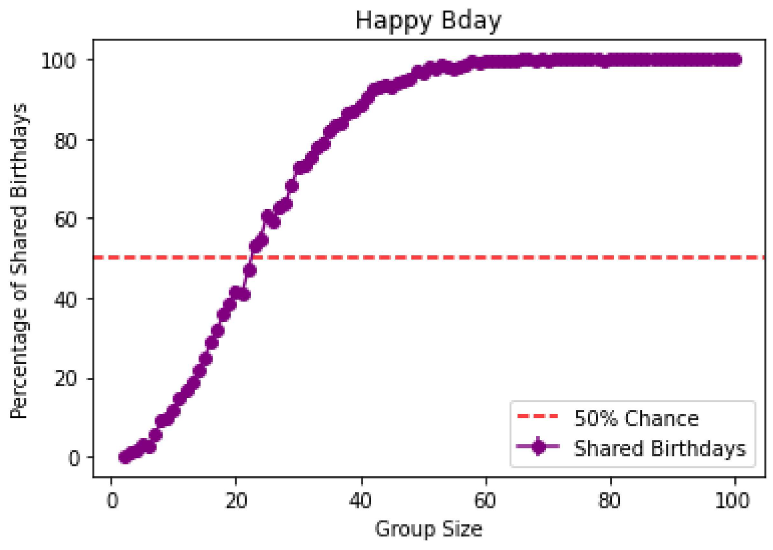
Computational Physics : Homework 9

In [8]:
```
import random
import numpy as np
import math
from math import cos,sin,exp
from matplotlib import cm
import matplotlib.pyplot as plt
pi = math.pi
```

Problem 1:

In [9]:
```python
def check_duplicates(birthdays):
    return len(birthdays) != len(set(birthdays))

def birthday_experiment(N, num_trials):
    shared_birthday_count = 0

    for _ in range(num_trials):
        birthdays = [random.randint(1, 365) for _ in range(N)]

        if check_duplicates(birthdays):
            shared_birthday_count += 1

    return shared_birthday_count / num_trials * 100

def main():
    np.random.seed(42)  # Set a random seed for reproducibility
    group_sizes = np.arange(2, 101)
    num_trials = 1000

    results = []

    for N in group_sizes:
        shared_percentage = birthday_experiment(N, num_trials)
        results.append(shared_percentage)

    # Calculate errors
    errors = np.sqrt(results * (100 - np.array(results)) / num_trials)

    # Plotting
    plt.errorbar(group_sizes, results, yerr=errors, fmt='o-', color = 'p
    plt.axhline(y=50, color='r', linestyle='--', label='50% Chance')
    plt.xlabel('Group Size')
    plt.ylabel('Percentage of Shared Birthdays')
    plt.title('Happy Bday')
    plt.legend()
    plt.show()

if __name__ == "__main__":
    main()
```


Happy Bday
Percentage of Shared Birthdays
Group Size
0
20
40
60
80
100
0
20
40
60
80
100
50% Chance
Shared Birthdays

Problem 2:

In [18]:

```python
def f1(x, y):
    return (x**2/16) + (y**2/9) - 1

def f2(x, y):
    return -((1 - (x**2/49))**0.5) + y/7 + 1

def f3(x, y):
    return ((1 - (x**2/49))**0.5) + y/7 + 1

def f4(x, y):
    return (1.5/0.3**2) * (x + 2.5)**2 + 1 - y

def f5(x, y):
    return (1.5/0.3**2) * (x - 2.5)**2 + 1 - y

def f6(x, y):
    return (x**2/49) + (y**2/9) - 1

def is_inside(x, y):
    return (
        f1(x, y).real <= 0 or
        f2(x, y).real <= 0 or
        f3(x, y).real <= 0 or
        f4(x, y).real <= 0 or
        f5(x, y).real <= 0 or
        f6(x, y).real <= 0
    )
def monte_carlo_simulation(num_points):
    inside_symbol = 0
    areas = []
    for _ in range(num_points):
        x = random.uniform(-8, 8)
        y = random.uniform(-6, 6)

        if is_inside(x, y):
            inside_symbol += 1
        bounding_box_area = 14 * 5  # (8 - (-8)) * (6 - (-6))
        symbol_area_estimate = (inside_symbol / num_points) * bounding_b
        areas.append(symbol_area_estimate)

    return symbol_area_estimate, areas

# Number of random points to generate
num_points = 1000000
points = np.linspace(1,1000000,1000000 )
# Run the Monte Carlo simulation
area_estimate, area_list = monte_carlo_simulation(num_points)

plt.plot(points,area_list)
plt.xlabel('')

print(f"Estimated area of the Batman symbol: {area_estimate} square unit
```

