Simulación del Modelo de Ising Kawasaki con Python

Jesús de la Oliva Iglesias

10/6/24

1 Introducción

Este documento describe un código en Python que implementa una simulación del modelo de Ising Kawasaki utilizando el método de Monte Carlo. La simulación calcula la magnetización y la energía del sistema en función de la temperatura y el tamaño de la malla.

2 Inicialización de Variables

El código comienza con la importación de las bibliotecas necesarias y la inicialización de las variables:

```
import numpy as np
from numpy import random
from numba import jit
import matplotlib.pyplot as plt
import time

# ====================================================
# ININICIAR VARIABLES

# Lado de la malla
lado_malla = np.full(5, 20).astype(np.int8)

# Temperatura
temperaturas = np.linspace(1, 5, 5).astype(np.float32)

# Numero de pasos Monte Carlo
pasos_monte = np.full(5, 50000).astype(np.int32)

# Calcular magnetizacion y energia
calcular_mag_ener = False
# ====================================================
```

Explicación:

- `import numpy as np`: Importa la biblioteca NumPy para operaciones numéricas.
- `from numpy import random`: Importa el submódulo random de NumPy para generar números aleatorios.
- `from numba import jit`: Importa el decorador jit de Numba para acelerar funciones mediante compilación Just-In-Time (JIT).
- `import matplotlib.pyplot as plt`: Importa Matplotlib para la visualización de datos.
- `import time`: Importa el módulo time para medir el tiempo de ejecución.

- `lado malla`: Define una matriz de tamaño 5 con todos los elementos iguales a 20, representando el tamaño de la malla.
- `temperaturas`: Genera un array de 5 elementos, linealmente espaciados entre 1 y 5, representando las diferentes temperaturas.
- `pasos monte`: Define una matriz de tamaño 5 con todos los elementos iguales a 50000, representando el número de pasos de Monte Carlo.
- `calcular mag ener`: Booleano que indica si se debe calcular la magnetización y la energía.

3 Generación de la Matriz Aleatoria

La siguiente sección del código define una función para generar una matriz aleatoria de estados de espín 1 y -1:

```
# Matriz aleatoria entre estado 1 y -1
@jit(nopython=True, fastmath=True)
def mtrz_aleatoria(M):
    matriz = 2 * np.random.randint(0, 2, size=(M, M)).astype(np.int8) - 1
    matriz[M-1] = np.ones(M).astype(np.int8)
    matriz[0] = np.ones(M).astype(np.int8)
    matriz[0] = -matriz[M-1]
    return matriz
```

Explicación:

- `@jit(nopython=True, fastmath=True)`: Decorador de Numba que compila la función para ejecutar sin Python, optimizando la velocidad.
- `mtrz aleatoria(M)`: Función que genera una matriz de tamaño `MxM` con valores aleatorios de 1 y -1.
- `np.random.randint(0, 2, size=(M, M))`: Genera una matriz `MxM` con valores enteros aleatorios entre 0 y 1.
- `2 * ... - 1`: Convierte los valores de la matriz a -1 y 1.
- `matriz[M-1] = np.ones(M)`: Establece la última fila de la matriz a 1.
- `matriz[0] = np.ones(M)`: Establece la primera fila de la matriz a 1.
- `matriz[0] = -matriz[M-1]`: Invierte el signo de la primera fila de la matriz.
- `return matriz`: Devuelve la matriz generada.

4 Condiciones de Contorno Periódicas

La siguiente sección define una función para aplicar condiciones de contorno periódicas a la matriz:

```
# Condiciones de contorno periodicas
@jit(nopython=True, fastmath=True)
def cond_contorno(M, i, j):
    if i == M-1:
        arriba = M-2
        abajo = M-1
    elif i == 0:
        arriba = 0
        abajo = 1
```

```
    else:
        arriba = i-1
        abajo = i+1

    if j == M-1:
        izquierda = j-1
        derecha = 0
    elif j == 0:
        izquierda = M-1
        derecha = 1
    else:
        izquierda = j-1
        derecha = j+1

    return izquierda, derecha, arriba, abajo
```

Explicación:

- `cond contorno(M, i, j)`: Función que calcula las posiciones de los vecinos de un elemento en una matriz `MxM` considerando condiciones de contorno periódicas.
- `if i == M-1` y `elif i == 0`: Manejan las condiciones de contorno para las filas (bordes superior e inferior).
- `if j == M-1` y `elif j == 0`: Manejan las condiciones de contorno para las columnas (bordes izquierdo y derecho).
- `return izquierda, derecha, arriba, abajo`: Devuelve las posiciones de los vecinos de un elemento.

5 Cálculo de la Matriz

Esta sección del código define una función para calcular la nueva matriz en función de la temperatura y las reglas del modelo de Ising:

```
# Calculo de la matriz
@jit(nopython=True, fastmath=True)
def calculo_matriz(matriz, M, T):
    cambio = True

    # Iteracion sobre la matriz
    i = np.random.randint(1, M-2)
    j = np.random.randint(0, M)

    # Eleccion de la pareja
    eje = np.random.randint(0, 2)
    if eje == 0:
        i_pareja = i + 1
        j_pareja = j
    else:
        i_pareja = i
        if j == M-1:
            j_pareja = 0
        else:
            j_pareja = j + 1

```

```
    if matriz[i, j] == matriz[i_pareja, j_pareja]:
        cambio = False
    else:
        izquierda, derecha, arriba, abajo = cond_contorno(M, i, j)
        izquierda_pareja, derecha_pareja, arriba_pareja, abajo_pareja =
            cond_contorno(M, i_pareja, j_pareja)

        # Calculo de la variacion de la energ a
        if eje == 0:
            delta_E = 2 * matriz[i, j] * (matriz[i, derecha] + matriz[i,
                izquierda] + matriz[arriba, j] - matriz[i_pareja,
                derecha_pareja] - matriz[i_pareja, izquierda_pareja] -
                matriz[abajo_pareja, j_pareja])
        else:
            delta_E = 2 * matriz[i, j] * (matriz[arriba, j] + matriz[abajo
                , j] + matriz[i, izquierda] - matriz[arriba_pareja,
                j_pareja] - matriz[abajo_pareja, j_pareja] - matriz[
                i_pareja, derecha_pareja])

        # Probabilidad de cambio
        p_0 = np.exp(-delta_E / T)

        # Evaluar probabilidad de cambio
        if p_0 > 1:
            p = 1
        else:
            p = p_0

        # Numero aleatorio para comparar con la probabilidad
        r = np.random.rand()

        # Comparar probabilidad para cambiar el spin
        if r < p:
            matriz[i, j], matriz[i_pareja, j_pareja] = matriz[i_pareja,
                j_pareja], matriz[i, j]

    return matriz, cambio
```

Explicación:

- `calculo matriz(matriz, M, T)`: Función que actualiza la matriz según las reglas del modelo de Ising.
- `i = np.random.randint(1, M-2)` y `j = np.random.randint(0, M)`: Selecciona una posición aleatoria en la matriz.
- `eje = np.random.randint(0, 2)`: Selecciona un eje (horizontal o vertical) para elegir la pareja.
- `if matriz[i, j] == matriz[i pareja, j pareja]`: Verifica si los spins de las posiciones seleccionadas son iguales.
- `delta E = 2 * matriz[i, j] * ...`: Calcula la variación de energía asociada al cambio de spin.
- `p 0 = np.exp(-delta E / T)`: Calcula la probabilidad de cambio basada en la variación de energía y la temperatura.
- `r = np.random.rand()`: Genera un número aleatorio para decidir si se realiza el cambio de spin.

