3D_run

June 14, 2023

**ML Visualisation of 3D and 2D Ising model

Ising 3d

Specifically, it utilizes a Monte Carlo simulation to evolve the system over time and study the behavior of the spins.

In Monte Carlo simulations, random sampling and probabilistic rules are employed to approximate and analyze complex systems. In the case of the Ising Model, the Monte Carlo simulation is used to update the spins at each step based on the current state and probabilistic rules. This allows for the exploration of different spin configurations and the study of the system's properties at different temperatures.

The specific Monte Carlo algorithm employed in the Ising Model simulation code is the Metropolis algorithm. In this algorithm, each spin is sequentially considered, and the probability of flipping its state is calculated based on the energy change and the temperature. The spin is then flipped with a certain probability determined by the Metropolis acceptance criteria.

In the Ising Model, the magnetization arises from the interaction between neighboring spins and the temperature of the system. The spins tend to align with their neighbors to minimize energy, leading to the formation of domains with aligned spins. The simulation code captures this behavior by updating the spins based on their local interactions and the temperature parameter.

Spins: The spins are represented by discrete values of -1 and +1, corresponding to the "down" and "up" states, respectively; the spins are stored in a 3D array called particles

Interaction: The spins interact with their nearest neighbors. The strength of this interaction is determined by the interaction parameter interaction passed to the IsingModel3d constructor; the interaction parameter is denoted as self.j

Temperature: The temperature of the system affects the probability of spin flips. Higher temperatures make it more likely for spins to change their state. The temperature is passed to the IsingModel3d constructor as temperature and stored in self.temperature.

Simulation Step: The simulation evolves the system over time by updating the spins according to a probabilistic rule. In the code, this is implemented in the make_simulation_step method. Each spin is considered in a sequential manner, and the probability of flipping its state is determined based on the energy change and the temperature. The spin is flipped with a certain probability calculated using the Metropolis algorithm.

Magnetism: The magnetism of the system is a measure of the net alignment of the spins. In the code, the magnetism is calculated using the get_magnetism method, which sums up the values of all spins.

The modeling approach used in the code follows a Monte Carlo simulation method. The simulation proceeds in discrete steps, with each step updating the spins based on the current state and the defined rules. The simulation can be run for a specified number of steps (n_max) and can be repeated multiple times at different temperatures (simulations_per_temperature).

Overall, the code provides a framework to simulate and visualize the behavior of the Ising Model in 3D, allowing you to observe the collective behavior of spins and explore phase transitions at different temperatures.

In the context of the Ising Model, the Metropolis algorithm is used to update the spins in each step of the simulation. The algorithm follows these steps:

1)Randomly select a spin in the lattice. 2)Calculate the energy change if the spin is flipped. 3)If the energy change is negative (i.e., the flipped configuration has lower energy), accept the flip and update the spin. 4)If the energy change is positive, accept the flip with a probability determined by the Boltzmann factor exp(-ΔE/kT), where ΔE is the energy change, k is the Boltzmann constant, and T is the temperature. 5)Repeat steps 1-4 for a specified number of iterations or until reaching a convergence criterion. By using the Metropolis algorithm, the Ising Model simulation explores the configuration space, allowing the system to equilibrate at a given temperature and reach a state that represents the thermal equilibrium of the system.

the Boltzmann factor is used implicitly in the calculation of the probability for accepting or rejecting a spin flip during the Metropolis algorithm. The Boltzmann factor incorporates the temperature of the system and relates it to the energy difference between two states.

In the Metropolis algorithm, the probability of accepting a spin flip is given by the Boltzmann factor, which is defined as:(P=exp(-ΔE/k_b*T)) where:

P is the probability of accepting the flip ΔE is the change in energy due to the spin flip k B is the Boltzmann constant

T is the temperature of the system In the else block, the Boltzmann factor is calculated as math.exp(-delta_h / self.temperature), where delta_h represents the energy difference and self.temperature is the temperature of the system. The Boltzmann factor represents the probability of accepting a spin flip with a positive energy difference. If a randomly generated number is less than the Boltzmann factor (random.random() < prob), the spin flip is accepted. Otherwise, it is rejected.

