TÉCNICO LISBOA

Física Computacional
LEFT 2024/2025

Projeto 1

Data e modo de entrega:

domingo, 1 de junho de 2025, 23:59, Projetos no Fénix

O trabalho deverá ser submetido no fenix (projectos) como um único notebook `Jupyter`. O nome do notebook deve conter os números de aluno de todos os elementos do grupo. Sempre que apropriado, devem ser usados métodos implementados no `numpy` e `scipy`. A biblioteca `matplotlib` deve ser usada para a parte gráfica. O código deve ser documentado apropriadamente com comentários curtos para cada uma das suas partes. Cada parte do projecto deve ser explicada em células de `markdown`.

Modelo de Ising e Algoritmo de Metropolis-Hastings

Um dos modelos mais simples de domínios magnéticos clássicos foi desenvolvido por Ernst Ising em 1925, ficando conhecido como Modelo de Ising. Este modelo consiste num sistema de spins s numa rede discreta (*lattice*), onde cada spin pode assumir um de dois estados: $s = +1$, spin apontado para cima, ou $s = -1$, spin apontado para baixo.

Cada spin interage apenas com os seus vizinhos mais próximos e a energia de interação depende apenas da direção relativa entre os spins. A energia do sistema é descrita por

$$E \equiv -J \sum_{\langle i,j \rangle} s_i s_j - h \sum_i s_i, \tag{1}$$

em que J é a constante de acoplamento, s_i e s_j são os spins (-1 ou 1) de partículas vizinhas, e h é um campo magnético externo. A primeira soma é sobre todos os pares de spins vizinhos, e a segunda é sobre todos os spins individuais. A magnetização total da *lattice* é simplesmente a soma de todos os spins,

$$M \equiv -J \sum_i s_i. \tag{2}$$

Considera o Modelo de Ising a duas dimensões, isto é, numa rede bidimensional como mostra a Fig. 1.

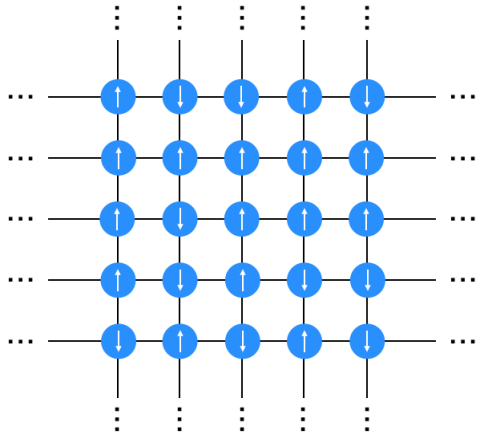

Figura 1: Exemplo de uma rede com spins $s_i = \{-1, 1\}$.

O modelo de Ising a duas dimensões possui uma transição de fase entre um estado ordenado e um estado desordenado. Esta transição de fase ocorre à temperatura crítica, T_c.

- Se $T < T_c$ o sistema está no estado ordenado. Os spins estão todos alinhados, e a magnetização do sistema é não nula
- Se $T > T_c$ o sistema está no estado desordenado. Os spins não se encontram alinhados, e a magnetização do sistema é nula (em equilíbrio).

No caso de um sistema infinito, a temperatura crítica é dada por

$$\frac{k_b T_c}{J} = \frac{2}{\ln(1+\sqrt{2})}. \tag{3}$$

No caso de um sistema finito, o valor da temperatura crítica pode ser ligeiramente diferente, porém quando $L \to +\infty$, temos que $T_c^L \to T_c$.

O modelo de Ising é muito estudado com recurso a métodos Monte-Carlo, nomeadamente, utilizando o algoritmo de Metropolis-Hastings. Este algoritmo é utilizado para obter uma sequência de amostras aleatórias a partir de uma distribuição de probabilidade, à semelhança do que foi feito nas aulas (semana 2). De uma forma simples, este algoritmo funciona em dois passos: primeiro, é proposta uma configuração para o sistema, com base na configuração anterior; em seguida, a nova configuração é ou não aceite dependendo da sua energia (que é utilizada como uma distribuição de probabilidades).

Em suma, para simular o modelo de Ising utilizando o algoritmo de Metropolis-Hastings, é necessário:

1. Escolher uma posição na rede aleatoriamente e trocar o valor do spin nessa posição, $s_i \leftarrow -s_i$
2. Calcular a diferença de energia ΔE entre a configuração anterior, X e a nova configuração, X^*
3. Aceitar a nova configuração caso $\Delta E < 0$ ou $e^{-\Delta E/k_b T} > u$, sendo u um número aleatório gerado a partir de uma distribuição uniforme, $u \sim U[0,1)$, isto é $X \leftarrow X^*$. Rejeitar caso contrário, mantendo a configuração original. Aceitar significa alterar o valor do spin da posição escolhida, e atualizar o valor da energia para a energia do novo sistema
4. Repetir N_{iter} vezes.