- `if r < p`: Compara el número aleatorio con la probabilidad calculada para decidir si se realiza el cambio.
- `return matriz, cambio`: Devuelve la matriz actualizada y un indicador de cambio.

6 Secuencia de Ising

Esta sección define una función para realizar una secuencia de pasos de Monte Carlo en la simulación:

```
# Secuencia de Ising
@jit(nopython=True, fastmath=True)
def secuencia_isin(M, T, matriz, n, magnt_prom, E, m_cuadrado,
    calcular_mag_ener):
    # Matriz de Ising
    matriz, cambio = calculo_matriz(matriz, M, T)

    magnt_prom = 0
    magnt_prom_superior = 0
    magnt_prom_inferior = 0
    E = 0
    E_sup = 0
    E_inf = 0

    if calcular_mag_ener == True:
        r = 0
        if n % 100 == 0:
            # Calculo de la energia y magnetizacion promedio
            i = 0
            j = 0
            while i < M:
                j = 0
                while j < M:
                    izquierda, derecha, arriba, abajo = cond_contorno(M, i
                        , j)
                    if i < M/2:
                        magnt_prom_superior += matriz[i, j]
                        E_sup += matriz[i, j] * (matriz[derecha, j] +
                            matriz[i, abajo] + matriz[izquierda, j] +
                            matriz[i, arriba])
                        r += 1
                    else:
                        magnt_prom_inferior += matriz[i, j]
                        E_inf += matriz[i, j] * (matriz[derecha, j] +
                            matriz[i, abajo] + matriz[izquierda, j] +
                            matriz[i, arriba])
                    j += 1
                i += 1
            E_sup = -E_sup / 2
            E_inf = -E_inf / 2
            E = (E_sup + E_inf) / (m_cuadrado)
            magnt_prom_superior = magnt_prom_superior / r
            magnt_prom_inferior = magnt_prom_inferior / (m_cuadrado - r)
            magnt_prom = magnt_prom_superior + magnt_prom_inferior
```

```
        return magnt_prom, E, matriz, cambio, magnt_prom_superior,
            magnt_prom_inferior, E_sup, E_inf
    else:
        return magnt_prom, E, matriz, cambio, magnt_prom_superior,
            magnt_prom_inferior, E_sup, E_inf
```

Explicación:

- `secuencia isin(M, T, matriz, n, magnt prom, E, m cuadrado, calcular mag ener)`: Función que realiza una secuencia de pasos de Monte Carlo para actualizar la matriz de Ising.
- `matriz, cambio = calculo matriz(matriz, M, T)`: Actualiza la matriz de Ising.
- `if calcular mag ener == True`: Bloque que calcula la energía y magnetización si `calcular_mag_ener` es `True`.
- `if n % 100 == 0`: Cada 100 pasos, se realiza
- `if n % 100 == 0`: Cada 100 pasos, se realiza el cálculo de la energía y magnetización.
- `while i < M` y `while j < M`: Iteraciones sobre la matriz para calcular la energía y magnetización.
- `E_sup = -E_sup / 2` y `E_inf = -E_inf / 2`: Cálculo de la energía en las mitades superior e inferior de la matriz.
- `E = (E_sup + E_inf) / (m_cuadrado)`: Cálculo de la energía total.
- `magnt_prom = magnt_prom_superior + magnt_prom_inferior`: Cálculo de la magnetización promedio.
- `return magnt_prom, E, matriz, cambio, magnt_prom_superior, magnt_prom_inferior, E_sup, E_inf`: Devuelve los resultados de la secuencia de Ising.

7 Modelo de Ising

Esta sección del código define una función para ejecutar el modelo de Ising y almacenar los resultados en un archivo de datos:

```
# Matriz de Ising
def ising_model(M, T, N, calcular_mag_ener):
    # Variables
    k = 0
    n = 0

    magnetizaciones = []
    energias = []
    magnetizaciones_superior = []
    magnetizaciones_inferior = []
    energias_superior = []
    energias_inferior = []
    magnt_prom = 0
    magnt_prom_superior = 0
    magnt_prom_inferior = 0
    E = 0
    E_sup = 0
    E_inf = 0

    # Matriz de Ising
```

```
    matriz = mtrz_aleatoria(M)
    m_cuadrado = M**2

    # Archivo de datos
    with open('ising_data_tem_{0:.2f}_malla_{1}.dat'.format(T, M), 'w') as
        file:
        while n < N:
            k = 0
            while k < m_cuadrado:
                # Resultados
                if calcular_mag_ener == True:
                    magnt_prom, E, matriz, cambio, magnt_prom_superior,
                       magnt_prom_inferior, E_sup, E_inf = secuencia_isin(
                       M, T, matriz, n, magnt_prom, E, m_cuadrado,
                       calcular_mag_ener)
                    if n % 100 == 0:
                        magnetizaciones.append(magnt_prom)
                        energias.append(E)
                        magnetizaciones_inferior.append(
                           magnt_prom_inferior)
                        magnetizaciones_superior.append(
                           magnt_prom_superior)
                        energias_superior.append(E_sup)
                        energias_inferior.append(E_inf)
                else:
                    magnt_prom, E, matriz, cambio, magnt_prom_superior,
                       magnt_prom_inferior, E_sup, E_inf = secuencia_isin(
                       M, T, matriz, n, magnt_prom, E, m_cuadrado,
                       calcular_mag_ener)
                k += 1

            if n % 100 == 0:
                file.write('\n')
                np.savetxt(file, matriz, fmt='%d', delimiter=',')
            n += 1

    return energias, magnetizaciones, magnetizaciones_superior,
       magnetizaciones_inferior, E_sup, E_inf
```

Explicación:

- `ising_model(M, T, N, calcular_mag_ener)`: Función principal para ejecutar el modelo de Ising.
- `matriz = mtrz_aleatoria(M)`: Inicializa la matriz de Ising.
- `with open('ising_data_tem_0:.2f_malla_1.dat'.format(T, M), 'w') as file`: Abre un archivo para guardar los resultados.
- `while n < N`: Itera sobre el número de pasos de Monte Carlo.
- `while k < m_cuadrado`: Itera sobre los elementos de la matriz.
- `if calcular_mag_ener == True`: Bloque que calcula y almacena la energía y magnetización si `calcular_mag_ener` es `True`.
- `if n % 100 == 0`: Cada 100 pasos, guarda la matriz actual en el archivo.