[1]:
```
from IsingModel import IsingModel
from PyQt5.QtWidgets import QApplication
from pyqtgraph.Qt import QtCore, QtGui
import pyqtgraph.opengl as gl
from typing import Iterable
import sys


class IsingModel3d(IsingModel):
    RED = (1, 0.1, 0.1, 0.1)
    BLUE = (0.1, 0.1, 1, 0.1)
```

```
  def __init__(self, x_size: int, y_size: int, z_size: int, temperature:␣
↪float, interaction: float,
               initializer: callable((int, int, int)) = lambda x, y, z: 1,␣
↪sphere_radius: float = 0.2):
      super().__init__(x_size, y_size, z_size, temperature, interaction,␣
↪initializer)
      self.sphere_radius = sphere_radius
      self.app = QApplication(sys.argv)
      self.w = gl.GLViewWidget()
      self.w.opts['distance'] = 40
      self.w.setWindowTitle('Ising Model')
      self.w.setGeometry(0, 50, 1920, 1080)

      # Create a grid on the sides of the model
      # gx = gl.GLGridItem(size=QtGui.QVector3D(y_size, z_size, 1))
      # gx.rotate(90, 0, 1, 0)
      # gx.translate(-x_size / 2, 0, 0)
      # self.w.addItem(gx)
      # gy = gl.GLGridItem(size=QtGui.QVector3D(x_size, z_size, 1))
      # gy.rotate(90, 1, 0, 0)
      # gy.translate(0, -y_size / 2, 0)
      # self.w.addItem(gy)
      # gz = gl.GLGridItem(size=QtGui.QVector3D(x_size, y_size, 1))
      # gz.translate(0, 0, -z_size / 2)
      # self.w.addItem(gz)

      self.points = []
      for i in range(self.x_size):
          self.points.append([])
          for j in range(self.y_size):
              self.points[i].append([])
              for k in range(self.z_size):
                  sphere_color = IsingModel3d.RED if self.particles[i][j][k]␣
↪== -1 else IsingModel3d.BLUE
                  sphere = self.create_sphere(sphere_color, (i, j, k))
                  self.points[i][j].append(sphere)
                  self.w.addItem(sphere)

  def create_sphere(self, color: tuple[float, float, float, float], position:␣
↪tuple[int, int, int]) -> gl.GLMeshItem:
      sphere = gl.MeshData.sphere(4, 4, radius=self.sphere_radius)
      item = gl.GLMeshItem(meshdata=sphere, smooth=False, color=color,␣
↪shader='balloon', glOptions='additive')
      item.translate(*position)
      return item
```

```
    def start(self):
        if (sys.flags.interactive != 1) or not hasattr(QtCore, 'PYQT_VERSION'):
            QApplication.instance().exec_()

    def update_points(self):
        for i in range(self.x_size):
            for j in range(self.y_size):
                for k in range(self.z_size):
                    self.points[i][j][k].setColor(
                        IsingModel3d.RED if self.particles[i][j][k] == -1 else␣
↪IsingModel3d.BLUE)

    def make_visualized_step(self):
        self.make_simulation_step()
        self.update()

    def run_simulation(self, n_max: int, simulations_per_temperature: int = 10,␣
↪generate_graphs: bool = True,
                       visualize: bool = False,
                       temperatures_list: Iterable[float] = ()):
        if visualize:
            self.w.show()
            timer = QtCore.QTimer()
            timer.timeout.connect(self.make_visualized_step)
            timer.start(100)
            self.start()
            return
        else:
            super().run_simulation(n_max, simulations_per_temperature,␣
↪generate_graphs, temperatures_list)

    def update(self):
        self.update_points()
```

[2]:
```
model = IsingModel3d(x_size=10, y_size=10, z_size=1, temperature=2.3,␣
↪interaction=0.5)

# Run the simulation without visualization
model.run_simulation(n_max=30)

# Run the simulation with visualization
model.run_simulation(n_max=30, visualize=True)
```

