Facciamo quindi un fit dei dati della magnetizzazione ottenuti per $N = 40$ in configurazione 1 (magnetizzazione positiva).

Per fare ciò in Python importiamo la libreria `scipy.optimize` e in particolare la funzione `curve_fit` [6].

Pertanto dai nostri dati `x_data` e `y_data` e dalla definizione di una funzione modello `f` che dipende da parametri non noti γ, vogliamo determinare i parametri ottimali γ tali che $y = f(x, \gamma)$ meglio rappresenti i dati.

Per fare ciò si deve minimizzare $\sum(f(x_i, \gamma) - y_i)^2$ dove (x_i, y_i) appartengono a `x_data` e `y_data`.

Dopo aver salvato i valori di temperatura e magnetizzazione rispettivamente in due array `x_data` e `y_data` [5], definiamo la funzione modello `model_f`:

```
def model_f(x, Tc, alpha, c):
    return (c*(x-Tc)**alpha)
```

che ha come parametri $\gamma = (Tc, \alpha, c)$.

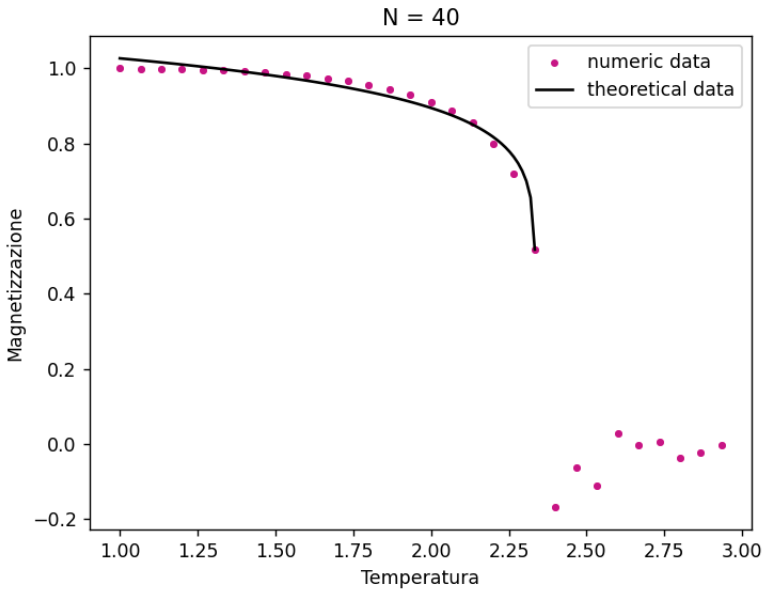
Successivamente utilizziamo `curve_fit` con un set tra cursori per specificare il set di dati in accordo con la teoria dove vale questa approssimazione e procediamo con il fitting.

[5] La temperatura è ottenuta con `temp = np.linspace(1,3,Npoints,endpoint=False)`, dove si è ristretto l'intervallo per avere la parte di grafico voluta.

```
popt, pcov = curve_fit(model_f, x_data2, y_data2, p0 = [2.2, 0.2, 1])
```

Otteniamo in output `popt`, il set di parametri ottimali, e `pcov`, la `covariance matrix`. Questa fornisce una stima dell'errore associato ai parametri lungo la diagonale (facendo la radice quadrata dei valori) e la correlazione tra questi, cioè il loro legame e come cambiano a seconda della variazione di uno o più.

Si ottengono:



Di seguito riportiamo i valori ottenuti dal `print` di `popt` e `pcov` selezionando i valori diagonali e facendone la radice:

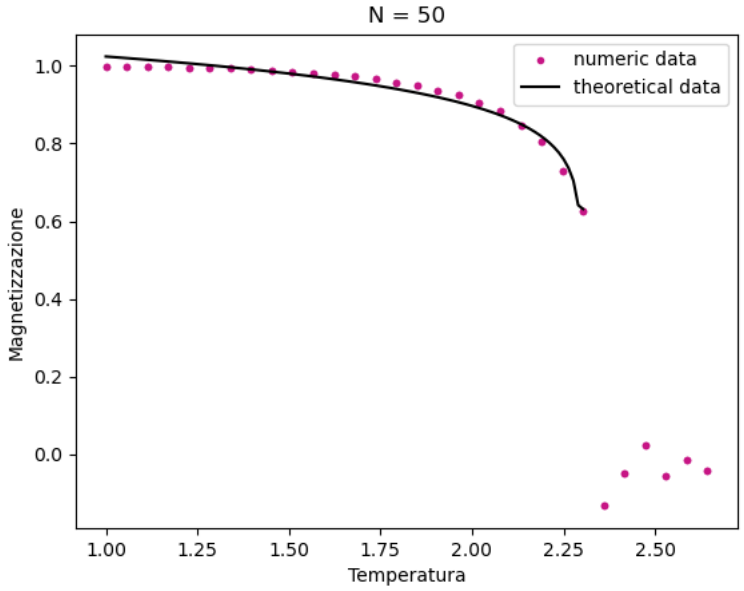
$$[2.33193586, 0.09927536, 0.99767451] = (Tc, \alpha, c) \tag{28}$$