- `return energias, magnetizaciones, magnetizaciones superior, magnetizaciones inferior, E sup, E inf`: Devuelve los resultados de la simulación.

8 Simulaciones de Monte Carlo

La última sección define una función para realizar simulaciones de Monte Carlo a diferentes temperaturas y tamaños de malla:

```
# Simulaciones de Monte Carlo distintas temperaturas y mallas
def simulaciones(lado_malla, temperaturas, pasos_monte, calcular_mag_ener):

    # Cantidad de archivos
    C = len(temperaturas)
    resultados = np.zeros((C, 2))

    for i in range(C):
        # Temperatura y lado de la malla
        T = temperaturas[i]
        M = lado_malla[i]
        N = pasos_monte[i]

        # Modelo y tiempo de ejecucion
        tiempo_0 = time.time()
        en, magn, magnetizaciones_superior, magnetizaciones_inferior,
            E_sup, E_inf = ising_model(M, T, N, calcular_mag_ener)
        tiempo_1 = time.time()

        tiempo = tiempo_1 - tiempo_0
        if calcular_mag_ener == True:
            magn = np.array(magn)
            en = np.array(en)
            magnetizaciones_inferior = np.array(magnetizaciones_inferior)
            magnetizaciones_superior = np.array(magnetizaciones_superior)
            E_inf = np.array(E_inf)
            E_sup = np.array(E_sup)
            energia_cuadra = en**2
            promedio_mag = np.mean(magn)
            media_eners = np.mean(en)
            promedio_energia = media_eners / (M**2)
            calor_especif = (np.mean(energia_cuadra) - media_eners**2) / (
                T * M)**2

            open('resultados_{0:.2f}_{1}.dat'.format(T, M), 'w').write(f'
                Magnetizacion promedio {promedio_mag} \n Energia promedio {
                promedio_energia} \n Calor especifico {calor_especif} \n
                Tiempo {tiempo}\n')

        # Guardar parametros de la simulacion
        resultados[i, 0] = T
        resultados[i, 1] = M

        print('Simulacion terminada para T = {0:.2f} y M = {1}'.format(T,
            M))
```

```
        print('Tiempo de ejecucion: {0:.2f} s'.format(tiempo))

    return magn, en, magnetizaciones_superior, magnetizaciones_inferior,
        E_sup, E_inf
```

Explicación:

- `simulaciones(lado_malla, temperaturas, pasos_monte, calcular_mag_ener)`: Función que realiza múltiples simulaciones de Monte Carlo.
- `C = len(temperaturas)`: Determina el número de simulaciones a realizar.
- `resultados = np.zeros((C, 2))`: Inicializa una matriz para almacenar los resultados.
- `for i in range(C)`: Itera sobre las distintas temperaturas y tamaños de malla.
- `T = temperaturas[i]`, `M = lado_malla[i]`, `N = pasos_monte[i]`: Asigna los valores de temperatura, tamaño de malla y número de pasos para cada simulación.
- `tiempo_0 = time.time()` y `tiempo_1 = time.time()`: Miden el tiempo de ejecución de la simulación.
- `if calcular_mag_ener == True`: Bloque que calcula y guarda los resultados si `calcular_mag_ener` es `True`.
- `open('resultados_0:.2f_1.dat'.format(T, M), 'w').write(...)`: Guarda los resultados en un archivo.
- `resultados[i, 0] = T` y `resultados[i, 1] = M`: Guarda la temperatura y
- `resultados[i, 0] = T` y `resultados[i, 1] = M`: Guarda la temperatura y el tamaño de la malla en la matriz de resultados.
- `print('Simulación terminada para T = {0:.2f} y M = {1}'.format(T, M))`: Imprime un mensaje indicando que la simulación ha terminado para una temperatura y tamaño de malla específicos.
- `print('Tiempo de ejecución: {0:.2f} s'.format(tiempo))`: Imprime el tiempo de ejecución de la simulación.
- `return magn, en, magnetizaciones_superior, magnetizaciones_inferior, E_sup, E_inf`: Devuelve los resultados de las simulaciones.