[5]:
```
!pip install -r requirements.txt
```

```
Requirement already satisfied: pyopengl in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
```

```
(from -r requirements.txt (line 1)) (3.1.7)
Requirement already satisfied: pyqt5 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from -r requirements.txt (line 2)) (5.15.9)
Requirement already satisfied: pyqtgraph in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from -r requirements.txt (line 3)) (0.13.3)
Requirement already satisfied: matplotlib in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from -r requirements.txt (line 4)) (3.7.0)
Requirement already satisfied: PyQt5-sip<13,>=12.11 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from pyqt5->-r requirements.txt (line 2)) (12.12.1)
Requirement already satisfied: PyQt5-Qt5>=5.15.2 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from pyqt5->-r requirements.txt (line 2)) (5.15.2)
Requirement already satisfied: numpy>=1.20.0 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from pyqtgraph->-r requirements.txt (line 3)) (1.24.2)
Requirement already satisfied: contourpy>=1.0.1 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 4)) (1.0.7)
Requirement already satisfied: cycler>=0.10 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 4)) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 4)) (4.38.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 4)) (1.4.4)
Requirement already satisfied: packaging>=20.0 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 4)) (23.0)
Requirement already satisfied: pillow>=6.2.0 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 4)) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 4)) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from matplotlib->-r requirements.txt (line 4)) (2.8.2)
Requirement already satisfied: six>=1.5 in
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
(from python-dateutil>=2.7->matplotlib->-r requirements.txt (line 4)) (1.16.0)
```

In the make_simulation_step() method, a random spin is selected, and the energy difference (ΔH)

is calculated using the calculate_h() method. If the energy difference is negative (ΔH 0), the spin flip is accepted. If the energy difference is positive (ΔH>0), the Metropolis criterion is applied by calculating the probability of accepting the spin flip using the Boltzmann factor. If the random number generated is less than the acceptance probability, the spin flip is accepted; otherwise, it is rejected.

Although the code does not explicitly mention the Metropolis algorithm, the acceptance or rejection of spin flips based on the energy difference and the Boltzmann factor is a characteristic of the Metropolis algorithm. Therefore, the code can be considered to implement a variation of the Metropolis algorithm for the Ising model simulation.

the Boltzmann factor is used in the code. It is utilized in the make_simulation_step() method when determining whether to accept or reject a spin flip based on the energy difference between two states; In the else block, the Boltzmann factor is calculated as math.exp(-delta_h / self.temperature), where delta_h represents the energy difference and self.temperature is the temperature of the system. The Boltzmann factor represents the probability of accepting a spin flip with a positive energy difference. If a randomly generated number is less than the Boltzmann factor (random.random() < prob), the spin flip is accepted. Otherwise, it is rejected.

