Universidad Nacional De Colombia
Facultad de Ciencias

Tarea 3: Muestreo MCMC vs Simulación perfecta aplicadas al modelo de Ising

Juan Sebastián Rueda Segura
Sara Sofía Rivera Sanabria
Santiago Rocha Pachón

Freddy Rolando Hernández Romero
Cadenas de Markov
Bogotá, Colombia
26 de diciembre de 2025

Tabla de contenidos

Introducción

El modelo de Ising, propuesto originalmente por Wilhelm Lenz en 1920 y desarrollado por su estudiante Ernst Ising en 1925, es usado para capturar fenómenos de transición de fase en materiales ferromagnéticos. Este modelo describe el comportamiento de espines magnéticos en una red y ha encontrado aplicaciones que trascienden la física.

El objetivo de este trabajo es generar muestras representativas del modelo de Ising para diferentes tamaños de red y temperaturas, utilizando métodos de Monte Carlo basados en cadenas de Markov. Específicamente, implementamos los algoritmos de Propp-Wilson (PW en lo que resta del documento) y de Metropolis-Hastings (MH en lo que resta del documento) para explorar el espacio de configuraciones y estudiar las propiedades termodinámicas del sistema.

1.0.1. Definición del Modelo de Ising

Consideremos una red cuadrada $G = (V, E)$ de tamaño $K \times K$, donde V representa el conjunto de vértices y E el conjunto de aristas que conectan vecinos. El modelo de Ising asigna a cada vértice $i \in V$ una variable de espín $\eta_i \in \{-1, +1\}$, donde $+1$ y -1 representan dos posibles orientaciones del momento magnético atómico.

Una configuración particular del sistema se describe mediante el vector ξ. La energía de una configuración ξ está determinada por el Hamiltoniano

$$H(\eta) = -\sum_{x \sim y} \eta_x \eta_y \tag{1.1}$$

donde la suma se realiza sobre todos los pares de vecinos más cercanos. Esta formulación implica que cada enlace entre vecinos con espines iguales contribuye con -1 a la energía total, mientras que espines diferentes contribuyen con $+1$. En consecuencia, las configuraciones de menor energía corresponden a aquellas con mayor acuerdo entre vecinos.

En equilibrio térmico a temperatura inversa, la probabilidad de observar una configuración específica ξ está dada por la distribución de Gibbs

$$\pi_\beta(\eta) = \frac{1}{Z_\beta} e^{-\beta H(\eta)} \tag{1.2}$$

Esta distribución refleja el principio de Boltzmann: las configuraciones con menor energía tienen mayor probabilidad de ocurrir, y esta tendencia se acentúa a medida que disminuye la temperatura (aumenta β). En el caso límite $\beta = 0$ (temperatura infinita), todas las configuraciones son equiprobables, resultando en espines independientes e idénticamente distribuidos. En el límite

opuesto $\beta \to \infty$ (temperatura cero), toda la masa de probabilidad se concentra en las configuraciones de mínima energía, que corresponden a las configuraciones completamente ordenadas donde todos los espines apuntan en la misma dirección.

Una característica fundamental del modelo de Ising es la existencia de una transición de fase a una temperatura crítica. Para una red cuadrada infinita, Lars Onsager demostró analíticamente en 1944 que existe un valor crítico

$$\beta_c = \frac{1}{2}\log(1+\sqrt{2}) \approx 0.441 \tag{1.3}$$

que separa dos regímenes cualitativamente distintos. Para $\beta < \beta_c$ (temperatura alta), las fluctuaciones térmicas dominan y el sistema exhibe comportamiento desordenado. Para $\beta > \beta_c$ (temperatura baja), emerge orden ferromagnético espontáneo.

2.1. Metodología

El algoritmo opera mediante un proceso iterativo de propuesta y aceptación. En cada paso, se selecciona aleatoriamente un espín en la red y se considera la posibilidad de invertir su orientación. Esta propuesta se acepta o rechaza según un criterio que balancea la exploración del espacio de estados con la convergencia hacia regiones de alta probabilidad.

Implementamos condiciones de frontera periódicas en la red, identificando los bordes opuestos para eliminar efectos de borde artificiales que serían particularmente problemáticos en redes pequeñas. Bajo estas condiciones, cada espín tiene exactamente cuatro vecinos, independientemente de su posición en la red.

Tamaño de red: Consideramos once tamaños diferentes, desde $K = 10$ hasta $K = 20$.

Temperatura inversa: Muestreamos once valores de β desde 0.0 hasta 1.0 en incrementos de 0.1. Este rango cubre tanto el régimen de alta temperatura (desordenado) como el régimen de baja temperatura (ordenado), incluyendo al valor crítico $\beta_c \approx 0.441$.

Para cada par (K, β), generamos 100 muestras independientes. Cada muestra se obtiene ejecutando una cadena de Markov durante 100000 barridos completos de la red, comenzando desde una configuración inicial aleatoria.

2.2. Implementación

La implementación hace uso del compilador *just-in-time* Numba para Python, que traduce funciones numéricas a código máquina optimizado. Esto resulta crítico para el rendimiento, ya que el algoritmo de Metropolis-Hastings requiere ejecutar muchisimas actualizaciones de espines individuales.

El núcleo del algoritmo se encapsula en la función `_metropolis` (2.1), que realiza un barrido completo de la red. Esta función opera *in-situ* sobre la matriz de espines (es decir, alterándola por referencia). Para cada actualización, calcula el cambio de energía considerando únicamente los vecinos del espín seleccionado.

```python
@numba.jit(nopython=True)
def _metropolis(spin, L, beta):

    for _ in range(L ** 2):
        x, y = random.randint(0, L - 1), random.randint(0, L - 1)
        s = spin[x, y]

        xpp = (x + 1) if (x + 1) < L else 0 #pp es plus plus
        ypp = (y + 1) if (y + 1) < L else 0
        xnn = (x - 1) if (x - 1) >= 0 else (L - 1)
        ynn = (y - 1) if (y - 1) >= 0 else (L - 1)

        R = spin[xpp, y] + spin[x, ypp] + spin[xnn, y] + spin[x, ynn]
        dH = 2 * s * R

        if dH < 0:
            s = -s
        elif beta > 0 and np.random.rand() < np.exp(-beta * dH):
            s = -s

        spin[x, y] = s

```

Celda de código 2.1: `_metropolis`, núcleo de la implementación del algoritmo MH.