9 Visualización de Resultados

En esta sección se definen funciones para visualizar los resultados de las simulaciones:

```
# Graficos de resultados
def graficar_resultados(temperaturas, magnetizaciones, energias,
    magnetizaciones_superior, magnetizaciones_inferior, E_sup, E_inf):
    import matplotlib.pyplot as plt

    # Grafico de Magnetizacion
    plt.figure(figsize=(10, 5))
    plt.plot(temperaturas, magnetizaciones, 'o-', label='Magnetizacion
        Promedio')
    plt.plot(temperaturas, magnetizaciones_superior, 's-', label='
        Magnetizacion Superior')
    plt.plot(temperaturas, magnetizaciones_inferior, 'd-', label='
        Magnetizacion Inferior')
    plt.xlabel('Temperatura')
```

```
    plt.ylabel('Magnetizacion')
    plt.title('Magnetizacion Promedio vs Temperatura')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Grafico de Energia
    plt.figure(figsize=(10, 5))
    plt.plot(temperaturas, energias, 'o-', label='Energia Promedio')
    plt.plot(temperaturas, E_sup, 's-', label='Energia Superior')
    plt.plot(temperaturas, E_inf, 'd-', label='Energia Inferior')
    plt.xlabel('Temperatura')
    plt.ylabel('Energia')
    plt.title('Energia Promedio vs Temperatura')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Explicación:

- `graficar_resultados(...)`: Función que genera gráficos para visualizar los resultados de las simulaciones.
- `import matplotlib.pyplot as plt`: Importa la biblioteca `matplotlib` para generar gráficos.
- `plt.figure(figsize=(10, 5))`: Crea una nueva figura con un tamaño específico.
- `plt.plot(...)`: Dibuja líneas y puntos en el gráfico.
- `plt.xlabel('Temperatura')` y `plt.ylabel('Magnetización')`: Etiquetas para los ejes X e Y.
- `plt.title('Magnetización Promedio vs Temperatura')`: Título del gráfico.
- `plt.legend()`: Muestra una leyenda para el gráfico.
- `plt.grid(True)`: Añade una cuadrícula al gráfico.