[1]:
```
import random
import math
from typing import Iterable, Sized
import matplotlib.pyplot as plt


class IsingModel:
    def __init__(self, x_size: int, y_size: int, z_size: int, temperature:␣
 ↪float, interaction: float,
                 initializer: callable((int, int, int)) = lambda x, y, z: 1):
        self.initializer = initializer
        self.particles = [[[initializer(x, y, z) for z in range(z_size)] for y␣
 ↪in range(y_size)] for x in range(x_size)]
        self.temperature = temperature
        self.x_size = x_size
        self.y_size = y_size
        self.z_size = z_size
        self.n = self.x_size * self.y_size * self.z_size
        self.j = interaction
        self.average_magnetism_points = []
        self.final_magnetism_points = []

    def initialize_particles(self):
        self.particles = [[[self.initializer(x, y, z) for z in range(self.
 ↪z_size)]
                           for y in range(self.y_size)] for x in range(self.
 ↪x_size)]

    @classmethod
```

```
    def from_file(cls, filename: str) -> 'IsingModel':
        with open(filename, 'r') as file:
            x, y, z, temperature, interaction = list(map(int, file.readline().
↪split()))
            model = IsingModel(x, y, z, temperature, interaction)
            for i in range(x):
                for j in range(y):
                    spins = list(map(int, file.readline().split()))
                    for k in range(z):
                        model[i, j, k] = spins[k]
            return model

    def get_magnetism(self) -> float:
        res = 0
        for i in range(self.n):
            res += self[i]
        return res / self.n


    def calculate_h(self, x: int, y: int, z: int):
        current = self.particles[x][y][z]
        res = 0
        for coords in ((-1, 0, 0), (1, 0, 0), (0, -1, 0), (0, 1, 0), (0, 0,␣
↪-1), (0, 0, 1)):
            res += current * self.particles[(x + coords[0]) % self.
↪x_size][(coords[1] + y) % self.y_size][(coords[2] + z) % self.z_size]
        return -self.j * res

    def make_simulation_step(self):
        for _ in range(self.n):
            x = random.randint(0, self.x_size - 1)
            y = random.randint(0, self.y_size - 1)
            z = random.randint(0, self.z_size - 1)
            h_old = self.calculate_h(x, y, z)
            h_new = -h_old
            delta_h = h_new - h_old
            if delta_h <= 0:
                self.particles[x][y][z] *= -1
            else:
                prob = math.exp(-delta_h / self.temperature)
                if random.random() < prob:
                    self.particles[x][y][z] *= -1

    def _n_index_to_xyz(self, key: int) -> (int, int, int):
        x = key // (self.y_size * self.z_size)
        key %= self.y_size * self.z_size
        y = key // self.z_size
```

```
        z = key % self.z_size
        return x, y, z

    def __getitem__(self, key: tuple[int, int, int] or int):
        if isinstance(key, int):
            x, y, z = self._n_index_to_xyz(key)
        else:
            x, y, z = key
        return self.particles[x][y][z]

    def __setitem__(self, key: tuple[int, int, int] or int, value: 1 or -1):
        if isinstance(key, int):
            x, y, z = self._n_index_to_xyz(key)
        else:
            x, y, z = key
        self.particles[x][y][z] = value

    @staticmethod
    def make_magnetism_progression_graph(magnetism: Sized and Iterable[float],␣
↪output_filename: str):
        plt.close()
        plt.plot(range(1, len(magnetism) + 1), magnetism)
        plt.xlabel('Iteration')
        plt.ylabel('Magnetism')
        plt.savefig(output_filename + '.png')

    @staticmethod
    def make_magnetism_by_temperature_graph(data: dict[float, float],␣
↪output_filename: str):
        plt.close()
        items = list(sorted(data.items()))
        plt.plot(list(map(lambda x: x[0], items)), list(map(lambda x: x[1],␣
↪items)))
        plt.xlabel('Temperature')
        plt.ylabel('Magnetism')
        plt.savefig(output_filename + '.png')

    def add_average_magnetism(self, temperature: float, magnetism: float):
        self.average_magnetism_points.append((temperature, magnetism))

    def add_final_magnetism(self, temperature: float, magnetism: float):
        self.final_magnetism_points.append((temperature, magnetism))

    @staticmethod
    def scatter_points(points: Iterable[tuple[float, float]], output_filename):
        plt.close()
        fig, ax = plt.subplots()
```