La función `get_samples_mcmc` (2.2) orquesta la generación de 100 muestras individuales, inicializando la red con espines aleatorios y aplicando 100000 de Metropolis-Hastings. Luego, calcula las propiedades termodinámicas de interés: energía promedio por espín y magnetización. Más adelante (4), se utilizará para guardar los resultados en un archivo csv.

```python
import os

from numba.typed import List
from numba import types, float64

def get_samples_mcmc(dimension: int, beta: float, mcmc_steps: int = 10000,

                     m_and_e_calc: bool = True, file_path: str = None, n_samples:

                     int = 100):

    data_list = []
    with tqdm(total=n_samples, desc="Generando muestras usando Metropolis") as pbar:
        for _ in range(n_samples):
            config = 2 * np.random.randint(2, size=(dimension, dimension)) - 1
            for _ in range(mcmc_steps):
                _metropolis(config, dimension, beta)
            pbar.update(1)

            if m_and_e_calc:
                data_list.append((config, calc_energy(config, dimension),
                calc_magnetization(config, dimension)))
            else:
                data_list.append(config)

    columns: list[str] = ['config', 'energy', 'magnetization'] if m_and_e_calc
                          else ['config']
    data_df: pd.DataFrame = pd.DataFrame(data_list, columns = columns)
    if file_path is not None:
        os.makedirs("H3/results", exist_ok=True)
        data_df.to_csv(f"H3/results/{file_path}")

    return data_df

```

Celda de código 2.2: `get_samples_mcmc`, función para tomar varias muestras del modelo de Ising usando MH.

La función principal `run_ising_experiment` (2.3) coordina la ejecución de todo el experimento, iterando sobre las combinaciones de parámetros y acumulando resultados. Para cada par (K, β), genera las 100 muestras requeridas y calcula tanto valores individuales como estadísticas agregadas, incluyendo promedios elemento por elemento de las configuraciones.

```python

def run_ising_experiment(K_range=(10, 21), betas=None, mcmc_steps=10000, seed=42):

    if betas is None:
        betas = [round(i * 0.1, 1) for i in range(11)]

    np.random.seed(seed)
    random.seed(seed)
    K_values = list(range(K_range[0], K_range[1]))

    results = {
        'K_values': K_values,
        'betas': betas,
        'energies': {},
        'magnetizations': {},
        'samples': {}
    }

    for K in K_values:
        results['energies'][K] = {}
        results['magnetizations'][K] = {}
        results['samples'][K] = {}

    total_iterations = len(K_values) * len(betas)

    with tqdm(total=total_iterations, desc="Generando muestras") as pbar:
        for K in K_values:
            for beta in betas:
                data_df = get_samples_mcmc(K, beta, mcmc_steps, n_samples=1)
                results['samples'][K][beta] = data_df.loc[0, 'config']
                results['energies'][K][beta] = data_df.loc[0, 'energy']
                results['magnetizations'][K][beta]=abs(data_df.loc[0, 'magnetization'])
                pbar.update(1)

    return results

```

Celda de código 2.3: `run_ising_experiment`, la función principal para obtener muestras usando MH y una función auxiliar de `visualize_results`.

Finalmente, para visualizar algunas de las muestras obtenidas, se creó la función `visualize_results`, cuya salida se muestra a continuación (2.1).

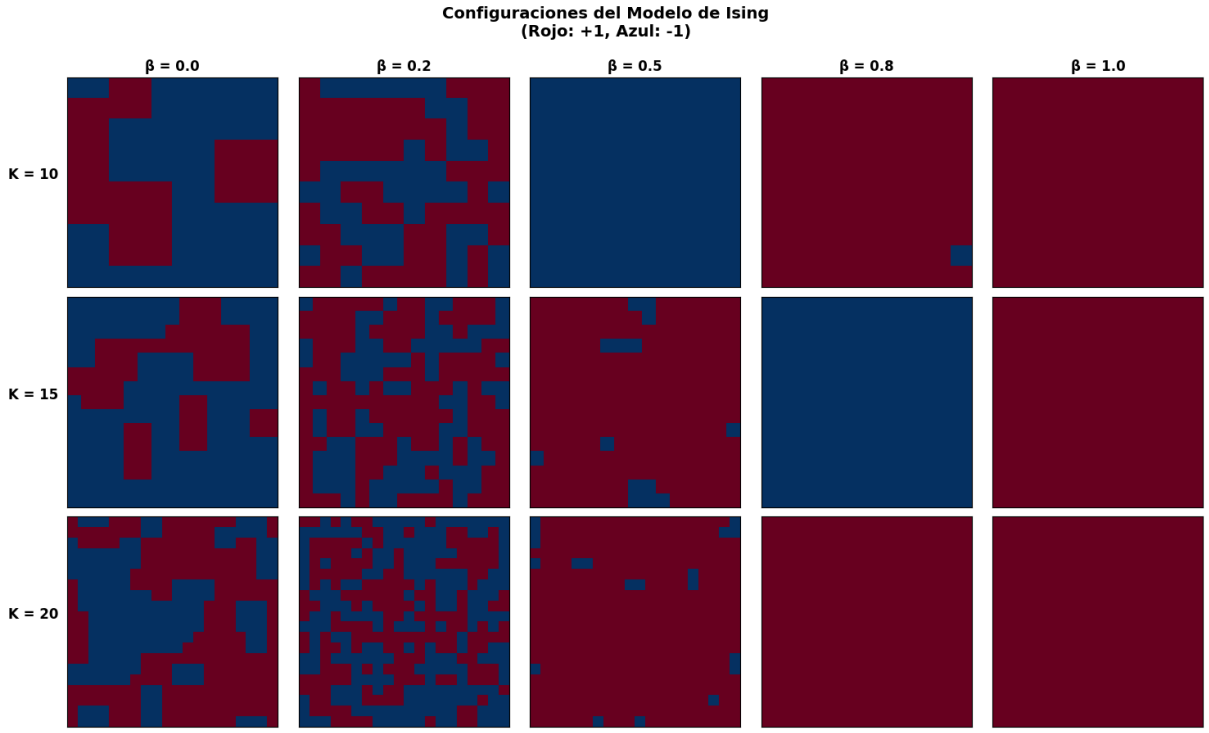

Figura 2.1: Algunas muestras obtenidas usando `run_ising_experiment` (MH) para diferentes valores de β y rejillas 10×10, 15×15 y 20×20.

Muestreo perfecto (Propp-Wilson)

3.1. Implementaciones

En el punto pasado utilizamos el Metrópolis-Hastings para generar muestras, pero en este caso debemos utilizar el Gibbs Sampler puesto que fue el que se comentó en clase al demostrarse que era monótono con la relación de orden que definiremos pronto.