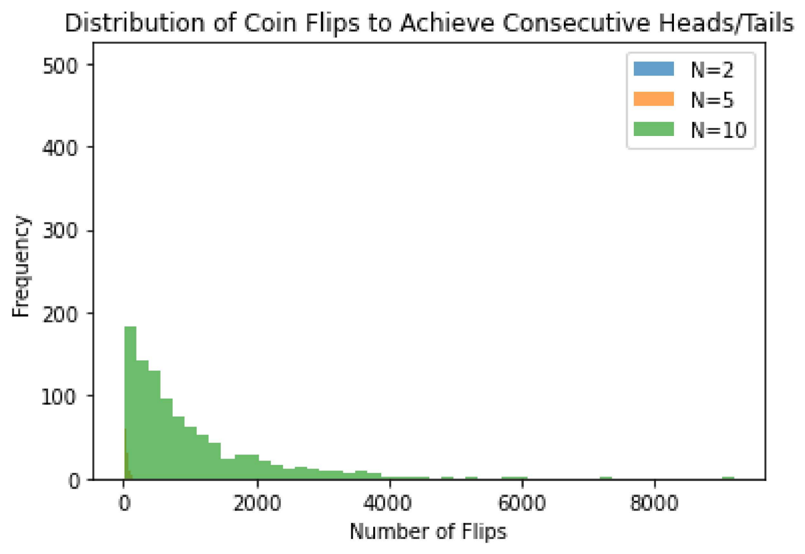
```
Estimated area of the Batman symbol: 41.5835 square units
```

Problem 3:

In [10]:

```
def simulate_coin_flips(N, trials):
    results = []

    for _ in range(trials):
        flips = 0
        consecutive_heads = 0
        consecutive_tails = 0

        while consecutive_heads < N and consecutive_tails < N:
            flip = random.choice(['H', 'T'])
            flips += 1

            if flip == 'H':
                consecutive_heads += 1
                consecutive_tails = 0
            else:
                consecutive_tails += 1
                consecutive_heads = 0

        results.append(flips)

    return results

def plot_results(N_values, trials):
    for N in N_values:
        random.seed(42)  # Set a random seed for reproducibility
        results = simulate_coin_flips(N, trials)
        average_flips = sum(results) / len(results)

        print(f"Average number of flips for {N} consecutive heads/tails:

        plt.hist(results, bins=50, alpha=0.7, label=f'N={N}')

    plt.title('Distribution of Coin Flips to Achieve Consecutive Heads/T
    plt.xlabel('Number of Flips')
    plt.ylabel('Frequency')
    plt.legend()
    plt.show()

# Define N values and the number of trials
N_values = [2, 5, 10]
trials = 1000

# Plot the results
plot_results(N_values, trials)
```

```
Average number of flips for 2 consecutive heads/tails: 2.962
Average number of flips for 5 consecutive heads/tails: 30.955
Average number of flips for 10 consecutive heads/tails: 990.375
```



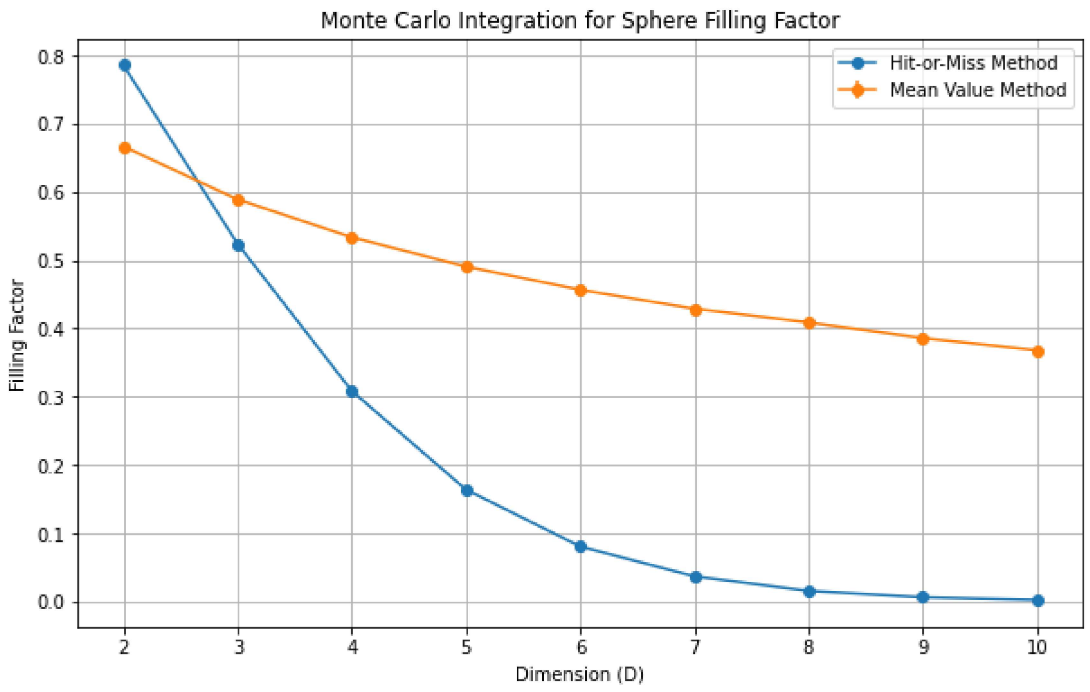
Problem 4:

In [6]:

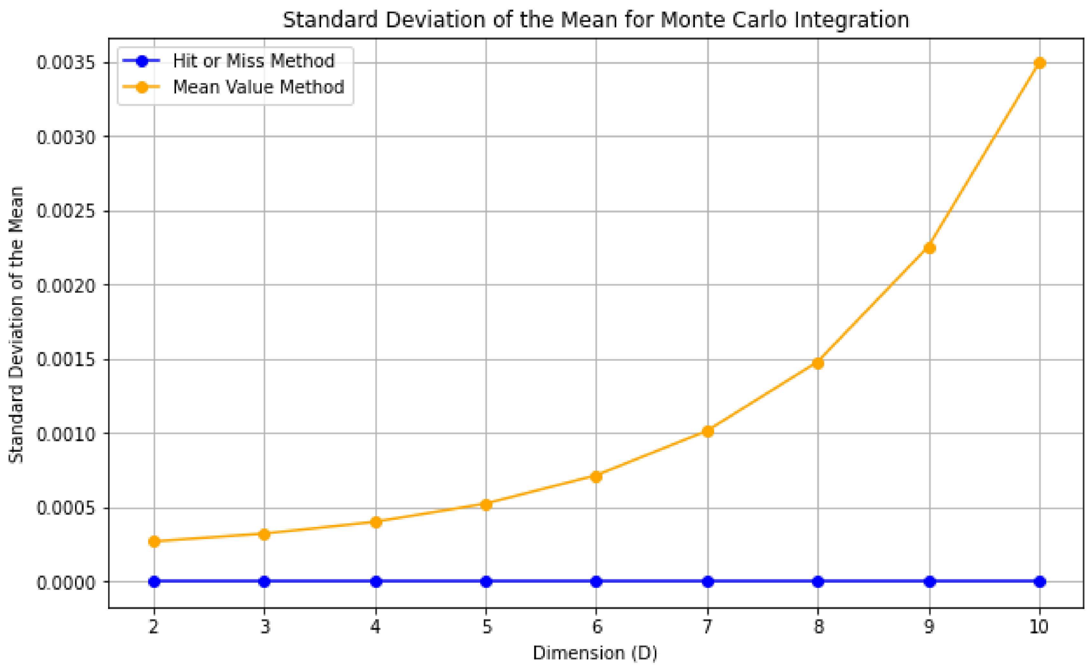
```
# part a: solve for filling factor using both monte carlo methods
# D=3
np.random.seed(42)
def hit_or_miss_method(dimension):
    points = np.random.rand(10**6, dimension)
    # Check if the point is inside the sphere
    inside_sphere = np.sum(points**2, axis=1) <= 1.0
    # Calculate the filling factor
    filling_factor = np.sum(inside_sphere) / len(inside_sphere)
    return filling_factor

def mean_value_method(dimension):
    points = np.random.rand(10**6, dimension)
    # Calculate the squared distance from the origin
    distance_squared = np.sum(points**2, axis=1)
    # Check if the point is inside the sphere
    inside_sphere = distance_squared <= 1.0
    # Calculate the filling factor using the mean value method
    filling_factor = np.mean(np.sqrt(1.0 - distance_squared[inside_spher
    std_dev = np.std(np.sqrt(1.0 - distance_squared[inside_sphere])) / n
    return filling_factor, std_dev

dimension = 3

# Hit-or-miss method
filling_factor_hit_or_miss = hit_or_miss_method(dimension)
print(f'Hit-or-Miss Method - Filling Factor (D={dimension}): {filling_fa

# Mean value method
filling_factor_mean_value, std_dev_mean_value = mean_value_method(dimens
print(f'Mean Value Method - Filling Factor (D={dimension}): {filling_fac
```

```
Hit-or-Miss Method - Filling Factor (D=3): 0.52343
Mean Value Method - Filling Factor (D=3): 0.5887370948885384 +/- 0.0003
179790797933394
```