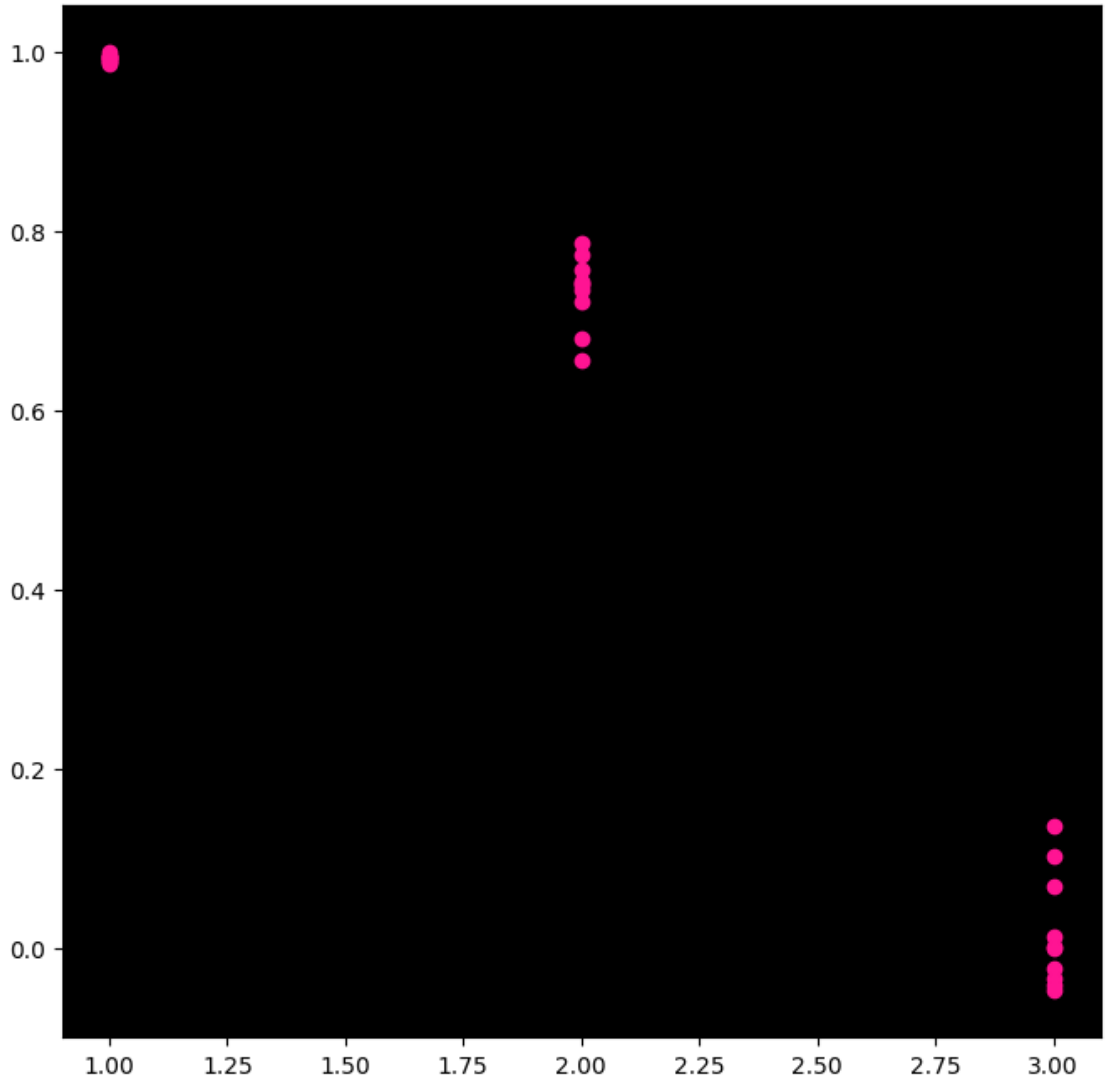
```
        x = list(map(lambda _x: _x[0], points))
        y = list(map(lambda _x: _x[1], points))

        ax.scatter(x, y, c='deeppink')


        ax.set_facecolor('black')

        fig.set_figwidth(8)
        fig.set_figheight(8)  #      "Figure"

        plt.savefig(output_filename)

    def make_points_graphs(self):
        self.scatter_points(self.average_magnetism_points,␣
↪'average_magnetism_each_simulation')
        self.scatter_points(self.final_magnetism_points,␣
↪'final_magnetism_each_simulation')

    def run_simulation(self, n_max: int, simulations_per_temperature: int = 10,␣
↪generate_graphs: bool = True,
                       temperatures_list: Iterable[float] = ()):

        average_magnetism_by_temperature = {}
        final_magnetism_by_temperature = {}
        for temperature in temperatures_list:
            print(f'Temperature: {temperature}')
            self.temperature = temperature
            average_magnetism = []
            final_magnetism = []
            for simulation_n in range(simulations_per_temperature):
                print(f'Simulation: {simulation_n}')
                self.initialize_particles()
                current_step_magnetism = []
                for step in range(n_max):
                    self.make_simulation_step()
                    magnetism = self.get_magnetism()
                    current_step_magnetism.append(magnetism)
                final_magnetism.append(self.get_magnetism())
                average_magnetism.append(sum(current_step_magnetism) /␣
↪len(current_step_magnetism))
                self.add_final_magnetism(temperature, final_magnetism[-1])
                self.add_average_magnetism(temperature, average_magnetism[-1])
                # make just one magnetism progression graph per one temperature
                if simulation_n == 0 and generate_graphs:
                    self.
↪make_magnetism_progression_graph(current_step_magnetism,
```

```
                                                                    ␣
↪f'temp_{temperature}_magnetism_progression')

            average_magnetism_by_temperature[temperature] = sum(map(abs,␣
↪average_magnetism)) / len(average_magnetism)
            final_magnetism_by_temperature[temperature] = sum(map(abs,␣
↪final_magnetism)) / len(final_magnetism)

            # Generate graphs after each temperature to observe changes␣
↪dynamically
            # and not just after the end of all the simulation
            if generate_graphs:
                self.
↪make_magnetism_by_temperature_graph(average_magnetism_by_temperature,␣
↪'average_magnetism_by_temperature')
                self.
↪make_magnetism_by_temperature_graph(final_magnetism_by_temperature,␣
↪'final_magnetism_by_temperature')
                self.make_points_graphs()
```