Recordemos la idea del Sandwiching: Si podemos introducir un orden parcial en el conjunto de estados S y ese orden es respetado por la dinámica de la cadena, si corremos las cadenas que empiezan con las configuraciones maximal y minimal, garantizamos la coalescencia de todos los estados posibles al estar siempre en medio de estos dos estados. Definamos pues el orden parcial entre los estados:

Dado $G = (V, E)$, sea $x' \in V$. Definimos la relación de orden entre los estados, dependiendo de x' como:

$$\xi \leq_{x'} \eta \iff \xi_x \leq \eta_x \qquad \forall x \neq x'$$

Donde $\xi, \eta \in \{-1, 1\}^V$ y $\xi_x \leq \eta_x \forall x \neq x'$ denota que todas las entradas de la configuración ξ, exceptuando x, son menores que todas las correspondientes entradas de η.

Una vez definido el orden, creemos las clases auxiliares para poder comparar configuraciones y aplicar el muestreador de Gibbs rápidamente.

3.1.1. Clase Configuration y Gibbs Sampler

Como explicamos en el método anterior, utilizaremos la librería Numba, la cual nos permite hacer cálculos más eficientemente. Sin embargo, nos toca ajustar cierto tipo de datos para que se pueda usar apropiadamente.

```python
numba_spec = [
    ('beta', numba.float64),
    ('dimension', numba.int64),
    ('matrix', numba.float64[:, :]),
]

@numba.experimental.jitclass(numba_spec)
class Configuration:
    def __init__(self, beta, dimension):
        self.beta = beta
        self.dimension = dimension
        self.matrix = np.empty((dimension, dimension))
        for idx in np.ndindex((dimension, dimension)):
            self.matrix[idx] = 1.0 if np.random.rand() < 0.5 else -1.0

    def set_maximal(self):
        for i in range(self.dimension):
            for j in range(self.dimension):
                self.matrix[i, j] = 1

    def set_minimal(self):
        for i in range(self.dimension):
            for j in range(self.dimension):
                self.matrix[i, j] = -1

    def k_plus(self, x, y):
        num = 0
        xpp = (x + 1) if (x + 1) < self.dimension else 0 #*right
        ypp = (y + 1) if (y + 1) < self.dimension else 0 #*up
        xnn = (x - 1) if (x - 1) >= 0 else (self.dimension - 1) #*left
        ynn = (y - 1) if (y - 1) >= 0 else (self.dimension - 1) #*down

        if self.matrix[xpp, y] == 1:
            num += 1
        if self.matrix[x, ypp] == 1:
            num += 1
        if self.matrix[xnn, y] == 1:
            num += 1
        if self.matrix[x, ynn] == 1:
            num += 1
        return num

    def gibbs_sampler_step_from(self, N, random_list):
        x,y = random_list.V_list[N]
        k_plus = self.k_plus(x, y)
        k_minus = self.k_minus(x, y)

        self.matrix[x, y] = 1 if random_list.U_list[N] <
        (np.exp(2*self.beta*(k_plus - k_minus)))/
        (np.exp(2*self.beta*(k_plus - k_minus)) + 1) else -1

        return x, y, self.matrix[x, y]

    def __eq__(self, other: "Configuration"):
        return np.array_equal(self.matrix, other.matrix)
```

Celda de código 3.1: Clase Configuración

```python
import numpy as np
from numba import njit, int64, float64, types
from numba.experimental import jitclass
from numba.typed import List


#*For compatibility with Numba, U_list and V_list must be defined as
#*Numba types
pair_type = types.UniTuple(int64, 2)

random_lists_spec = [
    ('U_list', types.ListType(float64)),
    ('V_list', types.ListType(pair_type)),
]

@jitclass(random_lists_spec)
class RandomLists:
    def __init__(self):
        self.U_list = List.empty_list(float64)
        self.V_list = List.empty_list(pair_type)

    def extend(self, n_new, dimension):
        for _ in range(n_new):
            u = np.random.rand()
            x = np.random.randint(0, dimension)
            y = np.random.randint(0, dimension)
            self.U_list.append(u)
            self.V_list.append((x, y))
```

Celda de código 3.2: Clase RandomLists y configuraciones maximal y minimal

De acá cabe resaltar que para el Sampler estamos usando una lista de $U_i's$ y de Vértices escogidos uniformemente del intervalo (0,1) y de los vértices de la matriz, respectivamente. Crearemos una clase de lista aleatoria, ya que para cada paso vamos a necesitar guardar los U_i y V_i y a su vez generar 2^{N_i} - 2^{N_i-1} nuevos, ya que estamos corriendo la cadena desde -2^{N_i}, la secuencia que vimos en clase y que recomienda usar el libro. Esto para poder aplicar los mismos números aleatorios sobre los mismos vértices en ambas configuraciones, maximal y minimal.

3.1.2. Propp-Wilson

Ahora ya estamos en condiciones para correr el algoritmo Propp-Wilson y aplicar Sandwiching:

```python
@numba.jit(nopython=True)
def propp_wilson_sandwiching(beta, dimension):
    N_j_minus_1 = 0
    N_j = 1
    random_list = RandomLists()
    max_config: Configuration = Configuration(beta, dimension)
    max_config.set_maximal()
    min_config: Configuration = Configuration(beta, dimension)
    min_config.set_minimal()

    #Vamos a correr hasta hallar el N_j tal que ambas cadenas X^{top}_{-N_j} y
    #X^{bottom}_{-N_j} coalescan
    while True:
        # Añadimos los nuevos U_i y V_i necesarios para
        # N_j - N_j_minus_1 iteraciones del sampler
        random_list.extend(N_j - N_j_minus_1, dimension)
        for i in range(N_j - 1, 0, -1):
            max_config.gibbs_sampler_step_from(i, random_list)
            min_config.gibbs_sampler_step_from(i, random_list)

        if max_config == min_config:
            return N_j - i + 1, max_config.matrix
            # Ambas cadenas coalescieron en esta configuración


        #En N_j - N_j_minus_1 iteraciones no coalescieron, duplicamos N_j
        N_j_minus_1 = N_j
        N_j *= 2
        max_config = Configuration(beta, dimension)
        max_config.set_maximal()
        min_config = Configuration(beta, dimension)
        min_config.set_minimal()
```

Celda de código 3.3: Algoritmo Propp-Wilson con Sandwiching

3.2. Comentarios sobre la implementación

Seguimos el libro guía del curso (5) para basarnos en la implementación presentada anteriormente y nos parece relevante comentar ciertas anotaciones ahí descritas:

La primera, en el ejemplo 11.1, se comenta que hay un número relevante, llamado el valor crítico Onsager, que tiene valor: $\beta_c = \frac{1}{2}\log(1+\sqrt{2}) \approx 0.441$. Para valores menores a este se espera que la proporción de espín +1 tienda a un medio para menores valores de la rejilla, mientras que para valores mayores, un valor de espín va a ser dominante. La explicación física es que el material ferromagnético se magnetiza espontáneamente a temperaturas bajas más no a altas.