- `plt.show()`: Muestra el gráfico.
- Se repite el mismo proceso para el gráfico de energía.

10 Resultados de las actividades propuestas

10.1 Evolución del modelo para diferentes temperaturas

Se ha realizado la simulación para un valor inicial de magnetización 0, es una malla de 120 por 120 y diferentes temperaturas.

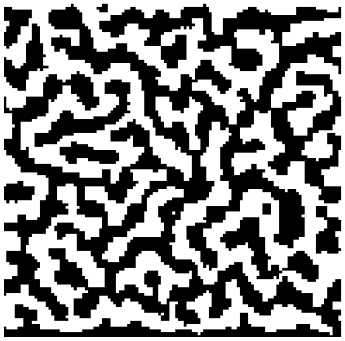

Figure 1: Imagen tras unos 10 pasos de montecarlo

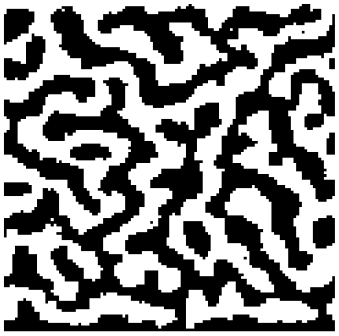

Figure 2: Imagen tras unos 10000 pasos de montecarlo

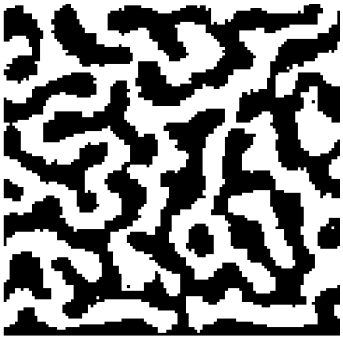

Figure 3: Imagen tras unos 30000 pasos de montecarlo

Figure 4: Malla 120 y temperatura 1 K

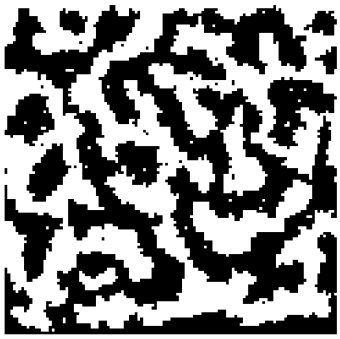

Figure 5: Imagen tras unos 10 pasos de montecarlo

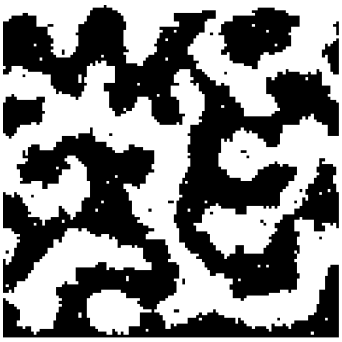

Figure 6: Imagen tras unos 10000 pasos de montecarlo

Figure 7: Imagen tras unos 30000 pasos de montecarlo

Figure 8: Malla 120 y temperatura 1.5 K

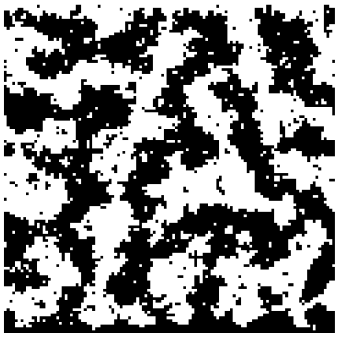

Figure 9: Imagen tras unos 10 pasos de montecarlo

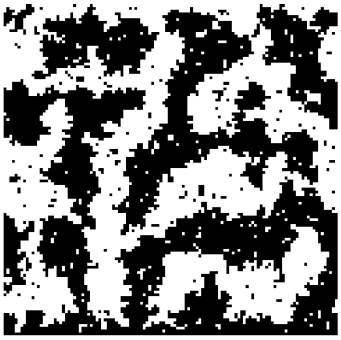

Figure 10: Imagen tras unos 10000 pasos de montecarlo

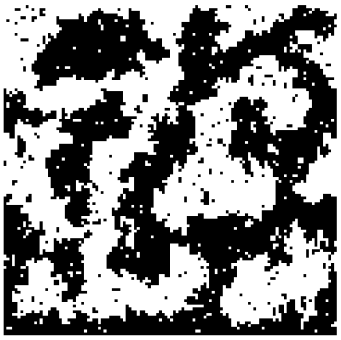

Figure 11: Imagen tras unos 30000 pasos de montecarlo

Figure 12: Malla 120 y temperatura 2 K

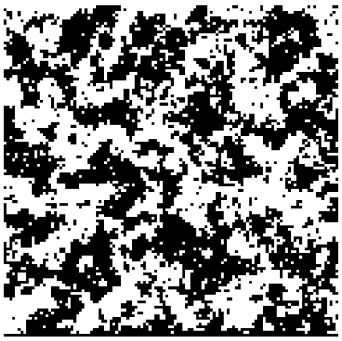

Figure 13: Imagen tras unos 10 pasos de montecarlo

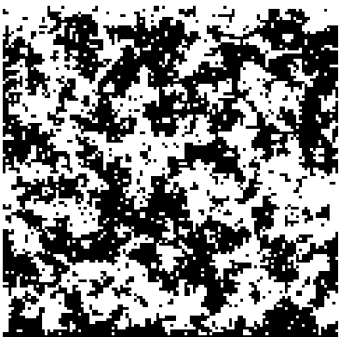

Figure 14: Imagen tras unos 10000 pasos de montecarlo

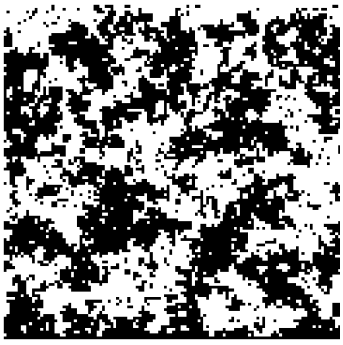

Figure 15: Imagen tras unos 30000 pasos de montecarlo

Figure 16: Malla 120 y temperatura 2.5 K

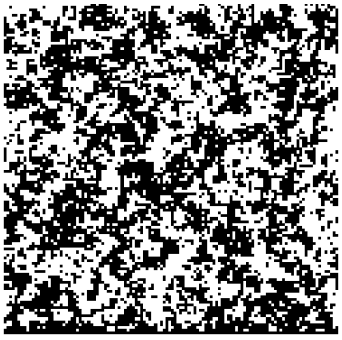

Figure 17: Imagen tras unos 10 pasos de montecarlo

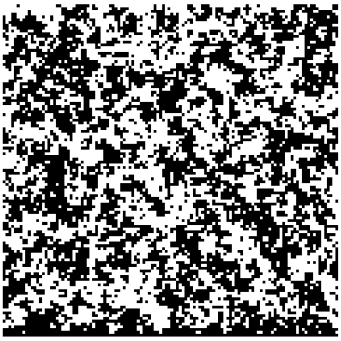

Figure 18: Imagen tras unos 10000 pasos de montecarlo

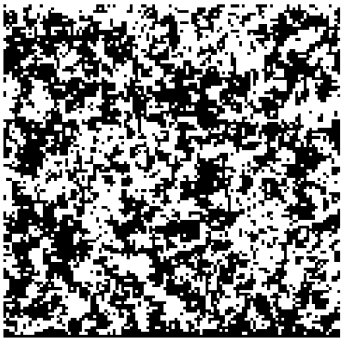

Figure 19: Imagen tras unos 30000 pasos de montecarlo