****

Implementa uma simulação do modelo de Ising utilizando o algoritmo de Metropolis-Hastings completando a classe **IsingModel** (ver esqueleto de código no fim deste documento) com os métodos em falta. **Os nomes, *inputs* e *returns* dos métodos não devem ser alterados**.

Utiliza uma rede 2D quadrada de lado L, e considera $J = 1$, $h = 0$ e $k_b = 1$. **Inicializa sempre todos os spins apontados para cima**, i.e., $s = +1$. Define um período de termalização N_{term} para garantir que, quando as medições são feitas, passadas N_{iter} iterações, o sistema já se encontra em equilíbrio térmico. Cada vez que corres a simulação para uma determinada temperatura, guarda o valor médio da energia e magnetização.

Uma dica para otimização do código: calcula a energia total apenas no início e atualiza o seu valor incrementalmente a cada passo aceite, $E_{new} = E_{old} + \Delta E$. Repara na Eq. (1) e pensa como podes facilmente calcular ΔE sem calcular a energia total de dois sistemas. O mesmo para a magnetização.

---

1) **Tempo de Termalização**

(a) **[3 pt]** De forma a determinar o tempo de termalização, τ_{term}, corre a simulação para $L \in [16, 32, 64, 128]$ e $T \in [1, 2, 3, 4]$ e, para cada condição, representa num gráfico a energia média, $e \equiv E/L^2$, em função do número da iteração N.

(b) **[3 pt]** A energia média em função da iteração pode ser aproximadamente dada pela seguinte função:

$$e(N) \equiv e_f + (e_0 - e_f)e^{-N/\tau_{\text{term}}} \tag{4}$$

onde $e_0 \equiv e_0(L)$ é a energia inicial da configuração, $e_f \equiv e_f(L, T)$ a energia final e $\tau_{\text{term}} \equiv \tau_{\text{term}}(L, T)$ o tempo de termalização. Faz um *fit* aos dados usando esta função de modo a determinar e_f e τ_{term} para cada combinação anterior de L e T. [Para fazer o *fit*, usa o método `curve_fit` do módulo `scipy.optimize`].

(c) **[3 pt]** Representa graficamente $\tau_{\text{term}}(L, T)$ e cria um interpolador para esta função.
Nota: Nos próximos exercícios, usa $N_{\text{term}}(L, T) \approx 4\tau_{\text{term}}(L, t)$ quando fores fazer medições.

2) **Determinação da Temperatura Crítica**

(a) **[3 pt]** Variando a temperatura de $T_{\min} = 1$ até $T_{\max} = 4$ em passos de $\Delta T = 0.2$, faz 1000000 medições da energia média, e, e magnetização média, $m \equiv M/L^2$ para cada valor da temperatura. Para o valor de cada quantidade, usa o valor médio das medições e para o erro associado, σ_e e σ_m, o desvio padrão das medições. Faz este estudo para sistemas de tamanhos $L \in [16, 32, 64, 128]$ e para cada L, representa graficamente e e m em função da temperatura.

(b) **[3 pt]** A derivada da energia média em ordem à temperatura é o capacidade calorífica,

$$c = \frac{\partial e}{\partial T}. \tag{5}$$

À temperatura crítica, T_c, a capacidade calorífica é máxima. Usando este facto, determina T_c usando uma derivada de primeira ordem com propagação do erro. Representa graficamente a derivada com as barras de erro associadas.
Nota: Usando um método de derivada central de primeira ordem, o erro da derivada $\sigma^i_{de/dT}$ na posição T^i é dado por:

$$\sigma^i_{de/dT} = \frac{\sqrt{(\sigma_e^{i+1})^2 + (\sigma_e^{i-1})^2}}{2h},$$

onde σ^i_e é o erro de $e(T^i)$.

(c) **[3 pt]** A capacidade calorífica também pode ser calculada tirando proveito da base estatistica do algoritmo de Metropolis-Hastings, isto é,

$$c = \left(\langle E^2 \rangle - \langle E \rangle^2\right) T^{-2}. \tag{6}$$

Edita o código de modo a calcular a média do quadrado da energia, e calcula c utilizando a Eq. (6). Compara com o resultado da alínea anterior.