Para la implementación se siguió el ejemplo 11.2, donde adicionalmente se prueba que el Gibbs Sampler es monótono con la relación de orden antes descrita y por eso podemos aplicar el Sandwiching.

Lo más importante a la hora de resultados viene de la nota 43. Dice principalmente que el algoritmo es bueno si encuentra un tiempo de coalescencia razonable. Esto sucede cuando $\beta < \beta_c$, ya que comenta que el tiempo de coalescencia crece como un polinomio de bajo grado, dependiendo de m, mientras que para $\beta > \beta_c$, el tiempo crece exponencialmente en m. Concluye que toma una cantidad de tiempo "astronómica" y por ende se vuelve inútil. Dice que se podrían obtener muestras exactas para grandes valores de β, pero trasladándolo a otro problema, el Fortuin-Kasteleyn random-cluster model.

En el ejercicio 11.4, se pide hacer este apartado de la tarea, pero la nota 45 dice que no deberíamos estar sorprendidos si no termina para N_m grande y $\beta > \beta_c$. Lo que se observó es que esto en efecto pasa y se debe a la regla usada en el muestreador de Gibbs para actualizar a la configuración en cuestión:

$$X_{n+1}(x) = \begin{cases} 1 & \text{si } U_{n+1} < \frac{\exp(2\cdot\beta\cdot(k^+(x,\eta)-k^-(x,\eta)))}{\exp(2\cdot\beta\cdot(k^+(x,\eta)-k^-(x,\eta)))+1} \\ -1 & \text{en otro caso} \end{cases}$$

por ejemplo, cuando $\beta = 0.5$, $q := 2\cdot\beta\cdot(k^+(x,\eta) - k^-(x,\eta))$ es, a lo más (si $k^+(x,\eta) = 4$), alrededor de 54.4982. Ello implica que $p := \exp(2\cdot\beta\cdot(k^+(x,\eta)-k^-(x,\eta)))/(\exp(2\cdot\beta\cdot(k^+(x,\eta)-k^-(x,\eta))) + 1)$ es a lo más, alrededor de 0.9820. Similarmente, q es a lo menos (si $k^-(x,\eta) = 4$) 0.0183, por lo que p es a lo menos 0.0179.

Recordando que el algoritmo (con *sandwiching*) corre las cadenas de las configuraciones minimal y maximal, al maximizar p, que sería el caso de la maximal (todos los espines son 1), la probabilidad de que cualquiera de los vértices de grado 4 cambie su espín (a -1) es del $(1 - 0.9820) * 100\,\% = 1.80\,\%$. Análogamente, en el caso de la minimal, la probabilidad de que algún vértice cambie su espín a +1 es del $0.0183 * 100\,\% = 1.83\,\%$.

Por lo tanto, 'salir' de las configuraciones minimales y maximales ocurre con muy baja probabilidad (lo cual, físicamente, tiene sentido, pues la magnetización absoluta está maximizada y esto le agrada al sistema después de β_c). En suma, se asume que si el algoritmo converge con probabilidad 1, entonces da una muestra exacta de la distribución estacionaria objetivo. Sin embargo, esto no está garantizado, y no es fácil determinar cuándo la probabilidad es 0, pues cuando k se acerca a 20 y $\beta = 0.5$ (más sobre ello en 4.1), usualmente aborta, pero ha habido casos en los que se tarda 2 minutos. En otras palabras, no se conoce de un criterio, al menos con respecto al tiempo de ejecución, que dé una indicación sobre si el algoritmo va a converger o no.

Estimación del valor esperado de la magnetización

4.1. Metodología

Para dar una estimación del valor esperado de la magnetización, se usará la Ley de los grandes números, cuyo enunciado es así: dada una variable aleatoria Y, su media muestral (de $n \in \mathbb{N}^+$ muestras), definida como

$$Y_n := \frac{1}{n} \sum_{i=1}^{n} Y_i$$

donde Y_i es la muestra i-ésima de Y, se aproxima al valor esperado de Y, $\mathbb{E}[Y]$, cuando n tiende al infinito, es decir,

$$\lim_{n \to \infty} Y_n = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} Y_i = \mathbb{E}[Y]$$

además, usando la Ley de los grandes números, se sabe que tomando 30 o más muestras, se logra conseguir una aproximación 'suficientemente buena' del valor esperado.

Sin embargo, al intentar simular una muestra de una rejilla 10×10 con $\beta = 0.6$ en una máquina virtual proveída por Google Colab con 12.7 GB de memoria RAM, se llenó la memoria, se abortó la ejecución y no se logró completar la muestra. Lo mismo ocurrió en un portátil con 16 GB de RAM al intentar tomar 10 muestras con los mismos parámetros, y con otro (Mac) que utilizó hasta 21.55 GB de RAM virtual y 8 GB de RAM física para tomar 10 muestras también con los mismos parámetros.

Similarmente, en el computador con 16 GB de RAM, se lograron obtener 100 muestras de la rejilla 10×10 con un β de 0.5 tras una hora, y con el de $21.55 + 8$ GB, 100 muestras de 12×12 con $\beta = 0.5$ tardó más de dos horas.

No obstante, en el de 16 GB de RAM, no se lograron obtener más de dos muestras en un total de 20 cuando la rejilla era 20×20 y β era 0.5, y considerando que el aborto ocurría tras alrededor de 13 minutos, se decidió asumir las dos muestras obtenidas como representativas y no muestrear para tamaños de rejilla más grande, pues seguramente no se podrán muestrear cuando $\beta \geq 0.5$.

4.2. Implementación

La implementación de dicha estimación tiene varios componentes. En primer lugar, las muestras se ejecutan y se guardan en archivos *.csv* con la función `estimate_magnetization` (4.1). Una vez se ejecuta por primera vez para cada par k y β, la función trae los resultados de cada archivo usando la librería *pandas* y calcula la estimación para cada par k y β tanto con Metropolis-Hastings como con Propp-Wilson.