In [7]:
```
def hit_or_miss_method(dimension):
    points = np.random.rand(10**6, dimension)
    # Check if the point is inside the sphere
    inside_sphere = np.sum(points**2, axis=1) <= 1.0
    # Calculate the filling factor
    filling_factor = np.sum(inside_sphere) / len(inside_sphere)
    return filling_factor

def mean_value_method(dimension):
    points = np.random.rand(10**6, dimension)
    # Calculate the squared distance from the origin
    distance_squared = np.sum(points**2, axis=1)
    # Check if the point is inside the sphere
    inside_sphere = distance_squared <= 1.0
    # Calculate the filling factor using the mean value method
    filling_factor = np.mean(np.sqrt(1.0 - distance_squared[inside_spher
    std_dev = np.std(np.sqrt(1.0 - distance_squared[inside_sphere])) / n
    return filling_factor, std_dev

dimensions = np.arange(2, 11)
filling_factors_hit_or_miss = np.zeros_like(dimensions, dtype=float)
filling_factors_mean_value = np.zeros_like(dimensions, dtype=float)
std_devs_mean_value = np.zeros_like(dimensions, dtype=float)

for i, dimension in enumerate(dimensions):
    filling_factors_hit_or_miss[i] = hit_or_miss_method(dimension)
    filling_factor_mean_value, std_dev_mean_value = mean_value_method(di
    filling_factors_mean_value[i] = filling_factor_mean_value
    std_devs_mean_value[i] = std_dev_mean_value

plt.figure(figsize=(10, 6))
plt.plot(dimensions, filling_factors_hit_or_miss, marker='o', label='Hit
plt.errorbar(dimensions, filling_factors_mean_value, yerr=std_devs_mean_
plt.title('Monte Carlo Integration for Sphere Filling Factor')
plt.xlabel('Dimension (D)')
plt.ylabel('Filling Factor')
plt.legend()
plt.grid(True)
plt.show()
```


Monte Carlo Integration for Sphere Filling Factor
Hit-or-Miss Method
Mean Value Method
Filling Factor
Dimension (D)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
2
3
4
5
6
7
8
9
10

In [11]:
```
np.random.seed(42)

def hit_or_miss_method(dimension):
    points = np.random.rand(10**6, dimension)
    # Check if the point is inside the sphere
    inside_sphere = np.sum(points**2, axis=1) <= 1.0
    # Calculate the filling factor
    filling_factor = np.sum(inside_sphere) / len(inside_sphere)
    return filling_factor

def mean_value_method(dimension):
    points = np.random.rand(10**6, dimension)
    # Calculate the squared distance from the origin
    distance_squared = np.sum(points**2, axis=1)
    # Check if the point is inside the sphere
    inside_sphere = distance_squared <= 1.0
    # Calculate the filling factor using the mean value method
    filling_factor = np.mean(np.sqrt(1.0 - distance_squared[inside_spher
    std_dev = np.std(np.sqrt(1.0 - distance_squared[inside_sphere])) / n
    return filling_factor, std_dev

dimensions = np.arange(2, 11)
filling_factors_hit_or_miss = np.zeros_like(dimensions, dtype=float)
filling_factors_mean_value = np.zeros_like(dimensions, dtype=float)
std_devs_hit_or_miss = np.zeros_like(dimensions, dtype=float)
std_devs_mean_value = np.zeros_like(dimensions, dtype=float)

for i, dimension in enumerate(dimensions):
    filling_factors_hit_or_miss[i] = hit_or_miss_method(dimension)
    filling_factor_mean_value, std_dev_mean_value = mean_value_method(di
    filling_factors_mean_value[i] = filling_factor_mean_value
    std_devs_mean_value[i] = std_dev_mean_value

# Plotting results
plt.figure(figsize=(10, 6))
plt.plot(dimensions, std_devs_hit_or_miss, 'o-', label='Hit or Miss Meth
plt.plot(dimensions, std_devs_mean_value, 'o-', label='Mean Value Method
plt.title('Standard Deviation of the Mean for Monte Carlo Integration')
plt.xlabel('Dimension (D)')
plt.ylabel('Standard Deviation of the Mean')
plt.legend()
plt.grid(True)
plt.show()
```