(d) **[3 pt]** A capacidade calorífica c pode ser modelada a uma função do tipo

$$c \sim A \cdot \ln\left|\frac{T - T_c}{T}\right| + B, \tag{7}$$

em que T_c é a temperatura crítica. Faz um *fit* dos dados da capacidade calorífica (obtidos anteriormente) à função da Eq. (7), e determina os valores de A, B e T_c que melhor se ajustam

aos dados. [Para fazer o *fit*, usa o método `curve_fit` do módulo `scipy.optimize`].

3) **Determinação da Energia Livre de Helmholtz, F**

(a) **[3 pt]** Uma variável física que tem relevância no cálculo de outros parâmetros, como o trabalho por exemplo, é a energia livre de Helmholtz, que pode ser definida como

$$F(T) = T\left[\int_T^\infty \frac{E(T')}{T'^2}dT' - N\ln 2\right]. \tag{8}$$

Para permitir o cálculo numérico, visto que para $T \gg T_c$, a energia deverá tender para zero e esta aparece dividida por T^2, o integrando tende rapidamente para zero com o aumento da temperatura. Assim, Consideremos que $F(T = 10) \approx F(T = +\infty)$ permitindo simplificar a expressão para:

$$F(T) \approx T\left[\int_T^{10} \frac{E(T')}{T'^2}dT' - N\ln 2\right] \text{ se } T < 10 \text{ , caso contrário } F(T) \approx -TN\ln 2. \tag{9}$$

Para $L = 128$, varia a temperatura entre $T_{\min} = 0.1$ e $T_{\max} = 10$ com $\Delta T = 0.1$ de modo a poder reconstruir a energia livre no intervalo $T \in [0.1, 10]$. Usa o método de integração por trapézios e um número de medições adequado, i.e., suficientemente grande para ter estatística.

(b) **[3 pt]** Interpola a função $F(T)$ calculada anteriormente com uma spline cúbica. Usando o método da bissetriz, calcula a temperatura onde a energia livre é nula.

(c) **[3 pt]** Calcula a segunda derivada da função interpolada $F(T)$ usando o método adequado do `scipy`. O que acontece à volta da temperatura crítica? Consegues explicar esse comportamento? **Ajuda:** nota que $\partial F/\partial T \sim e$ e portanto $\partial^2 F/\partial T^2 \sim c$.

Esqueleto do código

Não alteres os *inputs* e *outputs* das funções, nem as linhas escritas.

A classe `tqdm` serve para apresentar barras de progresso, como por exemplo no loop da função `iter_monte_carlo`. Assim, podes ter uma noção de quanto as simulações demoram. Se não tiveres o módulo instalado, abre a pasta da cadeira pelo terminal e instala com o comando "`uv pip install tqdm`"(caso não utilizes o UV, podes instalar com "`pip install tqdm`").

```python
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from scipy.optimize import curve_fit
from scipy.integrate import cumulative_trapezoid
from scipy.interpolate import CubicSpline

# Classe do Model de Ising
class IsingModel:

    def __init__( self, L, T ):
        # parametros do modelo
        self.L = L
        self.T = T

        # array com os spins
        self.spins = np.ones( (L, L) )

        # arrays para guardar evolução das variáveis
        self.energies = []
        self.magnetizations = []

    def calc_ener_spin( self, i, j ):
        # calcular a energia de um spin
        # ...
        return energy

    def calc_ener( self ):
        # calcular a energia por spin do sistema
        # e = E / L^2
        # ...
        return energy

    def calc_mag( self ):
        # calcular a magnetização por spin do sistema
        # m = M / L^2
        # ...
        return mag

    def iter_monte_carlo( self, n_iter ):
        # iterar com o método de Metropolis Hastings
        for i in tqdm( range(n_iter), desc=f"L={self.L:6d}, T={self.T:8f}" ):
            # ...

    @property
    def energy(self):
        # usa para aceder ao array com as energias
        return np.array(self.energies)

    @property
    def magnetization(self):
        # usa para aceder ao array com as magnetizações
        return np.array(self.magnetizations)
```

Para usares a classe, podes adaptar o *script* seguinte:

```python
ising = IsingModel( L, T ) # criar um objecto da classe IsingModel
ising.iter_monte_carlo( n_iter ) # correr o algoritmo de Metropolis-Hastings
# aceder aos np.array que contêm a energia e a magnetização do sistema a cada iteração MC
energy = ising.energy
magnetization = ising.magnetization
```