Esta función se ejecutó variando el tamaño de la rejilla y el β usado para tomar las muestras, teniendo que, con tamaños de rejilla 19×19 y 20×20 y con $\beta = 0.5$, tomar una única muestra por cada ejecución y apilarlas manualmente en un solo archivo para evitar que se abortara el proceso.

```
def estimate_magnetization(k: int, beta: float, mcmc_steps: int = 10000,
n_samples: int = 100):
    if not os.path.exists(f'H3/results/mcmc_{k}_{round(beta, 2)}.csv'):
        data_mcmc = get_samples_mcmc(dimension=k, beta=beta, mcmc_steps=mcmc_steps,
        m_and_e_calc=True, file_path=f'mcmc_{k}_{round(beta, 2)}.csv',
        n_samples=n_samples)
    else:
        data_mcmc = pd.read_csv(f'H3/results/mcmc_{k}_{round(beta, 2)}.csv')
    if not os.path.exists(f'H3/results/pw_{k}_{round(beta, 2)}.csv'):
        data_pw = get_samples_pw(beta, dimension=k, m_and_e_calc=True,
        file_path=f'pw_{k}_{round(beta, 2)}.csv', n_samples=n_samples)
    else:
        data_pw = pd.read_csv(f'H3/results/pw_{k}_{round(beta, 2)}.csv')

    if 'magnetization' in data_mcmc.columns:
        estimate_mcmc: float = data_mcmc['magnetization'].sum()
    else:
        raise ValueError(f"Column magnetization not in
        H3/results/mcmc_{k}_{round(beta, 2)}.csv's df: {data_mcmc.columns}")

    if 'magnetization' in data_pw.columns:
        estimate_pw: float = data_pw['magnetization'].sum()
    else:
        raise ValueError(f"Column magnetization not in
        H3/results/pw_{k}_{round(beta, 2)}.csv's df: {data_pw.columns}")

    return estimate_mcmc, estimate_pw
```

Celda de código 4.1: Función para estimar la magnetización de una rejilla $k \times k$ dada bajo un β dado usando los algoritmos de Metropolis-Hastings y Propp-Wilson o previas ejecuciones de la función almacenadas en archivos *.csv*.

En segundo lugar, la función `summarize_results` crea, como su nombre lo indica, un resumen de los resultados con Propp-Wilson o con Metropolis-Hastings, promediando la magnetización y, en el caso de la simulación perfecta, el tiempo de coalescencia para cada par tamaño de rejilla-β.

Esta función es usada para generar los argumentos que se pasarán a las cinco funciones que generan las seis gráficas que se consideraron propicias para visualizar los resultados (a continuación, el valor absoluto de la estimación de la magnetización se denotará como M^*):

- `plot_method` genera una gráfica de M^* vs. β usando los distintos puntos de las estimaciones para cada tamaño de rejilla ($k \times k$) juntándolos mediante una poligonal. Esto tanto para las estimaciones obtenidas con Propp-Wilson como con Metropolis-Hastings.
- `plot_combined` genera una gráfica como las generadas por `plot_method` pero usando los resultados de ambos métodos (las estimaciones con Propp-Wilson corresponden a las poligonales continuas y las de Metropolis-Hastings a poligonales punteadas).

- `plot_delta_errorbars` grafica el error absoluto de M^* usando Propp-Wilson comparado con el M^* usando Metropolis-Hastings en comparación con β con polígonales, y a cada valor calculado de β, aparte de añadir el error absoluto como un punto, agrega un 'rango' que corresponde a la desviación estandar la desviación estandar del mismo.
- `plot_heatmaps_with_shared_scale` construye dos mapas de calor de M^* para cada método/algoritmo comparando el tamaño de la rejilla vs. β.
- Finalmente, `plot_coupling_time` grafica el tiempo de coalescencia que, en promedio, se tardó el algoritmo de Propp-Wilson para que las configuraciones minimal y maximal coalescieran.

El código de todas estas funciones se encuentra en la penúltima celda del cuaderno de Jupyter. No se añaden aquí al considerarlos demasiado extensos e innecesarios cuando se tienen las gráficas ya compiladas. En suma, mencionar que todas salvo `plot_coupling_time` usan los valores verdaderos (es decir, no absolutos) de la magnetización si se pasa falso al parámetro `abs_magnetization`.

4.3. Análisis gráfico de la estimación

Por un lado (4.1 y 4.2, obtenidos con `plot_method`), como era de esperarse, el valor absoluto de la estimación de la magnetización, M^*, se mantiene cercano a 0 y crece muy lentamente cuando $\beta \in [0, 0.4]$. Al llegar a β_c, empieza a crecer más rápidamente, y para $\beta = 0.5$, algunos valores de k alcanzan valores por encima de 0.5. Al haber hecho la estimación con 10 muestras para $\beta = 0.5$, puede que la gráfica no se aproxime lo suficiente a la esperada, pero ciertamente se observa que M^* crece mucho más rápidamente al pasar β_c (en especial, usando Propp-Wilson).

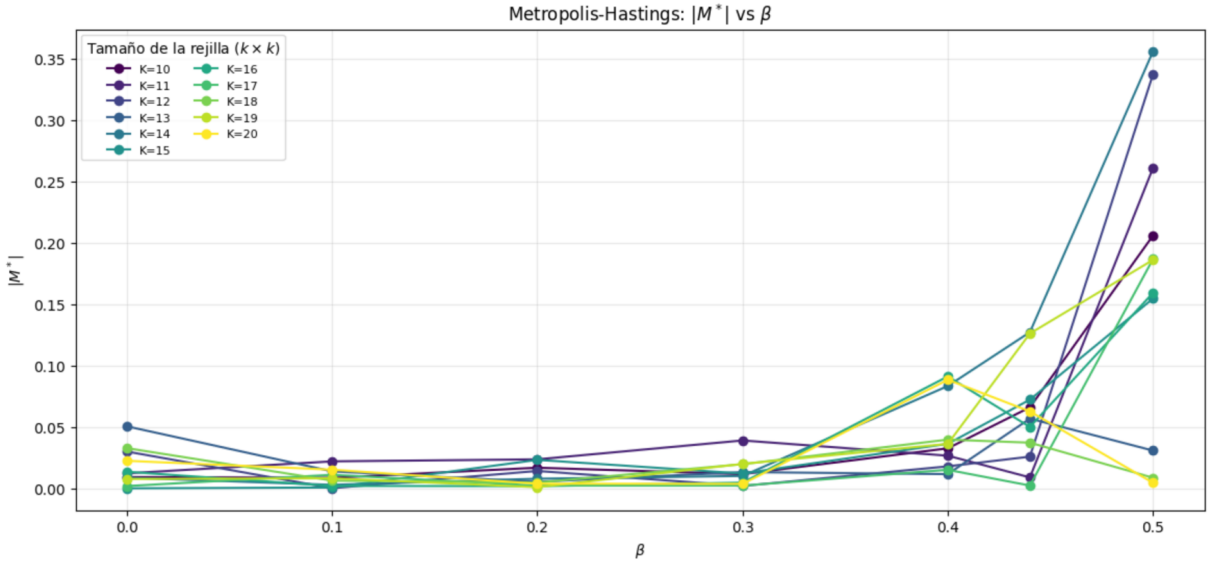

Figura 4.1: Estimación del valor esperado de la magnetización vs. β para las muestras tomadas usando MH para cada valor de k (en una rejilla $k \times k$).