Figure 20: Malla 120 y temperatura 3 K

Podemos observar que a medida que incrementamos la temperatura la red está cada vez más desestructurada y se identifican menos patrones. Las pruebas con temperatura entre 1 y 1.5 podríamos suponer que se encuentra el punto con la malla más polarizada de todas, or tanto si dejáramos suficiente tiempo se acabarían distribuyendo los espines mitad y mitad polarizando completamente la muestra de la simulación.

10.2 Cálculo de las magnetizaciones opr dominios y energía del sistema

Para este apartado se han realizado diferentes simulaciones variando la temperatura del sistema y el tamaño del mismo para poder comparar los resultados. La magnetización total del sistema al comienzo de la simulación es 0.

Queremos comprobar la evolución de las magnetizaciones superior e inferior del sistema a la vez que la evolución de la energía total del sistema. Para calcular la magnetización se han sumado los valores de todos los espines correspondientes en cada caso. Para el caso de la energía se ha aplicado:

$$E(C) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} s_{i,j}(s_{i+1,j} + s_{i-1,j} + s_{i,j+1} + s_{i,j-1})$$

Para cada uno de las posiciones estudiadas y se han sumado todos los valores.

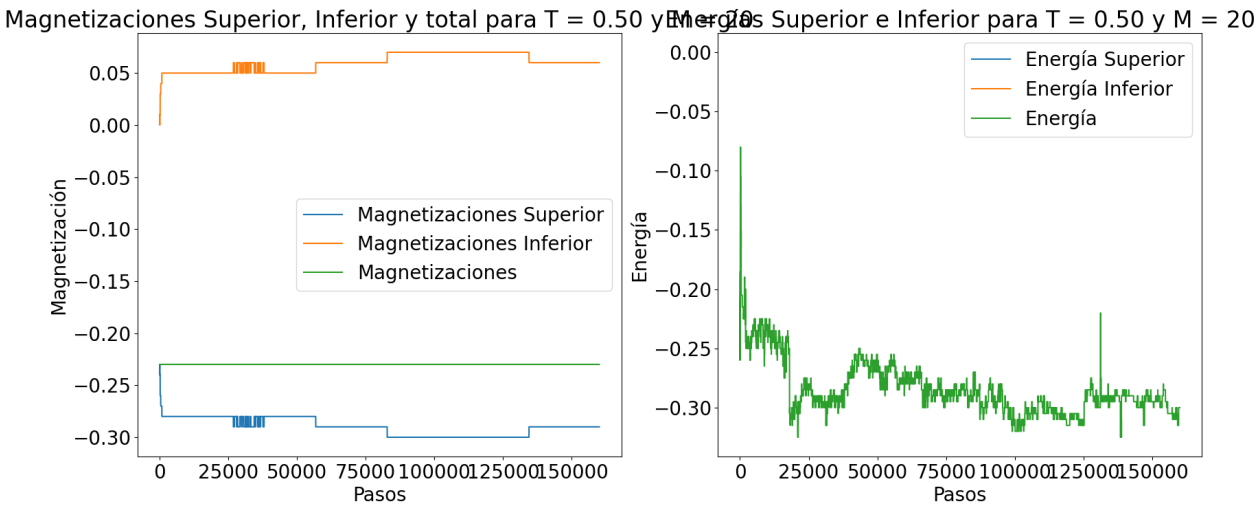

Figure 21: Temperatura 1 y 20 de malla

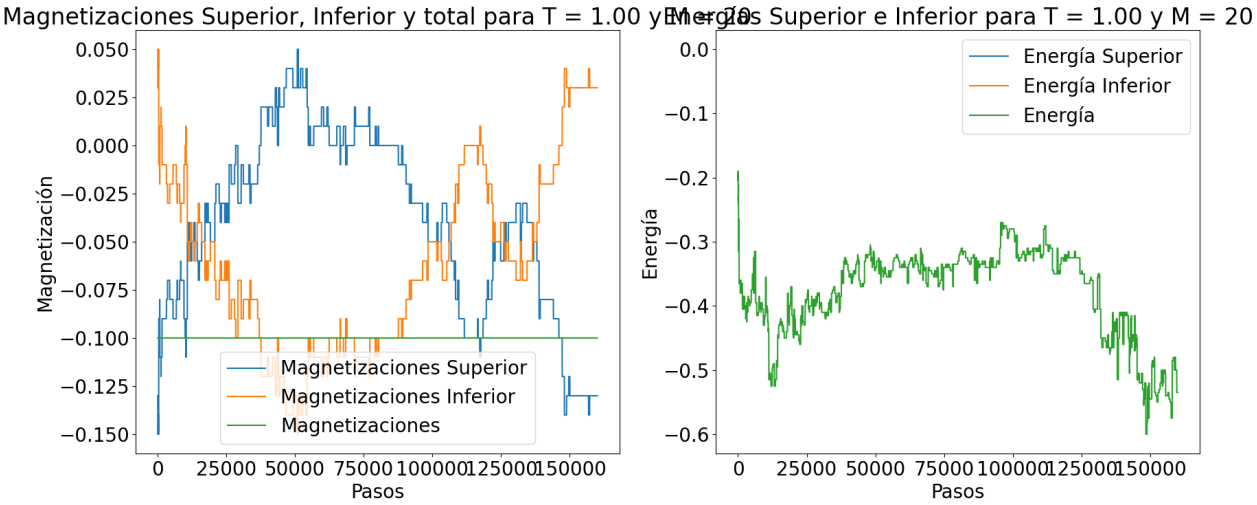

Figure 22: Temperatura 1.5 y 20 de malla

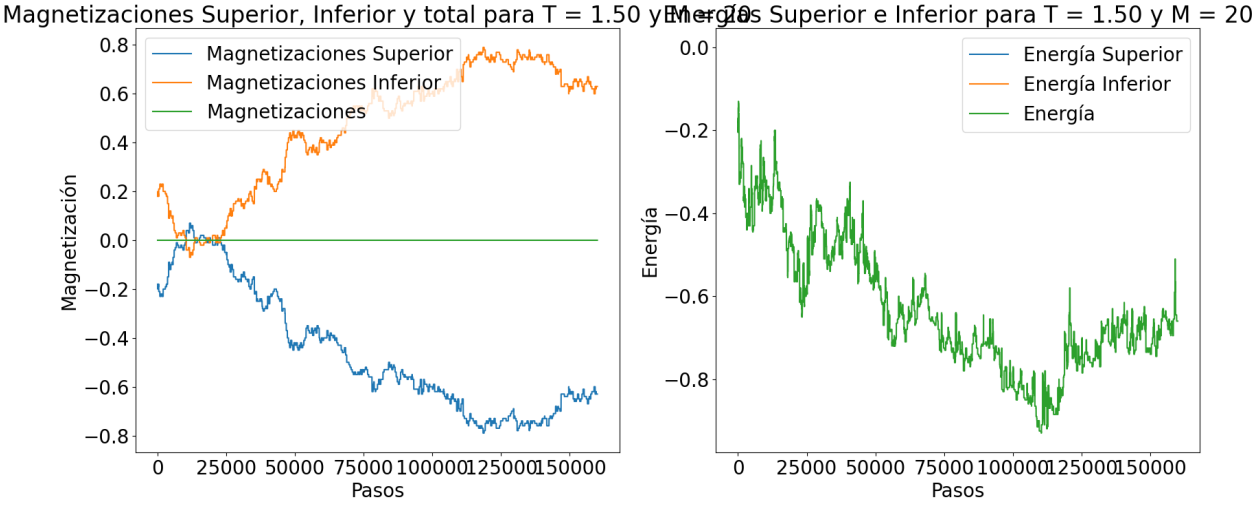

Figure 23: Temperatura 2 y 20 de malla

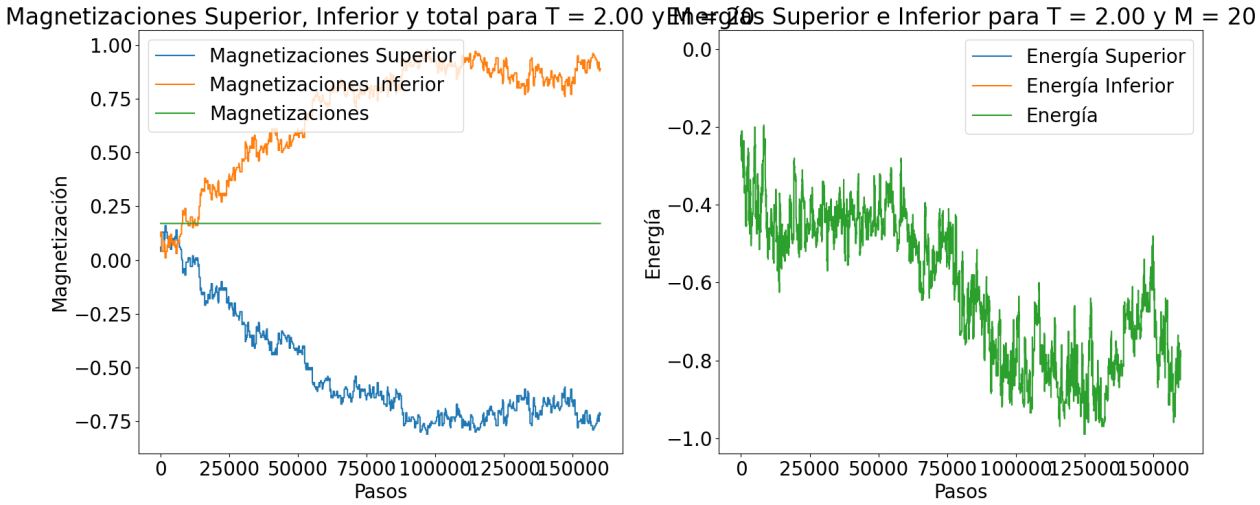

Figure 24: Temperatura 2.5 y 20 de malla

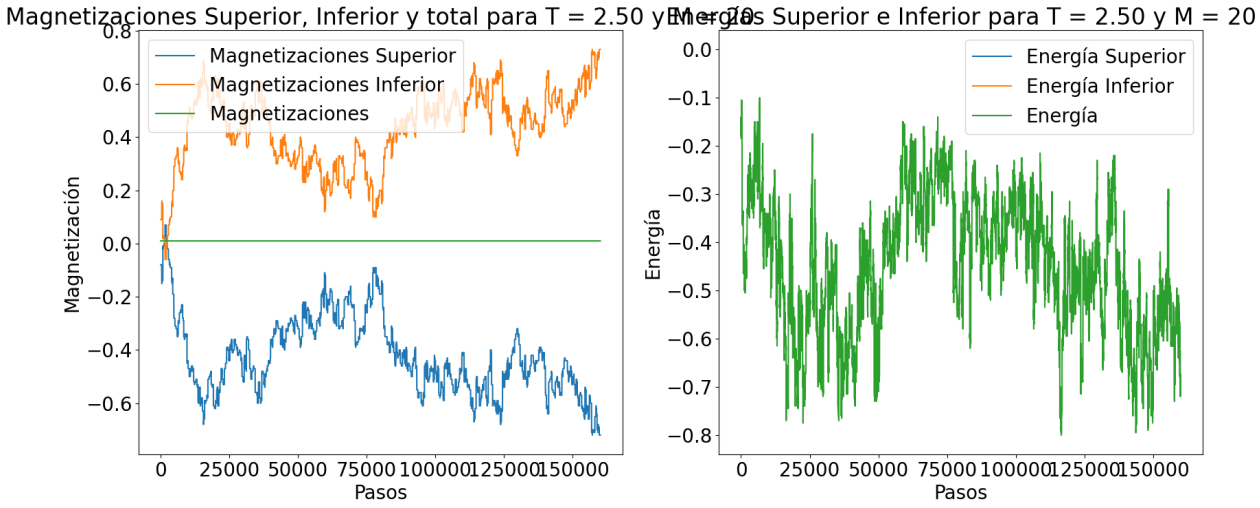

Figure 25: Temperatura 3 y 20 de malla

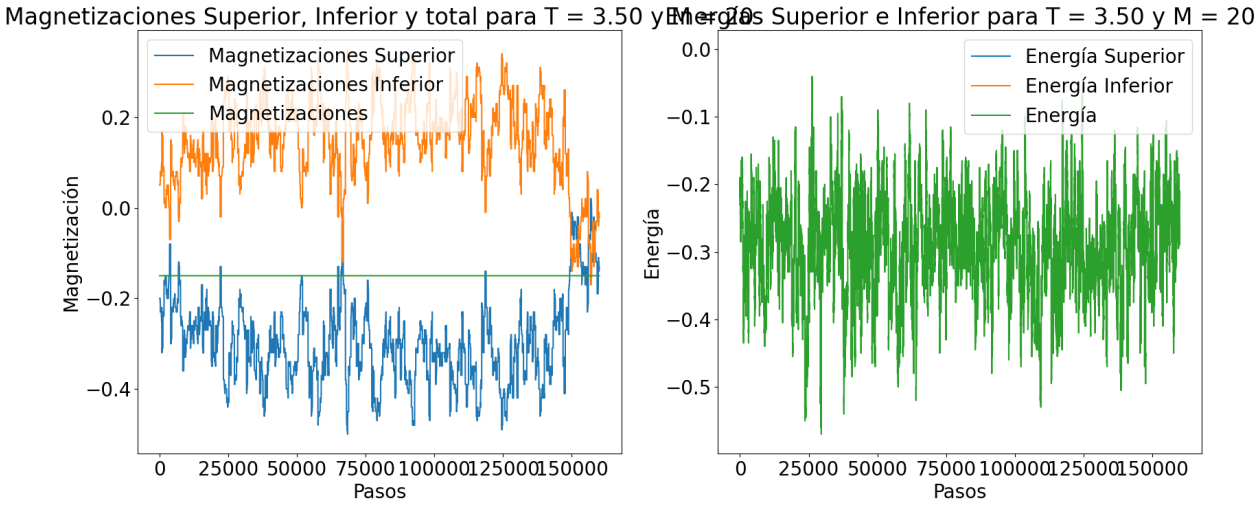

Figure 26: Temperatura 3.5 y 20 de malla

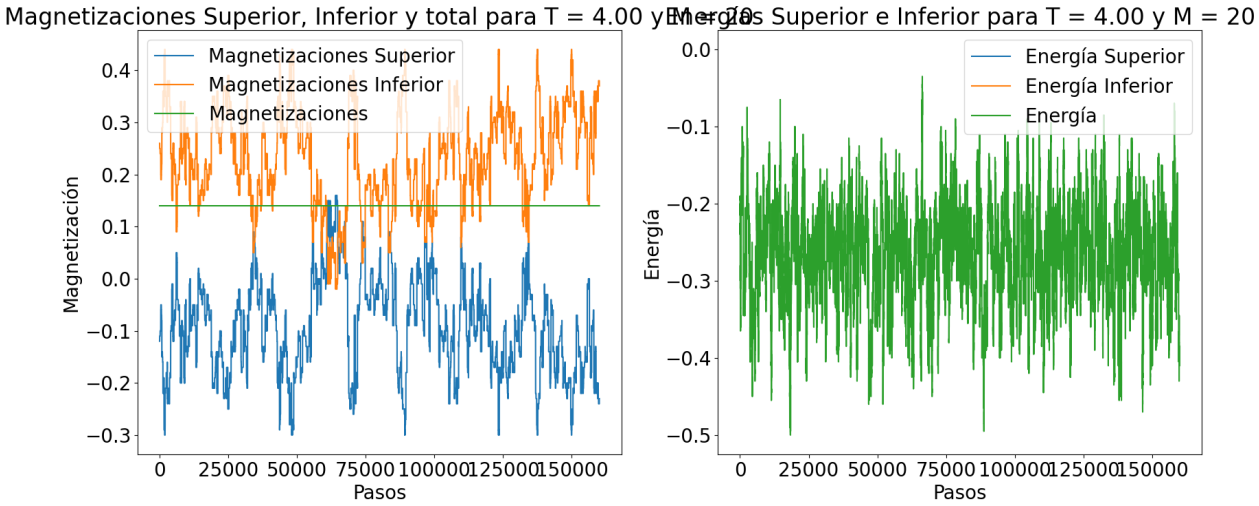

Figure 27: Temperatura 4 y 20 de malla

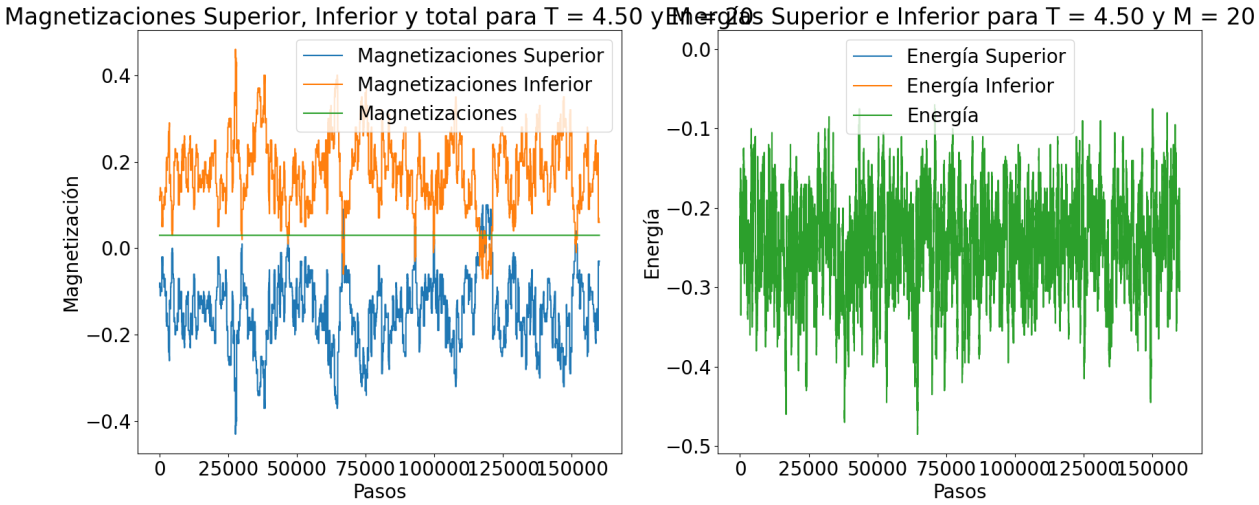