$$[0.00069584, 0.0058238, 0.00521798] = (\delta Tc, \delta\alpha, \delta c) \tag{29}$$

Rifacendo il calcolo con $N = 50$ e

```
popt, pcov = curve_fit(model_f, x_data2, y_data2, p0 = [2.3, 0.1, 1])}
```

Si ottiene:

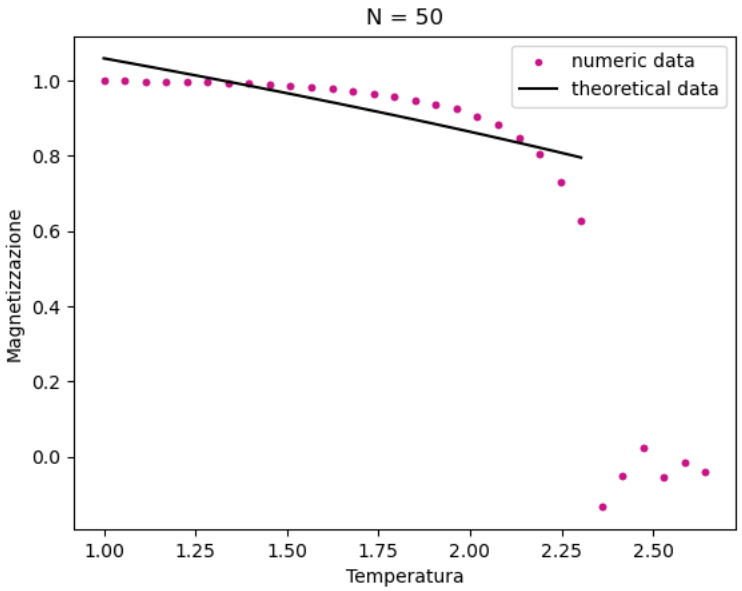


$$[2.29731567, 0.09007709, 1.00070402] = (Tc, \alpha, c) \tag{30}$$

$$[0.00200395, 0.0043007, 0.00439518] = (\delta Tc, \delta\alpha, \delta c) \tag{31}$$

Si vede subito che il fit ha errori bassi ma il coefficiente α non è $1/2$ come predetto dalla teoria in approssimazione di campo medio.

Se però proviamo a fornire noi tale coefficiente senza farlo calcolare al fit otteniamo il seguente grafico e come valore di temperatura critica $T_c = 3.99 \pm 0.32$.



Queste simulazioni ci fanno comprendere che la funzione utilizzata non rispecchia il vero comportamento del modello di Ising e che l'approssimazione a primi vicini va rivisitata.

5 Conclusioni

Le simulazioni condotte in approssimazione di campo medio mostrano come il modello di Ising presenta una transizione di fase continua da stato ordinato a stato disordinato. Al tempo stesso con l'algoritmo Metropolis in questa approssimazione abbiamo potuto verificarne l'esistenza e osservare come il sistema si evolve.

Possiamo pertanto concludere che in prima analisi l'appossimazione di campo medio è un metodo efficace per studiare questo sistema fisico ma per maggiori dettagli è preferibile ricorrere alle soluzioni esatte del problema proposte successivamente dai fisici teorici.

References

[1] Stephen J. Blundell and Katherine M. Blundell. *Concepts in thermal physics*. Oup Oxford, 2010.

[2] David Chandler. *Introduction to Modern Statistical Mechanics*. 1988.

[3] Mauro Ferrario e Guido Goldoni. *Dispense di laboratorio di fisica computazionale*. 2021.

[4] *Matplotlib documentation*. https://matplotlib.org/3.1.1/index.html. January 05, 2020.

[5] *Numpy documentation*. https://numpy.org/doc/stable/reference/. September 16, 2023.

[6] *SciPy documentation*. https://docs.scipy.org/doc/scipy/. September 27, 2023.