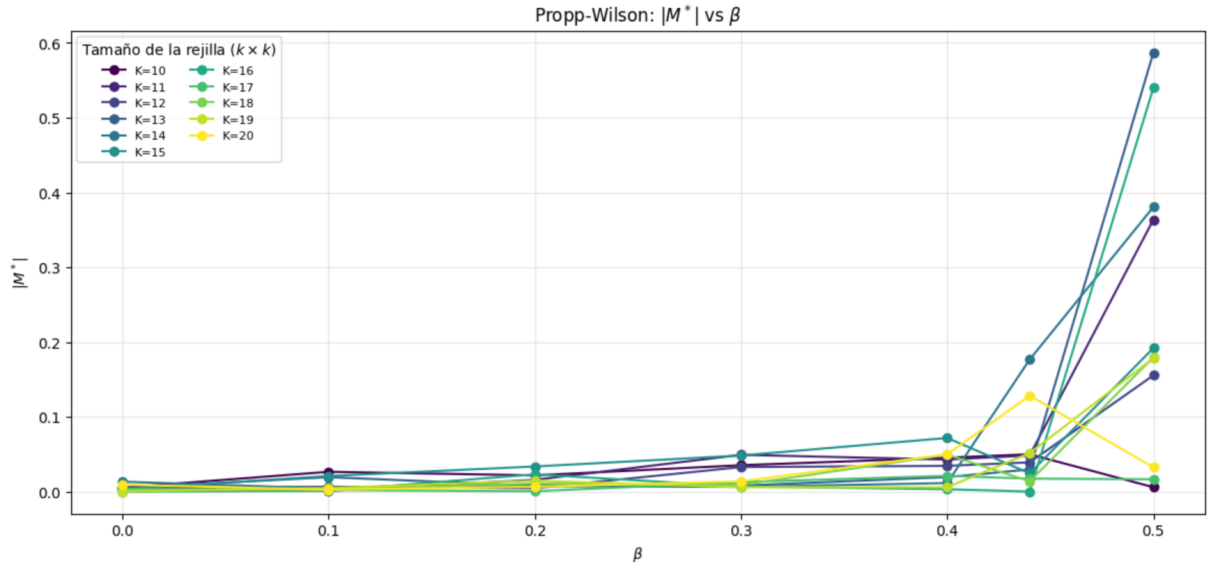

Figura 4.2: Estimación del valor esperado de la magnetización vs. β para las muestras tomadas usando PW para cada valor de k (en una rejilla $k \times k$).

Por otro lado (4.3 obtenido con `plot_combined`), aunque no consideramos la mejor forma de presentar los resultados, se presenta a continuación la 'unión' de los dos gráficos anteriores en uno solo. Lo más relevante es que la tendencia a crecer más rápido tras el β_c se evidencia más claramente en el caso de Propp-Wilson que en el de Metropolis-Hastings.

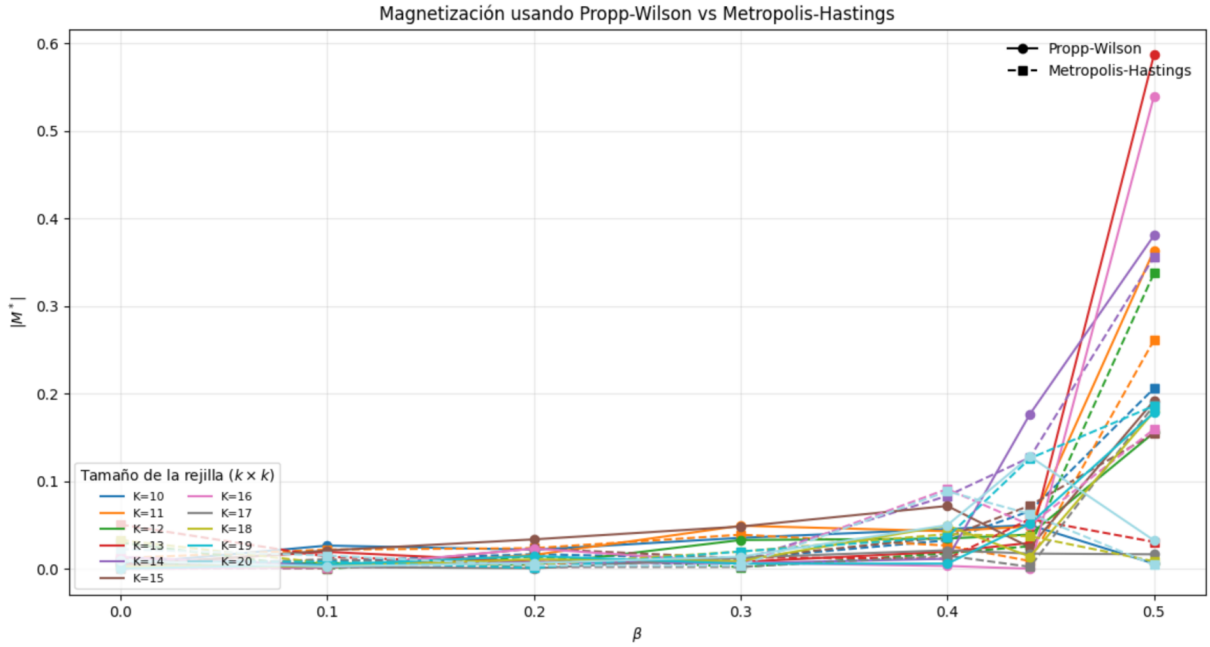

Figura 4.3: Estimación del valor esperado de la magnetización vs. β para las muestras tomadas usando MH (punteado) y PW (continuo) para cada valor de k (en una rejilla $k \times k$).