Problem 5:

In [19]:

```
# part a
m = 5.31e-26 # kg
r = 10e-15 # m (radius of a proton)

# R1 - R2 in terms of impact parameter
# theta = arcsin(b/2r)
def R1R2(r, b):
    return 2*r*(cos(pi - np.arcsin(b/2*r)) + sin(pi - np.arcsin(b/2*r)))

def R2R1(r, b):
    return 2*r*(cos(np.arcsin(b/2*r) - pi) + sin(np.arcsin(b/2*r) - pi))

# resulting velocities in terms of impact parameter
# mass cancels out since m1 = m2
def v1_out(v1, v2, R):
    return v1 - (((v1-v2) * R)/(abs(R)**2)) * R

def v2_out(v1, v2, R, R2):
    return v2 - (((v1-v2) * R)/(abs(R)**2)) * R2
```

In [20]:
```
# part b
mass = 5.31e-26  # Mass of particles in kg
num_particles = 1000

# Generate particle speeds
speeds = np.full(num_particles, 100.0)  # Initialize all speeds to 100 m
speeds[:10] = 500.0  # Set speeds of the first 10 particles to 500 m/s

# Compute average speed
average_speed = np.mean(speeds)

# Compute root mean square (RMS) speed
rms_speed = np.sqrt(np.mean(speeds**2))

# Print results
print(f"Average Speed: {average_speed:.2f} m/s")
print(f"RMS Speed: {rms_speed:.2f} m/s")
speeds = speeds.tolist()
```

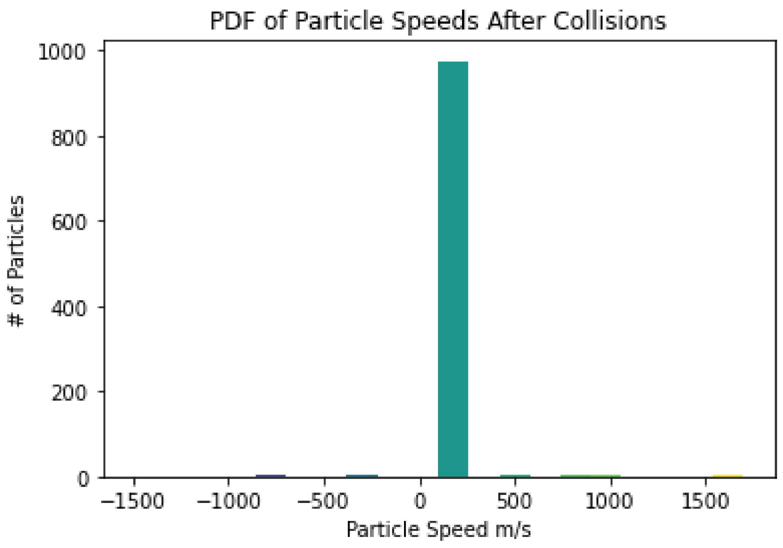
```
Average Speed: 104.00 m/s
RMS Speed: 111.36 m/s
```

In [13]:

```
# part c
theta_vals = np.linspace(0, 2*pi, 1000, endpoint = False)
b_vals = np.linspace(-2*r, 2*r, 1000)

i = 0
for i in range(1000):
    # step 1: choose 2 particles
    random_particles = random.sample(speeds, k=2)
    particle1 = random_particles[0]
    particle2 = random_particles[1]
    indices = [speeds.index(v) for v in random_particles]

    # step 2: choose random theta
    theta1 = np.random.choice(theta_vals, 1, replace = True)
    theta2 = 0

    # step 3: choose random b
    b_val = np.random.choice(b_vals, 1, replace = True)

    R1minusR2 = R1R2(r, b_val)
    R2minusR1 = R2R1(r, b_val)

    # step 4: compute velocities
    v1_final = v1_out(particle1, particle2, R1minusR2)
    v2_final = v2_out(particle1, particle2, R1minusR2, R2minusR1)

    # step 5: update speeds with new velocities
    speeds[indices[0]] = v1_final
    speeds[indices[1]] = v2_final

# part d
speeds = np.array(speeds)
# Compute average speed
average_speed = np.mean(speeds)

# Compute root mean square (RMS) speed
rms_speed = np.sqrt(np.mean(speeds**2))

# Print results
print(f"Average Speed: {average_speed:.2f} m/s")
print(f"RMS Speed: {rms_speed:.2f} m/s")
```

```
Average Speed: 107.20 m/s
RMS Speed: 170.41 m/s
```