[2]:
```
model = IsingModel(x_size=10, y_size=10, z_size=10, temperature=3.0,␣
↪interaction=0.5)
model.run_simulation(
    n_max=100,
    simulations_per_temperature=10,
    generate_graphs=True,
    temperatures_list=[1.0, 2.0, 3.0]
)
```

```
Temperature: 1.0
Simulation: 0
Simulation: 1
Simulation: 2
Simulation: 3
Simulation: 4
Simulation: 5
Simulation: 6
Simulation: 7
Simulation: 8
Simulation: 9
Temperature: 2.0
Simulation: 0
Simulation: 1
Simulation: 2
Simulation: 3
Simulation: 4
Simulation: 5
```

```
Simulation: 6
Simulation: 7
Simulation: 8
Simulation: 9
Temperature: 3.0
Simulation: 0
Simulation: 1
Simulation: 2
Simulation: 3
Simulation: 4
Simulation: 5
Simulation: 6
Simulation: 7
Simulation: 8
Simulation: 9
```



For J > 0 the state of lowest energy is when all spins are aligned. The state has macroscopic magnetizaion, i.e. it is ferromagnetic. When temperature is low Ising spins minimize energy. Interaction aligns all spin vectors in the same direction, giving huge total magnetic fields. At high temperature thermal fluctuations break spins order. The randomness of the spin configuration tends to wash out the large scale magnetism. In the 2D Ising model there is a phase transition at Tc = 2.269 from disordered (non-magnetic) to ordered magnetic state

If the exchange interaction constant J < 0 , then for low temperature nearest spins are anti-aligned. In the simplest Ising antiferromagnetic on square grid they form two ordered sub-lattices (disposed as cells on a chess-board). Starting at random configuration you can see clusters of ordered phase formation and movement of domain walls. Antiferromagnetic ordering appears in the thermostat algorithm for negative temperature.

The Metropolis algorithm In the Metropolis algorithm we try to turn over a single spin direction with transition probability W12 = exp[(E1-E2)/T] if E1 < E2 W12 = 1 if E1 > E2 where E1, E2 are energies of the old and new configurations (see details in the Gould and Tobochnik book). Statistical averages may then be computed as simple arithmetic means. The Metropolis algorithm can by applied easy e.g. to the XY model simulation. There are many possible choices for W12. To get equilibrium final spins distribution it is enough that Wij satisfy the detailed balance conditions W12 exp(-E1/T) = W21 exp(-E2/T) or W12 / W21 = exp[(E1-E2)/T].