Justamente para visualizar mejor la información, se graficaron los mapas de calor (4.4 y 4.5 usando `plot_heatmaps_with_shared_scale`) para cada método, y la tendencia a crecer más rápido tras β_c es aún más evidente: antes de él, en ambos métodos se evidencian colores muy

oscuros ($M^* \approx 0$) sin importar el tamaño de la rejilla; en él, hay algunas tonalidades de morado para algunas rejillas ($M^* \approx 0.1$); y tras él, en $\beta = 0.5$, hay colores naranjas ($M^* \approx 0.5$) e incluso amarillos ($M^* \approx 0.8$).

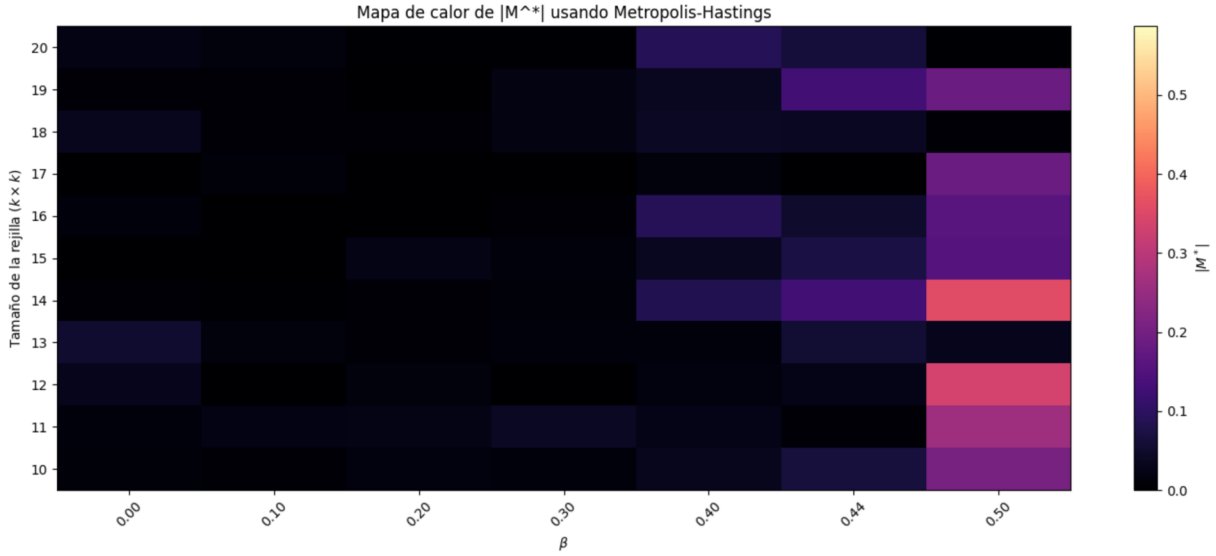

Figura 4.4: Mapa de calor en función del tamaño de la rejilla $k \times k$ y de β, con la escala de colores dada por el valor de la estimación de la magnetización usando MH.

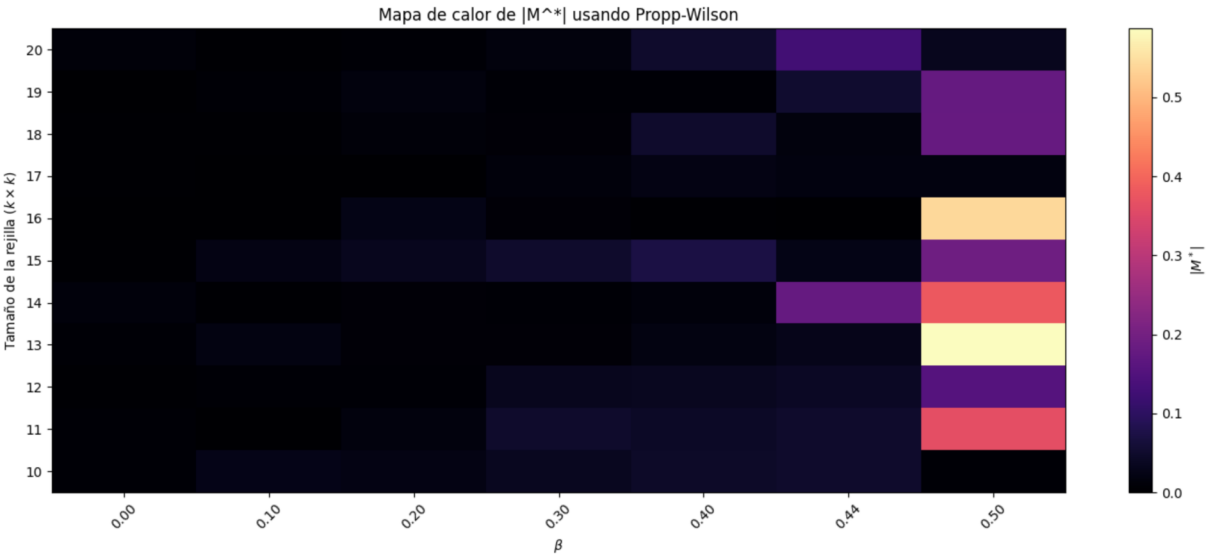

Figura 4.5: Mapa de calor en función del tamaño de la rejilla $k \times k$ y de β, con la escala de colores dada por el valor de la estimación de la magnetización usando PW.

Con el mismo objetivo de las dos gráficas anteriores, surgió una pregunta natural y es *¿qué tanto divergen los resultados de Metropolis-Hastings con respecto a los de Propp-Wilson?*, en especial considerando que con MH se obtiene solo una muestra aproximada de la distribución estacionaria objetivo mientras que con Propp-Wilson se garantiza una muestra exacta. Para ello, un gráfico del error absoluto entre las estimaciones con ambos métodos se realizó (4.6 usando `plot_delta_errorbars`), añadiéndole las desviaciones estándar para cada valor de k y β.