Figure 28: Temperatura 4.5 y 20 de malla

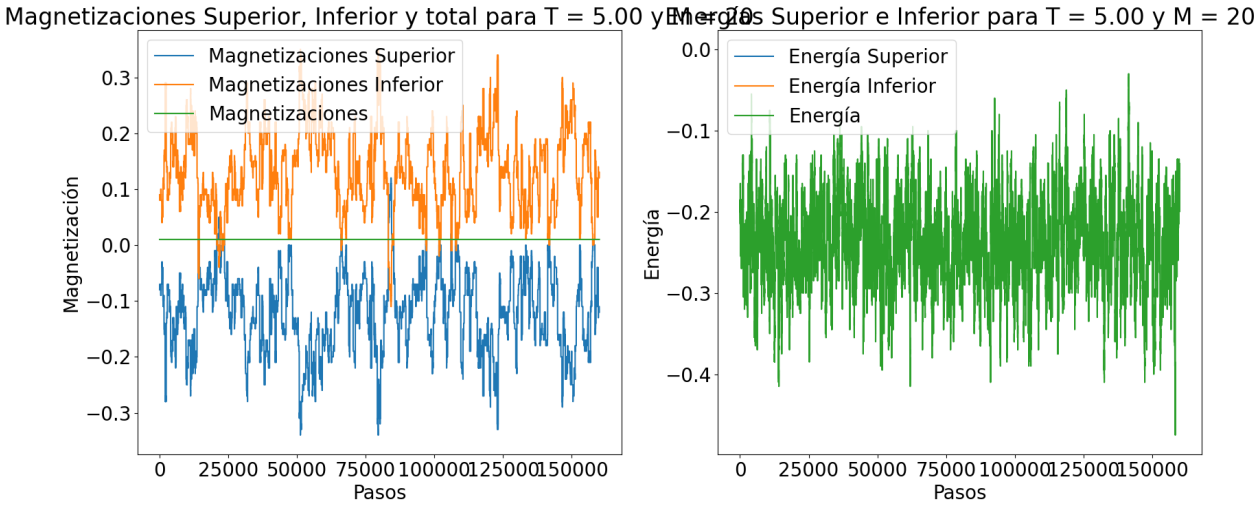

Figure 29: Temperatura 5 y 20 de malla

En este apartado podemos concluir que el sistema a bajas temperauras tiende a configurarse de forma que su energía sea la menor posible y la magnetización de cada mitad tienda al valor de spin igual a la de su borde fijo correspondiente. Si el sistema no tiene suficiente temperatura prácticamente no fluctúa una vez llegada la posición de equilibrio.

10.3 Calcular el calor específico en función de las variaciones de energía del sistema

En esta sección se ha procedido a calcular el calor específico de diferentes simulaciones haciendo uso de:

$$c_N = \frac{1}{N^2 T^2} \left[\langle E^2 \rangle - \langle E \rangle^2 \right] \tag{1}$$

Tabla de los valores calculados del calor específico para una malla fija 10 alrrededor de la temperatura crítica.

Tamaño de malla	**Temperatura**	**Calor específico**
10	1.1428572	$(1.85 \pm 0.01) \cdot 10^{-04}$
10	1.2857143	$(1.10 \pm 0.03) \cdot 10^{-04}$
10	1.4285715	$(6.11 \pm 0.03) \cdot 10^{-05}$
10	1.5714285	$(6.05 \pm 0.04) \cdot 10^{-05}$
10	1.6071428	$(5.49 \pm 0.02) \cdot 10^{-05}$
10	2.14	$(3.86 \pm 0.02) \cdot 10^{-05}$
10	2.25	$(3.84 \pm 0.04) \cdot 10^{-05}$
10	2.29	$(4.48 \pm 0.03) \cdot 10^{-05}$
10	2.43	$(3.20 \pm 0.03) \cdot 10^{-05}$
10	2.5	$(3.11 \pm 0.03) \cdot 10^{-05}$

Observamos como el calor específico es menor para temperaturas cercanas a 1.2 K eso nos puede indicar que ese punto está cercano a la temperatura crítica, ya que para esa temperatura el sistema es más susceptible a cambios por cualquier pequeño cambio que se realice en el mismo.

10.4 Comparativa entre Joel y Ordenador personal

Para lleavar a acabo la comparativa se han realizado la misma serie de simulaciones tres veces paar determinar el error en el tiempo de ejecución para cada uno de los cosasos estudiados.