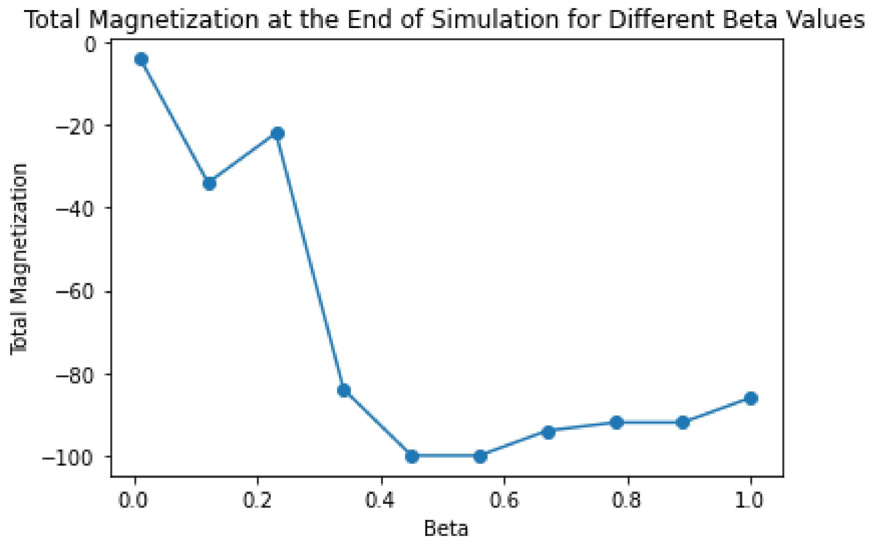
In [14]:
```
# part e
cm = plt.cm.get_cmap('viridis')
N, bins, patches = plt.hist(speeds, bins=20)
plt.xlabel('Particle Speed m/s')
plt.ylabel('# of Particles')
plt.title('PDF of Particle Speeds After Collisions')

bin_centers = 0.5 * (bins[:-1] + bins[1:])
col = bin_centers - min(bin_centers)
col /= max(col)
for c, p in zip(col, patches):
    plt.setp(p, 'facecolor', cm(c))
```



Problem 6

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt

def initialize_grid(N):
    return 2 * np.random.randint(2, size=(N, N)) - 1

def calculate_magnetization(grid):
    return np.sum(grid)

def update_spin(beta, spin, neighbors_sum):
    m = spin * (4 * neighbors_sum + 2)
    probabilities = np.array([np.exp(-beta * m), np.exp(-beta * (-m))])
    normalized_probabilities = probabilities / np.sum(probabilities)
    return np.random.choice([1, -1], p=normalized_probabilities)

def update_grid(beta, grid):
    N = len(grid)
    updated_grid = np.copy(grid)

    for i in range(N):
        for j in range(N):
            neighbors_sum = (
                grid[(i - 1) % N, j]
                + grid[(i + 1) % N, j]
                + grid[i, (j - 1) % N]
                + grid[i, (j + 1) % N]
            )
            updated_grid[i, j] = update_spin(beta, grid[i, j], neighbors

    return updated_grid

def run_simulation(N, steps, beta):
    grid = initialize_grid(N)

    for _ in range(steps):
        grid = update_grid(beta, grid)

    return calculate_magnetization(grid)

# Set random seed for reproducibility
np.random.seed(42)

# Parameters
N = 10
steps = 5000

# Generate values of beta
beta_values = np.linspace(0.01, 1, 10)

# Run the simulation for each beta value
final_magnetizations = []
for beta in beta_values:
    final_magnetization = run_simulation(N, steps, beta)
    final_magnetizations.append(final_magnetization)

# Plot the results
plt.plot(beta_values, final_magnetizations, marker='o')
plt.xlabel("Beta")
plt.ylabel("Total Magnetization")
plt.title("Total Magnetization at the End of Simulation for Different Be
plt.show()
```


Total Magnetization at the End of Simulation for Different Beta Values
Total Magnetization
Beta