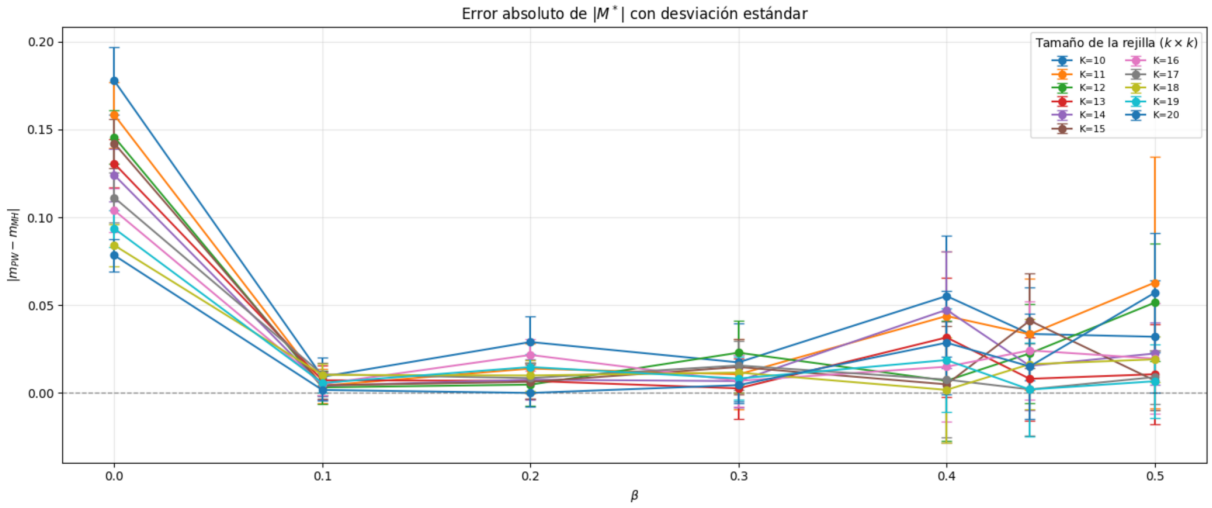

Figura 4.6: Error absoluto de las estimaciones usando MH y PW junto con su desviación estándar para cada tamaño de la rejilla $k \times k$.

4.4. Tiempo de coalescencia

Para recordar, el tiempo de coalescencia se define como el mínimo número de la sucesión de números positivos $\{N_m\}_{m\in\mathbb{N}}$ elegida para correr el algoritmo de Propp-Wilson, N_p, tal que al correr la simulación desde $-N_m$, para el tiempo 0, todos los estados han llegado al mismo estado. En el caso del modelo Ising, por el orden establecido, esto se reduce a que desde $-N_p$ hasta 0, las cadenas minimal y maximal sean la misma en algún tiempo en ese rango. A continuación, se muestra una tabla (4.2) con dichos tiempos de coalescencia en función de β y el tamaño de la rejilla ($k \times k$) y una gráfica con la misma información (4.7).

Cuadro 4.1: Data Table

k	0.00	0.10	0.20	0.30	0.40	0.44	0.50
10	747.52	1116.16	2048.00	4812.80	23961.60	113459.20	2136473.6
11	993.28	1382.40	2396.16	6656.00	34897.92	149094.40	5400166.4
12	1116.16	1761.28	3031.04	7741.44	43581.44	224624.64	13631488.0
13	1402.88	1976.32	3993.60	9297.92	63773.76	304087.04	24143462.4
14	1689.60	2375.68	4239.36	11550.72	71270.40	413532.16	37853593.6
15	2129.92	2949.12	5017.60	13271.04	87162.88	575733.76	119537664.0
16	2150.40	3543.04	6184.96	16875.52	106168.32	774307.84	263402291.2
17	2396.16	4157.44	7495.68	17694.72	110428.16	1087242.24	228170137.6
18	2949.12	4648.96	8355.84	20561.92	154664.96	1193410.16	516738252.8
19	3584.00	4833.28	9666.56	23920.64	174891.12	1578762.24	270113177.6
20	3911.68	5980.16	9830.40	28835.84	201195.52	1758968.24	272629760.0

Cuadro 4.2: Número de configuraciones factibles obtenidas con 100 muestras y 1000 pasos

Vale la pena resaltar que el tiempo de coalescencia crece mucho más rápido en función de β que del tamaño de la rejilla, lo cual sigue el comentario de 3.2: β influye en el cálculo de la exponencial, mientras que el tamaño de la rejilla solo hace que haya más variedad de vértices, pero a fin de cuentas, se elegirá uno siempre.

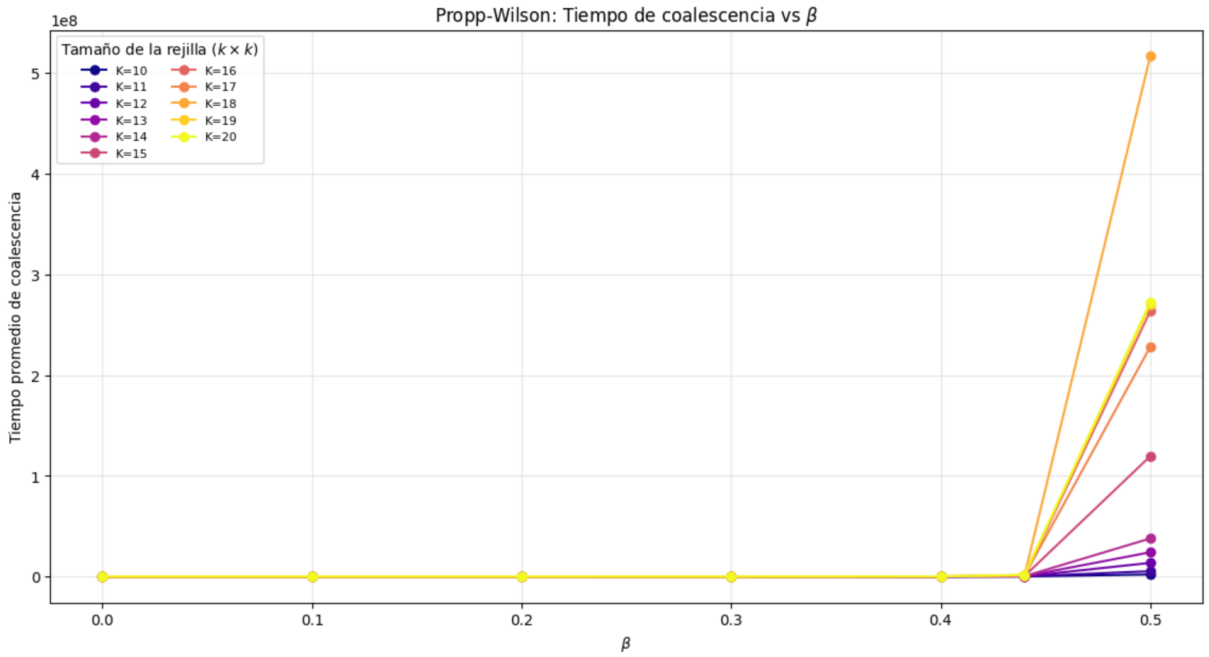

Figura 4.7: Tiempo de coalescencia vs. β (desde 0.0 hasta 0.5 con β_c) para cada tamaño de rejilla $k \times k$.

Bibliografía y referencias

---

1. Olle Häggström. Finite Markov Chains and Algorithmic Applications. Cambridge University Press (2010).