En todos los casos se ha usado numba en el código, de no ser así el tiempo de ejecución es demasiado elevado para un estudio efectivo del mismo. Para una malla de más de 40 ya superaba la media hora de cálculo par una simulación.

El ordenador personal tiene un procesador 12th Gen Intel(R) Core(TM) i5-12400F con seis núcleas de frecuencia base de 2.50 GHz y de máxima de 4.4 GHz. En todos los casos se ha usado un sólo núcleo de cada uno de los equipos. En todos los casos el ordeador personal ha sido aproximadamente el doble de rápido que Joel.

En este caso tampoco sería aplicable la paralelización a menos que se asignara a cada núcle una ejecución completamente independiente y que realizara varias simultáneamente.

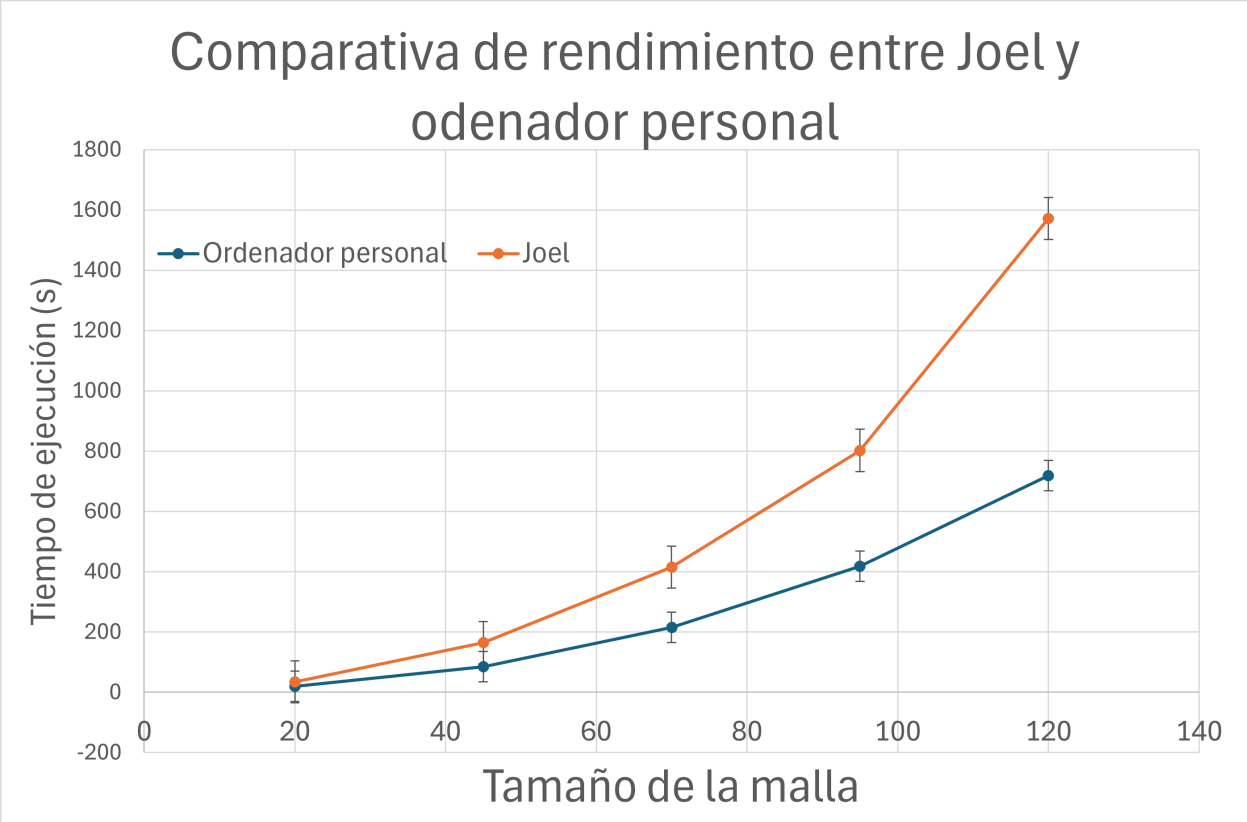

Figure 30: Tiempo de ejecución del programa en Joel y en al ordenador personal

11 Conclusiones

El modelo de Ising Kawasaki proporciona un marco teórico robusto para estudiar sistemas magnéticos mediante simulaciones computacionales. A través de este modelo, es posible calcular propiedades fundamentales como la magnetización, la energía y el calor específico del sistema. La magnetización representa la medida de la intensidad de la magnetización promedio del sistema, mientras que la energía refleja la suma de las interacciones entre los espines vecinos. Por otro lado, el calor específico ofrece información crucial sobre la capacidad del sistema para almacenar energía térmica.

Mediante simulaciones basadas en el modelo de Ising Kawasaki, es factible explorar cómo las variables clave como la temperatura y la densidad afectan estas propiedades. Esto no solo permite entender mejor el comportamiento de los materiales magnéticos a nivel microscópico, sino que también facilita la predicción y el diseño de sistemas con aplicaciones en diversas áreas, incluyendo la nanotecnología y la física de materiales.